

Algorithmen für Routenplanung

19. Vorlesung, Sommersemester 2023

Adrian Feilhauer | 26. Juni/1. Juli 2023



Graph-basiert:

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

Graph-basiert:

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

Näher an den Eingabedaten:

- RAPTOR

Aber: Daten müssen pro Route zusammengefasst werden

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?

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: $\text{trip}(c)$

Earliest Arrival Connection Scan

Connection Scan

Idee:

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

Idee:

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

Datenstruktur:

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

Idee:

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

Datenstruktur:

- Provisorische 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

Idee:

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

Datenstruktur:

- Provisorische 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: $d[v_{\text{dep}}(c)] \leq \tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c))$
 $\tau_{\text{ch}}(\cdot)$ ist Umstiegszeit
 - bereits im Trip sitzt: $r[\text{trip}(c)] = \text{true}$

Idee:

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

Datenstruktur:

- Provisorische 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: $d[v_{\text{dep}}(c)] \leq \tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c))$
 $\tau_{\text{ch}}(\cdot)$ ist Umstiegszeit
 - bereits im Trip sitzt: $r[\text{trip}(c)] = \text{true}$
- Wenn c erreichbar ist, dann

Idee:

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

Datenstruktur:

- Provisorische 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: $d[v_{\text{dep}}(c)] \leq \tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c))$
 $\tau_{\text{ch}}(\cdot)$ ist Umstiegszeit
 - bereits im Trip sitzt: $r[\text{trip}(c)] = \text{true}$
- Wenn c erreichbar ist, dann
 - könnte man sitzen bleiben: $r[\text{trip}(c)] \leftarrow \text{true}$
 - könnte man aussteigen: $d[v_{\text{arr}}(c)] \leftarrow \min\{d[v_{\text{arr}}(c)], \tau_{\text{arr}}(c)\}$

Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

Ausgabe: Früheste Ankunftszeit an Zielstop

Umstiegszeit (in min)		4	10	5	7	3											
Ankunftszeit an Stop	...	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$...										
Connections																	
sortiert nach	...	V_{dep}	V_{arr}	τ_{dep}	τ_{arr}	trip	V_{dep}	V_{arr}	τ_{dep}	τ_{arr}	trip	V_{dep}	V_{arr}	τ_{dep}	τ_{arr}	trip	...
Abfahrtszeit																	
Ist Trip erreichbar?	...	false					false					...					

Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

Ausgabe: Früheste Ankunftszeit an Zielstop

Umstiegszeit (in min)	4	10	5	7	3	...
Ankunftszeit an Stop	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$...

Connections													...			
sortiert nach	V_{dep}	V_{arr}	9:00	9:25	trip	V_{dep}	V_{arr}	9:15	9:45	trip	V_{dep}	V_{arr}	9:25	9:55	trip	...
Abfahrtszeit													...			

Ist Trip erreichbar?	false					false					...
----------------------	-------	--	--	--	--	-------	--	--	--	--	-----

Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

Ausgabe: Früheste Ankunftszeit an Zielstop

Umstiegszeit (in min)		4	10	5	7	3	
Ankunftszeit an Stop	...	$+\infty$	8:00	$+\infty$	$+\infty$	$+\infty$...

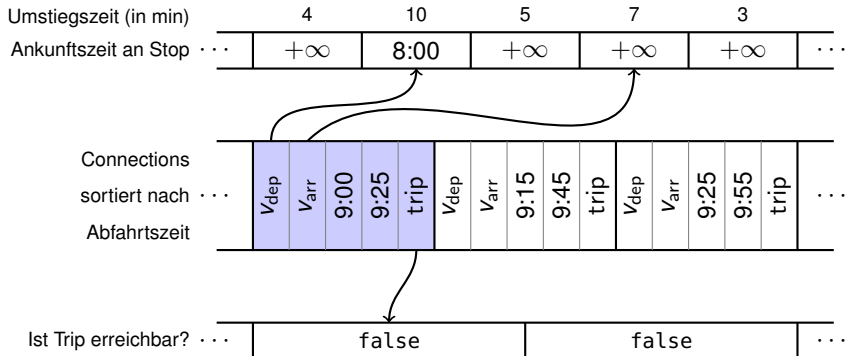
Connections																	
sortiert nach	...	V_{dep}	V_{arr}	9:00	9:25	trip	V_{dep}	V_{arr}	9:15	9:45	trip	V_{dep}	V_{arr}	9:25	9:55	trip	...
Abfahrtszeit																	

Ist Trip erreichbar?	...	false					false					...
----------------------	-----	-------	--	--	--	--	-------	--	--	--	--	-----

Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

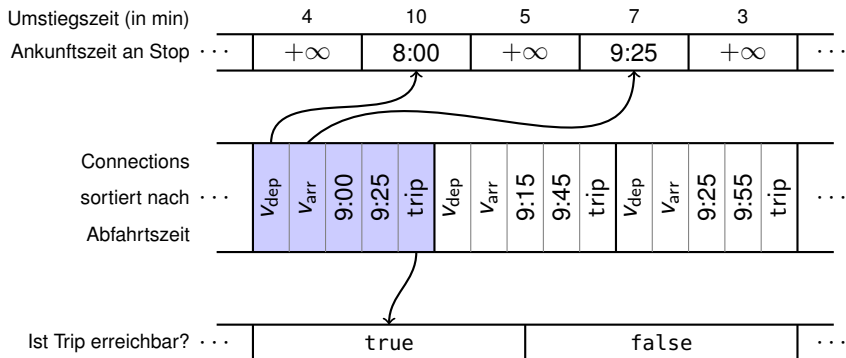
Ausgabe: Früheste Ankunftszeit an Zielstop



Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

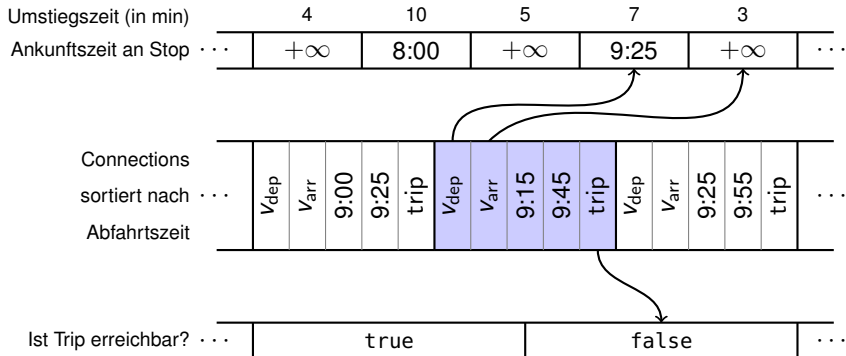
Ausgabe: Früheste Ankunftszeit an Zielstop



Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

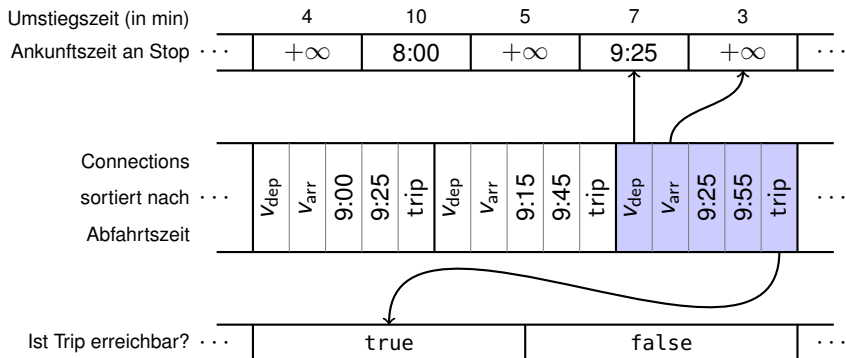
Ausgabe: Früheste Ankunftszeit an Zielstop



Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

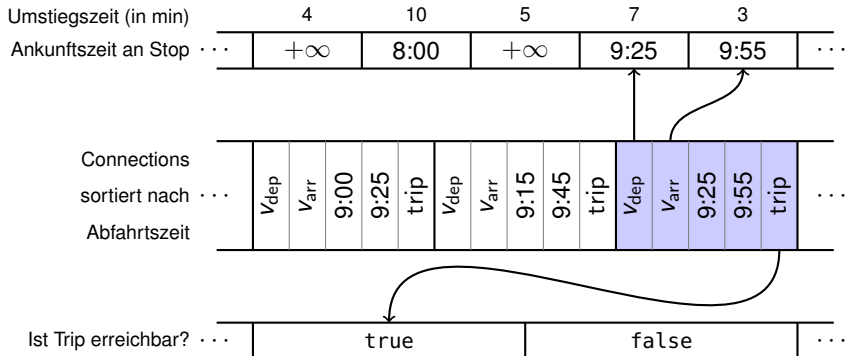
Ausgabe: Früheste Ankunftszeit an Zielstop



Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

Ausgabe: Früheste Ankunftszeit an Zielstop



Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop

Ausgabe: Früheste Ankunftszeit an Zielstop

Umstiegszeit (in min)		4	10	5	7	3	
Ankunftszeit an Stop ...		$+\infty$	8:00	$+\infty$	9:25	9:55	...

