

Algorithmen für Routenplanung

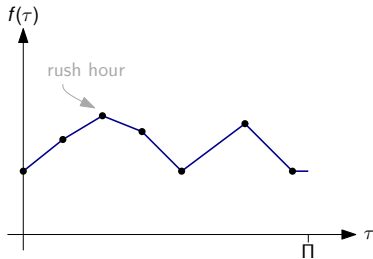
14. Sitzung, Sommersemester 2011

Thomas Pajor | 16. Juni 2011

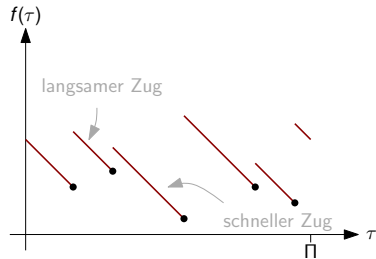
INSTITUT FÜR THEORETISCHE INFORMATIK · ALGORITHMIK I · PROF. DR. DOROTHEA WAGNER



Einführung in zeitabhängige Routenplanung



Straße



Schiene

Funktion f durch Interpolationspunkte: $I^f := \{(t_1^f, w_1^f), \dots, (t_k^f, w_k^f)\}$

Zeit-Anfrage:

- finde kürzesten Weg für Abfahrtszeit τ
- analog zu Dijkstra

Profil-Anfrage:

- finde kürzesten Weg für alle Abfahrtszeitpunkte
- Label-Correcting Algorithmus

Zeit-Anfrage:

- finde kürzesten Weg für Abfahrtszeit τ
- analog zu Dijkstra

Profil-Anfrage:

- finde kürzesten Weg für alle Abfahrtszeitpunkte
- Label-Correcting Algorithmus

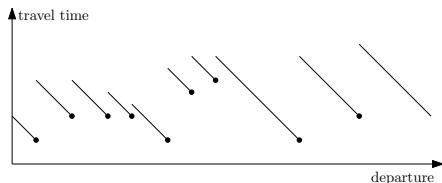
Evaluation von $f(\tau)$:

- Suche Punkte mit $t_i \geq \tau$ und $t_i - \tau$ minimal
- dann Evaluation durch

$$f(\tau) = w_i + (t_i - \tau)$$

Problem:

- Finden von t_i und t_{i+1}
- Theoretisch:
 - Lineare Suche: $\mathcal{O}(|I|)$
 - Binäre Suche: $\mathcal{O}(\log_2 |I|)$
- praktisch:
 - $|I| < 30$: Lineare Suche
 - Sonst: Lineare Suche mit Startpunkt $\frac{\tau}{\bar{t}} \cdot |I|$



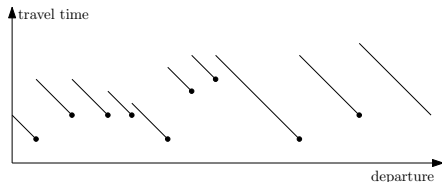
Evaluation von $f(\tau)$:

- Suche Punkte mit $t_i \geq \tau$ und $t_i - \tau$ minimal
- dann Evaluation durch

$$f(\tau) = w_i + (t_i - \tau)$$

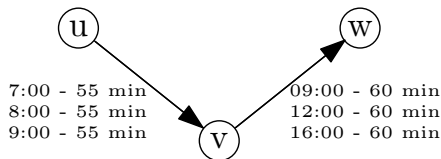
Problem:

- Finden von t_i und t_{i+1}
- Theoretisch:
 - Lineare Suche: $\mathcal{O}(|I|)$
 - Binäre Suche: $\mathcal{O}(\log_2 |I|)$
- praktisch:
 - $|I| < 30$: Lineare Suche
 - Sonst: Lineare Suche mit Startpunkt $\frac{\tau}{\bar{t}} \cdot |I|$



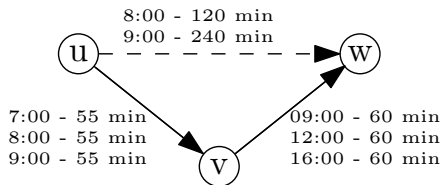
Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



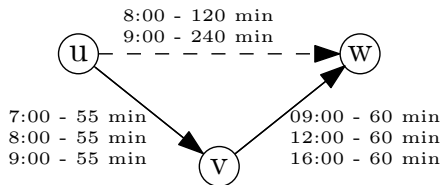
Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



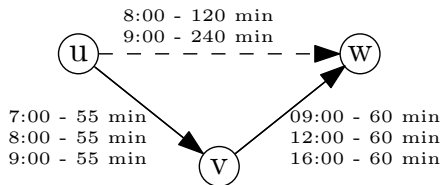
Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



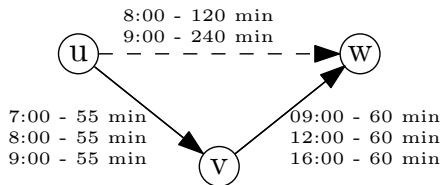
Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



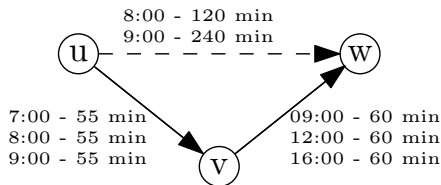
Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



Linken zweier Funktionen f und g

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - t_i^f - w_i^f \geq 0$ minimal
- Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g + w_j^g - t_i^f)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Wieder Sweep-Algorithmus



Laufzeit

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

Speicherverbrauch

- Gelinkte Funktion hat $\min\{|I^f|, |I^g|\}$ Interpolationspunkte

Somit:

- Deutlich gutmütiger als Straßengraph-Funktionen

Laufzeit

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

Speicherverbrauch

- Gelinkte Funktion hat $\min\{|I^f|, |I^g|\}$ Interpolationspunkte

Somit:

- Deutlich gutmütiger als Straßengraph-Funktionen

Laufzeit

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

Speicherverbrauch

- Gelinkte Funktion hat $\min\{|I^f|, |I^g|\}$ Interpolationspunkte

Somit:

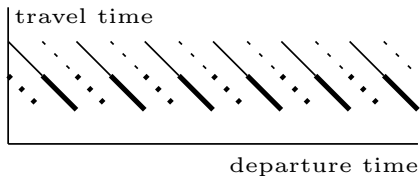
- Deutlich gutmütiger als Straßengraph-Funktionen

Minimum zweier Funktionen f und g

- Für alle (t_j^f, w_j^f) : behalte Punkt, wenn $w_j^f < g(t_j^f)$
- Für alle (t_j^g, w_j^g) : behalte Punkt, wenn $w_j^g < f(t_j^g)$
- Keine Schnittpunkte möglich(!)

