Karlsruher Institut für Technologie

Algorithmen für Routenplanung

2. Vorlesung, Sommersemester 2023

Michael Zündorf | 24. April 2023

Wdh.: Komplexität von Dijkstra

Dijkstra($G = (V, E)$, s)

```
forall nodes v ∈ V do
    d[v] = ∞, p[v] = NULL                          // n Mal
d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
while !Q.empty() do
    u ← Q.deleteMin()                              // n Mal
    forall edges e = (u, v) ∈ E do
        if d[u] + len(e) < d[v] then
            d[v] ← d[u] + len(e)
            p[v] ← u
            if v ∈ Q then Q.decreaseKey(v, d[v])   // m Mal
            else Q.insert(v, d[v])                 // n Mal
```

$$T_{\text{Dijkstra}} = T_{init} + n \cdot T_{deleteMin} + m \cdot T_{decreaseKey} + n \cdot T_{insert}$$

Wdh.: Laufzeit Dijkstra

$$T_{\text{Dijkstra}} = T_{init} + n \cdot T_{deleteMin} + m \cdot T_{decreaseKey} + n \cdot T_{insert}$$

Operation	Liste (worst-case)	Binary Heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Init	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log k)$	$\mathcal{O}(\log k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\log k)$	$\Theta(1)$
DeleteMin	$\Theta(n)$	$\Theta(\log k)$	$\Theta(\log k)$	$\mathcal{O}(\log k)$
Union	$\Theta(1)$	$\Theta(k)$	$\mathcal{O}(\log k)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\log k)$	$\Theta(\log k)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\log k)$	$\Theta(\log k)$	$\mathcal{O}(\log k)$
Dijkstra	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}(m + n\log n)$
$m \in \mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Wdh.: Laufzeit Dijkstra Experimente

k-ary Heap: Baum mit (max.) k Kindern je Vorgängerknoten

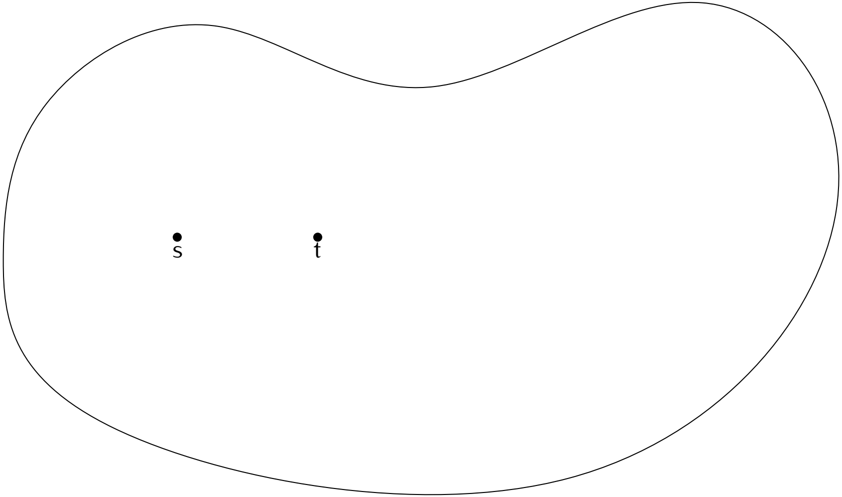
k	Query [sec]
2	1.834
3	1.595
4	1.507
5	1.525
8	1.561

Graph: 18M Knoten, 42M Kanten

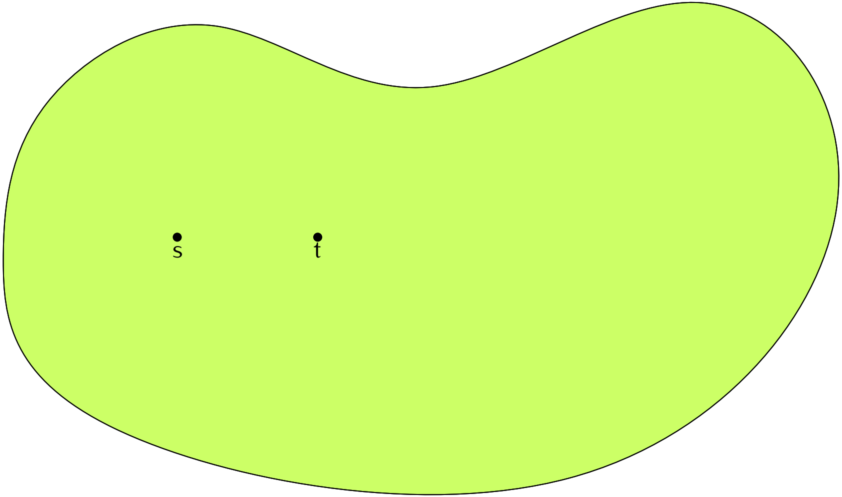
Dijkstra:	$\approx 1.5\,\text{s}$	$\Rightarrow$ nicht interaktiv
$n + m$ CPU clock cycles:	$\approx 20\,\text{ms}$	$\Rightarrow$ viel schneller
BFS:	$\approx 1.2\,\text{s}$	$\Rightarrow$ an der Queue liegt's nicht

Performanz von Graphsuchen ist speicher-begrenzt

Schematischer Suchraum, Dijkstra



Schematischer Suchraum, Dijkstra



Abbruch

Beobachtung

- Dijkstra's Algorithmus durchsucht den ganzen Graphen
- Viel unnütze Information, vor allem wenn s und t nahe beinander

Abbruch

Beobachtung

- Dijkstra's Algorithmus durchsucht den ganzen Graphen
- Viel unnütze Information, vor allem wenn s und t nahe beinander

Idee

- stoppe die Anfrage, sobald t aus der Queue entfernt wurde
- **Suchraum**: Menge der abgearbeiteten Knoten

Abbruch

Beobachtung

- Dijkstra's Algorithmus durchsucht den ganzen Graphen
- Viel unnütze Information, vor allem wenn s und t nahe beinander

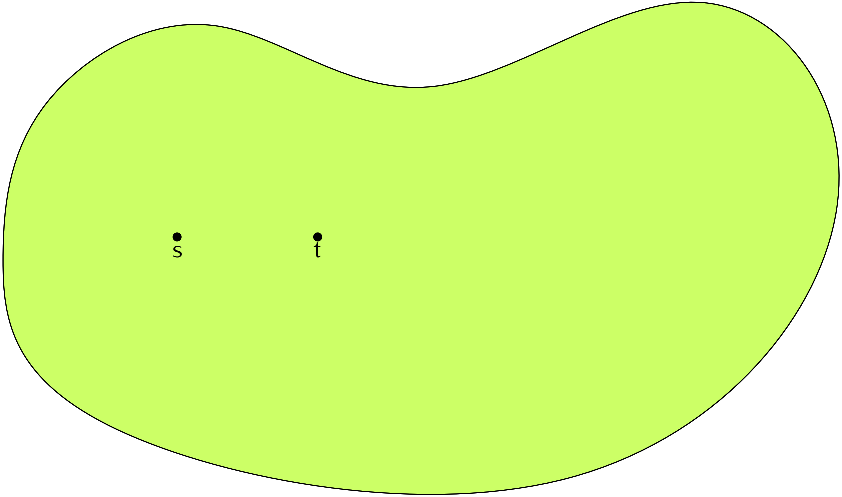
Idee

- stoppe die Anfrage, sobald t aus der Queue entfernt wurde
- **Suchraum**: Menge der abgearbeiteten Knoten

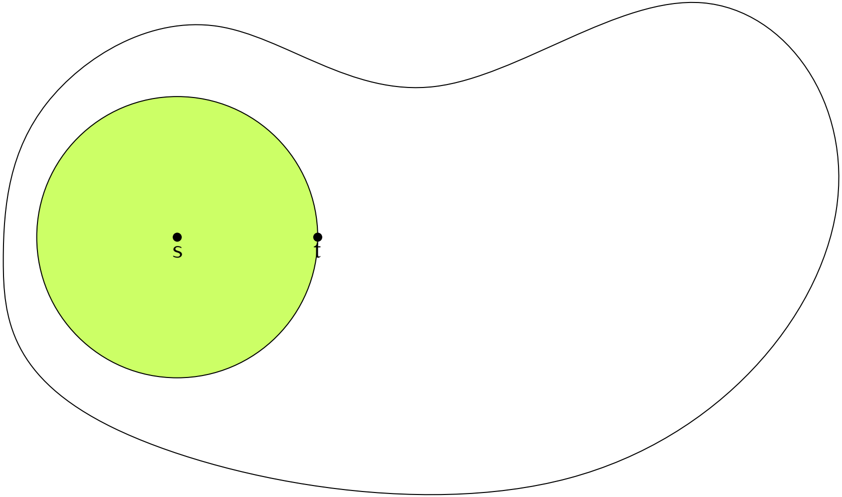
Korrektheit

- Wert $d[v]$ ändert sich nicht mehr, sobald v abgearbeitet wurde
- Korrektheit des Vorgehens bleibt also erhalten
- Reduziert durchschnittlichen Suchraum von n auf $\approx n/2$

Schematischer Suchraum, Dijkstra



Schematischer Suchraum, Dijkstra



Dijkstra mit Abbruchkriterium

Dijkstra($G = (V, E)$, s, t)

```
forall nodes v ∈ V do
    d[v] = ∞, p[v] = NULL                          // distances, parents
d[s] = 0, Q.clear(), Q.add(s, 0)                   // container
while !Q.empty() do
    u ← Q.deleteMin()                              // settling node u
    if u = t then return
    forall edges e = (u, v) ∈ E do
        if d[u] + len(e) < d[v] then
            d[v] ← d[u] + len(e)
            p[v] ← u
            if v ∈ Q then Q.decreaseKey(v, d[v])
            else Q.insert(v, d[v])
```

Frage

- Häufig werden viele Anfragen auf gleichem Netzwerk gestellt.
- Wo könnte ein Problem bzgl. der Laufzeit liegen?