Connections																
sortiert nach ...	V_{dep}	V_{arr}	9:00	9:25	trip	V_{dep}	V_{arr}	9:15	9:45	trip	V_{dep}	V_{arr}	9:25	9:55	trip	...
Abfahrtszeit																

Ist Trip erreichbar? ...		true				false				
--------------------------	--	------	--	--	--	-------	--	--	--	--

$d[v] \leftarrow \infty$ für alle Stops v

$d[s] \leftarrow \tau_{\text{dep}} - \tau_{\text{ch}}(s)$

$r[T] \leftarrow \text{false}$ 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** $r[\text{trip}(c)] = \text{true}$ **then**

$r[\text{trip}(c)] \leftarrow \text{true}$

$d[v_{\text{arr}}(c)] \leftarrow \min\{d[v_{\text{arr}}(c)], \tau_{\text{arr}}(c)\}$

Nach Scannen von Connection c :

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

Nach Scannen von Connection c :

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

Begründung:

- Angenommen, c verbessert $d[v_{arr}(c)]$ nicht
- Dann wurde bereits eine bessere Journey J nach $v_{arr}(c)$ gefunden
- Für jeden Fußweg $(v_{arr}(c), v)$: 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!

$d[v] \leftarrow \infty$ für alle Stops v

$d[s] \leftarrow \tau_{\text{dep}} - \tau_{\text{ch}}(s)$

$r[T] \leftarrow \text{false}$ für alle Trips T

for alle Fußwege (s, v) mit Länge l do

└ $d[v] \leftarrow \min\{d[v], \tau_{\text{dep}} + l\}$

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 $r[\text{trip}(c)] = \text{true}$ then**

└ $r[\text{trip}(c)] \leftarrow \text{true}$

┌ **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**

└└ $d[v] \leftarrow \min\{d[v], \tau_{\text{arr}}(c) + l\}$

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?

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

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

Profile Connection Scan

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

Screenshot von [bahn.de](https://www.bahn.de)

Profil-Anfragen

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

Lösung: Journeys für eine Zeitspanne angeben.

Startstop →

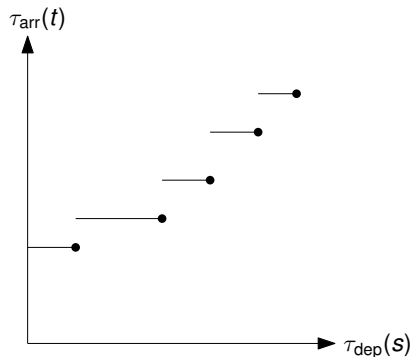
Zielstop →

→ 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	

minimale Abfahrtszeit

maximale Ankunftszeit

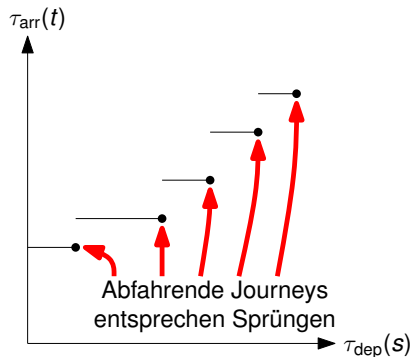
Screenshot von bahn.de



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t , Zeitspanne $[\tau_{\min}, \tau_{\max}]$

Ausgabe: s - t -Profil für jeden Stop s (außer t)



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t , Zeitspanne $[\tau_{min}, \tau_{max}]$

Ausgabe: s - t -Profil für jeden Stop s (außer t)

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  $P[v_{\text{arr}}(c)]$ )
```

```
//  $\tau_c$ : Optimale Ankunftszeit, falls  $c$  benutzt wird
```

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

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

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

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

Idee:

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

Idee:

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

Beobachtung:

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

- 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**

$d_t[v] \leftarrow \ell$

$d_t[t] \leftarrow 0$

...

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

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

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

- Aus

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

- wird

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

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

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

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

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

Stop-Datenstruktur

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

- Aus

Initialisiere Profil $P[v]$ pro Stop v

- wird

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

Stop-Datenstruktur

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

- Aus

$\tau_3 \leftarrow$ Ankunftszeit, wenn man umsteigt

- wird

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

Stop-Datenstruktur

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

- Aus

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

- wird

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

- $P[v]$ ist stückweise lineare Ankunftszeit-Funktion
- ⇒ Speichere $P[v]$ als Array von Breakpoints (d, a)
 $d \hat{=}$ Abfahrtszeit, $a \hat{=}$ 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

- 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  
   $P[v] \leftarrow \{(\infty, \infty)\}$ 
```