Vorgehen:

- Linearer Sweep

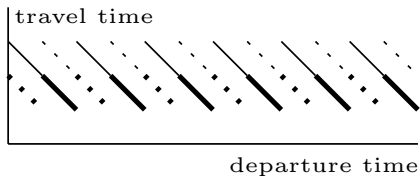


Minimum zweier Funktionen f und g

- 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 $w_j^g < f(t_j^g)$
- Keine Schnittpunkte möglich(!)

Vorgehen:

- Linearer Sweep

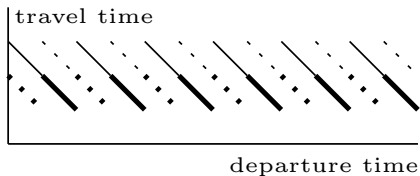


Minimum zweier Funktionen f und g

- 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 $w_j^g < f(t_j^g)$
- Keine Schnittpunkte möglich(!)

Vorgehen:

- Linearer Sweep



Laufzeit

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

Speicherverbrauch

- Keine Schnittpunkte
- ⇒ Minimum-Funktion kann maximal $|I^f| + |I^g|$ Interpolationspunkte enthalten

Laufzeit

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

Speicherverbrauch

- Keine Schnittpunkte
- ⇒ Minimum-Funktion kann maximal $|I^f| + |I^g|$ Interpolationspunkte enthalten

Schiene vs. Straße

Laufzeit Operationen

- gleich für beide
- $\mathcal{O}(\log |I|)$ für Auswertung
- $\mathcal{O}(|I^f| + |I^g|)$ für Linken und Minimum

Speicherverbrauch

- Public Transport deutlich geringer
- Link:

$$|I^{f \oplus g}| \leq \min\{|I^f|, |I^g|\} \quad \text{vs.} \quad |I^{f \oplus g}| \approx |I^f| + |I^g|$$

- Merge:

$$|I^{\min\{f,g\}}| \leq |I^f| + |I^g| \quad \text{vs. eventuell} \quad |I^{\min\{f,g\}}| > (|I^f| + |I^g|)$$

Profilsuchen

- Somit in Public Transport Netzen wahrscheinlich schneller

Schiene vs. Straße

Laufzeit Operationen

- gleich für beide
- $\mathcal{O}(\log |I|)$ für Auswertung
- $\mathcal{O}(|I^f| + |I^g|)$ für Linken und Minimum

Speicherverbrauch

- Public Transport deutlich geringer
- Link:

$$|I^{f \oplus g}| \leq \min\{|I^f|, |I^g|\} \quad \text{vs.} \quad |I^{f \oplus g}| \approx |I^f| + |I^g|$$

- Merge:

$$|I^{\min\{f,g\}}| \leq |I^f| + |I^g| \quad \text{vs. eventuell} \quad |I^{\min\{f,g\}}| > (|I^f| + |I^g|)$$

Profilsuchen

- Somit in Public Transport Netzen wahrscheinlich schneller

Laufzeit Operationen

- gleich für beide
- $\mathcal{O}(\log |I|)$ für Auswertung
- $\mathcal{O}(|I^f| + |I^g|)$ für Linken und Minimum

Speicherverbrauch

- Public Transport deutlich geringer
- Link:

$$|I^{f \oplus g}| \leq \min\{|I^f|, |I^g|\} \quad \text{vs.} \quad |I^{f \oplus g}| \approx |I^f| + |I^g|$$

- Merge:

$$|I^{\min\{f,g\}}| \leq |I^f| + |I^g| \quad \text{vs. eventuell} \quad |I^{\min\{f,g\}}| > (|I^f| + |I^g|)$$

Profilsuchen

- Somit in Public Transport Netzen wahrscheinlich schneller

Straße:

- Netzwerk Deutschland $|V| \approx 4.7$ Mio., $|E| \approx 10.8$ Mio.
- 5 Verkehrsszenarien:
 - Montag: $\approx 8\%$ Kanten zeitabhängig
 - Dienstag - Donnerstag: $\approx 8\%$
 - Freitag: $\approx 7\%$
 - Samstag: $\approx 5\%$
 - Sonntag: $\approx 3\%$

Schiene:

- Europa Fernverbindungen
- 30 156 Stationen, 1.8 Millionen Verbindungen
- $|V| = 0.4$ Mio., $|E| = 1.4$ Mio.

”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



Beobachtung:

- Nicht durchführbar durch zu großen Speicherbedarf (> 32 GiB RAM)
- Interpoliert:
 - Suchraum steigt um ca. 10%
 - Suchzeiten um einen Faktor von bis zu 2 500

⇒ inpraktikabel

	#delete mins	time [ms]
Zeit-Anfragen	260 095	125
Profil-Anfragen	1 919 662	5 327

Beobachtung:

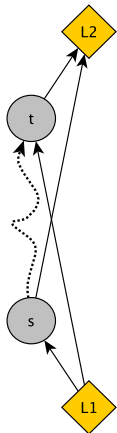
- Deletemins steigen an (ungefähr Faktor 8)
- Queryzeit steigt an um Faktor 42
- Verlust für Operationen ist ca. 5

Zeitabhängige Netzwerke (Basics)

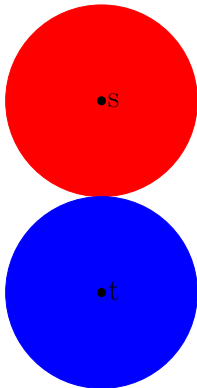
- Funktionen statt Konstanten an Kanten
- Operationen werden teurer
 - $\mathcal{O}(\log |I|)$ für Auswertung
 - $\mathcal{O}(|I^r| + |I^g|)$ für Linken und Minimum
 - Straßennetze: Speicherverbrauch explodiert
 - Eisenbahn: gutartiger
- Zeitanfragen:
 - Normaler Dijkstra
 - Kaum langsamer (lediglich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

Jetzt: Beschleunigungstechniken

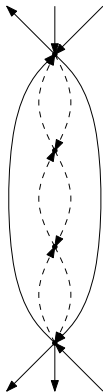
Landmarken



Bidirektionale Suche



Kontraktion



Arc-Flags

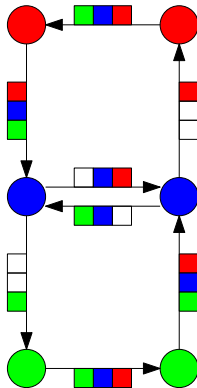
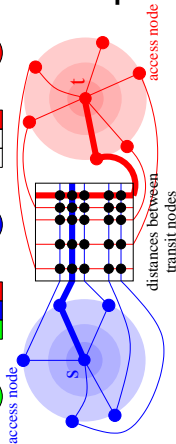


Table-Lookups



Vorbereitung:

- wähle eine Hand voll (≈ 16) Knoten als **Landmarken**
- berechne Abstände von und zu allen Landmarken

