

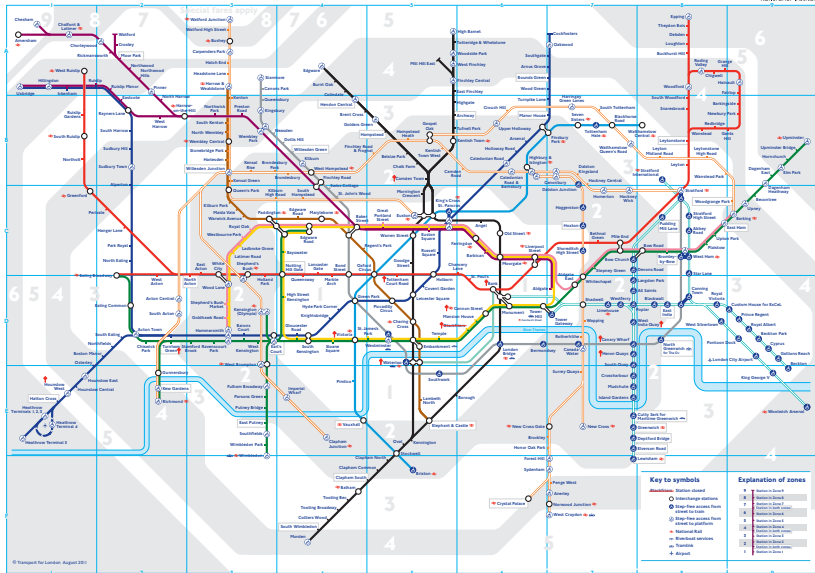
# Algorithmen für Routenplanung

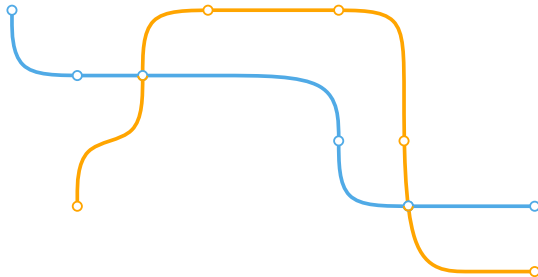
18. Vorlesung, Sommersemester 2022

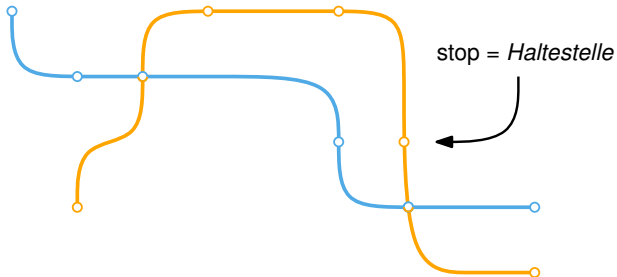
Adrian Feilhauer | 29. Juni 2022

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

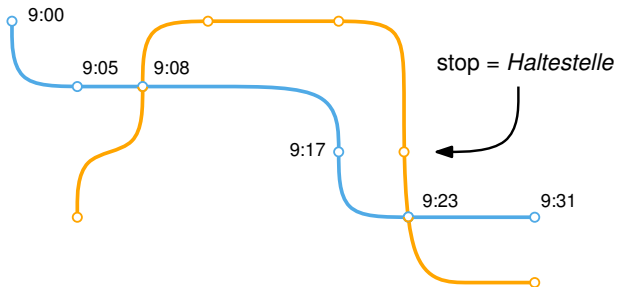
# Fahrplanauskunft



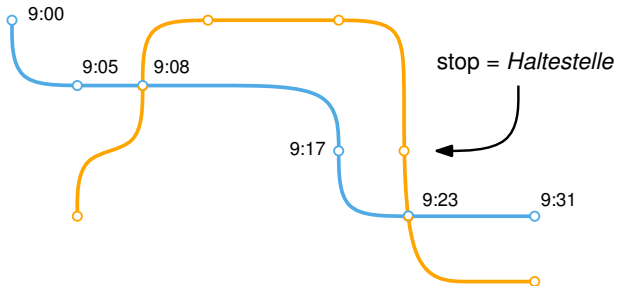
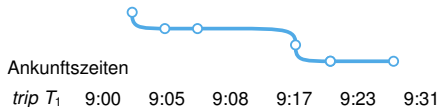




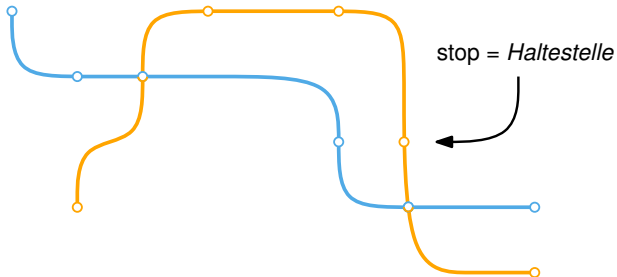
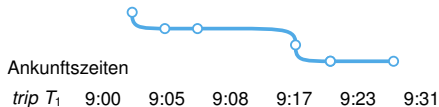
# Wdh. Begriffe



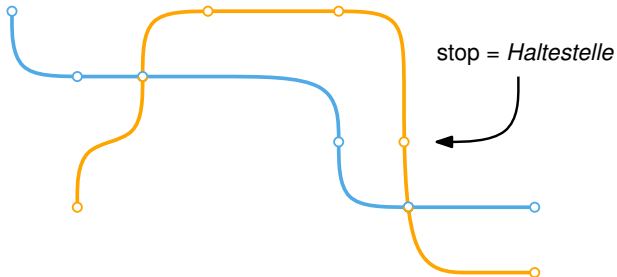
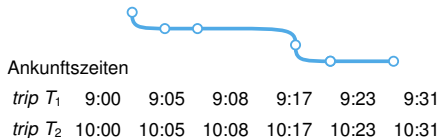
# Wdh. Begriffe



