

# Algorithmen für Routenplanung

2. Vorlesung, Sommersemester 2019

Tim Zeitz | 29. April 2019

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



---

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

---

```
1 forall nodes  $v \in V$  do
2    $d[v] = \infty, p[v] = \text{NULL}$  // n Mal
3  $d[s] = 0, Q.\text{clear}(), Q.\text{insert}(s, 0)$  // 1 Mal
4 while ! $Q.\text{empty}()$  do
5    $u \leftarrow Q.\text{deleteMin}()$  // n Mal
6   forall edges  $e = (u, v) \in E$  do
7     if  $d[u] + \text{len}(e) < d[v]$  then
8        $d[v] \leftarrow d[u] + \text{len}(e)$ 
9        $p[v] \leftarrow u$ 
10      if  $v \in Q$  then  $Q.\text{decreaseKey}(v, d[v])$  // m Mal
11      else  $Q.\text{insert}(v, d[v])$  // n Mal
```

---

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

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

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	$\Theta(1)$	$\Theta(\log k)$	$\Theta(\log k)$	$\mathcal{O}(\log k)$
<b>Dijkstra</b>	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}(m + n \log n)$
$m \in \mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Transportnetzwerke sind dünn  $\Rightarrow$  Binary Heaps

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

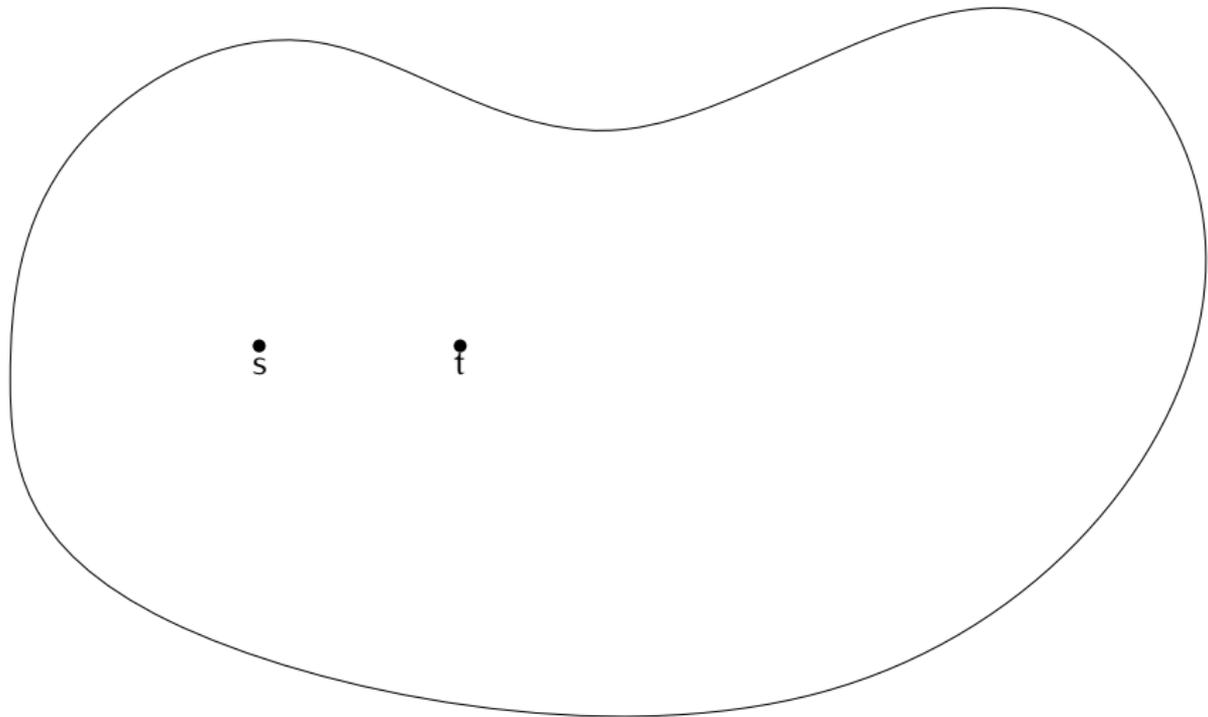
k	Query [sec]
2	1.834
3	1.595
4	1.507
5	1.525
8	1.561