- Auswertung mit binärer oder linearer Suche
- Hier ist lineare Suche besser
- Aus

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

- wird

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

- 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

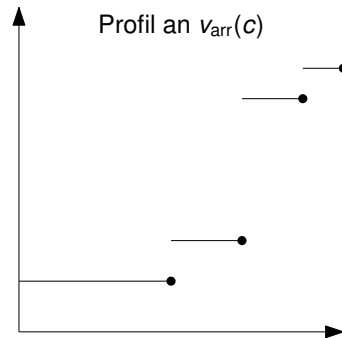
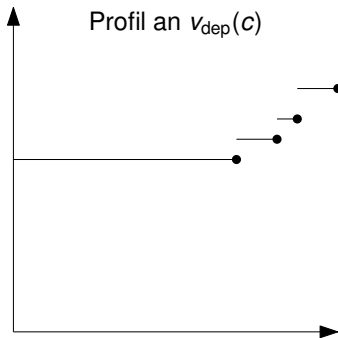
Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c)), \tau_c)$ in $P[v_{\text{dep}}(c)]$ ein

- wird

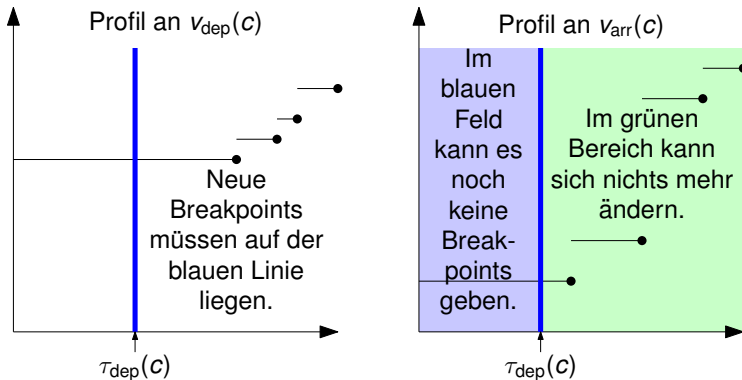
```
(d, a) ← (τdep(c) - τch(vdep(c)), τc)  
if a < P[vdep(c)][0].a then  
  | if d = P[vdep(c)][0].d then  
  | | P[vdep(c)][0].a ← a  
  | else  
  | | Füge (d, a) am Anfang von P[vdep(c)] ein
```


Profile Connection Scan

Für jede Connection c **absteigend** nach Abfahrtszeit:

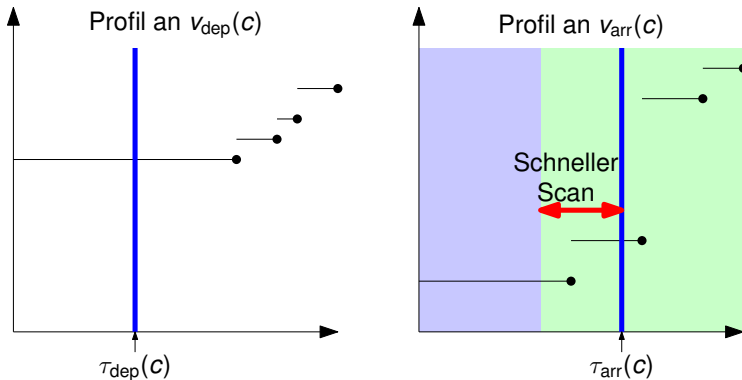


Für jede Connection c **absteigend** nach Abfahrtszeit:



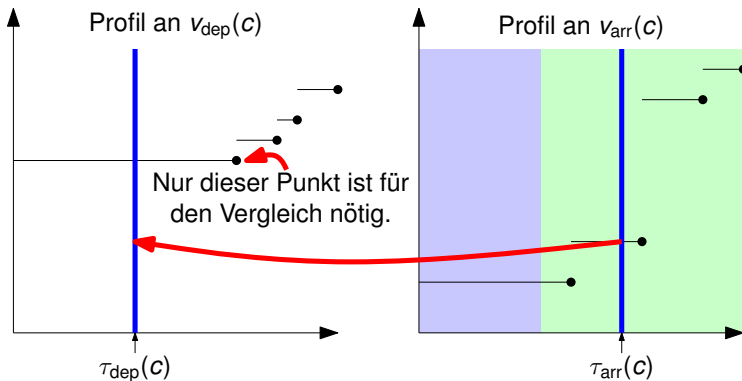
Profile Connection Scan

Für jede Connection c **absteigend** nach Abfahrtszeit:



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

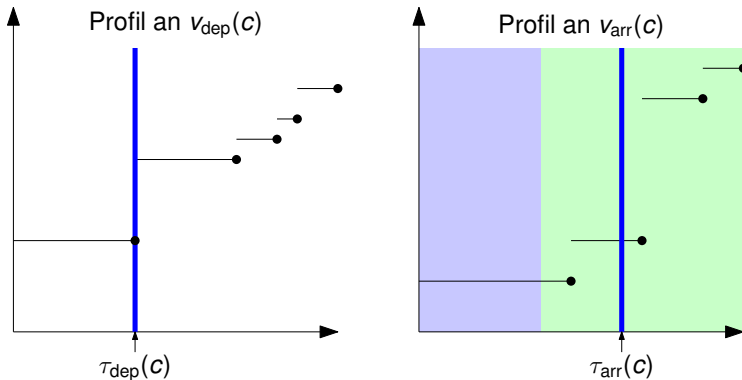
Für jede Connection c **absteigend** nach Abfahrtszeit:



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.

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

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

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

- **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

Warum?

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

Warum?

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

Antwort:

- Auswertung von $P[v_{arr}(c)]$ zum Zeitpunkt $\tau_{arr}(c)$:
 - Finde erste erreichbare Connection c' , die an $v_{arr}(c)$ abfährt
 - $P[v_{arr}(c)]$ hat einen Breakpoint für c'
 - Gib Ankunftszeit dieses Breakpoints aus
 - Index des Breakpoints für c' ist unabhängig von t
- Berechne Index von c' für jede Connection c vor
- Profilauswertung in $\mathcal{O}(1)$
- Zeit für Profilsuche beweisbar linear in der Anzahl an Connections

τ_{\min} und τ_{\max} verwenden

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

τ_{\min} und τ_{\max} verwenden

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

τ_{\min} und τ_{\max} verwenden:

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

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

- **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

- **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

- 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

Problem:

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

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

Aus

Füge (d, a) in $P[v]$ ein

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  
     $P[v][0].a \leftarrow a$   
  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
```

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

Umstiege Pareto-optimieren

Ziel:

- Ankunftszeit und Umstiege im Pareto-Sinn 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

Broadcast

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

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 $z_i = \min\{x_i, y_i\}$

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: $(\infty, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$

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

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  $P[v_{\text{arr}}(c)]$ )
```

```
//  $\tau_c$ : Optimale Ankunftszeit, falls  $c$  benutzt wird
```

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

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

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

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

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

- 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

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  
   $d_{\text{trip}}[T] \leftarrow \text{broadcast}(\infty)$ 
```

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

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

Stop-Datenstruktur

- Profil bildet Abfahrtszeit auf Vektor von Ankunftszeiten ab

- Aus

Initialisiere Profil $P[v]$ pro Stop v

- wird

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

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

- **Bisher:** $P[v]$ ist Array von Breakpoints (d, a)
- **Nun:** $P[v]$ ist Array von Breakpoints $(d, a = (a_1, \dots, a_8))$
- 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 $(\tau_{\text{arr}}^1, \dots, \tau_{\text{arr}}^8)$
- In τ_{arr}^i steht, wann ich an t ankomme, wenn ich
 - um τ_{dep}
 - an v losfahre
 - und höchstens i mal aussteigen darf

- 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  
   $P[v] \leftarrow \{(\infty, \text{broadcast}(\infty))\}$ 
```

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

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

- wird

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

- 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  $a \not\geq P[v_{\text{dep}}(c)][0].a$  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
```

- Vektor a der Länge n hat die Komponenten:

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

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

- Vektor a der Länge n hat die Komponenten:

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

- 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

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

Problem: Algorithmus ist memory-bound

Der Speicher ist fast vollständig mit Ankunftszeiten gefüllt

→ Komprimiere Ankunftszeiten

Beobachtung: Nicht an jedem Zeitpunkt fährt ein Zug

Idee: Berechne für jeden Stop ein geordnetes Array von Zeitpunkten $\tau_0, \tau_1, \dots, \tau_n$, 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

Problem: Algorithmus ist memory-bound

Der Speicher ist fast vollständig mit Ankunftszeiten gefüllt

→ Komprimiere Ankunftszeiten

Beobachtung: Nicht an jedem Zeitpunkt fährt ein Zug

Idee: Berechne für jeden Stop ein geordnetes Array von Zeitpunkten $\tau_0, \tau_1, \dots, \tau_n$, 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)

Option 1:

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

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

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.)

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



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.