# Wdh. Begriffe




# Wdh. Begriffe



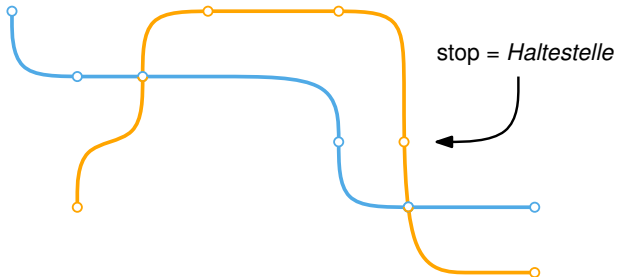


# Wdh. Begriffe



Ankunftszeiten

trip $T_1$	9:00	9:05	9:08	9:17	9:23	9:31
trip $T_2$	10:00	10:05	10:08	10:17	10:23	10:31
trip $T_3$	10:30	10:35	10:38	—	—	10:53



# Wdh. Begriffe

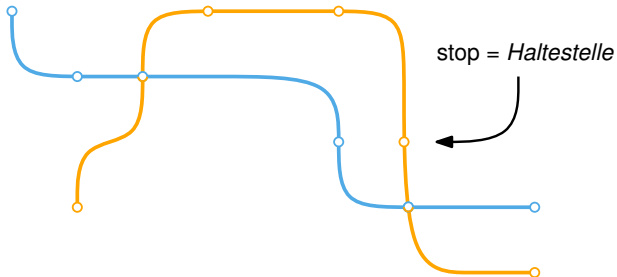


Ankunftszeiten

<i>trip</i> $T_1$	9:00	9:05	9:08	9:17	9:23	9:31
<i>trip</i> $T_2$	10:00	10:05	10:08	10:17	10:23	10:31
<i>trip</i> $T_3$	10:30	10:35	10:38	—	—	10:53

Route  $R_1$

Route  $R_2$



# Wdh. Begriffe

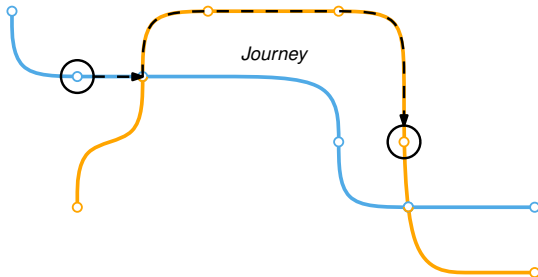


Ankunftszeiten

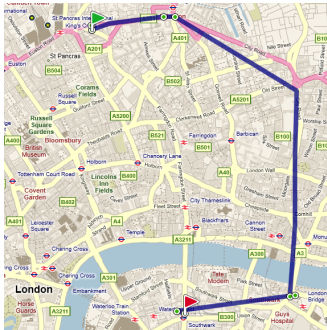
<i>trip</i> $T_1$	9:00	9:05	9:08	9:17	9:23	9:31
<i>trip</i> $T_2$	10:00	10:05	10:08	10:17	10:23	10:31
<i>trip</i> $T_3$	10:30	10:35	10:38	—	—	10:53

Route  $R_1$

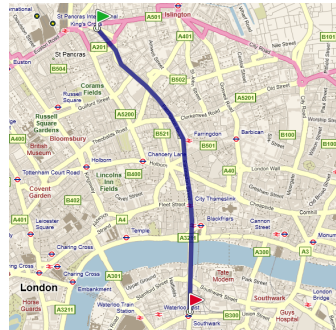
Route  $R_2$



Gesucht: „Gute“ Journeys für Ankunftszeit **und** Anzahl Umstiege

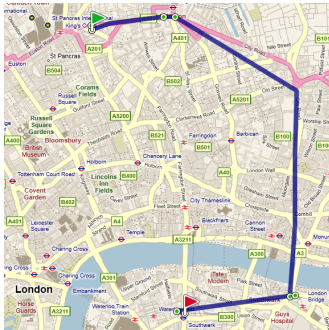


Ankunft 11:08 Uhr, 2 Umstiege

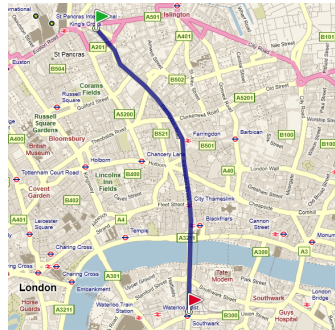


Ankunft 11:09 Uhr, 0 Umstiege

Gesucht: „Gute“ Journeys für Ankunftszeit **und** Anzahl Umstiege



Ankunft 11:08 Uhr, 2 Umstiege



Ankunft 11:09 Uhr, 0 Umstiege

**Problem:**  
Dijkstra-basierter Multi-Label-Correcting-Ansatz zu langsam

# Graph-Modelle?

## Bis jetzt:

- Modelliere Fahrplan als gerichteten Graphen
- Zeitexpandiert vs. zeitabhängig
- Verschiedene Varianten von Dijkstras Algorithmus
- Earliest Arrival-, Profil-, Multi-Criteria-Suchen

## Bis jetzt:

- Modelliere Fahrplan als gerichteten Graphen
- Zeitexpandiert vs. zeitabhängig
- Verschiedene Varianten von Dijkstras Algorithmus
- Earliest Arrival-, Profil-, Multi-Criteria-Suchen

## Probleme:

- Viele Knoten und Kanten
- Overhead von Priority Queue
- Wenig explizites Ausnutzen der Fahrplanstruktur
- Dynamische Szenarien erfordern Updates der Graph-Topologie
- Außerdem: Beschleunigungstechniken funktionieren nicht gut

## Bis jetzt:

- Modelliere Fahrplan als gerichteten Graphen
- Zeitexpandiert vs. zeitabhängig
- Verschiedene Varianten von Dijkstras Algorithmus
- Earliest Arrival-, Profil-, Multi-Criteria-Suchen

## Probleme:

- Viele Knoten und Kanten
- Overhead von Priority Queue
- Wenig explizites Ausnutzen der Fahrplanstruktur
- Dynamische Szenarien erfordern Updates der Graph-Topologie
- Außerdem: Beschleunigungstechniken funktionieren nicht gut

Sind Graphen die beste Art, Fahrpläne zu modellieren?



## Anforderungen:

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege
- Nutzt die Struktur der Fahrpläne aus  
benutzt Routen und Trips explizit?

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege
- Nutzt die Struktur der Fahrpläne aus  
benutzt Routen und Trips explizit?
- Funktioniert in dynamischen Szenarien  
Verspätungen, Zugausfälle, Routenänderungen; keine Vorberechnung

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege
- Nutzt die Struktur der Fahrpläne aus  
benutzt Routen und Trips explizit?
- Funktioniert in dynamischen Szenarien  
Verspätungen, Zugausfälle, Routenänderungen; keine Vorberechnung
- Kann auf zusätzliche Kriterien erweitert werden  
z.B. Tarifzonen, Umstiegssicherheit, etc.