Graph: 18M Knoten, 42M Kanten

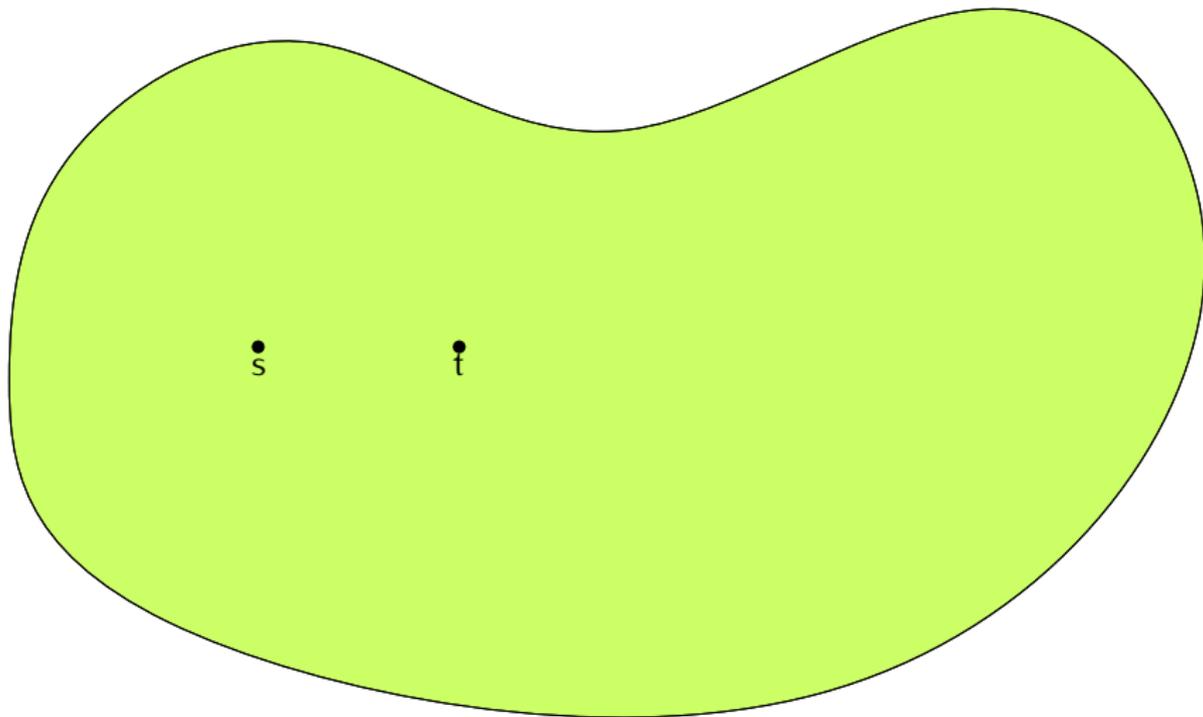
Dijkstra:  $\approx 1.5$  s  $\Rightarrow$  nicht interaktiv  
 $n + m$  CPU clock cycles:  $\approx 20$  ms  $\Rightarrow$  viel schneller  
BFS:  $\approx 1.2$  s  $\Rightarrow$  an der Queue liegt's nicht