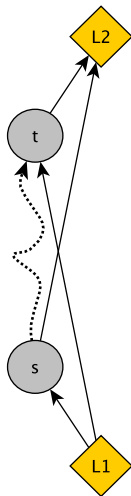
Anfrage:

- 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 größergleich 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 $\underline{\text{len}} := \min \text{len}$
- Vorbereitung auf lowerbound-Graph
- korrekt aber eventuell langsamere Anfragezeiten

Beobachtung:

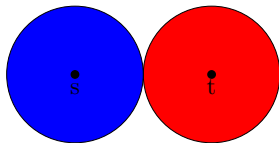
- Korrektheit von ALT basiert darauf, dass reduzierte Kosten größergleich 0 sind

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

- 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?
- Rückwärtssuche nur zum Einschränken der Vorwärtssuche benutzen
- je nach Beschleunigungstechnik verschieden \rightsquigarrow später

Profilanfragen:

- Anfrage zu allen Startzeitpunkten
- somit Rückwärtsuche kein Problem
- μ temporäre Abstandsfunktion
- breche ab, wenn $\bar{\mu} \leq \minKey(\vec{Q}) + \minKey(\overleftarrow{Q})$

Zeitanfragen:

- Ankunft unbekannt \Rightarrow Rückwärtsuche?
- Rückwärtssuche nur zum Einschränken der Vorwärtssuche benutzen
- je nach Beschleunigungstechnik verschieden \rightsquigarrow später

Profilanfragen:

- Anfrage zu allen Startzeitpunkten
- somit Rückwärtsuche kein Problem
- μ temporäre Abstandsfunktion
- breche ab, wenn $\bar{\mu} \leq \minKey(\vec{Q}) + \minKey(\overleftarrow{Q})$

Zeitanfragen:

- Ankunft unbekannt \Rightarrow Rückwärtsuche?
- Rückwärtssuche nur zum Einschränken der Vorwärtssuche benutzen
- je nach Beschleunigungstechnik verschieden \rightsquigarrow später

Profilanfragen:

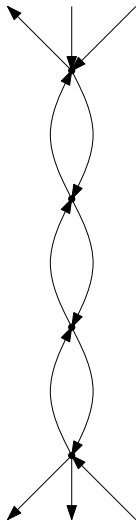
- Anfrage zu allen Startzeitpunkten
- somit Rückwärtsuche kein Problem
- μ temporäre Abstandsfunktion
- breche ab, wenn $\bar{\mu} \leq \minKey(\vec{Q}) + \minKey(\overleftarrow{Q})$

Knoten-Reduktion:

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

Kanten-Reduktion:

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

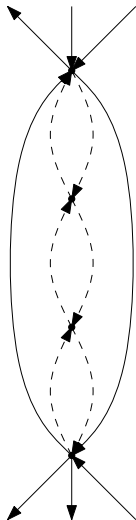


Knoten-Reduktion:

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

Kanten-Reduktion:

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

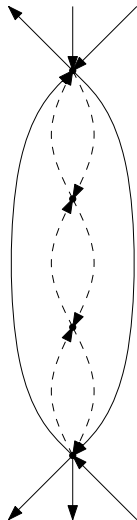


Beobachtung:

- Verfahren ist unabhängig von Metrik
- Shortcuts müssen nur Pfad entsprechen

Somit:

- Linken der Funktionen zu einer Shortcut-Funktion
- Speicherverbrauch steigt an (Straße)!



Zeitunabhängig:

- Lösche Kante (u, v) , 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:

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

Idee:

- lösche zunächst Kanten (u, v) für die $\text{len}(u, v) > \overline{d_*(u, v)}$ gilt
- danach lokale Profilsuche

Zeitunabhängig:

- Lösche Kante (u, v) , 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:

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

Idee:

- lösche zunächst Kanten (u, v) für die $\text{len}(u, v) > \overline{d_*(u, v)}$ gilt
- danach lokale Profilsuche

Zeitunabhängig:

- Lösche Kante (u, v) , 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:

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

Idee:

- lösche zunächst Kanten (u, v) für die $\text{len}(u, v) > \overline{d_*(u, v)}$ gilt
- danach lokale Profilsuche

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

Anmerkung:

- kann auch zur Beschleunigung einer s - t Profil-Suche genutzt werden

Problem:

- hoher Speicherbedarf der Shortcuts (Straße)

Ideen:

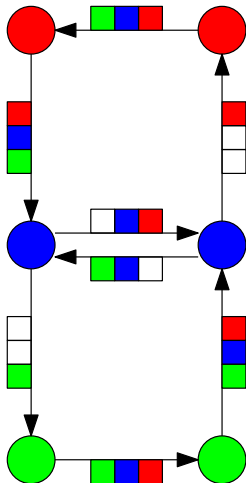
- Shortcuts on-the-fly entpacken und dann Gewicht des Pfades berechnen
- speichere Approximationen der Funktionen, führe dann Korridorsuche bei Query auf Originalgraphen durch
- durch speichern von Approximationen genauer

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 aktualisiert werden



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:

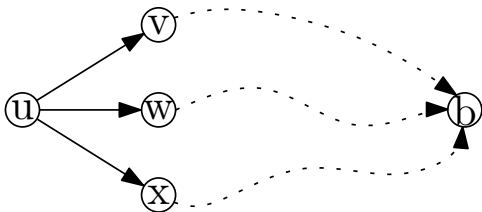
- für alle Randknoten b und alle Knoten u :
- Berechne Abstandsfunktion $d_*(u, b)$
- setze Flagge wenn gilt $\text{len}(u, v) \oplus d_*(v, b) \not\geq d_*(u, b)$

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 $\text{len}(u, v) \oplus d_*(v, b) \not\leq d_*(u, b)$

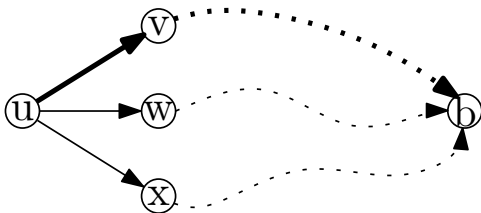


Idee:

- 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 $\text{len}(u, v) \oplus d_*(v, b) \not\leq d_*(u, b)$

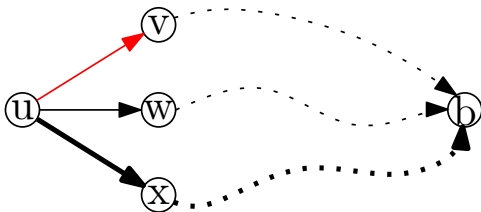


Idee:

- 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 $\text{len}(u, v) \oplus d_*(v, b) \neq d_*(u, b)$

