

# Algorithmen für Routenplanung

15. Vorlesung, Sommersemester 2021

Tim Zeitz | 20. Juni 2022



# 1. Dynamische Szenarien

## Szenario:

- Unfall auf einer Straße
- Reisezeit ändert sich auf dieser Straße
- berechne schnellsten Weg bezüglich der aktualisierten Reisezeiten



## Hauptproblem:

- Kantengewichte ändern sich
- Vorberechnung basiert auf ursprünglichen Kantengewichten
- komplette Vorberechnung für jeden Stau wenig sinnvoll

## Lösung:

- “Customizable” Techniken (MLD, CCH)
- Anpassungen auch für ALT und “klassische” CH möglich

## 2. Zeitabhängiges Szenario

### Motivation:

- Stau um Städte herum folgt vorhersehbaren Mustern
- Morgens geht jeder zur Arbeit → Stau
- Mittags gibt es weniger Stau
- Abends fährt jeder nach Hause → Stau in die andere Richtung
- Aber nicht Sonntags



## 2. Zeitabhängiges Szenario

### Motivation:

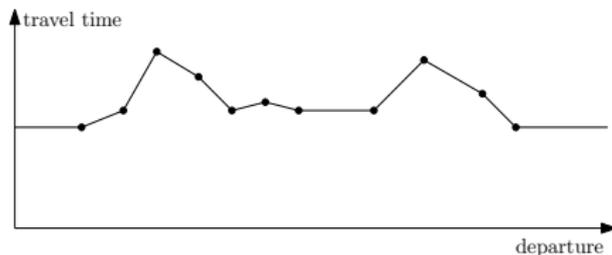
- Stau um Städte herum folgt vorhersehbaren Mustern
- Morgens geht jeder zur Arbeit → Stau
- Mittags gibt es weniger Stau
- Abends fährt jeder nach Hause → Stau in die andere Richtung
- Aber nicht Sonntags



- Aggregiere historische Daten um eine Vorhersage für jeden Wochentag zu machen

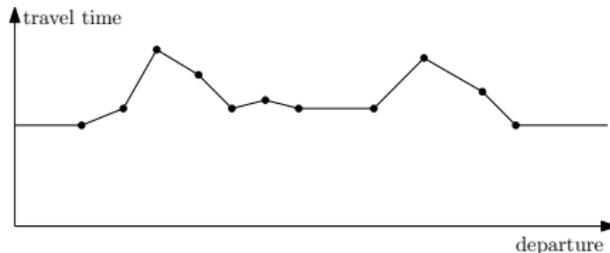
# Zeitabhängige Kantengewichte

- **bisher:** An jeder Kante steht ein skalares Kantengewicht
- **neu:** An jeder Kanten steht eine Funktion
- Die Funktion bildet den Zeitpunkt an dem eine Kante betreten wird auf die Fahrzeit ab



# Zeitabhängige Kantengewichte

- **bisher:** An jeder Kante steht ein skalares Kantengewicht
- **neu:** An jeder Kanten steht eine Funktion
- Die Funktion bildet den Zeitpunkt an dem eine Kante betreten wird auf die Fahrzeit ab



## Problemstellung

Mit zeitabhängigen Gewichten ist die Frage

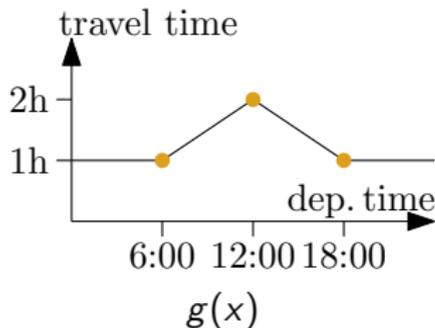
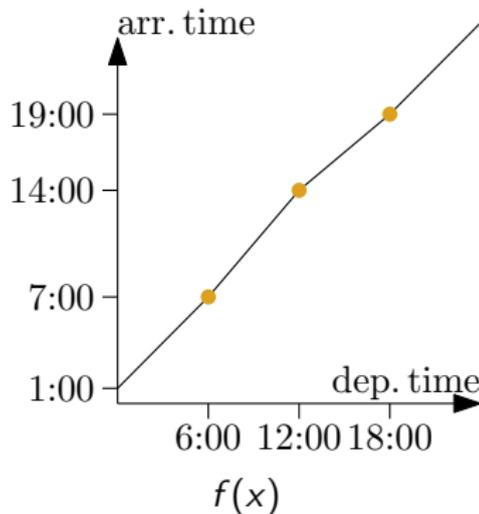
- Wie komme ich von  $s$  nach  $t$ ?

nicht mehr wohl geformt.

Wir betrachten nun das Problem der frühesten Ankunft:

- Wie komme ich von  $s$  nach  $t$  wenn ich um  $\tau$  losfahre?

# Zwei Sichtweisen



- Zwei Sichtweise auf die selbe Information
- $f(x) = g(x) + x$
- Aussagen in der einen Sichtweise lassen sich immer auf die andere übertragen
- Wir wechseln ständig zwischen beiden Sichtweisen und nehmen die, die gerade am besten passt

## Definition

Sei  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  eine Funktion.  $f$  erfüllt die *FIFO-Eigenschaft*, wenn für jedes  $\varepsilon > 0$  und alle  $\tau \in \mathbb{R}_0^+$  gilt, dass

$$\tau + f(\tau) \leq \tau + \varepsilon + f(\tau + \varepsilon).$$

## Definition

Sei  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  eine Funktion.  $f$  erfüllt die *FIFO-Eigenschaft*, wenn für jedes  $\varepsilon > 0$  und alle  $\tau \in \mathbb{R}_0^+$  gilt, dass

$$\tau + f(\tau) \leq \tau + \varepsilon + f(\tau + \varepsilon).$$

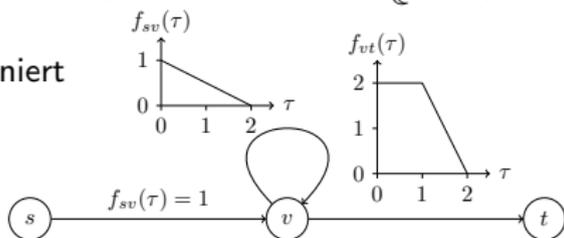
- Interpretation: “Warten/Später ankommen lohnt sich nie”
- Kürzeste Distanzen auf Graphen mit non-FIFO Funktionen finden ist NP-schwer (wenn warten an Knoten nicht erlaubt ist)
  - Schwach NP-schwer auf Funktion in  $\mathbb{N}$  (Reduktion von Partition)
  - Stark NP-schwer schon auf stückweise linearen Funktionen in  $\mathbb{Q}$  mit allen Stützstellen in  $\mathbb{N}^2$

## Definition

Sei  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  eine Funktion.  $f$  erfüllt die *FIFO-Eigenschaft*, wenn für jedes  $\varepsilon > 0$  und alle  $\tau \in \mathbb{R}_0^+$  gilt, dass

$$\tau + f(\tau) \leq \tau + \varepsilon + f(\tau + \varepsilon).$$

- Interpretation: “Warten/Später ankommen lohnt sich nie”
- Kürzeste Distanzen auf Graphen mit non-FIFO Funktionen finden ist NP-schwer (wenn warten an Knoten nicht erlaubt ist)
  - Schwach NP-schwer auf Funktion in  $\mathbb{N}$  (Reduktion von Partition)
  - Stark NP-schwer schon auf stückweise linearen Funktionen in  $\mathbb{Q}$  mit allen Stützstellen in  $\mathbb{N}^2$
  - Kürzeste Wege nicht mehr wohldefiniert



## Definition

Sei  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  eine Funktion.  $f$  erfüllt die *FIFO-Eigenschaft*, wenn für jedes  $\varepsilon > 0$  und alle  $\tau \in \mathbb{R}_0^+$  gilt, dass

$$\tau + f(\tau) \leq \tau + \varepsilon + f(\tau + \varepsilon).$$

- Interpretation: “Warten/Später ankommen lohnt sich nie”
- Kürzeste Distanzen auf Graphen mit non-FIFO Funktionen finden ist NP-schwer (wenn warten an Knoten nicht erlaubt ist)
  - Schwach NP-schwer auf Funktion in  $\mathbb{N}$  (Reduktion von Partition)
  - Stark NP-schwer schon auf stückweise linearen Funktionen in  $\mathbb{Q}$  mit allen Stützstellen in  $\mathbb{N}^2$
  - Kürzeste Wege nicht mehr wohldefiniert

⇒ Sicherstellen, dass Funktionen FIFO-Eigenschaft erfüllen.