Performanz von Graphsuchen ist speicher-begrenzt

# Schematischer Suchraum, Dijkstra



# Schematischer Suchraum, Dijkstra



## Beobachtung

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

## 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

## Beobachtung

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

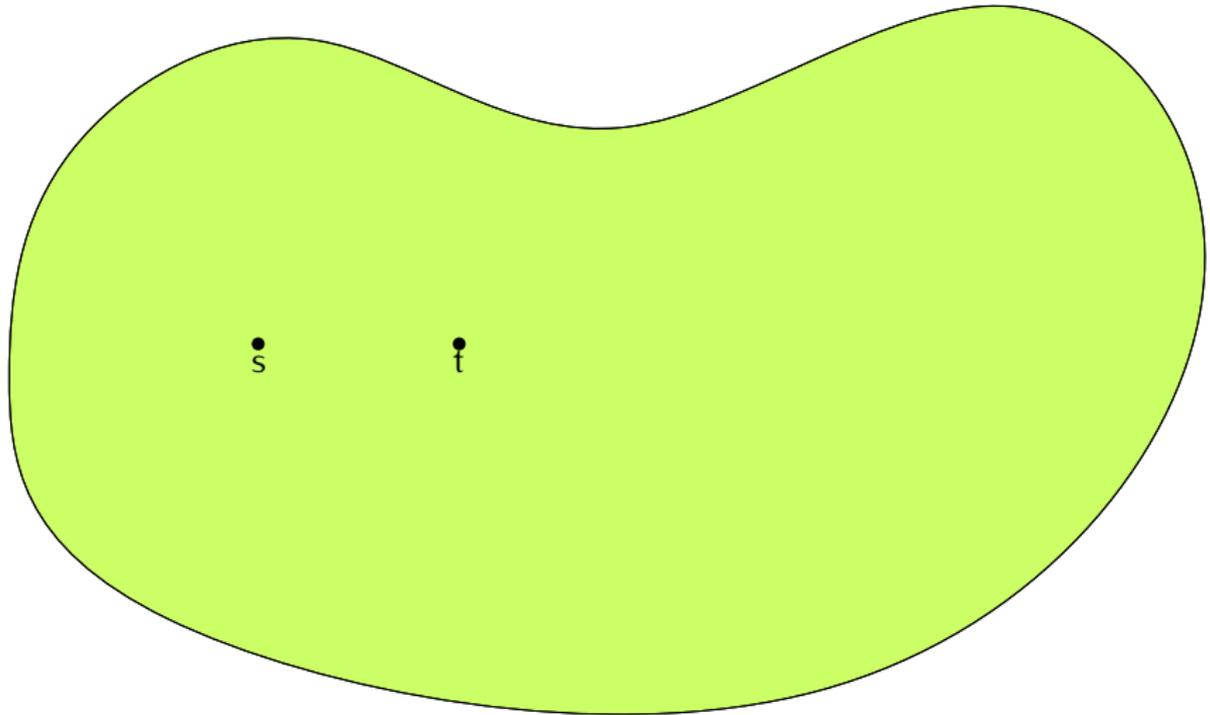
## Idee

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

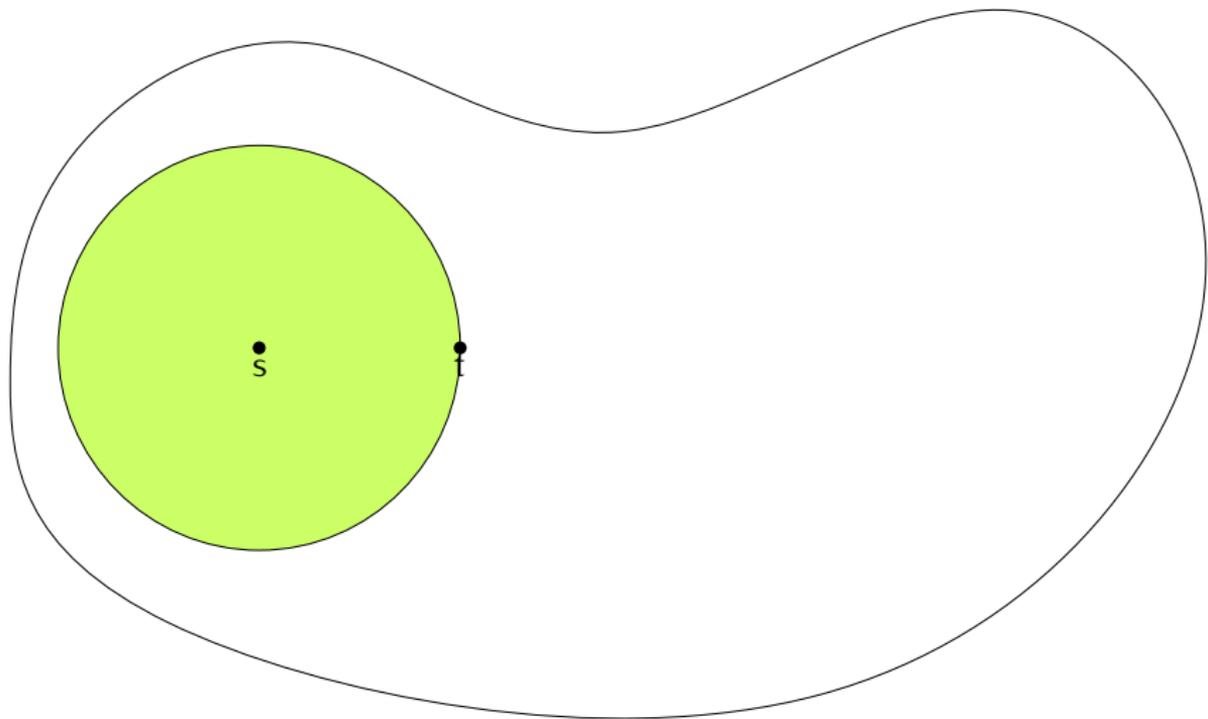
## Korrektheit

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

```
1 forall nodes  $v \in V$  do
2    $d[v] = \infty$ ,  $p[v] = \text{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.\text{clear}()$ ,  $Q.\text{add}(s, 0)$            // container
5 while  $!Q.\text{empty}()$  do
6    $u \leftarrow Q.\text{deleteMin}()$        // settling node u
7   if  $u = t$  then return
8   forall edges  $e = (u, v) \in E$  do
9     if  $d[u] + \text{len}(e) < d[v]$  then
10       $d[v] \leftarrow d[u] + \text{len}(e)$ 
11       $p[v] \leftarrow u$ 
12      if  $v \in Q$  then  $Q.\text{decreaseKey}(v, d[v])$ 
13
14      else  $Q.\text{insert}(v, d[v])$ 
```

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

---

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

---

```
1 forall nodes  $v \in V$  do