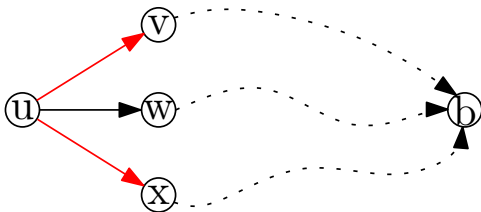


Idee:

- 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 $\text{len}(u, v) \oplus d_*(v, b) \not\leq d_*(u, b)$

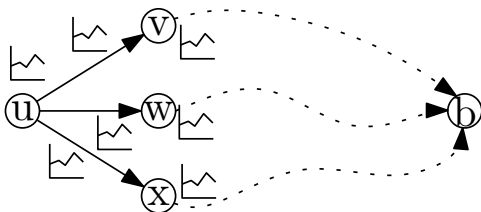


Beobachtung:

- **viele** Interpolationspunkte (Straße)
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

Idee:

- benutze Über- und Unterapproximation
- ⇒ schnellere Vorberechnung, langsamere Anfragen
- ⇒ aber immer noch korrekt

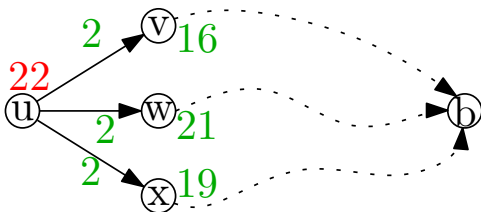


Beobachtung:

- viele Interpolationspunkte (Straße)
- 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

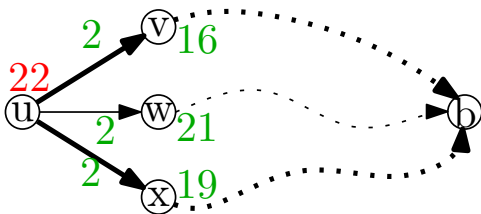


Beobachtung:

- **viele** Interpolationspunkte (Straße)
- 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**

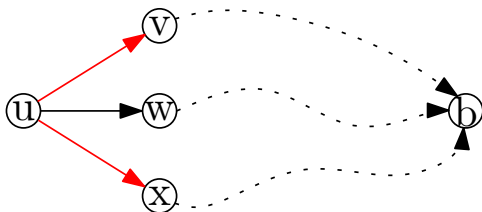


Beobachtung:

- **viele** Interpolationspunkte (Straße)
- Berechnung der Abstandsfunktionen ist sehr zeitintensiv
- Laufzeit stark abhängig von der Komplexität der Funktionen

Idee:

- benutze **Über-** und **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

Beobachtungen:

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

Idee:

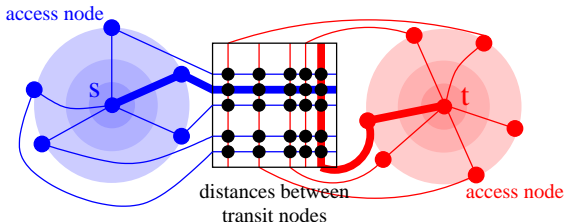
- führe von jedem Randknoten K Zeitanfragen aus
- mit fester Ankunftszeit
- setze Flagge, wenn Kante auf einem dem Bäume eine Baumkante ist

Beobachtungen:

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

Idee:

- speichere Distanztabellen
- nur für "wichtige" Teile des Graphen
- Suchen laufen nur bis zur Tabelle
- harmoniert gut mit hierarchischen Techniken



Beobachtung:

- Distanz-Tabelle muss aktualisiert werden
- ein Eintrag entspricht effektiv einem Shortcut
- vollständiger Overlay-Graph
- viele Shortcuts

also:

- Speicherverbrauch deutlich zu groß?

Basismodule:

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

Somit sind folgende Algorithmen gute Kandidaten

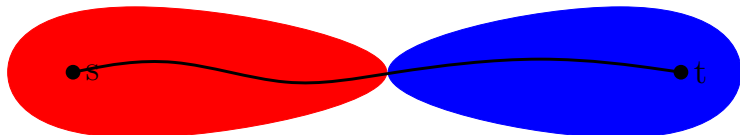
- ALT
- Core-ALT
- SHARC
- Contraction Hierarchies

● s

● t

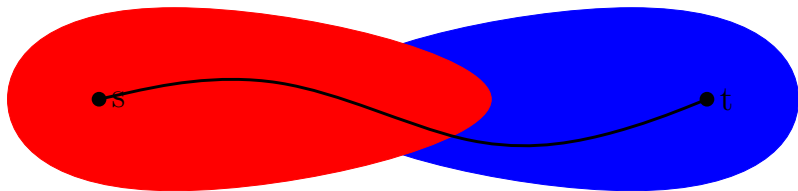
Idee - Drei Phasen:

- 1 Vorwärts zeitabhängig, Rückwärtssuche benutzt **Minima** der Funktionen. Fertig wenn Suchen sich treffen. Berechne **zeitabhängige** Distanz μ (durch Auswerten des gefundenen Weges).
- 2 Rückwärtssuche arbeitet weiter bis $\minKey(\overleftarrow{Q}) > \mu$
- 3 Vorwärtssuche arbeitet weiter bis t abgearbeitet worden ist und besucht nur Knoten, die die Rückwärtssuche zuvor besucht hat



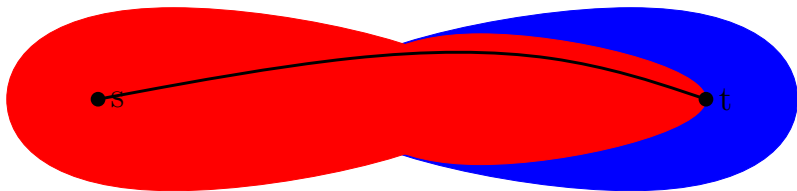
Idee - Drei Phasen:

- 1 Vorwärts zeitabhängig, Rückwärtssuche benutzt **Minima** der Funktionen. Fertig wenn Suchen sich treffen. Berechne **zeitabhängige** Distanz μ (durch Auswerten des gefundenen Weges).
- 2 Rückwärtssuche arbeitet weiter bis $\text{minKey}(\overleftarrow{Q}) > \mu$
- 3 Vorwärtssuche arbeitet weiter bis t abgearbeitet worden ist und besucht nur Knoten, die die Rückwärtssuche zuvor besucht hat



Idee - Drei Phasen:

- 1 Vorwärts zeitabhängig, Rückwärtssuche benutzt **Minima** der Funktionen. Fertig wenn Suchen sich treffen. Berechne **zeitabhängige** Distanz μ (durch Auswerten des gefundenen Weges).
- 2 Rückwärtssuche arbeitet weiter bis $\minKey(\overleftarrow{Q}) > \mu$
- 3 Vorwärtssuche arbeitet weiter bis t abgearbeitet worden ist und besucht nur Knoten, die die Rückwärtssuche zuvor besucht hat