- In der akademischen Forschung nimmt man oft an, dass Interpolationspunkte pro Kante gegeben sind
- Viele Probleme mit dieser Annahme:
  - Viele Interpolationspunkte pro Graph
    - Großer Speicherverbrauch
    - Sehr problematisch
  - Erhobene Daten sind stark verrauscht und Vorhersagen ungenau
    - Interpolationspunkte suggerieren eine Genauigkeit die reale Daten nicht hergeben
- **Deswegen:** in der Praxis auch andere Darstellungsformen verbreitet
- In der Vorlesung bleiben wir aber bei der akademischen Sichtweise

## Option 1

- Teile den Tag in Abschnitte ein, z.B., in 24 Stunden
  - Abschnitte müssen nicht gleich groß sein
  - Unterteilung kann pro Kante variieren
- Speichere für jeden Abschnitt eine Reisegeschwindigkeit
  - Geschwindigkeiten können gerundet sein: z.B. 5 km/h Schritte
- Alle Fahrzeuge auf einer Kanten ändern spontan ihre Reisegeschwindigkeit wenn die Tageszeit über eine Abschnittsgrenzen fortschreitet
  - Aus dieser Eigenschaft folgt FIFO

## Option 2

- Speichere kleine globale Menge von Funktionskurven  $\mathbb{F}$  explizit
  - Funktionen aus  $\mathbb{F}$  werden meistens mittels Interpolationspunkte und linearer Interpolation dargestellt
  - Jede dieser Funktionen ist FIFO
- An jeder Kante speichert man einen fixen Satz an skalaren Attributen:
  - Funktion-ID in  $\mathbb{F}$
  - Kanten-Länge
  - Diverse Strauchungs- und Streckungsfaktoren die die Funktion aus  $\mathbb{F}$  transformieren

## Hauptprobleme:

- Kürzester Weg hängt von Abfahrtszeitpunkt ab
- Eingabegröße steigt massiv an

## Hauptprobleme:

- Kürzester Weg hängt von Abfahrtszeitpunkt ab
- Eingabegröße steigt massiv an

## Vorgehen:

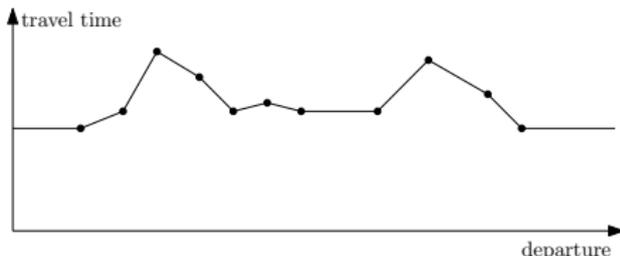
- Modellierung
- Anpassung Dijkstra
- Anpassung Beschleunigungstechniken

## Eingabe:

- Durchschnittliche Reisezeit zu bestimmten Zeitpunkten
- Jeden Wochentag verschieden
- Sonderfälle: Urlaubszeit

## Somit an jeder Kante:

- Periodische stückweise lineare Funktion
- Definiert durch Stützpunkte
- Interpoliere linear zwischen Stützpunkten



## Eigenschaften “Zeitabhängigkeit”:

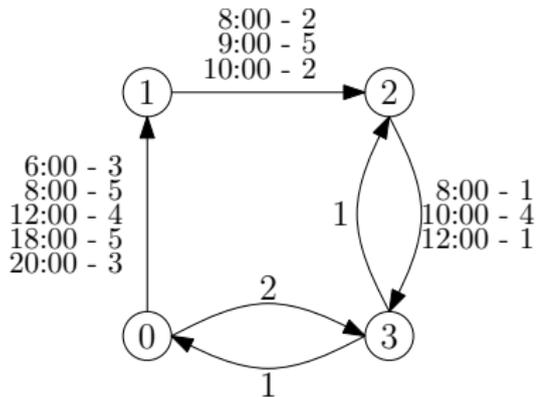
- Topologie ändert sich nicht
- Kanten gemischt zeitabhängig und konstant
- variable (!) Anzahl Interpolationspunkte pro Kante

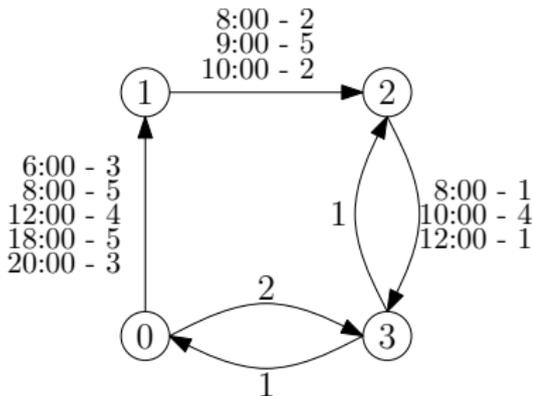
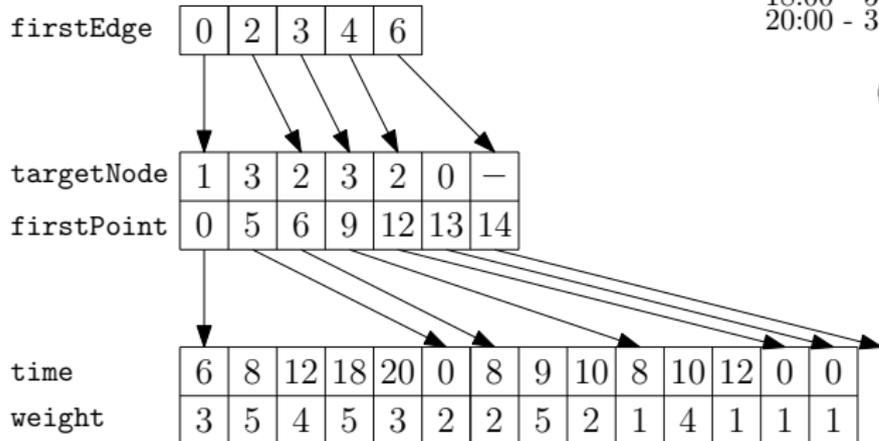
## Eigenschaften “Zeitabhängigkeit”:

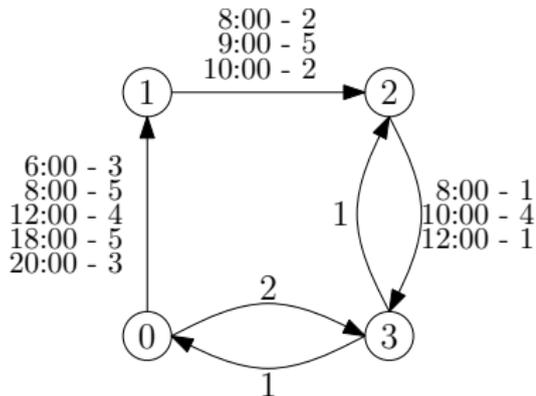
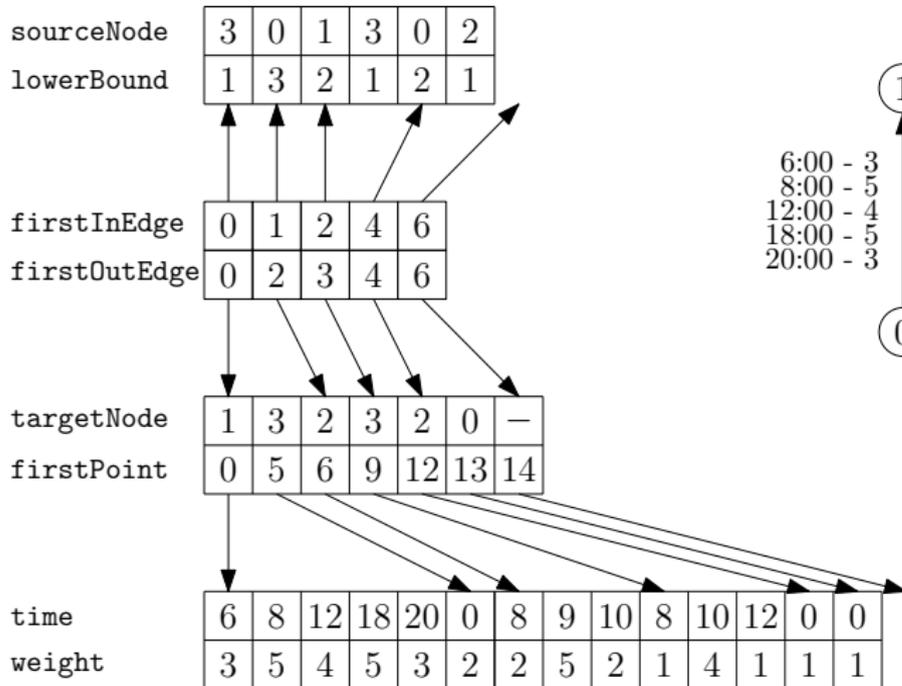
- Topologie ändert sich nicht
- Kanten gemischt zeitabhängig und konstant
- variable (!) Anzahl Interpolationspunkte pro Kante

## Voraussetzung:

- FIFO gilt auf allen Kanten







## Zeit-Anfrage:

- finde kürzesten Weg für Abfahrtszeit  $\tau$
- analog zu Dijkstra?

## Zeit-Anfrage:

- finde kürzesten Weg für Abfahrtszeit  $\tau$
- analog zu Dijkstra?

## Profil-Anfrage:

- finde kürzeste Reisezeit für alle Abfahrtszeitpunkte
- analog zu Dijkstra?

Ziel: finde kürzesten Weg für Abfahrtszeit  $\tau$

---

Time-Dijkstra( $G = (V, E), s, \tau$ )

---

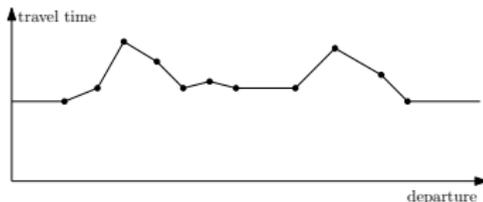
```
1  $d_\tau[s] = 0$ 
2  $Q.clear(), Q.add(s, 0)$ 
3 while ! $Q.empty()$  do
4    $u \leftarrow Q.deleteMin()$ 
5   for all edges  $e = (u, v) \in E$  do
6     if  $d_\tau[u] + \text{len}(e, \tau + d_\tau[u]) < d_\tau[v]$  then
7        $d_\tau[v] \leftarrow d_\tau[u] + \text{len}(e, \tau + d_\tau[u])$ 
8        $p_\tau[v] \leftarrow u$ 
9       if  $v \in Q$  then  $Q.decreaseKey(v, d_\tau[v])$ 
10
11     else  $Q.insert(v, d_\tau[v])$ 
```

# Auswertung

## Evaluation von $f(\tau)$ :

- Suche Punkte mit  $t_i \leq \tau$  und  $t_{i+1} \geq \tau$
- dann Evaluation durch

$$f(\tau) = w_i + \frac{\tau - t_i}{t_{i+1} - t_i} \cdot (w_{i+1} - w_i)$$

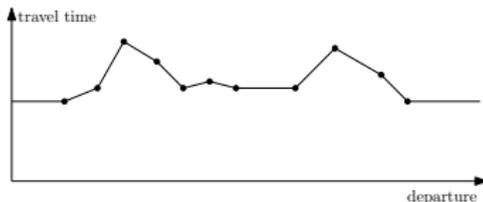


# Auswertung

## Evaluation von $f(\tau)$ :

- Suche Punkte mit  $t_i \leq \tau$  und  $t_{i+1} \geq \tau$
- dann Evaluation durch

$$f(\tau) = w_i + \frac{\tau - t_i}{t_{i+1} - t_i} \cdot (w_{i+1} - w_i)$$



## Problem:

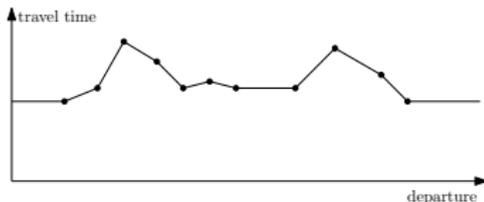
- Finden von  $t_i$  und  $t_{i+1}$ 
  - Achtung: Sonderfall am Periodenrand
- Theoretisch:
  - Lineare Suche:  $\mathcal{O}(|I|)$
  - Binäre Suche:  $\mathcal{O}(\log_2 |I|)$

# Auswertung

## Evaluation von $f(\tau)$ :

- Suche Punkte mit  $t_i \leq \tau$  und  $t_{i+1} \geq \tau$
- dann Evaluation durch

$$f(\tau) = w_i + \frac{\tau - t_i}{t_{i+1} - t_i} \cdot (w_{i+1} - w_i)$$



## Problem:

- Finden von  $t_i$  und  $t_{i+1}$ 
  - Achtung: Sonderfall am Periodenrand
- Praktisch:
  - Binäre Suche:  $\mathcal{O}(\log_2 |I|)$  aber schlecht für Pipelining
  - Oder lineare Suche mit Startpunkt  $\frac{\tau}{\Pi} \cdot |I|$   
wobei  $\Pi$  die Periodendauer ist
    - Gut, wenn Punkte uniform verteilt
  - Oder Startpunkte für Zeitintervalle merken
  - Kommt drauf an:
    - Maschine, Cacheeffekte, Größe der Punktearrays, Verteilung der Punkte, ...
    - Benchmarken!

Ziel: finde kürzesten Weg für alle Abfahrtszeitpunkte

---

Profile-Search( $G = (V, E), s$ )

---

```
1  $d_*[s] = 0$ 
2  $Q.clear(), Q.add(s, 0)$ 
3 while ! $Q.empty()$  do
4    $u \leftarrow Q.deleteMin()$ 
5   for all edges  $e = (u, v) \in E$  do
6     if  $d_*[u] \oplus \text{len}(e) \not\geq d_*[v]$  then
7        $d_*[v] \leftarrow \min(d_*[u] \oplus \text{len}(e), d_*[v])$ 
8       if  $v \in Q$  then  $Q.decreaseKey(v, \underline{d}[v])$ 
9
10      else  $Q.insert(v, \underline{d}[v])$ 
```

---

## Beobachtungen:

- Operationen auf Funktionen
- Priorität im Prinzip frei wählbar  
( $\underline{d}[u]$  ist das Minimum der Funktion  $d_*[u]$ )
- Knoten können mehrfach besucht werden  $\Rightarrow$  label-correcting

## Herausforderungen:

- Wie effizient  $\oplus$  berechnen (Linken)?
- Wie effizient Minimum bilden?