2    $d[v] = \infty$ ,  $p[v] = \text{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.\text{clear}()$ ,  $Q.\text{add}(s, 0)$            // container
5 while ! $Q.\text{empty}()$  do
6    $u \leftarrow Q.\text{deleteMin}()$          // settling node u
7   break if  $u = t$ 
8   forall edges  $e = (u, v) \in E$  do
9     // relaxing edges
10    if  $d[u] + \text{len}(e) < d[v]$  then
11       $d[v] \leftarrow d[u] + \text{len}(e)$ 
12       $p[v] \leftarrow u$ 
13      if  $v \in Q$  then  $Q.\text{decreaseKey}(v, d[v])$ 
14    else  $Q.\text{insert}(v, d[v])$ 
```

# Dijkstra mit Abbruchkriterium

---

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

---

```
1 forall nodes  $v \in V$  do
2    $d[v] = \infty$ ,  $p[v] = \text{NULL}$            // distances, parents
3    $d[s] = 0$ 
4    $Q.\text{clear}()$ ,  $Q.\text{add}(s, 0)$            // container
5   while ! $Q.\text{empty}()$  do
6      $u \leftarrow Q.\text{deleteMin}()$          // settling node u
7     break if  $u = t$ 
8     forall edges  $e = (u, v) \in E$  do
9       // relaxing edges
10      if  $d[u] + \text{len}(e) < d[v]$  then
11         $d[v] \leftarrow d[u] + \text{len}(e)$ 
12         $p[v] \leftarrow u$ 
13        if  $v \in Q$  then  $Q.\text{decreaseKey}(v, d[v])$ 
14      else  $Q.\text{insert}(v, d[v])$ 
```