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege
- Nutzt die Struktur der Fahrpläne aus  
benutzt Routen und Trips explizit?
- Funktioniert in dynamischen Szenarien  
Verspätungen, Zugausfälle, Routenänderungen; keine Vorberechnung
- Kann auf zusätzliche Kriterien erweitert werden  
z.B. Tarifzonen, Umstiegssicherheit, etc.
- ... und ist hinreichend schnell  
für interaktive Szenarien

## Anforderungen:

- Berechnet Pareto-Mengen  
mindestens Ankunftszeit und # Umstiege
- Nutzt die Struktur der Fahrpläne aus  
benutzt Routen und Trips explizit?
- Funktioniert in dynamischen Szenarien  
Verspätungen, Zugausfälle, Routenänderungen; keine Vorberechnung
- Kann auf zusätzliche Kriterien erweitert werden  
z.B. Tarifzonen, Umstiegssicherheit, etc.
- ... und ist hinreichend schnell  
für interaktive Szenarien

**RAPTOR: Round-bAseed Public Transit Optimized Router**

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.



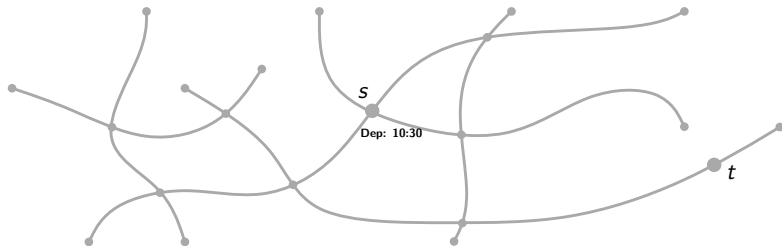
# Runden

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

**Idee:** Eine **Runde** für jeden genommenen Trip.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

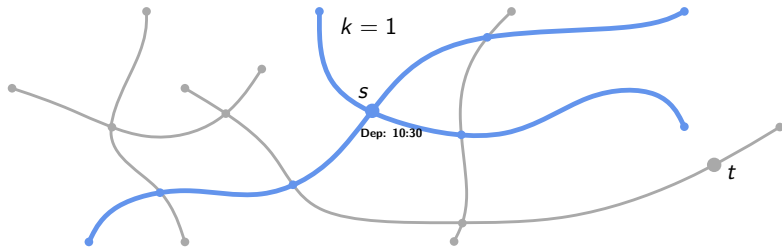
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

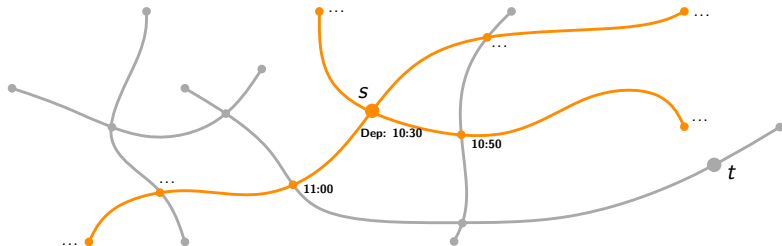
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

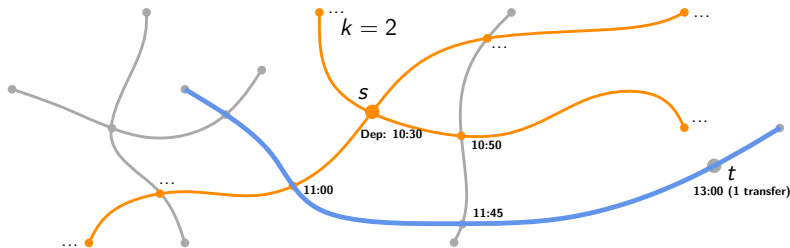
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

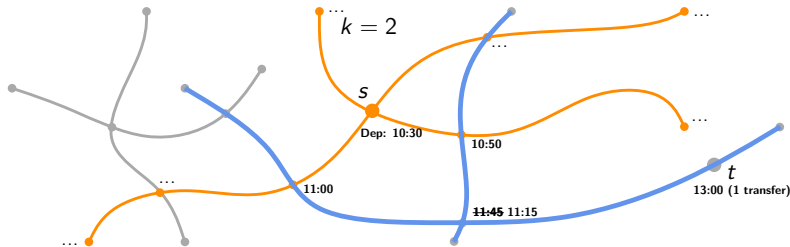
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

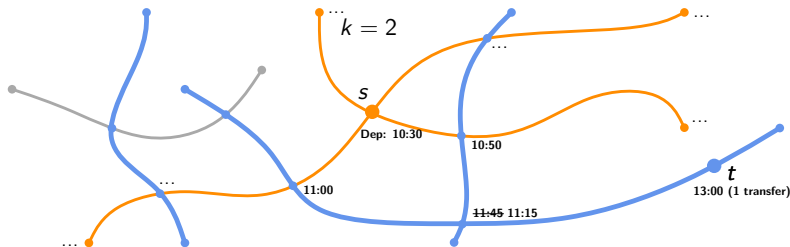
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

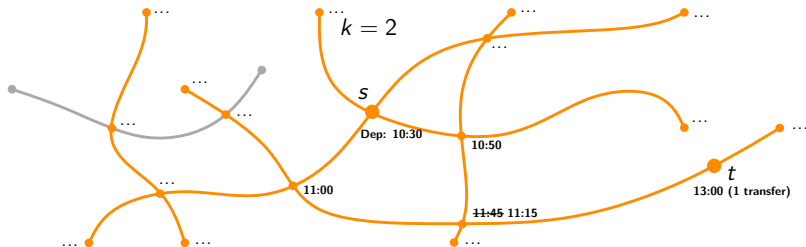
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