Dijkstra mit Abbruchkriterium

Dijkstra($G = (V, E)$, s, t)

forall *nodes* $v \in V$ **do**
    $d[v] = \infty$, $p[v] = \mathrm{NULL}$ `// distances, parents`
$d[s] = 0$, $Q.clear()$, $Q.add(s, 0)$ `// container`
while $!Q.empty()$ **do**
    $u \leftarrow Q.deleteMin()$ `// settling node u`
    break if $u = t$
    forall *edges* $e = (u, v) \in E$ **do**
        `// relaxing edges`
        if $d[u] + \mathrm{len}(e) < d[v]$ **then**
            $d[v] \leftarrow d[u] + \mathrm{len}(e)$
            $p[v] \leftarrow u$
            if $v \in Q$ **then** $Q.decreaseKey(v, d[v])$
            else $Q.insert(v, d[v])$

Dijkstra mit Abbruchkriterium

Dijkstra($G = (V, E)$, s, t)

```
forall nodes v ∈ V do
    d[v] = ∞, p[v] = NULL                      // distances, parents
d[s] = 0, Q.clear(), Q.add(s, 0)               // container
while !Q.empty() do
    u ← Q.deleteMin()                          // settling node u
    break if u = t
    forall edges e = (u, v) ∈ E do
        // relaxing edges
        if d[u] + len(e) < d[v] then
            d[v] ← d[u] + len(e)
            p[v] ← u
            if v ∈ Q then Q.decreaseKey(v, d[v])
            else Q.insert(v, d[v])
```

Viele Anfragen

Problem

- Häufig viele Anfragen auf gleichem Graphen
- Die Initialisierung muss immer für alle Knoten neu ausgeführt werden

Viele Anfragen

Problem

- Häufig viele Anfragen auf gleichem Graphen
- Die Initialisierung muss immer für alle Knoten neu ausgeführt werden

Idee

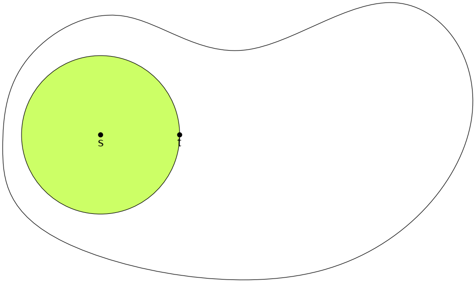
- Speiche zusätzlichen „Timestamp“ $run[v]$ für jeden Knoten
- Benutze Zähler *count*
- Damit kann abgefragt werden, ob ein Knoten im aktuellen Lauf schon besucht wurde.

Dijkstra mit Timestamps

```
count ← count + 1                                        // Überlauf
d[s] = 0, Q.clear(), Q.add(s, 0)
while !Q.empty() do
    u ← Q.deleteMin()
    if u = t then return
    forall edges e = (u, v) ∈ E do
        if run[v] ≠ count then
            d[v] ← d[u] + len(e)
            Q.insert(v, d[v])
            run[v] ← count
        else if d[u] + len(e) < d[v] then
            d[v] ← d[u] + len(e)
            Q.decreaseKey(v, d[v])
```

Bidirektionale Suche

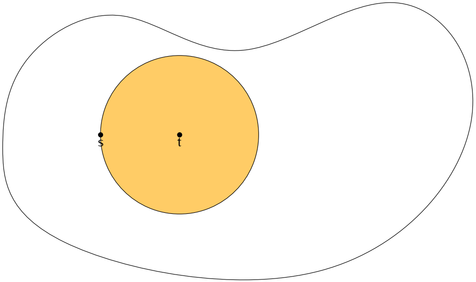
Bidirektionale Suche



Beobachtung: Ein kürzester s-t-Weg lässt sich finden durch

- Normaler Dijkstra (Vorwärtssuche) von s
- Dijkstra auf Graph mit umgedrehten Kantenrichtungen (Rückwärtssuche) von t

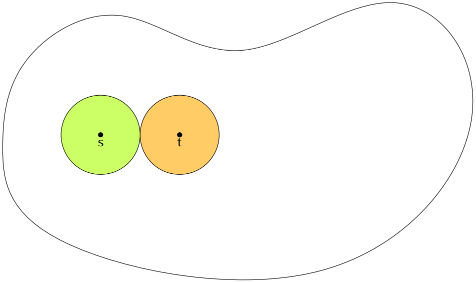
Bidirektionale Suche



Beobachtung: Ein kürzester s-t-Weg lässt sich finden durch

- Normaler Dijkstra (Vorwärtssuche) von s
- Dijkstra auf Graph mit umgedrehten Kantenrichtungen (Rückwärtssuche) von t

Bidirektionale Suche



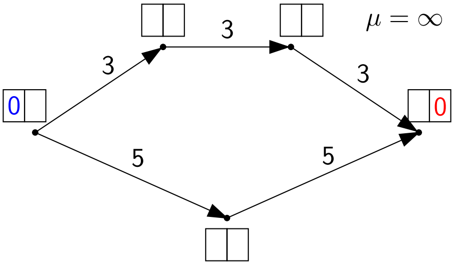
Idee: Kombiniere beide Suchen

- „Gleichzeitig“ Vor- und Rückwärtssuche
- Abbruch wenn beide Suchen „weit genug fortgeschritten“
- Weg dann zusammensetzen

Bidirektionale Suche

Anfrage:

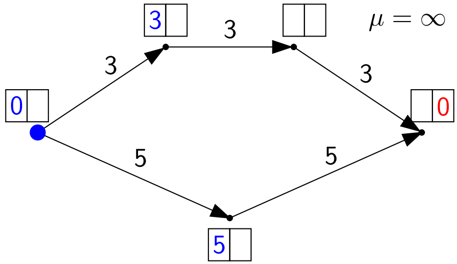
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

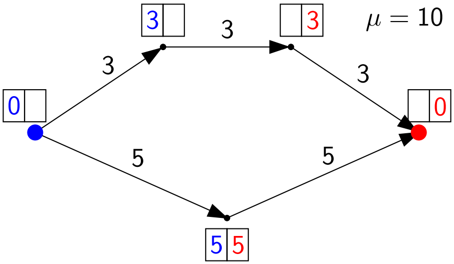
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

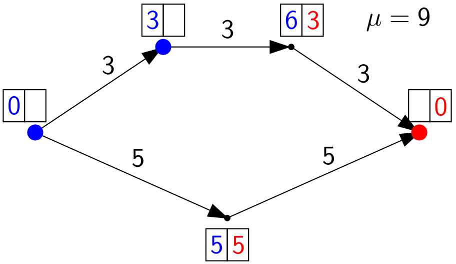
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

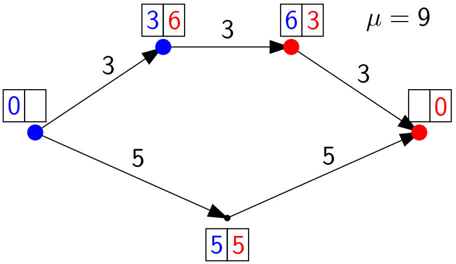
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

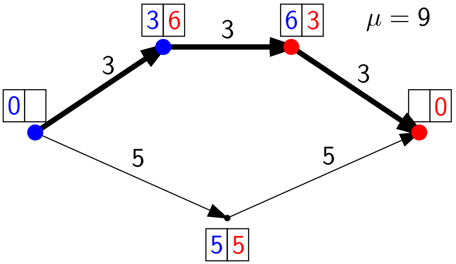
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

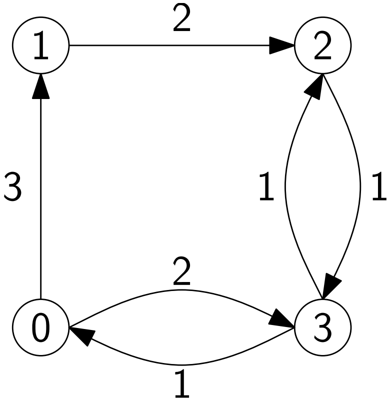
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Graph-Datenstruktur