## Problem

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

## 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.

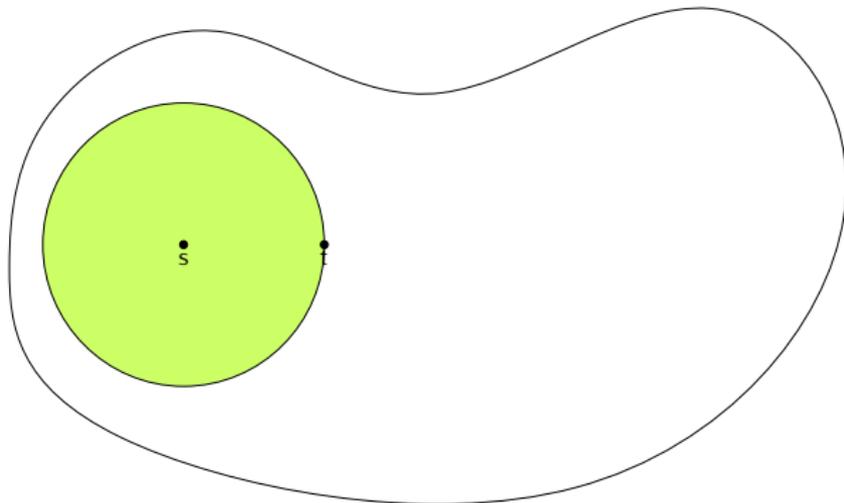
# Dijkstra mit Timestamps

// Überlauf

```
1 count ← count + 1
2 d[s] = 0
3 Q.clear(), Q.add(s, 0)
4 while !Q.empty() do
5     u ← Q.deleteMin()
6     if u = t then return
7     forall edges e = (u, v) ∈ E do
8         if run[v] ≠ count then
9             d[v] ← d[u] + len(e)
10            Q.insert(v, d[v])
11            run[v] ← count
12        else if d[u] + len(e) < d[v] then
13            d[v] ← d[u] + len(e)
14            Q.decreaseKey(v, d[v])
```

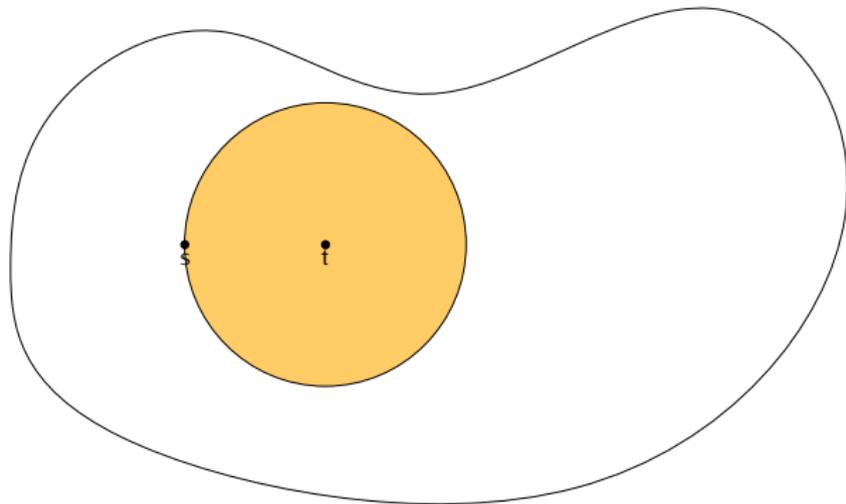
# Bidirektionale Suche





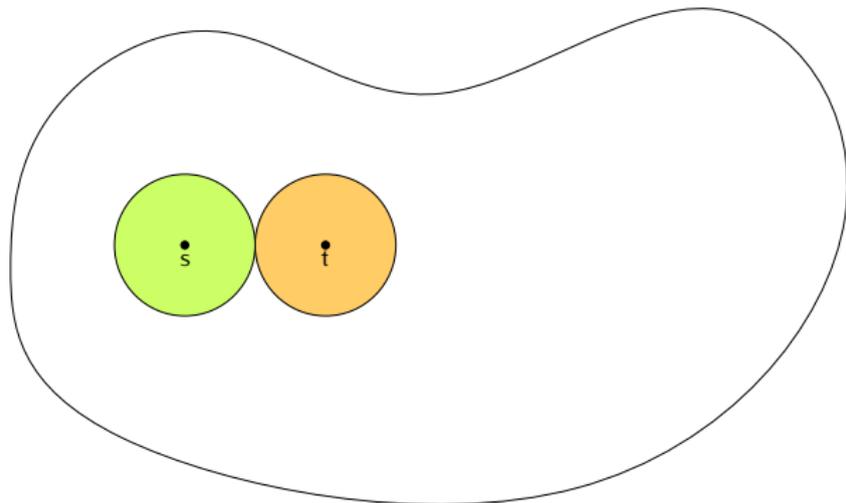
**Beobachtung:** Ein kürzester  $s$ - $t$ -Weg lässt sich finden durch