**Idee:** Eine **Runde** für jeden genommenen Trip.

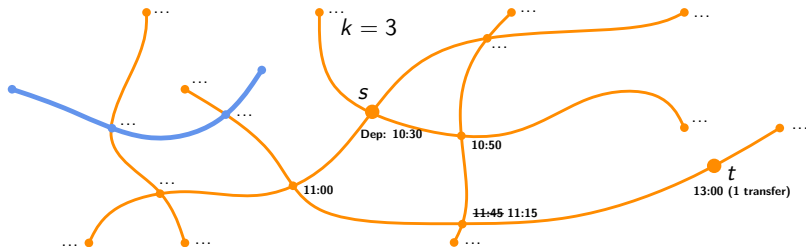


**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.



**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

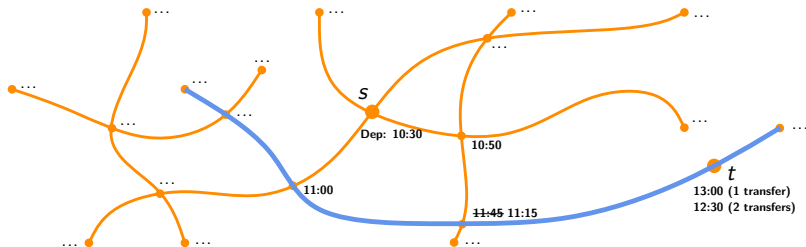
**Idee:** Eine **Runde** für jeden genommenen Trip.



**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

**Beobachtung:** Wechseln zw. Trips führt immer zu einem Umstieg.

**Idee:** Eine **Runde** für jeden genommenen Trip.

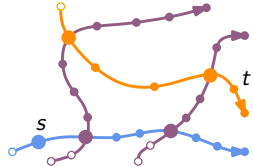


**Ansatz:** Runde  $k$  berechnet Ankunftszeiten für Journeys mit  $k$  Trips.

Scanne jede **Route** höchstens einmal pro Runde.

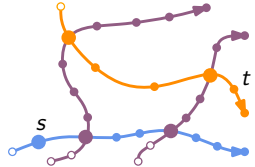
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Scanne jede Route
- 2 Relaxiere Fußwege



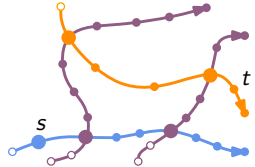
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Scanne jede Route
- 2 Relaxiere Fußwege



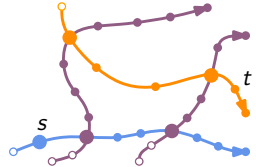
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Scanne jede Route
- 2 Relaxiere Fußwege

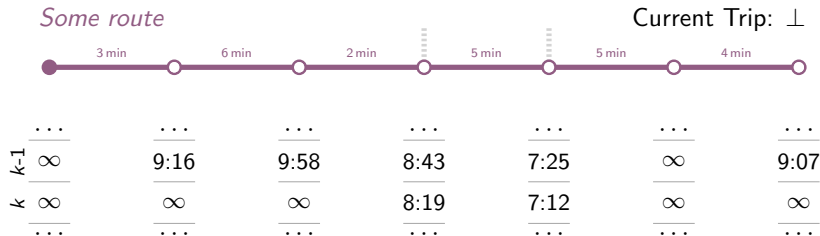


Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Scanne jede Route
- 2 Relaxiere Fußwege



Terminiere, wenn ... ?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?

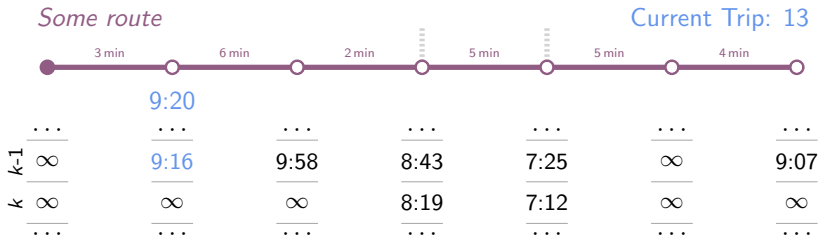




- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

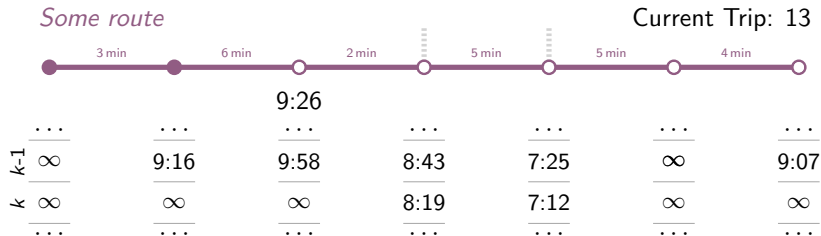
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

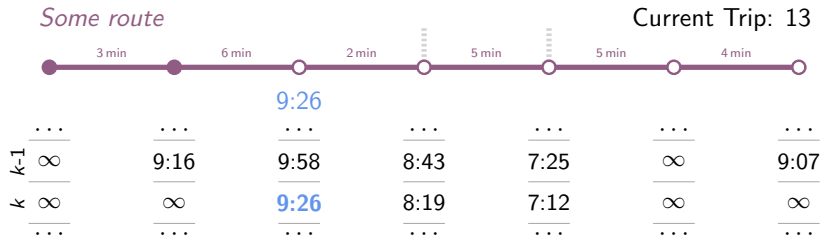
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?

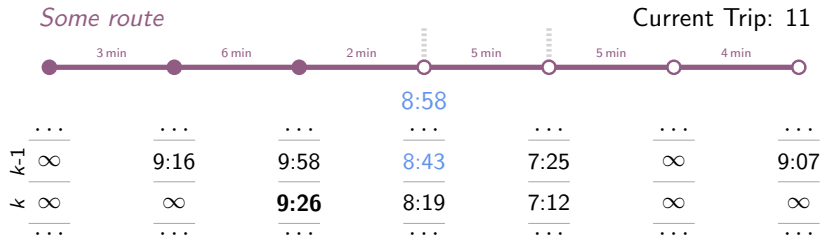
Current Trip: 13



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

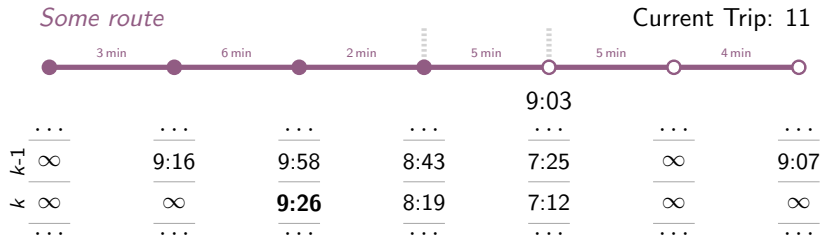
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

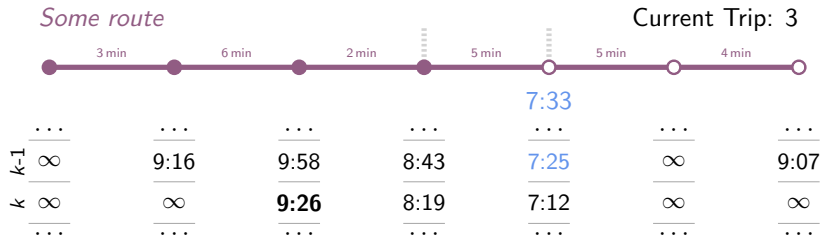
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?