Problem:

- ein- und ausgehende Kanten-Inzidenz benötigt
- Graph (fast) ungerichtet



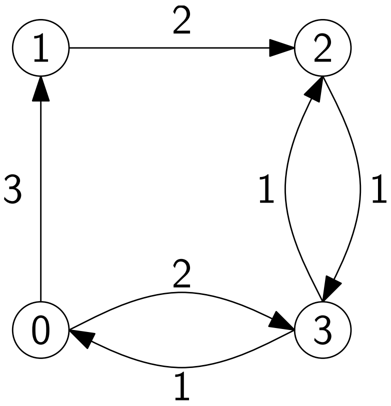
Graph-Datenstruktur

Problem:

- ein- und ausgehende Kanten-Inzidenz benötigt
- Graph (fast) ungerichtet

`firstEdge`	0	3	5	7	10

`targetNode`	1	3	3	0	2	1	3	0	0	2
`weight`	3	2	1	3	2	2	1	1	2	1
`isIncoming`	—	—	$\surd$	$\surd$	—	$\surd$	$\surd$	—	$\surd$	$\surd$
`isOutgoing`	$\surd$	$\surd$	—	—	$\surd$	—	$\surd$	$\surd$	—	$\surd$



Bidirektionale Suche - Definitionen

- Input ist Graph $G = (V, E, \text{len})$ und Knoten $s, t \in V$.
- Für Inputgraphen G bezeichne $\overleftarrow{G} := (V, \overleftarrow{E}, \overleftarrow{\text{len}})$ den *umgekehrten Graphen*, d.h.

$$\begin{aligned} \overleftarrow{E} &:= \{(v,u) \in V \times V \mid (u,v) \in E\} \\ \overleftarrow{\text{len}}(u,v) &= \text{len}(v,u) \end{aligned}$$

- Die **Vorwärtssuche** ist Dijkstra's Algo mit Start s auf G
- Die **Rückwärtssuche** ist Dijkstra's Algo mit Start t auf $\overleftarrow{G}$
- Die Queue der Vorwärtssuche ist $\overrightarrow{Q}$
- Die Queue der Rückwärtssuche ist $\overleftarrow{Q}$
- Der Distanzvektor der Vorwärtssuche ist $\overrightarrow{d}[]$
- Der Distanzvektor der Rückwärtssuche ist $\overleftarrow{d}[]$

Bidirektionale Suche - Vorgehen

- Vor- und Rückwärtssuche werden abwechselnd ausgeführt
- Es wird zusätzlich die vorläufige Distanz

$$\mu := \min_{v \in V}(\overrightarrow{d}[v] + \overleftarrow{d}[v])$$

  berechnet, sowie der (vorläufige) Mittelknoten m.
- Dazu wird bei der Relaxierung von Kante (u, v) zusätzlich

$$\mu := \min\{\mu, \overrightarrow{d}[v] + \overleftarrow{d}[v]\}$$

  ausgeführt und m ggfs. aktualisiert. (Initial ist $\mu = \infty$).
- Nach Terminierung beinhaltet μ die Distanz $d(s, t)$,
  $P = s, \ldots, m, \ldots, t$ ist kürzester Pfad.

Bidirektionale Suche - Vorgehen

Karlsruher Institut für Technologie

- Vor- und Rückwärtssuche werden abwechselnd ausgeführt
- Es wird zusätzlich die vorläufige Distanz

$$\mu := \min_{v \in V}(\overrightarrow{d}[v] + \overleftarrow{d}[v])$$

  berechnet, sowie der (vorläufige) Mittelknoten m.
- Dazu wird bei der Relaxierung von Kante (u, v) zusätzlich

$$\mu := \min\{\mu, \overrightarrow{d}[v] + \overleftarrow{d}[v]\}$$

  ausgeführt und m ggfs. aktualisiert. (Initial ist $\mu = \infty$).
- Nach Terminierung beinhaltet μ die Distanz $d(s, t)$,
  $P = s, \dots, m, \dots, t$ ist kürzester Pfad.

Was sind gute Abbruchstrategien?

Abbruchstrategie (1)

Abbruchstrategie (1)

Abbruch, sobald ein Knoten v^* existiert, der von beiden Suchen abgearbeitet wurde.

Abbruchstrategie (1)

Abbruchstrategie (1)

Abbruch, sobald ein Knoten v^* existiert, der von beiden Suchen abgearbeitet wurde.

Abbruchstrategie (1) berechnet $\mathbf{d(s,t)}$ korrekt. Beweisskizze:

- O.B.d.A: Sei $d(s,t) < \infty$ (andernfalls klar)
- Klar: $\overrightarrow{d}[v] + \overleftarrow{d}[v] \geq \mathrm{dist}(s,t)$ für alle $v \in V$
- Seien $\overrightarrow{S}, \overleftarrow{S}$ die abgearbeiteten Knoten von Vor- und Rückwärtssuche nach Terminierung
- Sei $P = (v_1, \dots v_k)$ ein kürzester s-t-Weg. Wir zeigen:

$$\{v_1, \dots v_k\} \subseteq \overrightarrow{S} \cup \overleftarrow{S} \quad \text{oder} \quad \mathrm{len}(P) = \mathrm{dist}(s, v^*) + \mathrm{dist}(v^*, t)$$

Abbruchstrategie (1)

Abbruchstrategie (1)

Abbruch, sobald ein Knoten v^* existiert, der von beiden Suchen abgearbeitet wurde.

- Sei $P = (v_1, \dots v_k)$ ein kürzester s-t-Weg. Wir zeigen:

$$\{v_1, \dots v_k\} \subseteq \overrightarrow{S} \cup \overleftarrow{S} \quad \text{oder} \quad \text{len}(P) = \text{dist}(s, v^*) + \text{dist}(v^*, t)$$

- Angenommen es gibt $v_i \notin \overrightarrow{S} \cup \overleftarrow{S}$. Dann gilt:

$$\begin{aligned} \text{dist}(s, v_i) &\geq \text{dist}(s, v^*) \\ \text{dist}(v_i, t) &\geq \text{dist}(v^*, t) \end{aligned}$$

- Woraus folgt:

$$\text{len}(P) = \text{dist}(s, v_i) + \text{dist}(v_i, t) \geq \text{dist}(s, v^*) + \text{dist}(v^*, t)$$

Abbruchstrategie (2)

Abbruchstrategie (2)

Abbruch, sobald $\mu \leq \mathrm{minKey}(\overrightarrow{Q}) + \mathrm{minKey}(\overleftarrow{Q})$

Abbruchstrategie (2)

Abbruchstrategie (2)

Abbruch, sobald $\mu \leq \text{minKey}(\overrightarrow{Q}) + \text{minKey}(\overleftarrow{Q})$

Abbruchstrategie (2) berechnet $\mathbf{d(s,t)}$ korrekt. Beweisskizze:

- O.B.d.A: Sei $d(s,t) < \infty$ (andernfalls klar)
- Klar: $\overrightarrow{d}[v] + \overleftarrow{d}[v] \geq \text{dist}(s,t)$ für alle $v \in V$

Abbruchstrategie (2)

Abbruchstrategie (2)

Abbruch, sobald $\mu \leq \text{minKey}(\overrightarrow{Q}) + \text{minKey}(\overleftarrow{Q})$

Annahme: $\mu > d(s,t)$ nach Terminierung

- Dann gibt es einen s-t Pfad P der echt kürzer als μ ist
- Auf P gibt es eine Kante (u,v) mit:

$$d(s,u) \leq minKey(\overrightarrow{Q}) \quad \text{und} \quad d(v,t) \leq minKey(\overleftarrow{Q}) \quad \text{und}$$

$$d(s,u) < minKey(\overrightarrow{Q}) \quad \text{oder} \quad d(v,t) < minKey(\overleftarrow{Q}).$$

- Es muss mindestens u oder v schon abgearbeitet worden sein
- Beim relaxieren von (u,v) wäre:
  - P entdeckt worden
  - μ aktualisiert worden

Damit ist $\mu \leq d(s,t)$. Widerspruch!

Wechselstrategien

Frage: Was sind mögliche Wechselstrategien?