## Funktion gegeben durch:

- Menge von Interpolationspunkten
- $I^f := \{(t_1^f, w_1^f), \dots, (t_k^f, w_k^f)\}$

## 3 Operationen notwendig:

- Auswertung
- Linken  $\oplus$
- Minimumsbildung
- Vergleich  $\not\leq$   
(ist analog zu Minimumsbildung)

## Definition

Seien  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  und  $g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  zwei Reisezeitfunktionen die die FIFO-Eigenschaft erfüllen. Die Linkoperation  $f \oplus g$  ist dann definiert durch

$$f \oplus g := f + g \circ (\text{id} + f)$$

Oder

$$(f \oplus g)(\tau) := f(\tau) + g(\tau + f(\tau))$$

## Definition

Seien  $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  und  $g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$  zwei Ankunftszeitfunktionen die die FIFO-Eigenschaft erfüllen. Die Linkoperation  $f \oplus g$  ist dann definiert durch

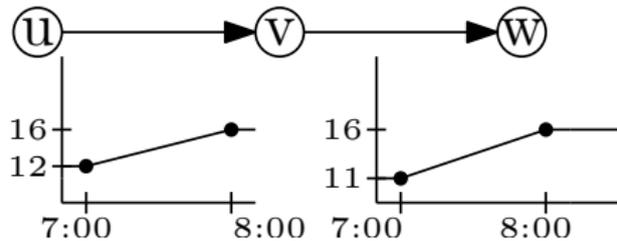
$$f \oplus g := g \circ f$$

Oder

$$(f \oplus g)(\tau) := g(f(\tau))$$

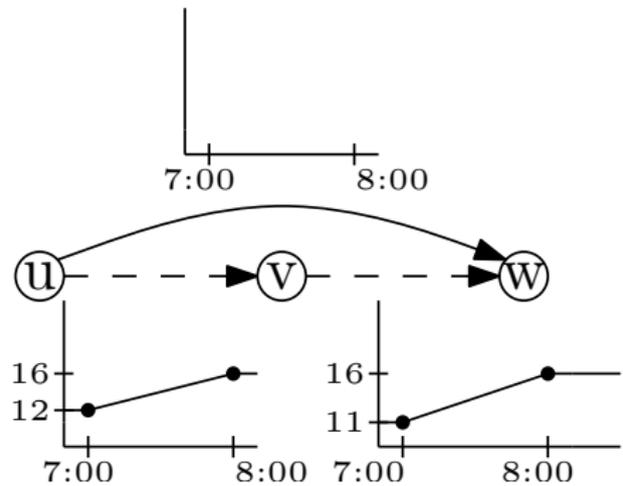
# Linken

## Linken zweier Funktionen $f$ und $g$



# Linken

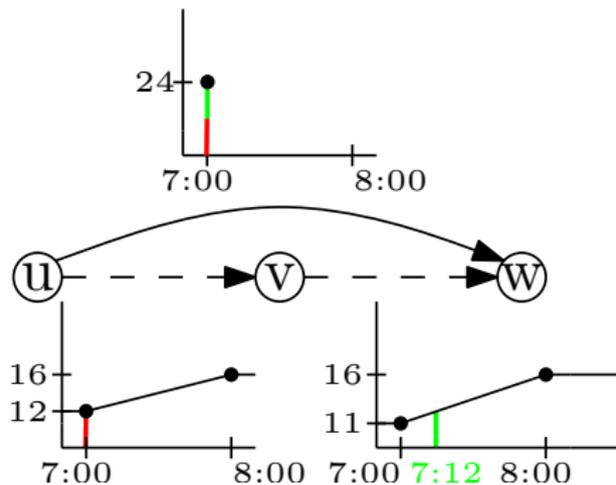
## Linken zweier Funktionen $f$ und $g$



## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall

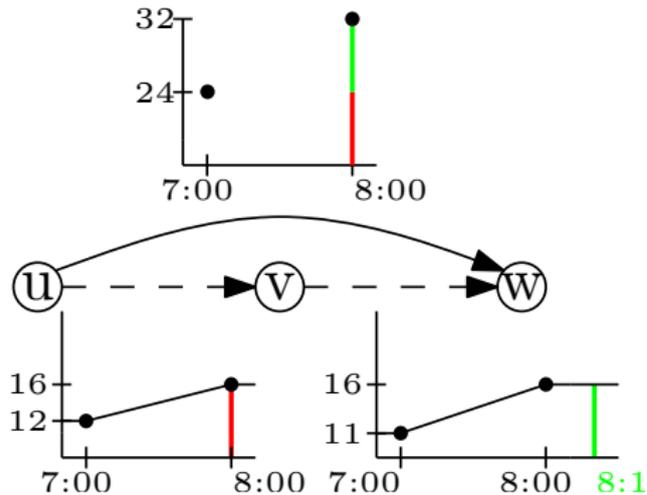
$$\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_i^f, w_i^f + g(t_i^f + w_i^f))\}$$



## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall

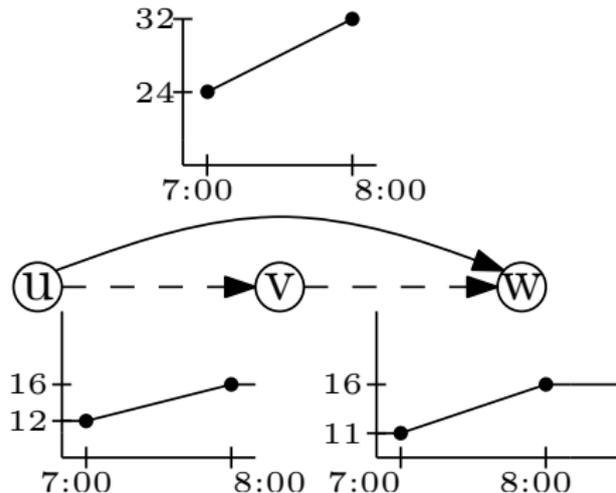
$$\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_i^f, w_i^f + g(t_i^f + w_i^f))\}$$



## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall

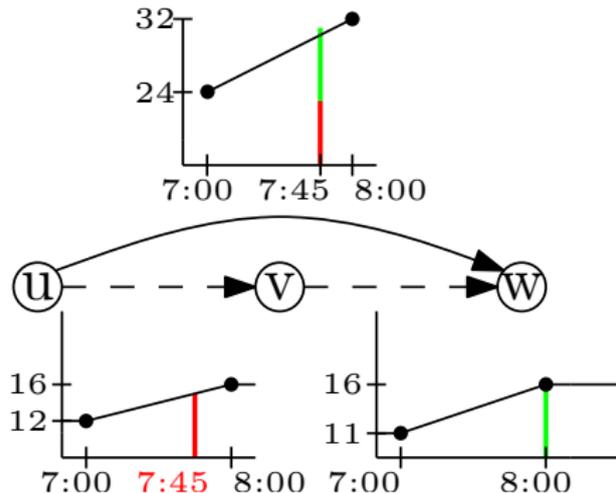
$$\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_i^f, w_i^f + g(t_i^f + w_i^f))\}$$



## Linken zweier Funktionen $f$ und $g$

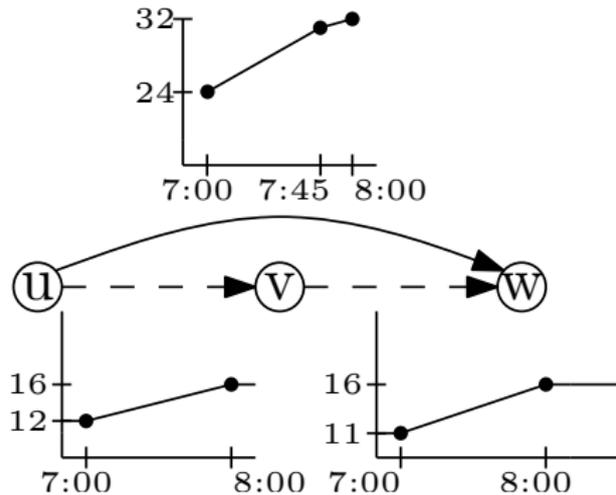
- $f \oplus g$  enthält auf jeden Fall

$$\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_i^f, w_i^f + g(t_i^f + w_i^f))\}$$



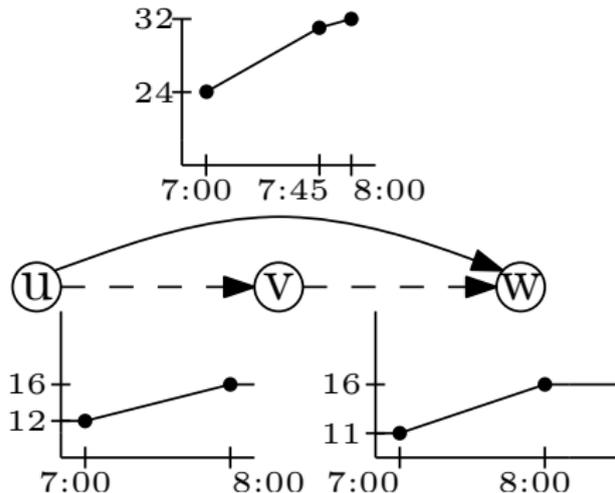
## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall  
 $\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_j^f, w_j^f + g(t_j^f + w_j^f))\}$
- Zusätzliche Interpolationspunkte  
an  $t_j^{-1}$  mit  $f(t_j^{-1}) + t_j^{-1} = t_j^g$