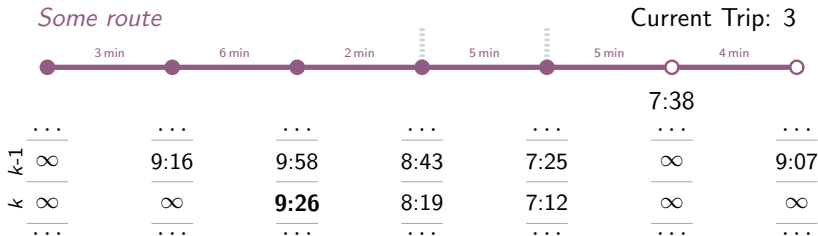


- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?

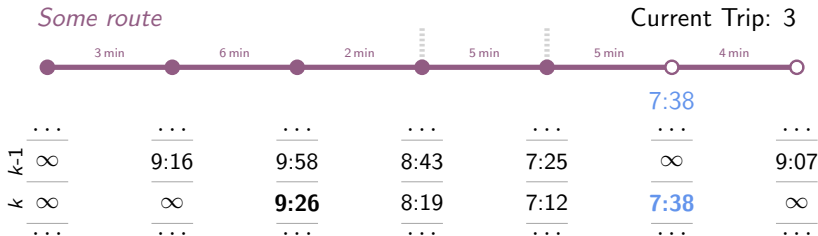




- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

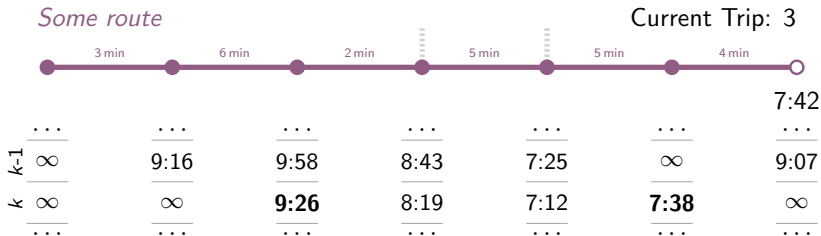
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

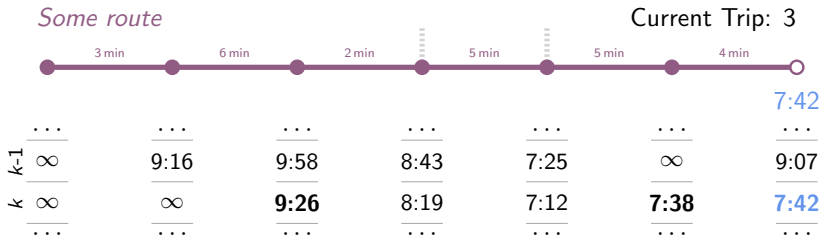
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

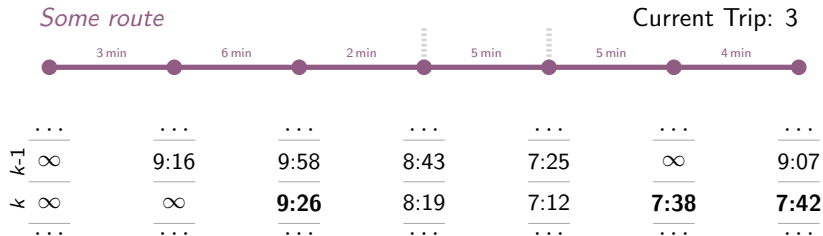
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

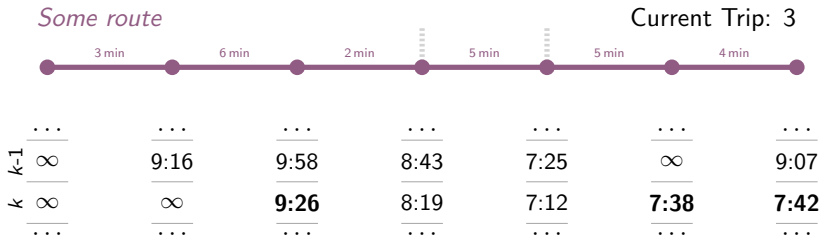
- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?



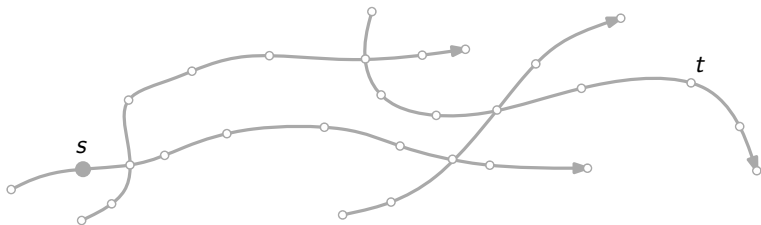
- Jeder Stop hat ein Label (Ankunftszeit) *pro Runde*
- Routenscan in Runde  $k$ : Iteriere über Stops der Route
- Aktiver Trip: Frühester Trip, in den wir bisher einsteigen konnten

Prüfe für jeden Stop:

- Aussteigen: Verbessert aktiver Trip das Label aus Runde  $k$ ?
- Einsteigen: Verbessert Label aus Runde  $k - 1$  den aktiven Trip?