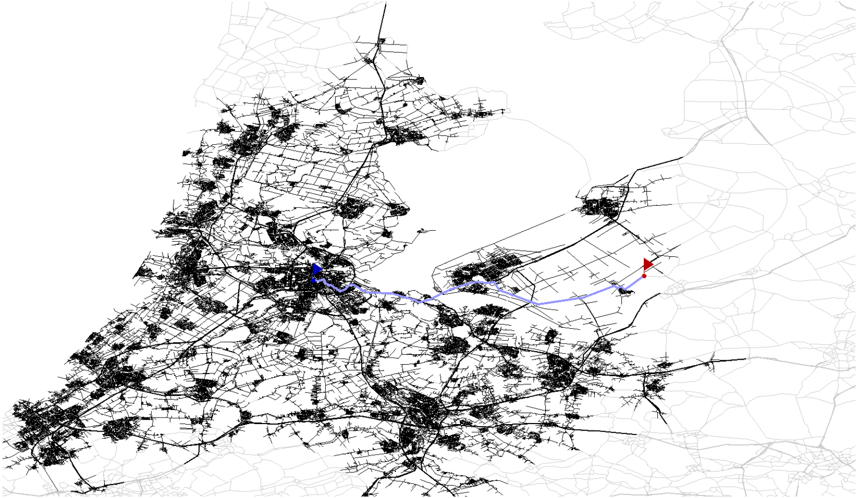
Wechselstrategien

Karlsruher Institut für Technologie

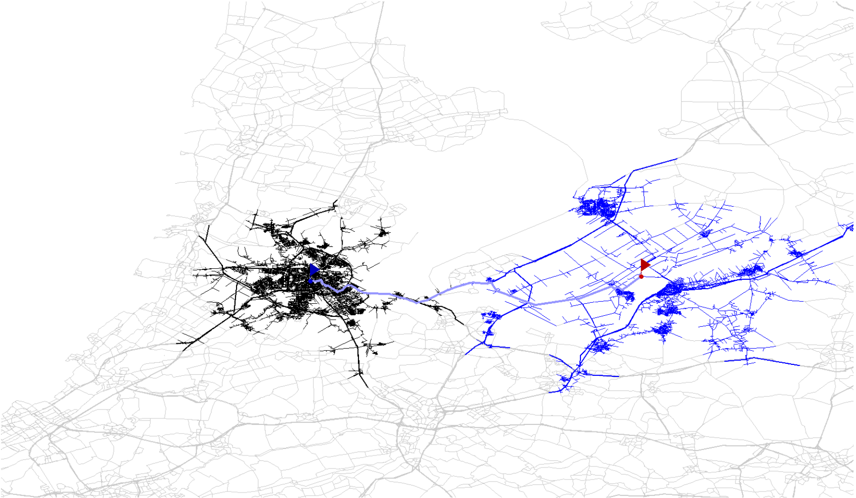
Mögliche Wechselstrategien

- Prinzipiell jede Wechselstrategie möglich
- Wechsle nach jedem Schritt zur entgegengesetzten Suche
- Wechsel zu Suche mit der weniger Elementen in der Queue
- Wechsel zu Suche mit dem kleineren minimalen Queueelement
- Oder: Parallele Ausführung auf zwei Kernen

Beispiel



Beispiel



Bemerkungen

Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius r

$\Rightarrow$ Speedup bzgl. Suchraum (ca.):

$$\frac{\text{Dijkstra}}{\text{Bidir. Suche}} \approx \frac{\pi r^2}{2 \cdot \pi \left(\frac{r}{2}\right)^2} = 2$$

- Führe Suchen *parallel* aus

$\Rightarrow$ Gesamtspeedup ca. 4

Bemerkungen

Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius r

$\Rightarrow$ Speedup bzgl. Suchraum (ca.):

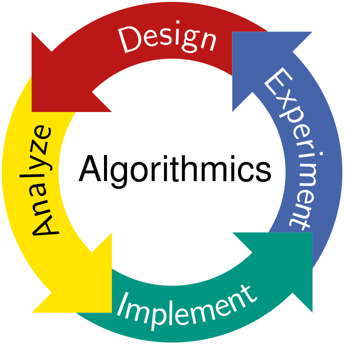
$$\frac{\text{Dijkstra}}{\text{Bidir. Suche}} \approx \frac{\pi r^2}{2 \cdot \pi \left(\frac{r}{2}\right)^2} = 2$$

- Führe Suchen *parallel* aus

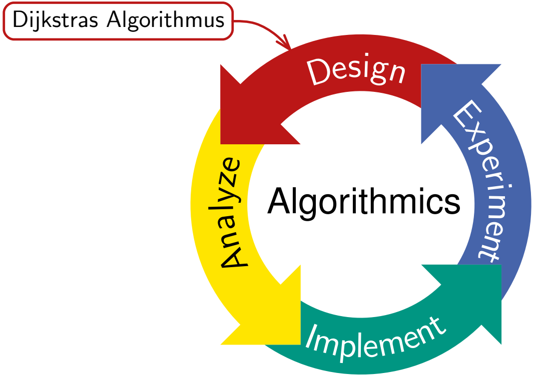
$\Rightarrow$ Gesamtspeedup ca. 4

Wichtiger Bestandteil vieler effizienter Techniken!

Algorithm Engineering

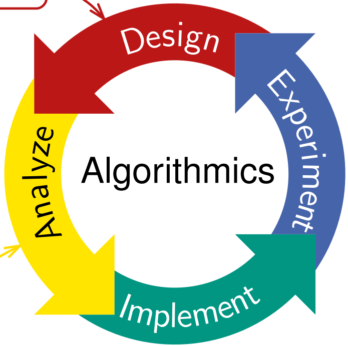


Algorithm Engineering



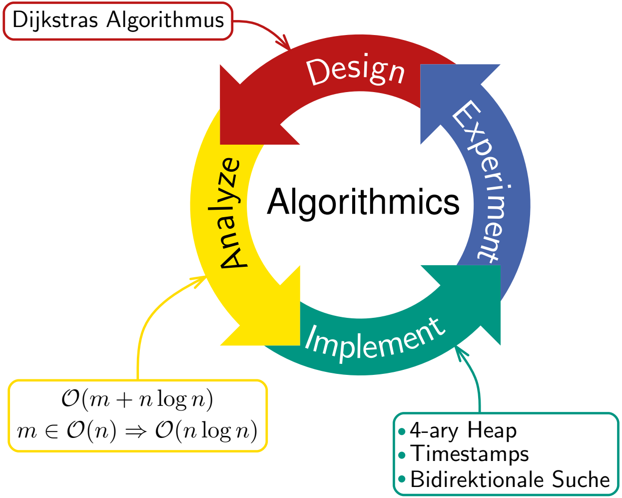
Algorithm Engineering

Dijkstras Algorithmus

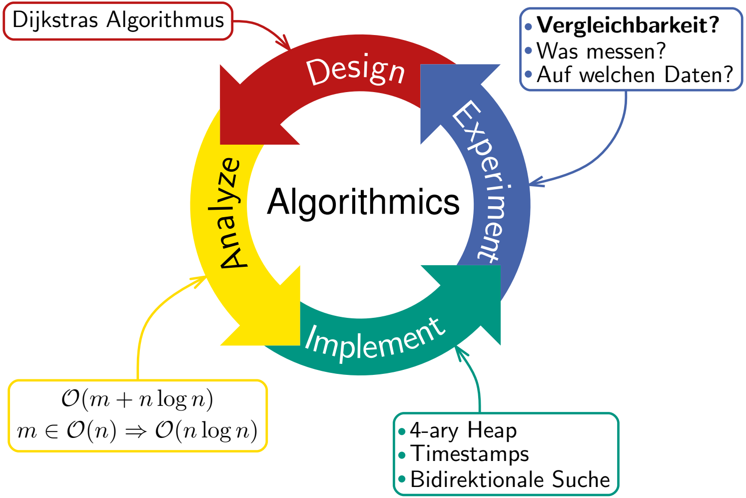


$$\mathcal{O}(m + n \log n)$$
$$m \in \mathcal{O}(n) \Rightarrow \mathcal{O}(n \log n)$$

Algorithm Engineering



Algorithm Engineering



Experiment – Dijkstra-Rank

Problem:

- Zufallsanfragen geben wenig Informationen
- Wie ist die Varianz?
- Werden nahe oder ferne Anfragen beschleunigt?

Experiment – Dijkstra-Rank

Problem:

- Zufallsanfragen geben wenig Informationen
- Wie ist die Varianz?
- Werden nahe oder ferne Anfragen beschleunigt?