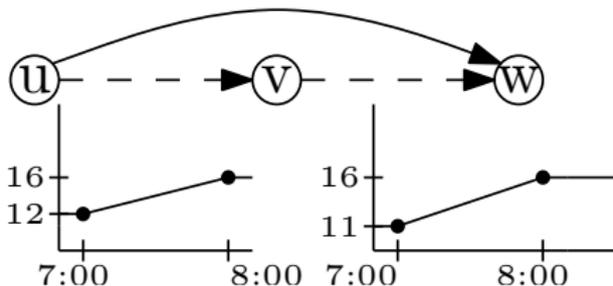
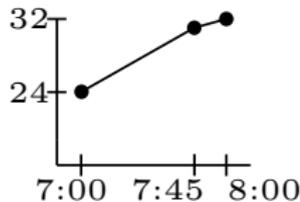
## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall  $\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_j^f, w_j^f + g(t_j^f + w_j^f))\}$
- Zusätzliche Interpolationspunkte an  $t_j^{-1}$  mit  $f(t_j^{-1}) + t_j^{-1} = t_j^g$
- Füge  $(t_j^{-1}, f(t_j^{-1}) + w_j^g)$  für alle Punkte von  $g$  zu  $f \oplus g$



## Linken zweier Funktionen $f$ und $g$

- $f \oplus g$  enthält auf jeden Fall  $\{(t_1^f, w_1^f + g(t_1^f + w_1^f)), \dots, (t_j^f, w_j^f + g(t_j^f + w_j^f))\}$
- Zusätzliche Interpolationspunkte an  $t_j^{-1}$  mit  $f(t_j^{-1}) + t_j^{-1} = t_j^g$
- Füge  $(t_j^{-1}, f(t_j^{-1}) + w_j^g)$  für alle Punkte von  $g$  zu  $f \oplus g$
- Durch linearen Sweeping-Algorithmus implementierbar



## Ankunftszeit-Funktion $f^{-1}$ an Stelle $x$ auswerten:

- Seien  $(x_i, y_i)$  die Interpolationspunkte von  $f$
- Die Interpolationspunkte von  $f^{-1}$  sind  $(y_i, x_i)$

## Ankunftszeit-Funktion $f^{-1}$ an Stelle $x$ auswerten:

- Seien  $(x_i, y_i)$  die Interpolationspunkte von  $f$
- Die Interpolationspunkte von  $f^{-1}$  sind  $(y_i, x_i)$
- Funktionsevaluation analog zu der von  $f$
- Die Interpolationspunkte von  $f^{-1}$  müssen nicht explizit gespeichert werden

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig  $\mathcal{O}(1)$

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig  $\mathcal{O}(1)$

## Speicherverbrauch

- Geknickte Funktion hat  $\approx |I^f| + |I^g|$  Interpolationspunkte

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig  $\mathcal{O}(1)$

## Speicherverbrauch

- Geknickte Funktion hat  $\approx |I^f| + |I^g|$  Interpolationspunkte

## Problem:

- In Profilsuchen können Pfade aus tausenden Kanten bestehen

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig  $\mathcal{O}(1)$

## Speicherverbrauch

- Gekante Funktion hat  $\approx |I^f| + |I^g|$  Interpolationspunkte

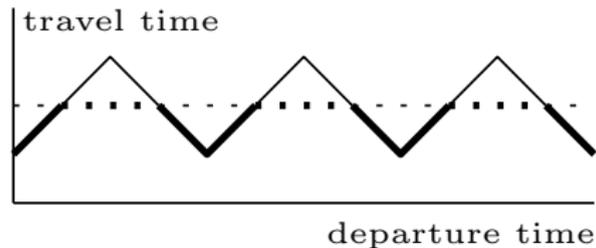
## Problem:

- In Profilsuchen können Pfade aus tausenden Kanten bestehen
- Shortcuts. . .

# Merge von Funktionen

## Minimum zweier Funktionen $f$ und $g$

- Für alle  $(t_i^f, w_i^f)$ : behalte Punkt, wenn  $w_i^f < g(t_i^f)$
- Für alle  $(t_j^g, w_j^g)$ : behalte Punkt, wenn  $w_j^g < f(t_j^g)$
- Schnittpunkte müssen ebenfalls eingefügt werden



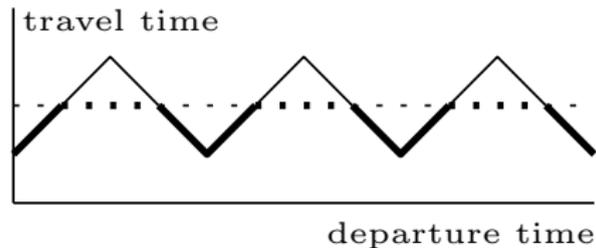
# Merge von Funktionen

## Minimum zweier Funktionen $f$ und $g$

- Für alle  $(t_i^f, w_i^f)$ : behalte Punkt, wenn  $w_i^f < g(t_i^f)$
- Für alle  $(t_j^g, w_j^g)$ : behalte Punkt, wenn  $w_j^g < f(t_j^g)$
- Schnittpunkte müssen ebenfalls eingefügt werden

## Vorgehen:

- Linearer sweep über die Stützstellen
- Evaluere, welcher Abschnitt oben
- Checke ob Schnittpunkt existiert
- Vorsicht bei der Numerik



## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig:  $\mathcal{O}(1)$

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig:  $\mathcal{O}(1)$

## Speicherverbrauch

- Minimum-Funktion kann mehr als  $|I^f| + |I^g|$  Interpolationspunkte enthalten

## Laufzeit

- Sweep Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig:  $\mathcal{O}(1)$

## Speicherverbrauch

- Minimum-Funktion kann mehr als  $|I^f| + |I^g|$  Interpolationspunkte enthalten

## Problem:

- Während Profilsuche werden Funktionen gemergt
- Laufzeit der Profilsuchen wird durch diese Operationen (link + merge) dominiert

## Problem 1: Anzahl Interpolationspunkte

- Je länger ein Shortcut desto mehr Interpolationspunkte
- Kleiner Suchraum bezüglich Anzahl Knoten und Kanten
- Aber trotzdem viele Interpolationspunkte
- Teure Vorberechnungen

## Problem 2: Numerische Instabilität

- Linken und Mergen nicht genau falls mit Gleitkommazahlen implementiert
- Fehler akkumulieren sich falls man Linken und Mergen verkettet

## Problem 2: Numerische Instabilität

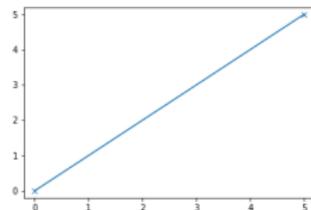
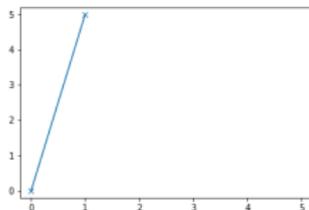
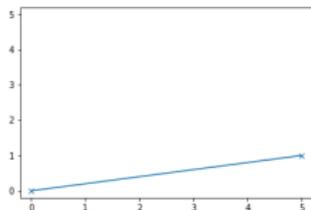
- Linken und Mergen nicht genau falls mit Gleitkommazahlen implementiert
- Fehler akkumulieren sich falls man Linken und Mergen verkettet
- Aber: Alle Operationen bleiben in den rationalen Zahlen.