Dynamischer Programmierungsansatz

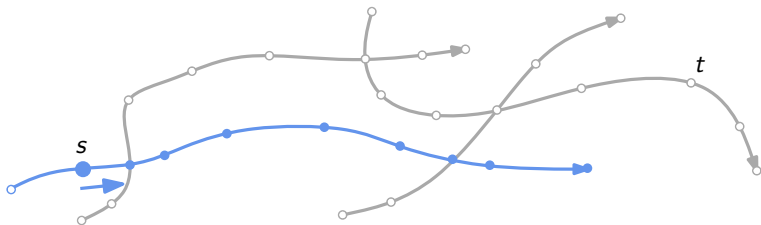
**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.



## Markieren und Pruning:

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop

**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.

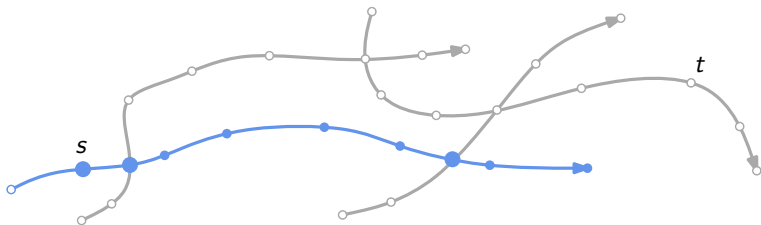


**Markieren und Pruning:**

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop



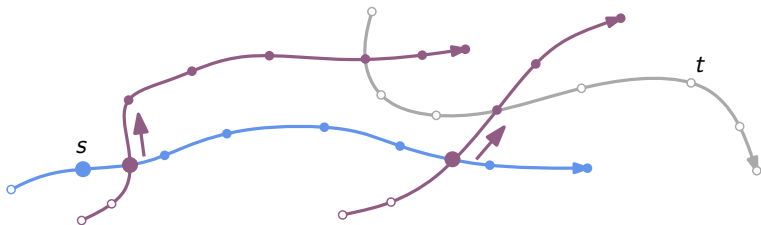
**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.



**Markieren und Pruning:**

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop

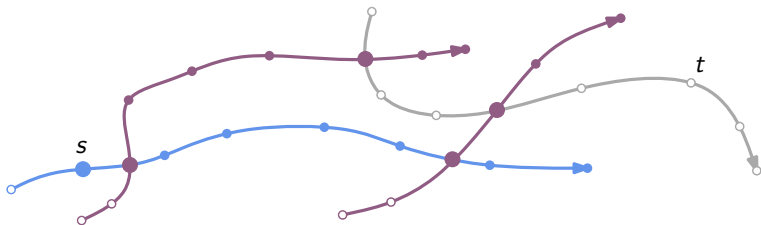
**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.



**Markieren und Pruning:**

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop

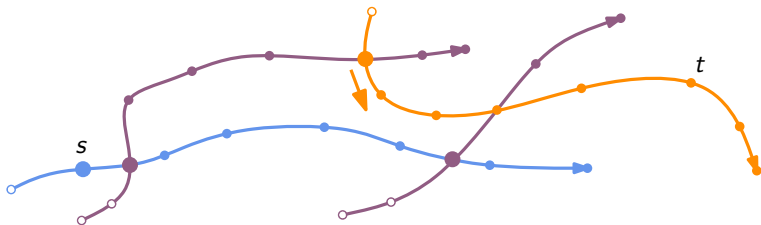
**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.



## Markieren und Pruning:

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop

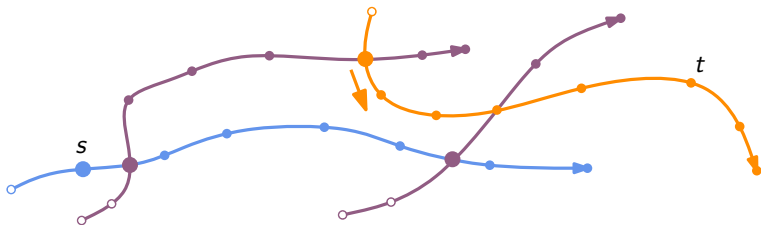
**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.



## Markieren und Pruning:

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop

**Beobachtung:** Nicht alle Routen werden in jeder Runde erreicht.

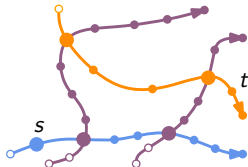


## Markieren und Pruning:

- Route scannen: Markiere Stop, falls Ankunftszeit verbessert
- Nächste Runde: Nur Routen von markierten Stops scannen
- Scanne jede Route ab ihrem ersten markierten Stop
- Markiere Stops nur wenn bessere Ankunftszeit als am Ziel

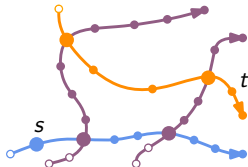
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Wähle erreichte Routen aus letzter Runde
- 2 Scanne diese Routen
- 3 Relaxiere Fußwege



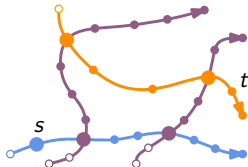
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Wähle erreichte Routen aus letzter Runde
- 2 Scanne diese Routen
- 3 Relaxiere Fußwege



Für jede Runde  $k \leftarrow 1, 2, \dots$

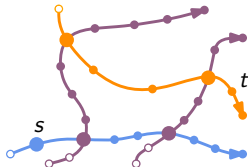
- 1 Wähle erreichte Routen aus letzter Runde
- 2 Scanne diese Routen
- 3 Relaxiere Fußwege





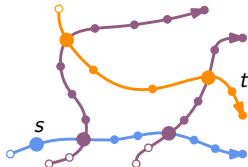
Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Wähle erreichte Routen aus letzter Runde
- 2 Scanne diese Routen
- 3 Relaxiere Fußwege



Für jede Runde  $k \leftarrow 1, 2, \dots$

- 1 Wähle erreichte Routen aus letzter Runde
- 2 Scanne diese Routen
- 3 Relaxiere Fußwege



Terminiere, wenn kein Stop markiert wurde.