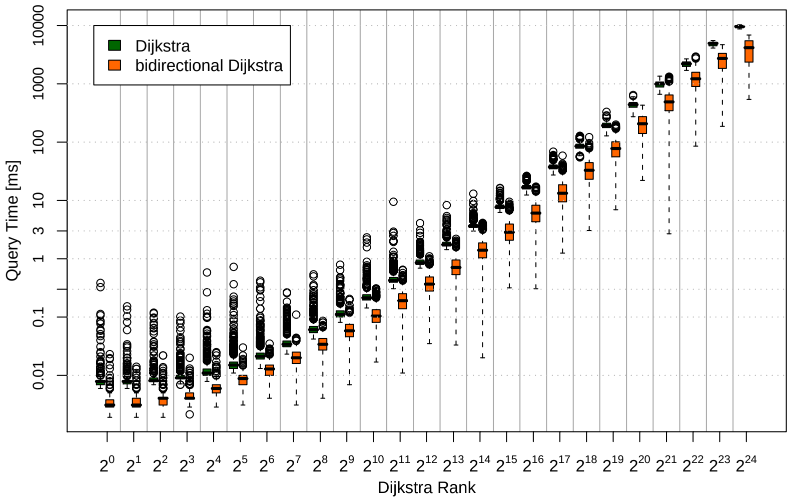
Idee:

- Dijkstra definiert für Startknoten s Ordnung auf den Knoten
- Dijkstra-Rang $r_s(u)$ eines Knoten u für gegebenes s
- Wähle 1000 zufällige Startknoten, analysiere jeweils den Suchraum

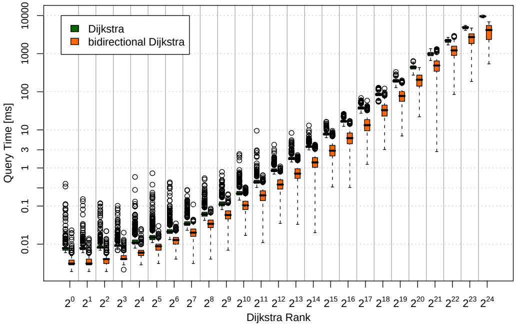
$$\text{Zieknoten } t := \text{ Knoten mit Rang } 2^1, \ldots, 2^{\log n}$$

- Zeichne Plot (x-Achse = Rang, y-Achse = Laufzeit)

Dijkstra-Rank – Bidirektionale Suche



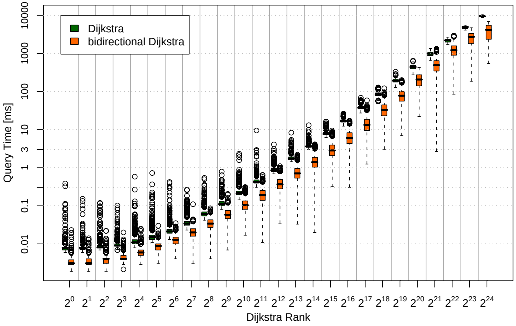
Dijkstra-Rank – Bidirektionale Suche



Boxplots

- Median
- Boxen: 25%-75% Perzentile
- Whiskers: 1.5facher Interquartilsabstand (meistens)
- Ausreißer

Dijkstra-Rank – Bidirektionale Suche



- Ausreißer bei nahen Anfragen (vor allem unidirektional)
- Beschleunigung unabhängig vom Rang (immer ca. Faktor 2)
- Varianz etwas höher als bei unidirektionaler Suche
- Manche Anfragen sehr schnell

Experiment – Datensatz

Welche Graphen für Experimente nutzen:

- Reales Straßennetz
- Synthetische Daten
- Zufallsgraphen

Experiment – Datensatz

Welche Graphen für Experimente nutzen:

- Reales Straßennetz
- Synthetische Daten
- Zufallsgraphen

Zielsetzung:

- Anwendungsnähe
- Effizienz auf echten Daten

Experiment – Datensatz

Welche Graphen für Experimente nutzen:

- Reales Straßennetz
  - Kartendaten: z.B.: OpenStreetMap
  - Benchmark-Instanzen: z.B.: Dimacs Implementation Challenge
- Synthetische Daten
- Zufallsgraphen

Zielsetzung:

- Anwendungsnähe
- Effizienz auf echten Daten

Experiment – Datensatz

Welche Graphen für Experimente nutzen:

- Reales Straßennetz
  - Kartendaten: z.B.: OpenStreetMap
  - Benchmark-Instanzen: z.B.: Dimacs Implementation Challenge
- Synthetische Daten
- Zufallsgraphen

Zielsetzung:

- Anwendungsnähe
- Effizienz auf echten Daten

Welchen Einfluss haben die Daten auf die Experimente?

OpenStreetMap (OSM)

Straßengraphen extrahiert von OSM:

+ Echtes Straßennetz
+ Frei verfügbar
+ Große Datenmengen ($>$ 173 Mio Knoten für Europa)

OpenStreetMap (OSM)

Straßengraphen extrahiert von OSM:

+ Echtes Straßennetz
+ Frei verfügbar
+ Große Datenmengen ($>$ 173 Mio Knoten für Europa)

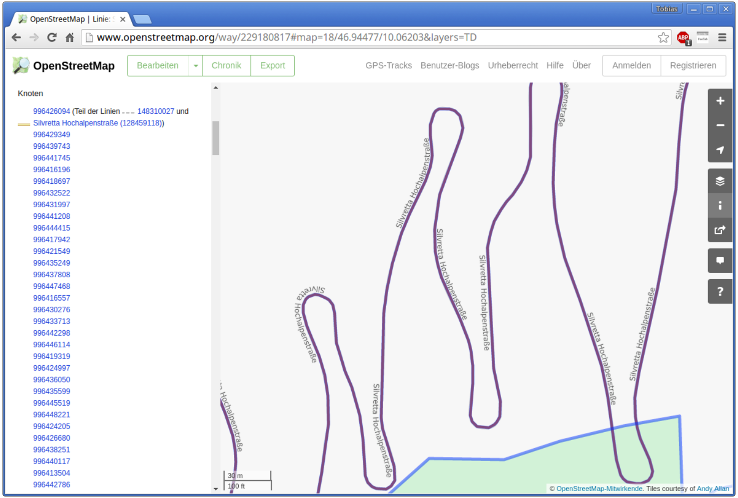
– Zur Darstellung in Karten gedacht
– Viele Freiheiten beim erstellen des Graphen

$\Rightarrow$ Experimente schwer vergleichbar

OpenStreetMap (OSM)

Karlsruher Institut für Technologie

OpenStreetMap (OSM)



OpenStreetMap (OSM)

OSM Graphen:

- Knoten zum Modellieren des Straßenverlaufs
- Knoten irrelevant für das Routing
- Hat großen Einfluss auf Laufzeit & gemessene Beschleunigung

OpenStreetMap (OSM)

OSM Graphen:

- Knoten zum Modellieren des Straßenverlaufs
- Knoten irrelevant für das Routing
- Hat großen Einfluss auf Laufzeit & gemessene Beschleunigung

Wie Graphen vereinheitlichen?

OpenStreetMap (OSM)

Karlsruher Institut für Technologie

OSM Graphen:

- Knoten zum Modellieren des Straßenverlaufs
- Knoten irrelevant für das Routing
- Hat großen Einfluss auf Laufzeit & gemessene Beschleunigung

Wie Graphen vereinheitlichen?

- Grad zwei Knoten löschen?

OpenStreetMap (OSM)

OSM Graphen:

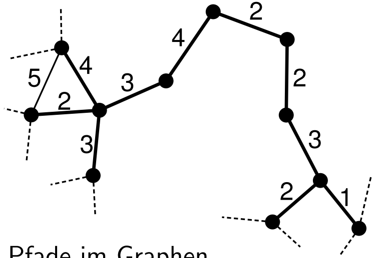
- Knoten zum Modellieren des Straßenverlaufs
- Knoten irrelevant für das Routing
- Hat großen Einfluss auf Laufzeit & gemessene Beschleunigung

Wie Graphen vereinheitlichen?

- Grad zwei Knoten löschen?
- Knoten kontrahieren und Overlay Graphen!

Knotenkontraktionen

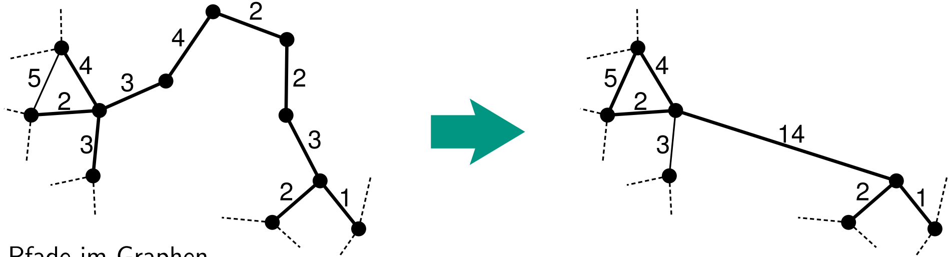
Beispiel: Kontraktion eines Pfades



- Finde Pfade im Graphen

Knotenkontraktionen

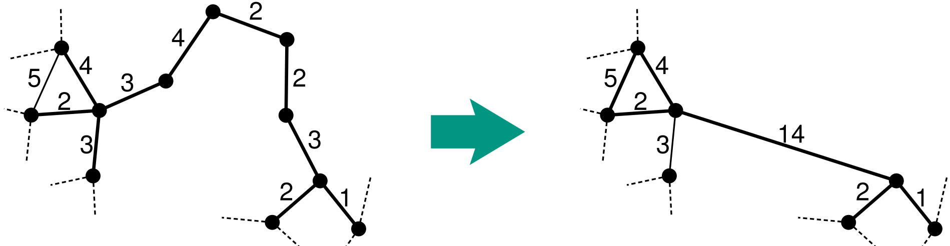
Beispiel: Kontraktion eines Pfades



- Finde Pfade im Graphen
- Ersetze jeden Pfad durch einen Shortcut
- Länge des Shortcut = Länge des Pfades

Knotenkontraktionen

Beispiel: Kontraktion eines Pfades

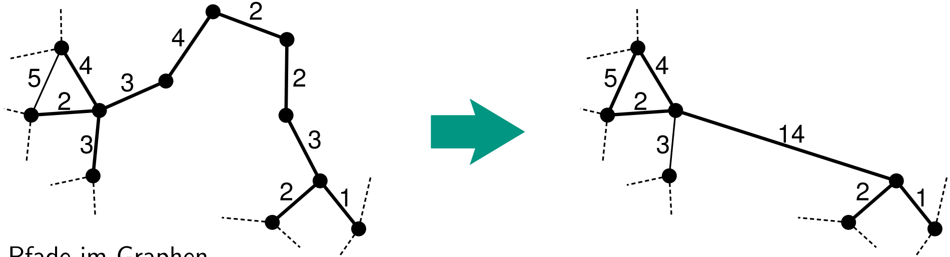


- Finde Pfade im Graphen
- Ersetze jeden Pfad durch einen Shortcut
- Länge des Shortcut = Länge des Pfades

Probleme?