Idee - Drei Phasen:

- 1 Vorwärts zeitabhängig, Rückwärtssuche benutzt **Minima** der Funktionen. Fertig wenn Suchen sich treffen. Berechne **zeitabhängige** Distanz μ (durch Auswerten des gefundenen Weges).
- 2 Rückwärtssuche arbeitet weiter bis $\text{minKey}(\overleftarrow{Q}) > \mu$
- 3 Vorwärtssuche arbeitet weiter bis t abgearbeitet worden ist und besucht nur Knoten, die die Rückwärtssuche zuvor besucht hat

Beobachtung:

- Phase 2 läuft recht lange weiter, bis $\min\text{Key}(\overleftarrow{Q}) > \mu$ gilt
- insbesondere dann schlecht, wenn die lower bounds stark vom echten Wert abweichen

Approximation:

- breche Phase 2 bereits ab, wenn $\min\text{Key}(\overleftarrow{Q}) \cdot K > \mu$ gilt
- dann ist der berechnete Weg eine K -Approximation des kürzesten Weges

scen.	algorithm	K	Error				Query		
			rate	relative av.	max	abs. max [s]	#sett. nodes	#rel. edges	time [ms]
mid	uni-ALT	–	0.0%	0.000%	0.00%	0	200 236	239 112	147.20
	TDALT	1.00	0.0%	0.000%	0.00%	0	116 476	138 696	98.27
		1.15	12.4%	0.094%	14.32%	1 892	50 764	60 398	36.91
		1.50	12.5%	0.097%	27.59%	1 892	50 742	60 371	36.86
Sat	uni-ALT	–	0.0%	0.000%	0.00%	0	148 331	177 568	100.07
	TDALT	1.00	0.0%	0.000%	0.00%	0	63 717	76 001	47.41
		1.15	10.5%	0.088%	13.97%	2 613	50 042	59 607	36.00
		1.50	10.6%	0.089%	26.17%	2 613	50 036	59 600	35.63
Sun	uni-ALT	–	0.0%	0.000%	0.00%	0	142 631	170 670	92.79
	TDALT	1.00	0.0%	0.000%	0.00%	0	58 956	70 333	42.96
		1.15	10.4%	0.088%	14.28%	1 753	50 349	59 994	36.04
		1.50	10.5%	0.089%	32.08%	1 753	50 345	59 988	35.74

Idee

- begrenze Beschleunigungstechnik auf kleinen Subgraphen (**Kern**)

s ●

● t

Vorbereitung

- kontrahiere Graphen zu einem Kern
- Landmarken nur im Kern

Anfrage

- Initialphase: normaler Dijkstra
- benutze Landmarken nur im Kern
- zeitabhängig:
 - Rückwärtssuche ist zeitunabhängig
 - Vorwärtssuche darf alle Knoten der Rückwärtssuche besuchen

Idee

- begrenze Beschleunigungstechnik auf kleinen Subgraphen (**Kern**)



Vorbereitung

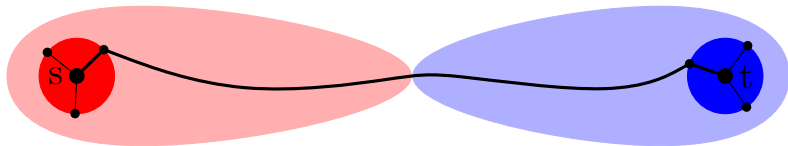
- kontrahiere Graphen zu einem Kern
- Landmarken nur im Kern

Anfrage

- Initialphase: normaler Dijkstra
- benutze Landmarken nur im Kern
- zeitabhängig:
 - Rückwärtssuche ist zeitunabhängig
 - Vorwärtssuche darf **alle** Knoten der Rückwärtssuche besuchen

Idee

- begrenze Beschleunigungstechnik auf kleinen Subgraphen (**Kern**)



Vorbereitung

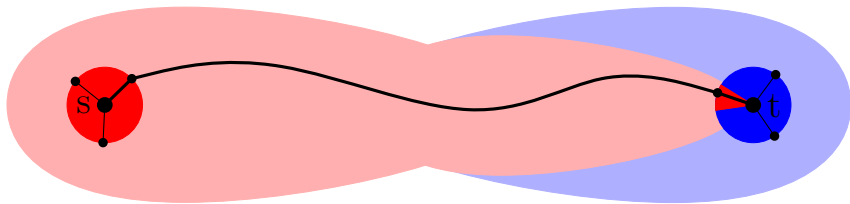
- kontrahiere Graphen zu einem Kern
- Landmarken nur im Kern

Anfrage

- Initialphase: normaler Dijkstra
- benutze Landmarken nur im Kern
- zeitabhängig:
 - Rückwärtssuche ist zeitunabhängig
 - Vorwärtssuche darf **alle** Knoten der Rückwärtssuche besuchen

Idee

- begrenze Beschleunigungstechnik auf kleinen Subgraphen (**Kern**)



Vorbereitung

- kontrahiere Graphen zu einem Kern
- Landmarken nur im Kern

Anfrage

- Initialphase: normaler Dijkstra
- benutze Landmarken nur im Kern
- zeitabhängig:
 - Rückwärtssuche ist zeitunabhängig
 - Vorwärtssuche darf **alle** Knoten der Rückwärtssuche besuchen

scenario	K	Preproc.		Error			Query			
		time [min]	space [B/n]	rate	relative av.	max	abs. max [s]	#settled nodes	#relaxed edges	time [ms]
Monday	1.00	9	50.3	0.0%	0.000%	0.00%	0	2984	11316	4.84
	1.15	9	50.3	8.3%	0.051%	11.00%	1618	1588	5303	1.84
	1.50	9	50.3	8.3%	0.052%	17.25%	1618	1587	5301	1.84
midweek	1.00	9	50.3	0.0%	0.000%	0.00%	0	3190	12255	5.36
	1.15	9	50.3	8.2%	0.051%	13.84%	2408	1593	5339	1.87
	1.50	9	50.3	8.2%	0.052%	13.84%	2408	1592	5337	1.86
Friday	1.00	8	44.9	0.0%	0.000%	0.00%	0	3097	12162	5.21
	1.15	8	44.9	7.8%	0.052%	11.29%	2348	1579	5376	1.82
	1.50	8	44.9	7.8%	0.054%	21.19%	2348	1579	5374	1.82
Saturday	1.00	6	27.8	0.0%	0.000%	0.00%	0	1856	7188	2.42
	1.15	6	27.8	4.4%	0.031%	11.50%	1913	1539	5542	1.71
	1.50	6	27.8	4.4%	0.031%	24.17%	1913	1539	5541	1.71
Sunday	1.00	5	19.1	0.0%	0.000%	0.00%	0	1773	6712	2.13
	1.15	5	19.1	4.0%	0.029%	12.72%	1400	1551	5541	1.68
	1.50	5	19.1	4.1%	0.029%	17.84%	1400	1550	5540	1.68

Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

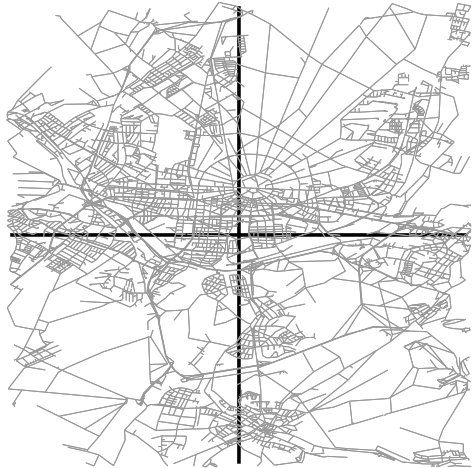


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

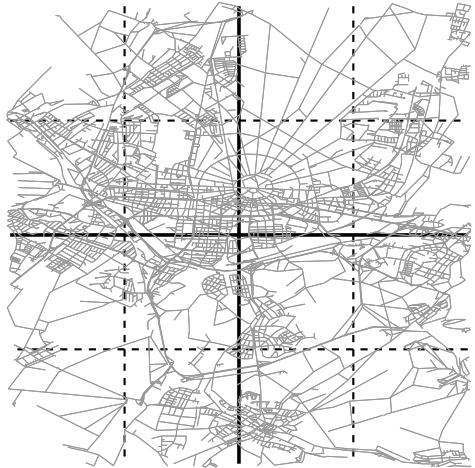


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

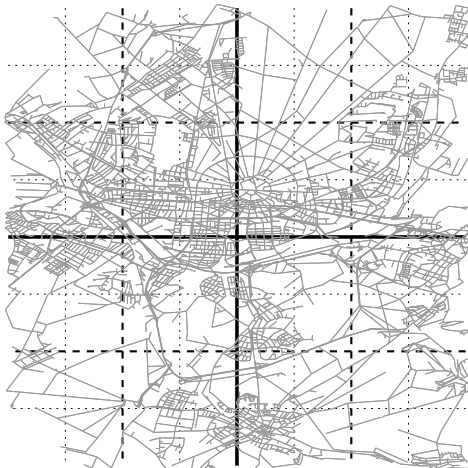


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

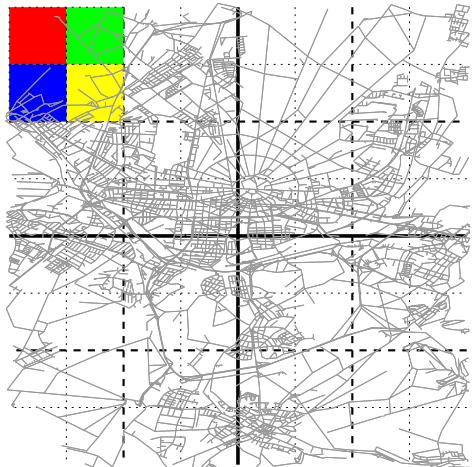


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

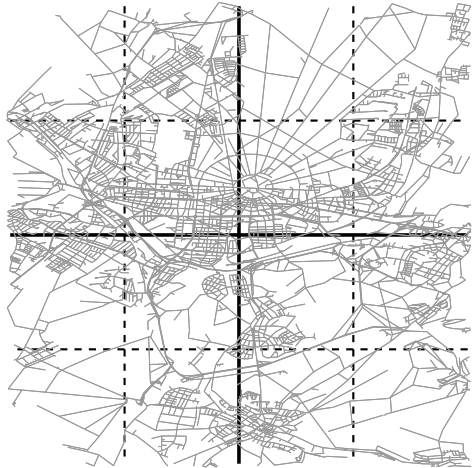


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

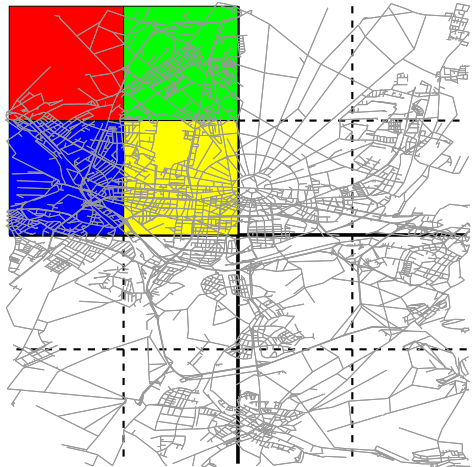


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?

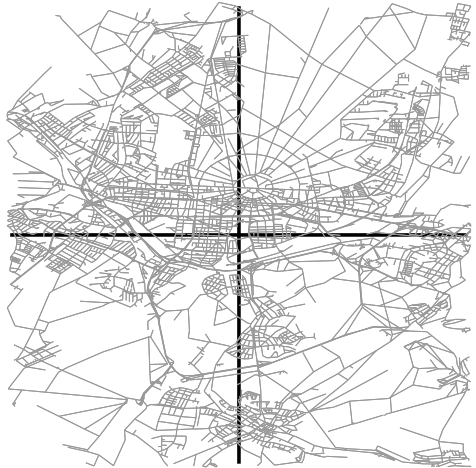


Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

- Kontraktion und Flaggen berechnung anpassen
- Verfeinerung?

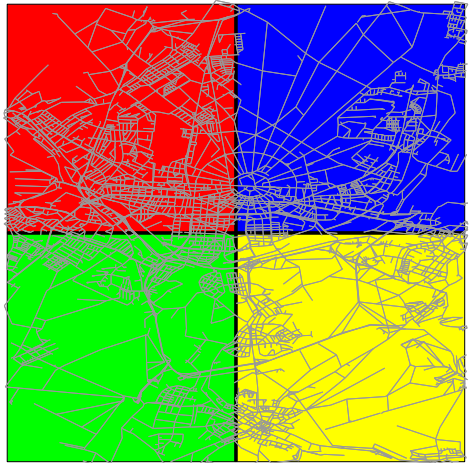


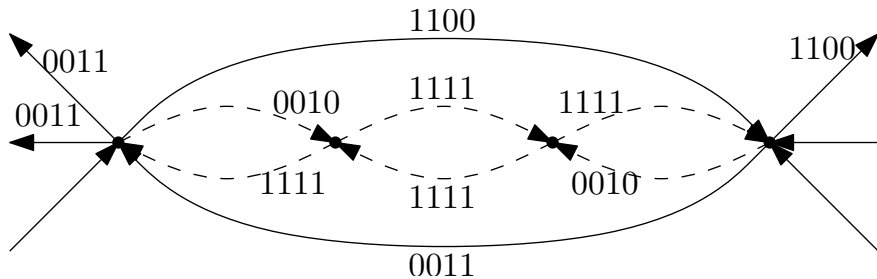
Vorbereitung:

- Multi-Level-Partition
- iterativer Prozess:
 - kontrahiere Subgraphen
 - berechne Flaggen
- Flaggenverfeinerung

Anpassung:

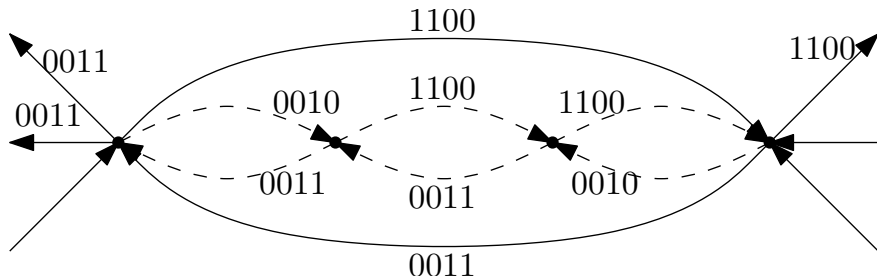
- Kontraktion und Flaggenberechnung anpassen
- Verfeinerung?





Vorgehen:

- verfeinere Flaggen
- propagiere Flaggen von wichtigen zu unwichtigen Kanten
- zeitunabhängig: mittels lokaler Suche
- zeitabhängig: mittels lokaler Profilsuche



Vorgehen:

- verfeinere Flaggen
- propagiere Flaggen von wichtigen zu unwichtigen Kanten
- zeitunabhängig: mittels lokaler Suche
- zeitabhängig: mittels lokaler Profilsuche

scenario	algom	Preprocessing				Time-Queries	
		time [h:m]	space [B/n]	edge inc.	points inc.	time [ms]	speed up
Monday	eco	1:16	156.6	25.4%	366.8%	24.55	63
midweek	eco	1:16	154.9	25.4%	363.8%	25.06	60
Friday	eco	1:10	142.0	25.4%	358.0%	22.07	69
Saturday	eco	0:42	90.3	25.0%	283.6%	5.34	276
	agg	48:57	84.3	24.5%	264.4%	0.58	2554
Sunday	eco	0:30	64.6	24.6%	215.8%	1.86	787
	agg	27:20	60.7	24.1%	202.6%	0.50	2904
no traffic	static	0:06	13.5	23.9%	23.9%	0.30	4075

scenario	algo	Prepro		Error			Time-Queries	
		time [h:m]	space [B/n]	error -rate	max rel.	max abs.[s]	time [ms]	spd up
Monday	heu	3:30	138.2	0.46%	0.54%	39.3	0.69	2 253
midweek	heu	3:26	137.2	0.82%	0.61%	48.3	0.69	2 164
Friday	heu	3:14	125.2	0.50%	0.50%	50.3	0.64	2 358
Saturday	heu	2:13	80.4	0.18%	0.23%	16.9	0.51	2 887
Sunday	heu	1:48	58.8	0.09%	0.36%	14.9	0.46	3 163
no	static	0:06	13.5	0.00%	0.00%	0.0	0.30	4 075

Beobachtung:

- Fehler sehr gering
- hoher Speicherverbrauch

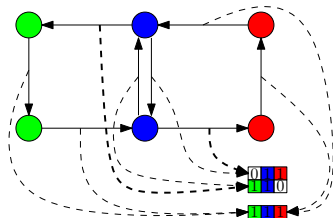
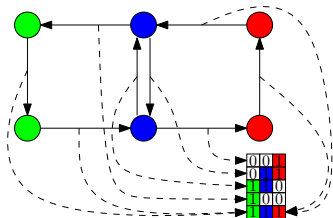
traffic	var.	Time-Queries		Profile-Queries			
		#del. mins	time [ms]	#del. mins	#re- ins.	time [ms]	profile /time
Monday	eco	19 136	24.55	19 768	402	51 122	2 082.6
	heu	810	0.69	1 071	24	1 008	1 460.9
midweek	eco	19 425	25.06	20 538	432	60 147	2 400.3
	heu	818	0.69	1 100	27	1 075	1 548.4
Friday	eco	17 412	22.07	19 530	346	52 780	2 391.9
	heu	769	0.64	1 049	21	832	1 293.2
Saturday	eco	5 284	5.34	5 495	44	3 330	624.0
	agg	721	0.58	865	9	134	232.5
	heu	666	0.51	798	8	98	191.9
Sunday	eco	2 142	1.86	2 294	12	536	288.1
	agg	670	0.50	781	5	57	113.5
	heu	635	0.46	738	5	45	97.9

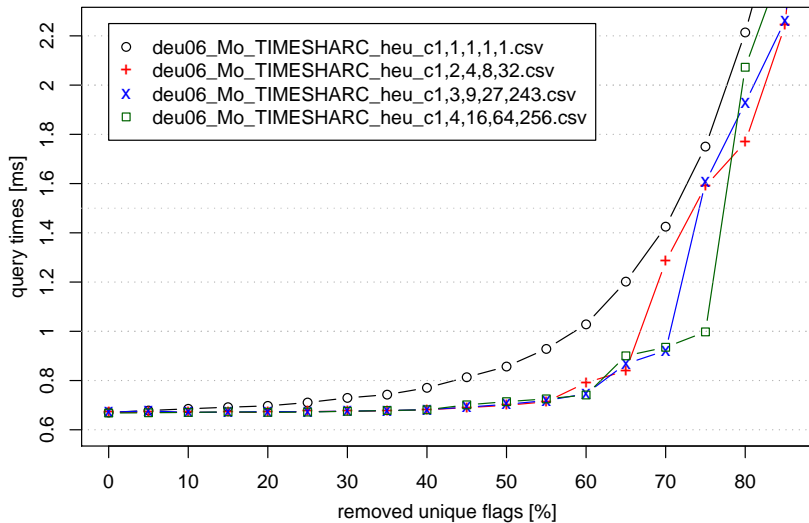
Gründe für Overhead:

- Regionsinformation
- Flaggen speichern
- Shortcuts (Einträge im Kantenarray)
- zusätzliche Interpolationspunkte

1. Arc-Flag Compression

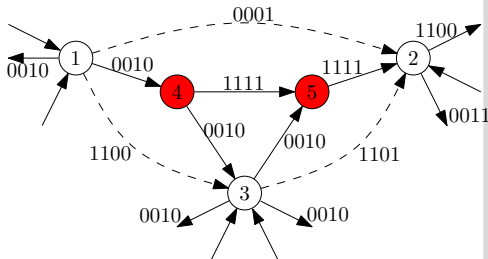
- Beobachtung: Es gibt weniger verschiedene Flaggen-Vektoren als Kanten
- Speichere Flaggen in externer Tabelle
- Reduziere die Anzahl verschiedener Flaggen (Bit-Flipping 0 \rightsquigarrow 1)
- Weniger Platzverbrauch, dafür größerer Suchraum