## Problem 2: Numerische Instabilität

- Linken und Mergen nicht genau falls mit Gleitkommazahlen implementiert
- Fehler akkumulieren sich falls man Linken und Mergen verkettet
- Aber: Alle Operationen bleiben in den rationalen Zahlen.
  - Mit Brüchen arbeiten?
  - Vorläufige Experimente zeigen, dass Nenner und Zähler unkontrolliert wachsen
  - Offene Frage: Kann man diese Divergenz zeigen?

## Problem 2: Numerische Instabilität

- Linken und Mergen nicht genau falls mit Gleitkommazahlen implementiert
- Fehler akkumulieren sich falls man Linken und Mergen verkettet
- Aber: Alle Operation bleiben in den rationalen Zahlen.
  - Mit Brüchen arbeiten?
  - Vorläufige Experimente zeigen, dass Nenner und Zähler unkontrolliert wachsen
  - Offene Frage: Kann man diese Divergenz zeigen?
- Integer auch kaputt



# Linken & Mergen: Probleme

**Problem 3:** Implementierung :'(

- Netzwerk Deutschland  $|V| \approx 4.7$  Mio.,  $|E| \approx 10.8$  Mio. von 2006
- Standard-Instanz in der Forschung
- Basierend auf realen Verkehrsdaten
- 5 Verkehrsszenarien:
  - Montag:  $\approx 8\%$  Kanten zeitabhängig
  - Dienstag - Donnerstag:  $\approx 8\%$
  - Freitag:  $\approx 7\%$
  - Samstag:  $\approx 5\%$
  - Sonntag:  $\approx 3\%$

# "Grad" der Zeitabhängigkeit

	#delete mins	slow-down	time [ms]	slow-down
kein	2,239,500	0.00%	1219.4	0.00%
Montag	2,377,830	6.18%	1553.5	27.40%
DiDo	2,305,440	2.94%	1502.9	23.25%
Freitag	2,340,360	4.50%	1517.2	24.42%
Samstag	2,329,250	4.01%	1470.4	20.59%
Sonntag	2,348,470	4.87%	1464.4	20.09%

## Beobachtung:

- kaum Veränderung in Suchraum
- Anfragen etwas langsamer durch Auswertung

		Nodes	Arcs [ $\cdot 10^6$ ]	Avg. $ f $	Rel. Delay [%]		Size
		[ $\cdot 10^6$ ]	(TD [%])	TD arcs	All	TD	[GB]
Ber	Tuesday	0.4	1.0 (27.4)	75.0	3.1	17.6	0.2
	Saturday	0.4	1.0 (20.2)	69.1	2.1	14.8	0.1
Ger06	midweek	4.7	10.8 (7.2)	19.5	1.7	33.1	0.3
	Saturday	4.7	10.8 (3.9)	15.8	0.8	28.5	0.2
SynEur	Low	18.0	42.2 (0.1)	13.2	0.3	125.2	0.8
	Medium	18.0	42.2 (1.0)	13.2	0.8	124.9	0.8
	High	18.0	42.2 (6.2)	13.2	4.6	124.8	1.0
Ger17	Tuesday	7.2	15.8 (29.2)	31.6	3.5	20.8	1.3
Eur17	Tuesday	25.8	55.5 (27.2)	29.5	2.7	19.0	4.2
Eur20	Tuesday	28.5	60.9 (76.3)	22.5	21.0	34.9	8.7

## Beobachtung:

- Nicht durchführbar auf Europa-Instanz durch zu großen Speicherbedarf (> 32 GiB RAM)
    - Speicherverbrauch explodiert wegen durch Linkoperationen hinzukommenden Stützstellen
    - Vergleich Theorie: Linken kein Problem, potentiell super-polynomiell viele Stützstellen durch Mergen
  - Extrapoliert:
    - Suchraum steigt um ca. 10%
    - Suchzeiten um einen Faktor von bis zu 2 500 über Dijkstra
- ⇒ impraktikabel

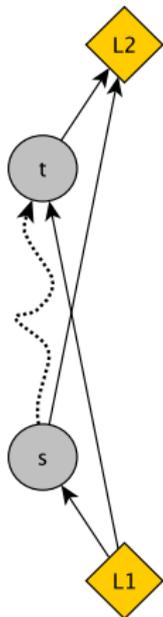
## Zeitabhängige Netzwerke (Basics)

- Funktionen statt Konstanten an Kanten
- Operationen werden teurer
  - $\mathcal{O}(\log |I|)$  für Auswertung
  - $\mathcal{O}(|I^f| + |I^g|)$  für Linken und Minimum
  - Speicherverbrauch explodiert
- Zeitanfragen:
  - Normaler Dijkstra
  - Kaum langsamer (lediglich Auswertung)
- Profilanfragen
  - nicht zu handhaben

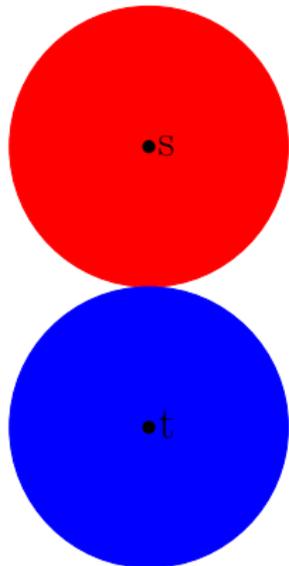
# Bausteine

# Beschleunigungstechniken

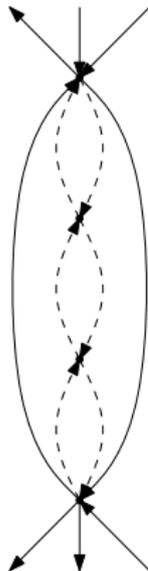
## Landmarken



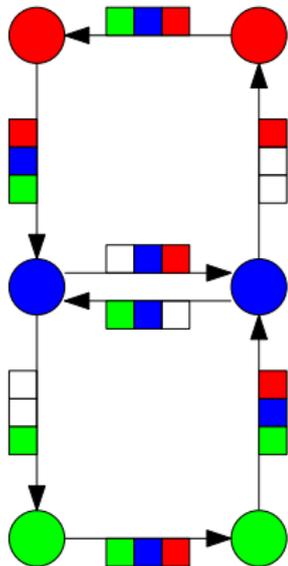
## Bidirektionale Suche



## Kontraktion



## Arc-Flags



## Vorbereitung:

- wähle eine Hand voll ( $\approx 16$ ) Knoten als Landmarken
- berechne Abstände von und zu allen Landmarken

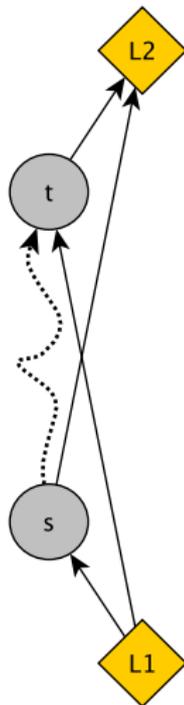
## Anfrage:

- benutze Landmarken und Dreiecksungleichung um eine untere Schranke für den Abstand zum Ziel zu bestimmen

$$d(s, t) \geq d(L_1, t) - d(L_1, s)$$

$$d(s, t) \geq d(s, L_2) - d(t, L_2)$$

- verändert Reihenfolge der besuchten Knoten



## Beobachtung:

- Korrektheit von ALT basiert darauf, dass reduzierte Kosten größer oder gleich 0 sind

$$\text{len}_\pi(u, v) = \text{len}(u, v) - \pi(u) + \pi(v) \stackrel{!}{\geq} 0$$

- durch Erhöhen der Kantengewichte wird dies nicht verletzt

## Beobachtung:

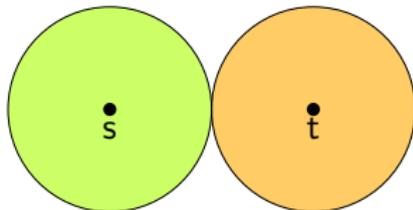
- Korrektheit von ALT basiert darauf, dass reduzierte Kosten größer oder gleich 0 sind

$$\text{len}_\pi(u, v) = \text{len}(u, v) - \pi(u) + \pi(v) \stackrel{!}{\geq} 0$$

- durch Erhöhen der Kantengewichte wird dies nicht verletzt

## Somit:

- Definiere lowerbound-Graph  $\underline{G} = (V, E, \underline{\text{len}})$  mit  $\underline{\text{len}} := \min \text{len}$
- Vorberechnung auf lowerbound-Graph
- korrekt aber eventuell langsamere Anfragezeiten



- starte zweite Suche von  $t$
- relaxiere rückwärts nur eingehende Kanten
- stoppe die Suche, wenn beide Suchräume sich treffen

## Zeitanfragen:

- Ankunft unbekannt  $\Rightarrow$  Rückwärtsuche?