Knotenkontraktionen

Beispiel: Kontraktion eines Pfades



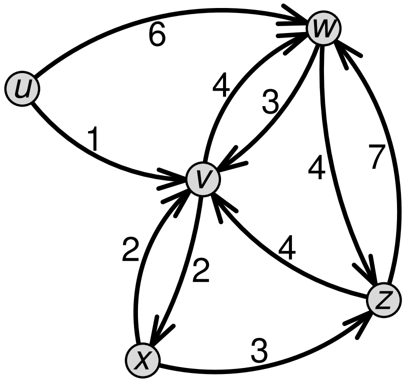
- Finde Pfade im Graphen
- Ersetze jeden Pfad durch einen Shortcut
- Länge des Shortcut = Länge des Pfades

Probleme:

- Gelöschte Knoten können nicht mehr Start/Ziel sein
- Entfernt nur Grad 2 Knoten – Gibt es weitere unnötige Strukturen?

Knotenkontraktionen

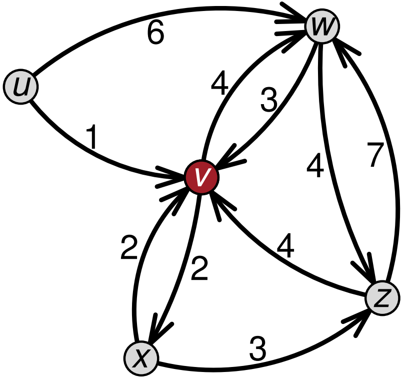
Allgemeine Knotenkontraktion:



Knotenkontraktionen

Allgemeine Knotenkontraktion:

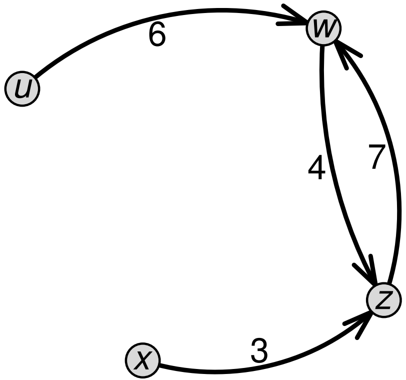
- Knoten v soll kontrahiert werden



Knotenkontraktionen

Allgemeine Knotenkontraktion:

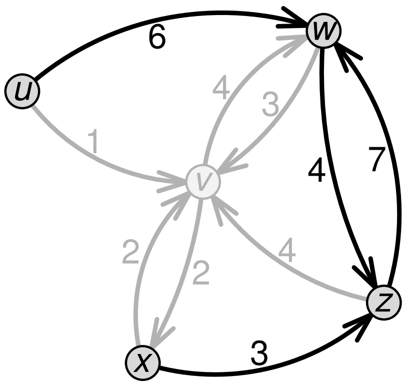
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$



Knotenkontraktionen

Allgemeine Knotenkontraktion:

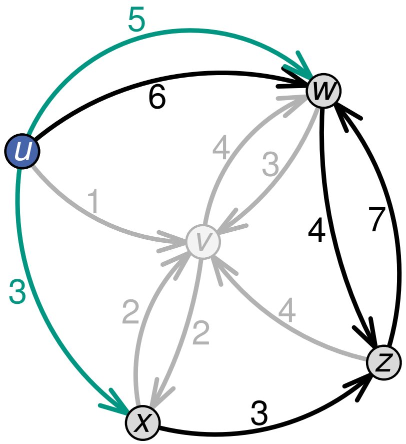
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)



Knotenkontraktionen

Allgemeine Knotenkontraktion:

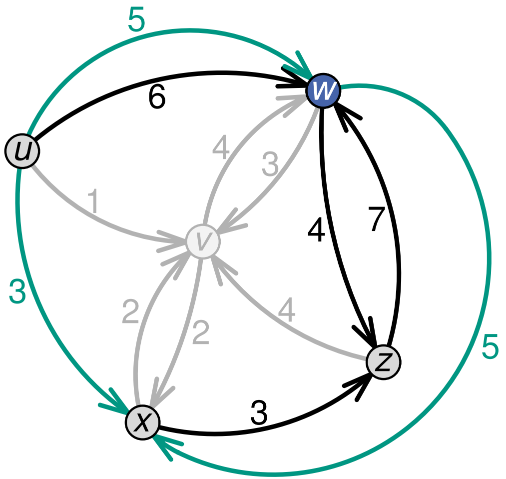
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)



Knotenkontraktionen

Allgemeine Knotenkontraktion:

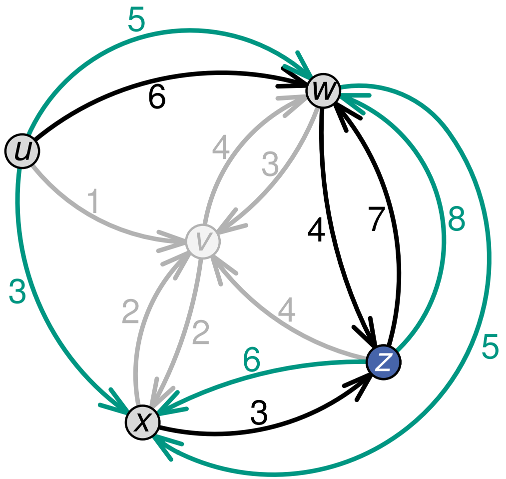
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)



Knotenkontraktionen

Allgemeine Knotenkontraktion:

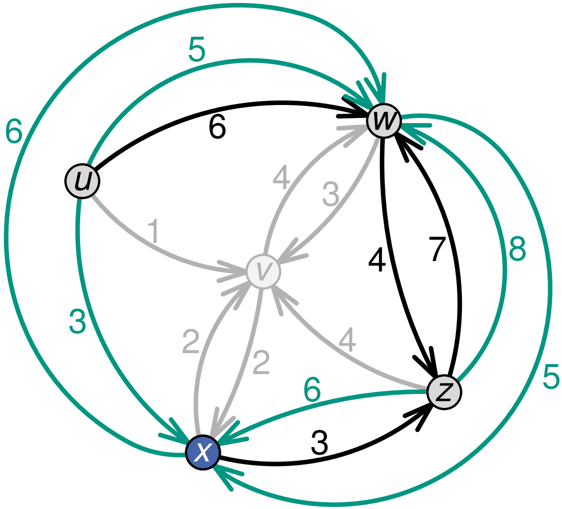
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)



Knotenkontraktionen

Allgemeine Knotenkontraktion:

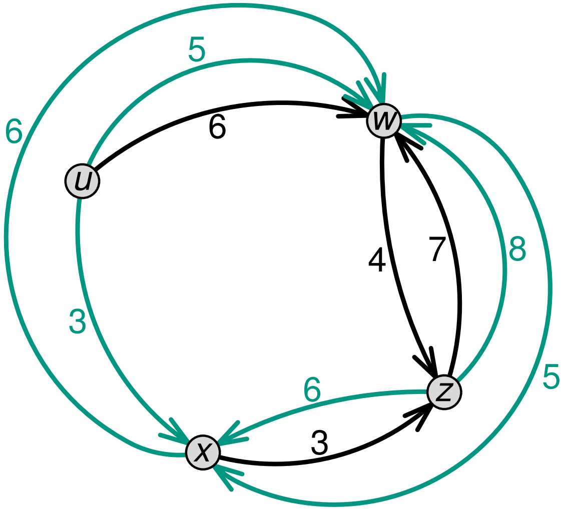
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)



Knotenkontraktionen

Allgemeine Knotenkontraktion:

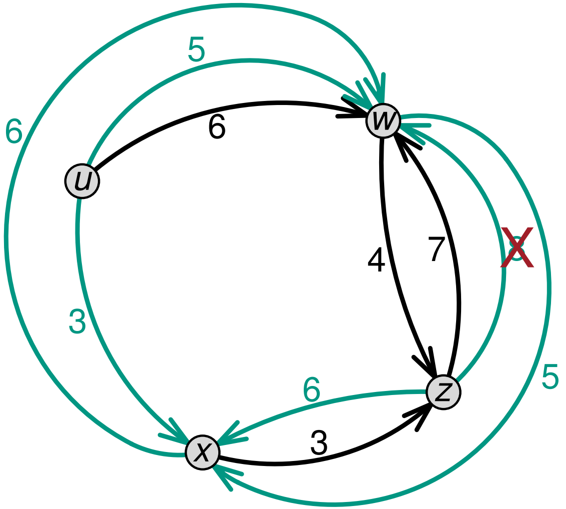
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)
- Lösche Knoten v



Knotenkontraktionen

Allgemeine Knotenkontraktion:

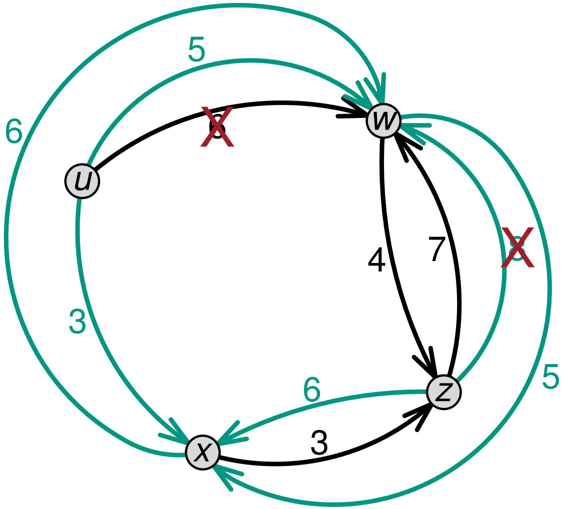
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)
- Lösche Knoten v
- Reduziere Multikanten



Knotenkontraktionen

Allgemeine Knotenkontraktion:

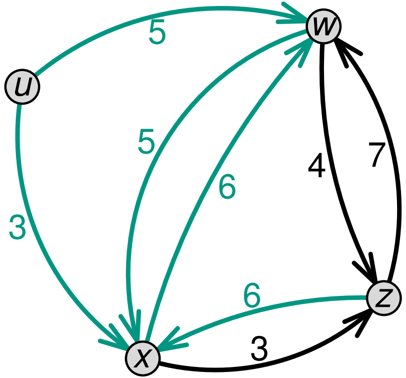
- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)
- Lösche Knoten v
- Reduziere Multikanten



Knotenkontraktionen

Allgemeine Knotenkontraktion:

- Knoten v soll kontrahiert werden
- Erhalte Distanzen in $G \setminus \{v\}$
- $\forall$ Nachbarpaare $\{a, b\}$ von v:
  - Sind (a, v) und (v, b) Kanten?
  - Dann erstelle Shortcut (a, b)
- Lösche Knoten v
- Reduziere Multikanten



Grundstein vieler weiterer Beschleunigungstechniken

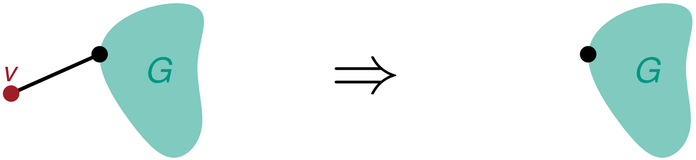
Knotenkontraktionen – TopoCore

Welche Kontraktionen reduzieren die Komplexität des Graphen?

Knotenkontraktionen – TopoCore

Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad 1 Knoten:

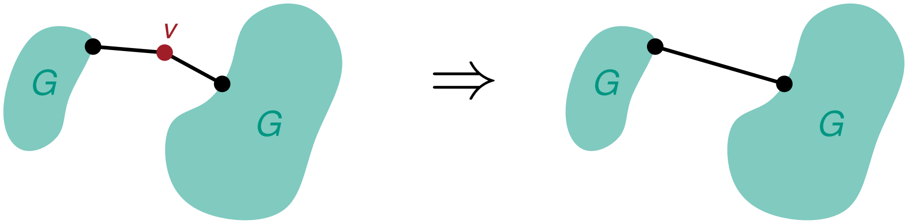


- Sackgasse liegen nie auf kürzesten Wegen
- Kontraktion reduziert Anzahl Kanten
- Entfernt komplette Bäume!

Knotenkontraktionen – TopoCore

Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad 2 Knoten:

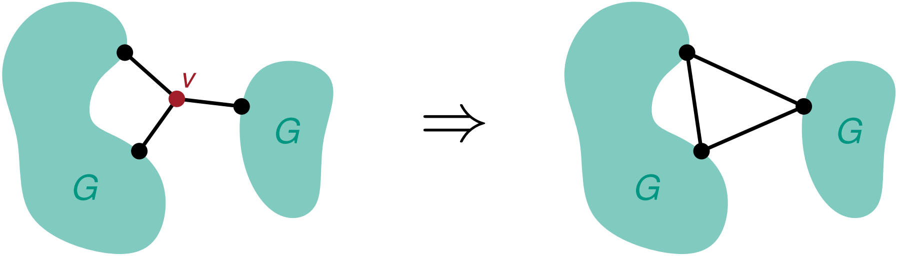


- Nur eine Möglichkeit für Routing auf Pfaden
- Kontraktion reduziert Anzahl Kanten

Knotenkontraktionen – TopoCore

Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad 3 Knoten:

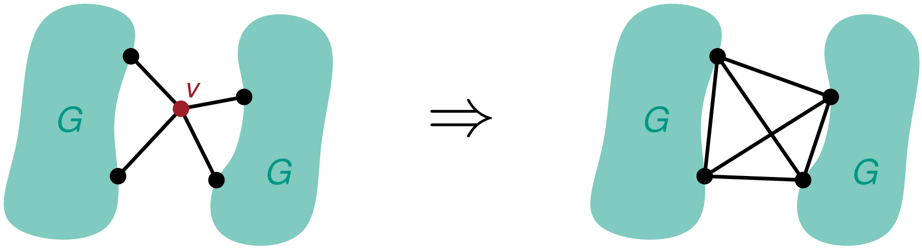


- Wenig Möglichkeiten während Routing (Alle vorberechnen)
- Kontraktion lässt Anzahl Kanten unverändert
  (Anzahl Knoten wird reduziert)

Knotenkontraktionen – TopoCore

Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad ≥ 4 Knoten:



- Struktur nicht mehr ausnutzbar
- Kontraktion erhöht komplexität des Graphen
- Nachbarknoten von v bilden Clique

Overlays

Bisher:

- Kontraktionen reduzieren komplexität des Graphen
- Knoten werden endgültig gelöscht
- Mögliche Start-/Zielpunkte gehen verloren

Overlays

Karlsruher Institut für Technologie

Bisher:

- Kontraktionen reduzieren komplexität des Graphen
- Knoten werden endgültig gelöscht
- Mögliche Start-/Zielpunkte gehen verloren

Gesucht:

- Technik die alle Knoten im Graphen beibehält
- Dabei weiterhin von Shortcuts profitieren

Overlays

Bisher:

- Kontraktionen reduzieren komplexität des Graphen
- Knoten werden endgültig gelöscht
- Mögliche Start-/Zielpunkte gehen verloren

Gesucht:

- Technik die alle Knoten im Graphen beibehält
- Dabei weiterhin von Shortcuts profitieren

Lösung: Overlays

Overlays

Definition:

- Ein Overlay O ist eine Teilmenge der Knoten $O \subset V$
- Für Knoten $s, t \in O$ gilt:

  Es gibt einen kürzesten s-t-Pfad der nur Knoten aus O enthält

Overlays

Definition:

- Ein Overlay O ist eine Teilmenge der Knoten $O \subset V$
- Für Knoten $s, t \in O$ gilt:

Es gibt einen kürzesten s-t-Pfad der nur Knoten aus O enthält

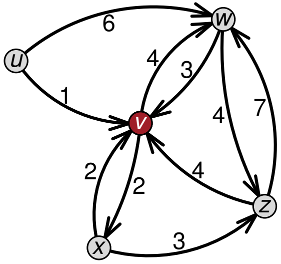
Kürzeste Wege Suche mit Overlays:

- Bidirektionale Variante von Dijkstra
- Suche erreicht Overlay $\Rightarrow$ Beschränke Suche auf Overlay

Prune Kanten (u, v) mit $u \in O$ und $v \notin O$

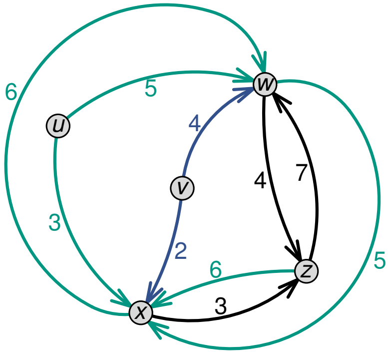
Overlays

Beispiel:

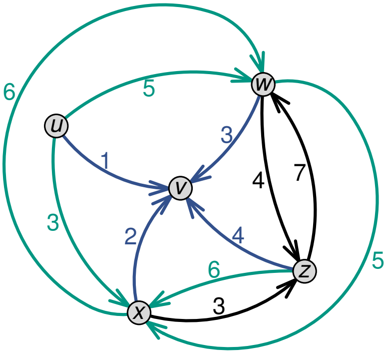


- Nutze speziellen Vorwärtsgraph $\overrightarrow{G}$ und Rückwärtsgraph $\overleftarrow{G}$
- $\overrightarrow{G}$ enthält Kanten ausgehend von kontrahierten Knoten
- $\overleftarrow{G}$ enthält Kanten zu kontrahierten Knoten

Beispiel:



$\overrightarrow{G}$:

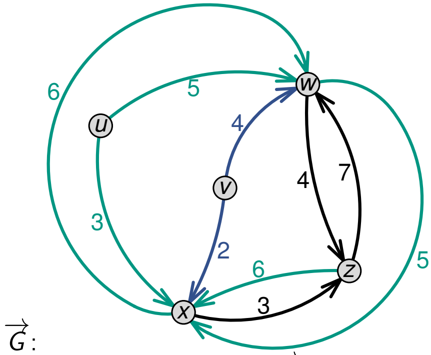


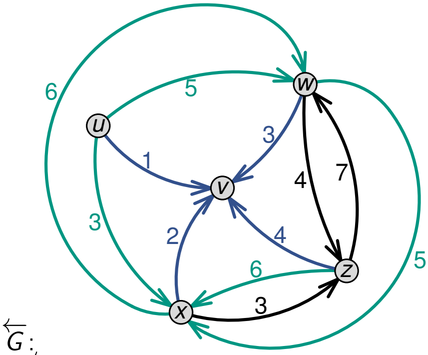
$\overleftarrow{G}$:

- Nutze speziellen Vorwärtsgraph $\overrightarrow{G}$ und Rückwärtsgraph $\overleftarrow{G}$
- $\overrightarrow{G}$ enthält Kanten ausgehend von kontrahierten Knoten
- $\overleftarrow{G}$ enthält Kanten zu kontrahierten Knoten

Overlays

Beispiel:





- Nutze speziellen Vorwärtsgraph $\overrightarrow{G}$ und Rückwärtsgraph $\overleftarrow{G}$
- $\overrightarrow{G}$ enthält Kanten ausgehend von kontrahierten Knoten
- $\overleftarrow{G}$ enthält Kanten zu kontrahierten Knoten

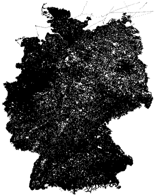
Danach: Bidirektionaler Dijkstra unverändert benutzbar

Graphen Vergleich

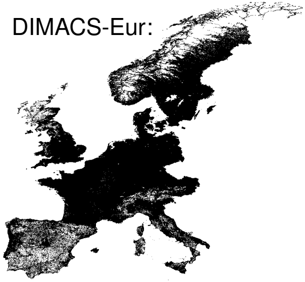
- Größe der Graphen:

Graph	$\lvert V \rvert [\cdot 10^3]$	$\lvert E \rvert [\cdot 10^3]$
OSM-Ger	20 690	41 792
OSM-Eur	173 789	347 997
DIMACS-Eur	18 010	42 189

OSM-Ger:



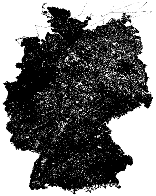
DIMACS-Eur:



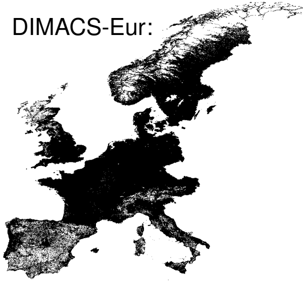
OSM-Eur:

Graphen Vergleich

OSM-Ger:



DIMACS-Eur:



- Größe der Graphen:

Graph	$\|V\|[\cdot 10^3]$	$\|E\|[\cdot 10^3]$
OSM-Ger	20 690	41 792
OSM-Eur	173 789	347 997
DIMACS-Eur	18 010	42 189

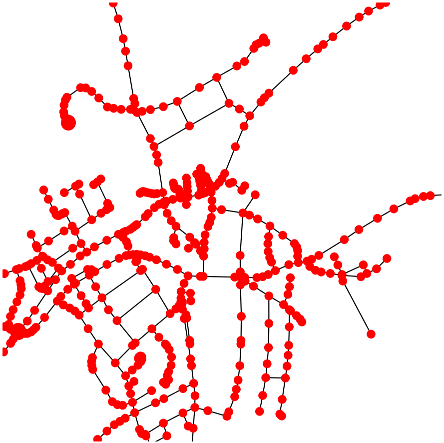
- Verteilung der Knotengrade:

Graph	#Knoten pro Grad:			
	1	2	3	≥ 4
OSM-Ger	14%	71%	14%	1%
OSM-Eur	12%	77%	10%	1%
DIMACS-Eur	26%	19%	49%	6%

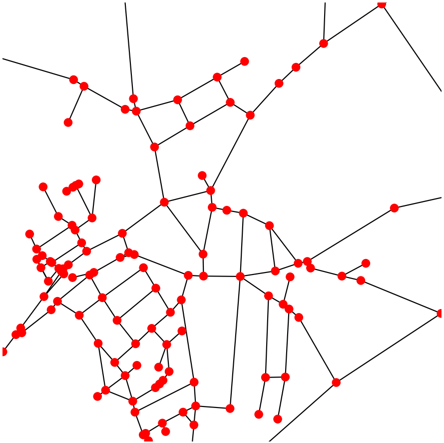
OSM-Eur:

Graphen Vergleich – TopoCore

OSM Input:



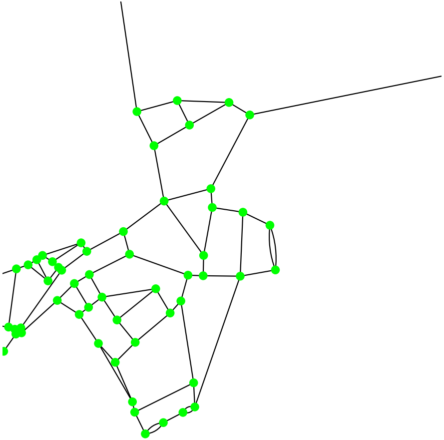
Dimacs Input:



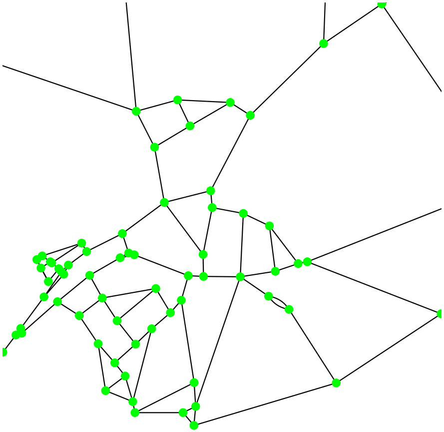
Graphen Vergleich – TopoCore

Karlsruher Institut für Technologie

OSM TopoCore:



Dimacs TopoCore:



Graphen Vergleich – TopoCore

Algorithmus	#Attr.	Graph	$\|V\|$ $[\cdot 10^6]$	$\|E\|$ $[\cdot 10^6]$	Prepro. [h:m:s]	Query [ms]
CH [GSSV'12]	1	DIMACS-Eur	18.0	42.2	2:45	0.15
Pareto-SHARC [DW'09]	2	DIMACS-Eur	18.0	42.2	7:12:00	35.4
FlexCH [GKS'10]	2	DIMACS-Eur	18.0	42.2	5:12:00	0.98
MultiCH [FS'13]	2	OSM-BaWü	2.5	5.0	2:01	0.42
MultiCH [FS'13]	3	OSM-BaWü	2.5	5.0	1:08	3.16
k-Path Cover [FNS'14]	8	OSM-BaWü	2.2	4.6	12	35
k-Path Cover [FNS'14]	8	**OSM-Ger**	17.7	36.1	**2:29**	**249**
PRP [FS'15]	10	**OSM-Ger**	21.7	44.1	**n/r**	**128**
TopoCore [DSW'15]	8	OSM-BaWü	3.1	6.2	3	9
TopoCore [DSW'15]	8	**OSM-Ger**	20.7	41.8	**35**	**86**
TopoCore [DSW'15]	8	OSM-Eur	173.8	348.0	10:57	621
TopoCore [DSW'15]	8	DIMACS-Eur	18.0	42.2	36	279
TopoCore [DSW'15]	8	DIMACS-US	23.9	57.7	43	386

Nächster Termin

Mittwoch, 26. April 2023

Bibliographie I

Edsger W. Dijkstra.
A Note on Two Problems in Connexion with Graphs.
Numerische Mathematik, 1(1):269–271, 1959.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner.
Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs.
In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015.

Hermann Kaindl and Gerhard Kainz.
Bidirectional Heuristic Search Reconsidered.
Journal of Artificial Intelligence Research, 7:283–317, December 1997.