## Datenstrukturen:

- $Q$ : Menge von Tupeln  $(R, i)$ 
  - Erreichte Route  $R$  mit Stopsequenz  $(v_0, v_1, \dots, v_{|R|-1})$
  - Frühester erreichter Stop auf  $R$  ist  $v_i$

---

Leere  $Q$

**for** alle markierten Stops  $v$  **do**

**for** alle Routen  $R$ , die  $v$  besuchen **do**

$i \leftarrow$  Position von  $v$  in  $R$

**if**  $\exists j$  mit  $(R, j) \in Q$  **then**

            Ersetze  $(R, j)$  durch  $(R, \min\{i, j\})$  in  $Q$

**else**

            Füge  $(R, i)$  in  $Q$  ein

Lösche alle Markierungen

---

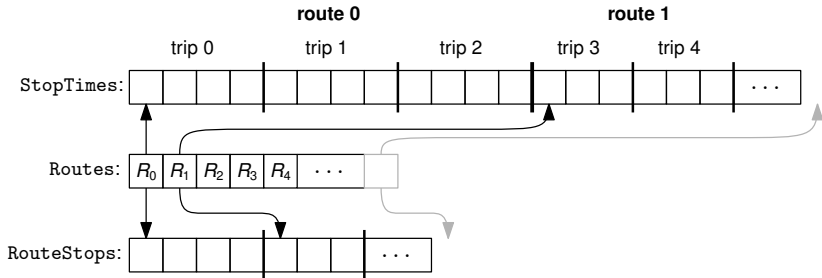
## Datenstrukturen:

- $\tau_k(v)$ : Ankunftszeit an  $v$  in Runde  $k$
- $\tau^*(v)$ : Beste Ankunftszeit an  $v$  bisher
- $\text{pred}(T)$ : Nächstfrüherer Trip vor  $T$  in der Route

---

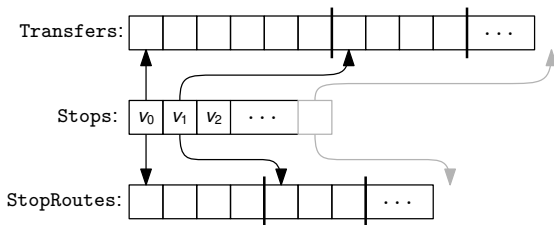
```
for alle  $(R, i) \in Q$  do  
   $T \leftarrow \arg \min_{T \in R} \{ \tau_{\text{dep}}(T, v_i) \geq \tau_{k-1}(v_i) \}$   
  for alle  $v_j \in R$  mit  $j > i$  do  
    if  $T \neq \perp$  und  $\tau_{\text{arr}}(T, v_j) < \min\{\tau^*(v_j), \tau^*(t)\}$  then  
       $\tau_k(v_j) \leftarrow \tau_{\text{arr}}(T, v_j)$   
       $\tau^*(v_j) \leftarrow \tau_{\text{arr}}(T, v_j)$   
      Markiere  $v_j$   
    while  $\text{pred}(T) \neq \perp$  und  $\tau_{k-1}(v_j) < \tau_{\text{dep}}(\text{pred}(T), v_j)$  do  
       $T \leftarrow \text{pred}(T)$ 
```

---



## Route $R_j$ scannen:

- Finde ersten Stop in RouteStops
- Finde aktiven Trip in StopTimes
- Iteriere simultan über RouteStops und aktiven Trip
- Nächstfrüherer Trip: Springe in StopTimes um  $|R_j|$  nach links



## Transfers relaxieren:

- Iteriere über `Transfers` von allen markierten Stops

## Erreichte Routen einsammeln:

- Iteriere über `StopRoutes` von allen markierten Stops

**Beobachtung:** Routen werden in beliebiger Reihenfolge gescannt.

**Beobachtung:** Routen werden in beliebiger Reihenfolge gescannt.

Verteile Routen auf verschiedene CPU-Kerne; scanne parallel.

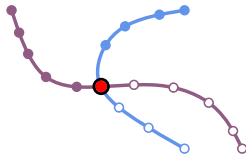


**Beobachtung:** Routen werden in beliebiger Reihenfolge gescannt.

Verteile Routen auf verschiedene CPU-Kerne; scanne parallel.

**Vermeiden von Race-Conditions:**

- Lock auf Schreiben von Labels (teuer)



**Beobachtung:** Routen werden in beliebiger Reihenfolge gescannt.

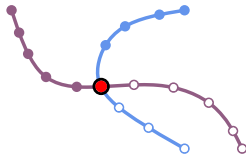
Verteile Routen auf verschiedene CPU-Kerne; scanne parallel.

## Vermeiden von Race-Conditions:

- Lock auf Schreiben von Labels (teuer)

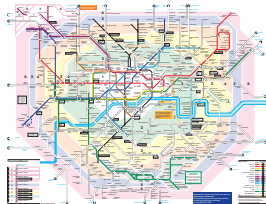
Oder ohne Locks:

- Synchronisiere Labels nach jeder Runde
- Sicherstellen, dass nur „unabhängige“ Routen gleichzeitig gescannt werden (Reduktion auf Färbeproblem)



## Mögliche Erweiterungen:

- Profil-Anfragen (Intervallanfragen)  
Flexible Abfahrtszeiten
- Tarifzonen  
Längere Routen könnten billiger sein.
- Umstiegssicherheit  
Routen könnten knappe Umstiege haben.
- ...



Performance hängt von Anzahl *nichtdominierter* Journeys ab.

# More Criteria: McRAPTOR

**Ziel:** Erweitern von RAPTOR auf zusätzliche Kriterien



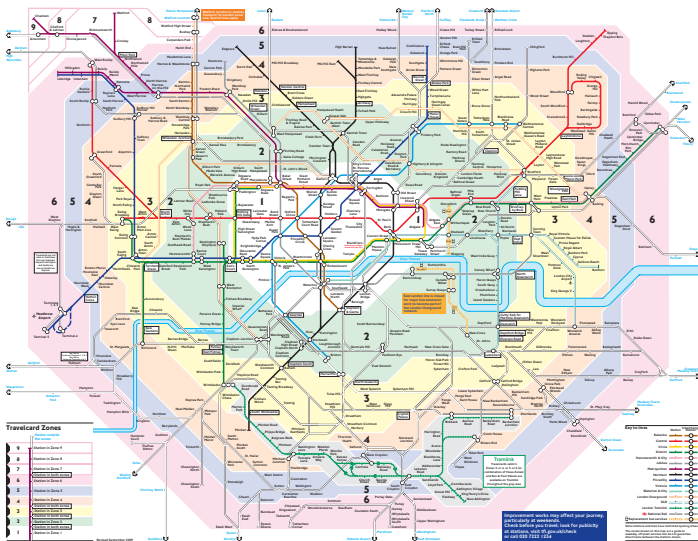
**Ziel:** Erweitern von RAPTOR auf zusätzliche Kriterien