## Zeitanfragen:

- Ankunft unbekannt  $\Rightarrow$  Rückwärtsuche?
- Rückwärtssuche nur zur Einschränkung der Vorwärtssuche
- je nach Beschleunigungstechnik verschieden  $\rightsquigarrow$  später

## Zeitanfragen:

- Ankunft unbekannt  $\Rightarrow$  Rückwärtsuche?
- Rückwärtssuche nur zur Einschränkung der Vorwärtssuche
- je nach Beschleunigungstechnik verschieden  $\rightsquigarrow$  später

## Profilanfragen:

- Anfrage zu allen Startzeitpunkten
- somit Rückwärtsuche kein Problem
- $\mu$ : tentative Abstandsfunktion
- breche ab, wenn  $\text{minKey}(\vec{Q}) + \text{minKey}(\overleftarrow{Q}) \geq \bar{\mu}$   
Erinnere: key von  $v$  ist der lowerbound seiner Profilfunktion

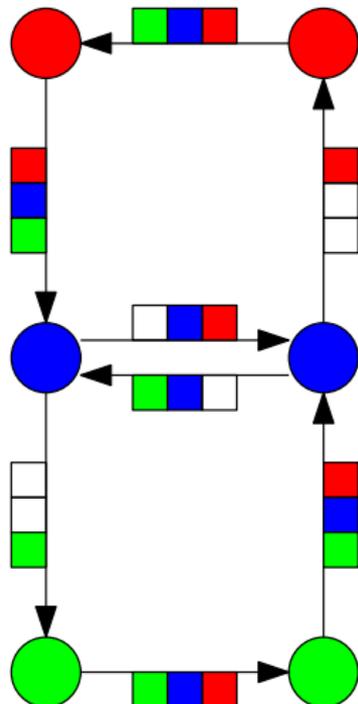
# Arc-Flags

## Idee:

- partitioniere den Graph in  $k$  Zellen
- hänge ein Label mit  $k$  Bits an jede Kante
- zeigt ob  $e$  wichtig für die Zielzelle ist
- modifizierter Dijkstra überspringt unwichtige Kanten

## Beobachtung:

- Partition wird auf ungewichtetem Graphen durchgeführt
- Flaggen müssen allerdings angepasst werden



# Anpassung

## Idee:

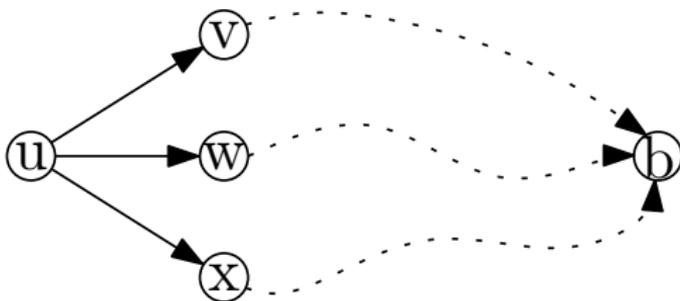
- ändere Intuition einer gesetzten Flagge
- Konzept bleibt gleich: Eine Flagge pro Kante und Region
- setze Flagge wenn Kante mindestens ein mal am Tag “wichtig” ist

## Idee:

- ändere Intuition einer gesetzten Flagge
- Konzept bleibt gleich: Eine Flagge pro Kante und Region
- setze Flagge wenn Kante mindestens ein mal am Tag “wichtig” ist

## Anpassung:

- für alle Randknoten  $b$  und alle Knoten  $u$ :
- Berechne Abstandsfunktion  $d_*(u, b)$
- setze Flagge wenn gilt  $\text{len}(u, v) \oplus d_*(v, b) \not\geq d_*(u, b)$

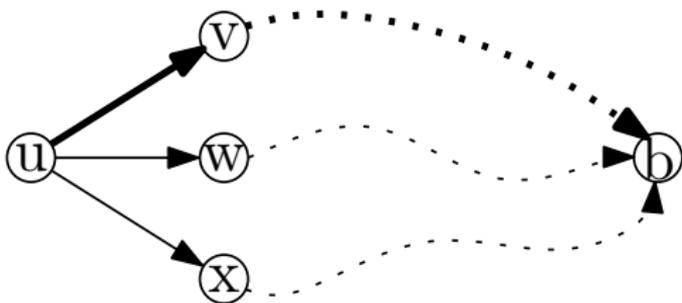


## Idee:

- ändere Intuition einer gesetzten Flagge
- Konzept bleibt gleich: Eine Flagge pro Kante und Region
- setze Flagge wenn Kante mindestens ein mal am Tag “wichtig” ist

## Anpassung:

- für alle Randknoten  $b$  und alle Knoten  $u$ :
- Berechne Abstandsfunktion  $d_*(u, b)$
- setze Flagge wenn gilt  $\text{len}(u, v) \oplus d_*(v, b) \not\geq d_*(u, b)$

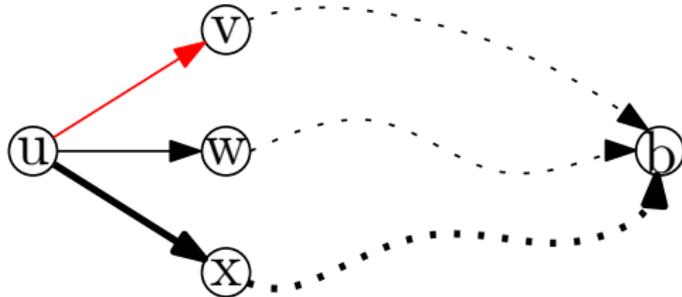


## Idee:

- ändere Intuition einer gesetzten Flagge
- Konzept bleibt gleich: Eine Flagge pro Kante und Region
- setze Flagge wenn Kante mindestens ein mal am Tag “wichtig” ist

## Anpassung:

- für alle Randknoten  $b$  und alle Knoten  $u$ :
- Berechne Abstandsfunktion  $d_*(u, b)$
- setze Flagge wenn gilt  $\text{len}(u, v) \oplus d_*(v, b) \not\geq d_*(u, b)$

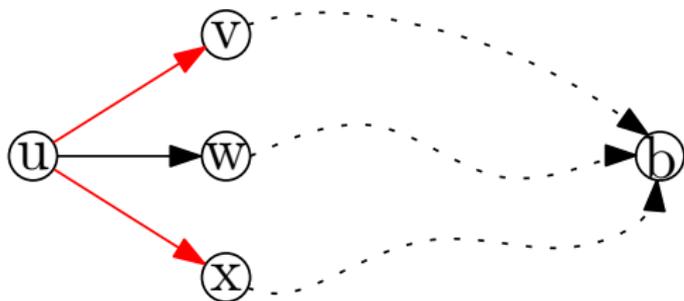


## Idee:

- ändere Intuition einer gesetzten Flagge
- Konzept bleibt gleich: Eine Flagge pro Kante und Region
- setze Flagge wenn Kante mindestens ein mal am Tag “wichtig” ist

## Anpassung:

- für alle Randknoten  $b$  und alle Knoten  $u$ :
- Berechne Abstandsfunktion  $d_*(u, b)$
- setze Flagge wenn gilt  $\text{len}(u, v) \oplus d_*(v, b) \not\geq d_*(u, b)$