Beobachtung:

- Shortcut entspricht einem Pfad
- manche Shortcuts erscheinen unwichtig

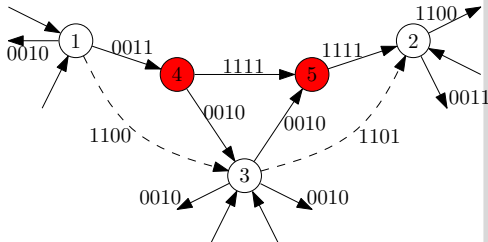


Idee:

- entferne (manche) Shortcuts nach Vorberechnung
- vererbe Flaggen an erste Kante des Pfades
- welche sind wichtig?
- Außerdem: entferne Interpolationspunkte und entpacke on-the-fly

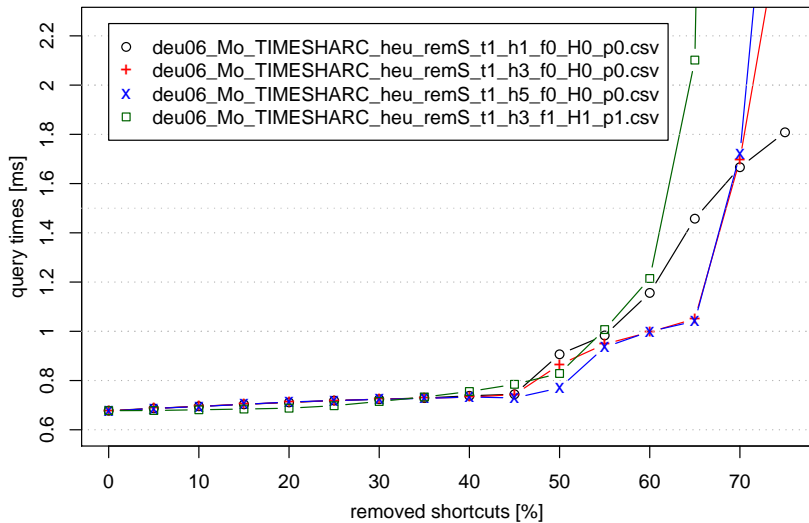
Beobachtung:

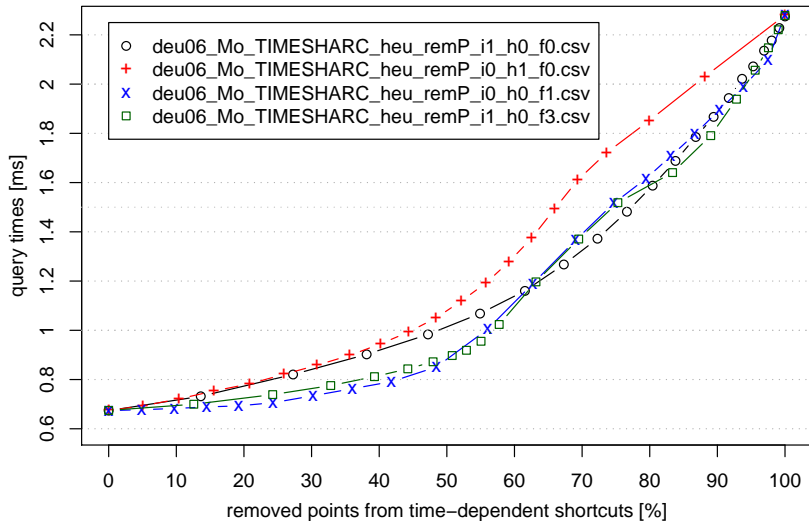
- Shortcut entspricht einem Pfad
- manche Shortcuts erscheinen unwichtig



Idee:

- entferne (manche) Shortcuts nach Vorberechnung
- vererbe Flaggen an erste Kante des Pfades
- welche sind wichtig?
- Außerdem: entferne Interpolationspunkte und entpacke on-the-fly





Ergebnis:

Speicherverbrauch kann auf 10–15 Bytes pro Knoten gedrückt werden

Vorbereitung:

- benutze gleiche Knotenordnung
- kontrahiere zeitabhängig
- erzeugt Suchgraphen $G' = (V, \uparrow E \cup \downarrow E)$

Anfrage

- Rückwärts aufwärts mittels min-max Suche
- markiere alle Kanten (u, v) aus $\downarrow E$ mit $\underline{d(u, v)} + \underline{d(v, t)} \leq \overline{d(u, v)}$
- diese Menge sei $\downarrow E'$
- zeitabhängige Vorwärtssuche in $(V, \uparrow E \cup \downarrow E')$

Vorbereitung:

- benutze gleiche Knotenordnung
- kontrahiere zeitabhängig
- erzeugt Suchgraphen $G' = (V, \uparrow E \cup \downarrow E)$

Anfrage

- Rückwärts aufwärts mittels min-max Suche
- markiere alle Kanten (u, v) aus $\downarrow E$ mit $\underline{d(u, v)} + \underline{d(v, t)} \leq \overline{d(u, v)}$
- diese Menge sei $\downarrow E'$
- zeitabhängige Vorwärtssuche in $(V, \uparrow E \cup \downarrow E')$

input	type of ordering	Contr.			Queries	
		ordering [h:m]	const. [h:m]	space [B/n]	time [ms]	speed up
Monday	static min	0:05	0:20	1 035	1.19	1 240
	timed	1:47	0:14	750	1.19	1 244
midweek	static min	0:05	0:20	1 029	1.22	1 212
	timed	1:48	0:14	743	1.19	1 242
Friday	static min	0:05	0:16	856	1.11	1 381
	timed	1:30	0:12	620	1.13	1 362
Saturday	static min	0:05	0:08	391	0.81	1 763
	timed	0:52	0:08	282	1.09	1 313
Sunday	static min	0:05	0:06	248	0.71	1 980
	timed	0:38	0:07	177	1.07	1 321

Literatur (Zeitabhängige Beschleunigungstechniken):

- Daniel Delling:
Engineering and Augmenting Route Planning Algorithms
Ph.D. Thesis, Universität Karlsruhe (TH), 2009.
- **Gernot Veit Batz, Daniel Delling, Peter Sanders, Christian Vetter**
Time-Dependent Contraction Hierarchies
In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, April 2009.
- **Edith Brunel, Daniel Delling, Andreas Gemsa, Dorothea Wagner**
Space-Efficient SHARC-Routing
In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, May 2010.