**Ansatz:**

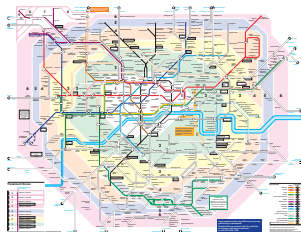
- Labels haben Wert für jedes zusätzliche Kriterium
- Mehrere nichtdominierte Labels pro Stop und Runde
- Beim Routenscan: Ein aktiver Trip pro nichtdominiertem Label
- Lösche dominierte Labels on-the-fly

# McRAPTOR Beispiel: Tarifzonen



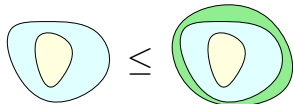
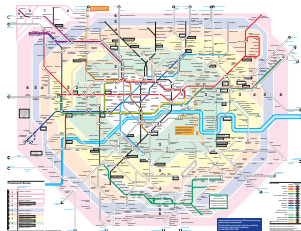
## Tarifzonen einbauen:

- Direkte Preise (£) nicht handhabbar
- ⇒ Berechne alle Kombinationen von Tarifzonen
- ... und filtere im Postprocessing



## Tarifzonen einbauen:

- Direkte Preise (£) nicht handhabbar
- ⇒ Berechne alle Kombinationen von Tarifzonen
- ... und filtere im Postprocessing



## Implementierung:

- Mengen von Tarifzonen als Kriterium
- Dominieren  $\hat{=}$  Teilmengenrelation
- Benutze Bits von `int64` für Mengen



# Profil-Anfragen: rRAPTOR

**Problem:** Finde alle optimalen Journeys,  
die in einem Zeitintervall  $\Delta$  abfahren.



**Problem:** Finde alle optimalen Journeys, die in einem Zeitintervall  $\Delta$  abfahren.

- Lösbar mit McRAPTOR...
- ... mit Abfahrtszeit als Kriterium.



**Problem:** Finde alle optimalen Journeys, die in einem Zeitintervall  $\Delta$  abfahren.

- Lösbar mit McRAPTOR...
- ... mit Abfahrtszeit als Kriterium.



**Effizienterer Ansatz: rRAPTOR (Self-Pruning)**

- Sammle alle Abfahrten aus Intervall  $\Delta$  in Menge  $\mathcal{D}$
- Dann: RAPTOR für jedes  $\tau \in \mathcal{D}$  geordnet absteigend nach Zeit
- Reinitialisiere keine Labels zwischen den Aufrufen!

**Problem:** Finde alle optimalen Journeys, die in einem Zeitintervall  $\Delta$  abfahren.

- Lösbar mit McRAPTOR...
- ... mit Abfahrtszeit als Kriterium.



**Effizienterer Ansatz: rRAPTOR (Self-Pruning)**

- Sammle alle Abfahrten aus Intervall  $\Delta$  in Menge  $\mathcal{D}$
- Dann: RAPTOR für jedes  $\tau \in \mathcal{D}$  geordnet absteigend nach Zeit
- Reinitialisiere keine Labels zwischen den Aufrufen!

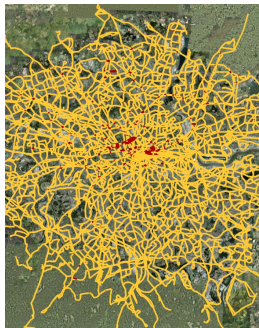
Prunt implizit Journeys, die früher abfahren und später ankommen.

# Experimente



## Das vollständige Londoner Netzwerk

- Ein Dienstag
- Beinhaltet Tube, Bus, DLR, Tram, ...
- 20 843 Stops
- 2 225 Routen mit 133 011 Trips
- 5 132 672 Connections pro Tag



## Das vollständige Londoner Netzwerk

- Ein Dienstag
- Beinhaltet Tube, Bus, DLR, Tram, ...
- 20 843 Stops
- 2 225 Routen mit 133 011 Trips
- 5 132 672 Connections pro Tag

Experimente: 10 000 zufällige  $s-t$ -Anfragen



# Vergleich der Algorithmen

(Hardware: Intel Xeon X5680 mit 3.33 GHz und 96 GiB DDR3-1333 RAM)

Algorithm	Ar	R	Tr	Fz	Rounds	Journeys	[ms]
Dijkstra	●	○	○	○	—	0.9	14.2
RAPTOR	●	○	●	○	8.4	1.9	7.3
LD	●	○	●	○	—	1.9	44.5
MLC	●	○	●	○	—	1.9	67.2
McRAPTOR	●	○	●	●	10.8	9.0	107.4
MLC	●	○	●	●	—	9.0	399.5
McRAPTOR	●	●	●	○	9.5	16.3	259.8
rRAPTOR	●	●	●	○	138.5	16.3	87.0
SPCS	●	●	○	○	—	7.8	183.6

(Ar: Arrival Time, R: Range, Tr: Transfers, Fz: Fare Zones)



Algorithm	Ar	R	Tr	Fz	1 core [ms]	3 cores [ms]	6 cores [ms]	12 cores [ms]
RAPTOR	●	○	●	○	7.7	5.0	4.1	3.7
McRAPTOR	●	○	●	●	118.6	49.4	29.9	26.1
rRAPTOR	●	●	●	○	92.3	39.5	26.8	21.6
SPCS	●	●	○	○	183.6	69.1	44.9	38.9

(Ar: Arrival Time, R: Range, Tr: Transfers, Fz: Fare Zones)

Algorithm	Ar	R	Tr	Fz	1 core [ms]	3 cores [ms]	6 cores [ms]	12 cores [ms]
RAPTOR	●	○	●	○	7.7	5.0	4.1	3.7
McRAPTOR	●	○	●	●	118.6	49.4	29.9	26.1
rRAPTOR	●	●	●	○	92.3	39.5	26.8	21.6
SPCS	●	●	○	○	183.6	69.1	44.9	38.9

(Ar: Arrival Time, R: Range, Tr: Transfers, Fz: Fare Zones)

- Gute Speedups auf bis zu 6 Kernen
- RAPTOR immer  $\leq 30$  ms



Daniel Delling, Thomas Pajor, and Renato F. Werneck.

Round-Based Public Transit Routing.

In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140. SIAM, 2012.