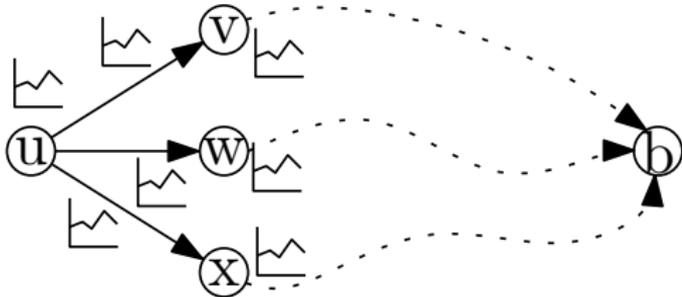


# Approximation Arc-Flags

## Beobachtung:

- viele Interpolationspunkte
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

## Idee:



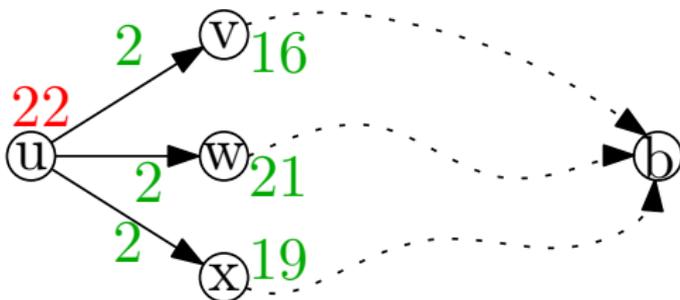
# Approximation Arc-Flags

## Beobachtung:

- viele Interpolationspunkte
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

## Idee:

- benutze über- and Unterapproximation



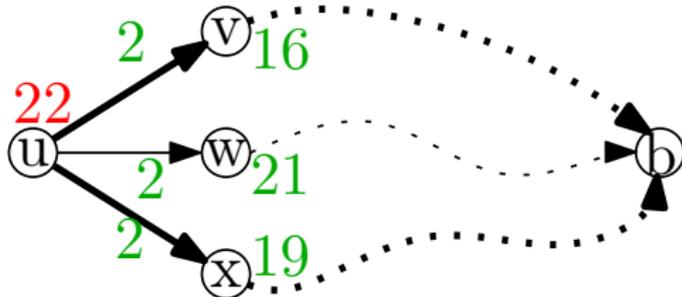
# Approximation Arc-Flags

## Beobachtung:

- viele Interpolationspunkte
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

## Idee:

- benutze über- and Unterapproximation



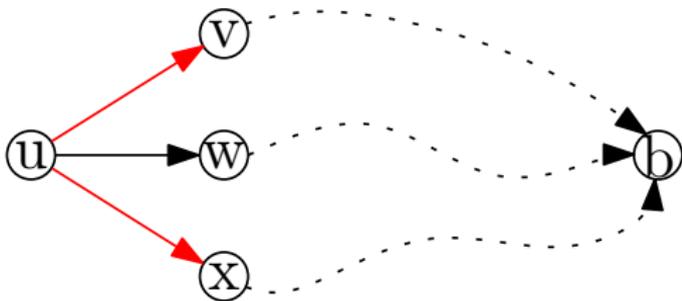
# Approximation Arc-Flags

## Beobachtung:

- viele Interpolationspunkte
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

## Idee:

- benutze über- and Unterapproximation
- ⇒ schnellere Vorberechnung, langsamere Anfragen
- ⇒ aber immer noch korrekt



## Idee:

- führe von jedem Randknoten  $K$  Zeitanfragen aus
- mit fester Ankunftszeit
- setze Flagge, wenn Kante auf einem dem Bäume eine Baumkante ist

## Idee:

- führe von jedem Randknoten  $K$  Zeitanfragen aus
- mit fester Ankunftszeit
- setze Flagge, wenn Kante auf einem dem Bäume eine Baumkante ist

## Beobachtungen:

- Flaggen eventuell nicht korrekt
- ein Pfad wird aber immer gefunden
- Fehlerrate?

# Kontraktion

## Knoten-Reduktion:

- entferne Knoten
- füge neue Kanten (Shortcuts) hinzu, um die Abstände zwischen verbleibenden Knoten zu erhalten

## Zeugensuche:

- behalte nur relevante Shortcuts
- lokale Suche während oder nach Knoten-Reduktion



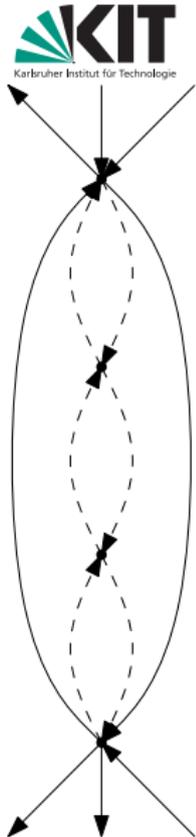
# Kontraktion

## Knoten-Reduktion:

- entferne Knoten
- füge neue Kanten (Shortcuts) hinzu, um die Abstände zwischen verbleibenden Knoten zu erhalten

## Zeugensuche:

- behalte nur relevante Shortcuts
- lokale Suche während oder nach Knoten-Reduktion



## Zeitunabhängig:

- Kante  $(u, v)$  nicht nötig, wenn  $(u, v)$  nicht Teil des kürzesten Weges von  $u$  nach  $v$  ist, also  $\text{len}(u, v) > d(u, v)$
- lokale Dijkstra-Suche von  $u$

## Zeitabhängig:

## Zeitunabhängig:

- Kante  $(u, v)$  nicht nötig, wenn  $(u, v)$  nicht Teil des kürzesten Weges von  $u$  nach  $v$  ist, also  $\text{len}(u, v) > d(u, v)$
- lokale Dijkstra-Suche von  $u$

## Zeitabhängig:

- Kante  $(u, v)$  nicht nötig, wenn  $(u, v)$  nicht Teil eines kürzesten Wege von  $u$  nach  $v$  ist, also  $\text{len}(u, v) > d_*(u, v)$
- lokale Profilsuche
- Problem: deutlich langsamer

## Idee:

- führe zunächst zwei Dijkstra-Suchen mit  $\underline{\text{len}}$  und  $\overline{\text{len}}$  durch
- relaxiere dann nur solche Kanten  $(u, v)$ , für die  $\underline{d(s, u)} + \underline{\text{len}(u, v)} \leq \overline{d(s, v)}$  gilt
- lokale Profilsuche in diesem Korridor

## Anmerkung:

- auch zur Beschleunigung von  $s$ - $t$  Profil-Suchen

## Problem:

- hoher Speicherbedarf der Shortcuts

## Ideen:

- Shortcuts nur approximieren, inexakte Anfragen
- Keine Gewichte am Shortcut speichern, stattdessen on-the-fly entpacken und Pfad linken spart Speicher, kostet Laufzeit
- speichere auf Shortcuts Über- und Unterapproximation der Funktionen
  - induzieren wieder Korridor (aber genaueren als nur Min/Max!)
  - entpacke Shortcuts im Korridor, dies gibt einen Teil des Originalgraphen
  - benutze nun die nicht-approximierten Originalkanten für eine exakte Suche

## Basismodule:

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

## Somit sind folgende Algorithmen gute Kandidaten

- ALT
- Core-ALT
- SHARC
- Contraction Hierarchies
- MLD (CRP)

# Mittwoch, 22. Juni 2022

- Daniel Delling:  
**Engineering and Augmenting Route Planning Algorithms**  
Ph.D. Thesis, Universität Karlsruhe (TH), 2009.
- Gernot Veit Batz, Robert Geisberger, Peter Sanders, Christian Vetter:  
**Minimum Time-Dependent Travel Times with Contraction Hierarchies**  
In: *Journal of Experimental Algorithmics*, 2013.
- Moritz Baum, Julian Dibbelt, Thomas Pajor, Dorothea Wagner:  
**Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**  
In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'16)*, 2016.

- Ben Strasser: **Dynamic Time-Dependent Routing in Road Networks Through Sampling**  
In: *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'17)*, 2017.
- Ben Strasser, Dorothea Wagner, Tim Zeitz: **Space-efficient, Fast and Exact Routing in Time-dependent Road Networks**  
In: *Proceedings of the 28th Annual European Symposium on Algorithms (ESA'20)*