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



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

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

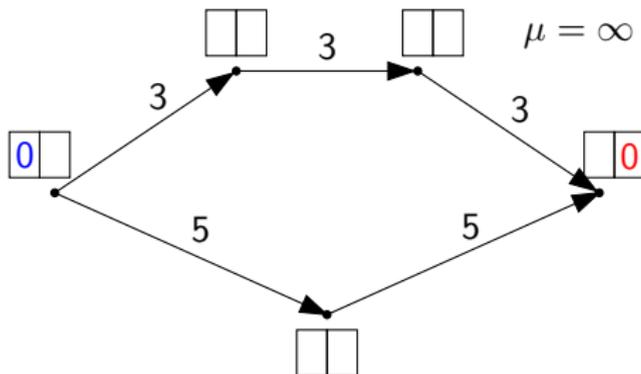


## Idee: Kombiniere beide Suchen

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

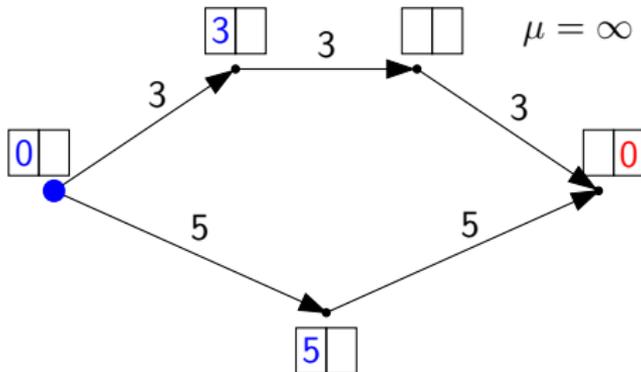
## Anfrage:

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



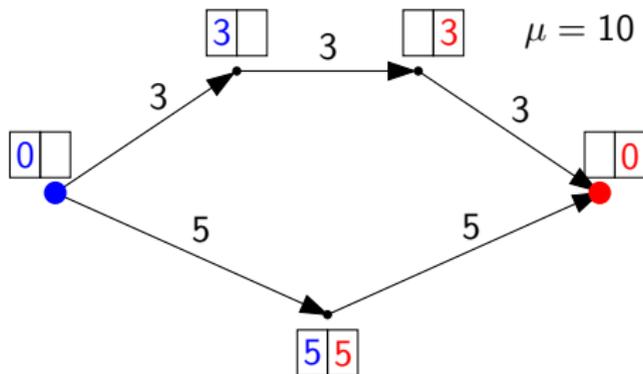
## Anfrage:

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



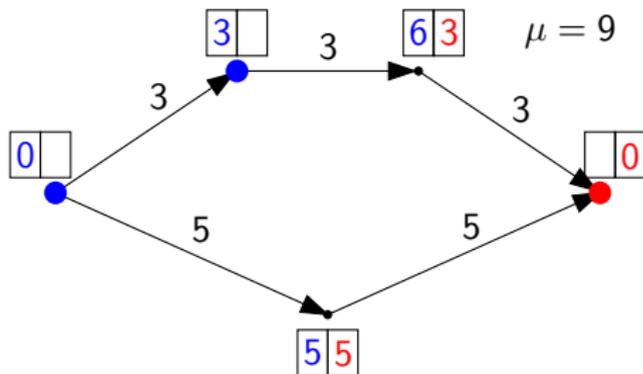
## Anfrage:

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



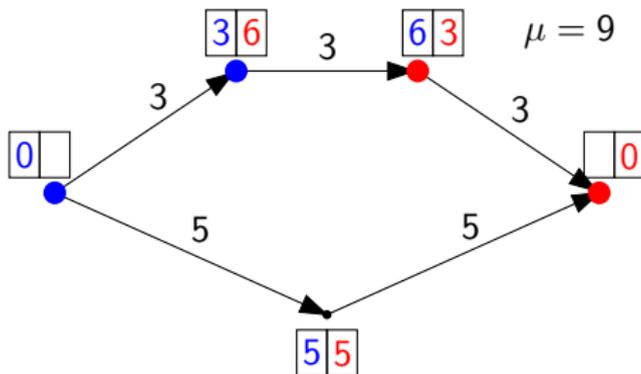
## Anfrage:

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



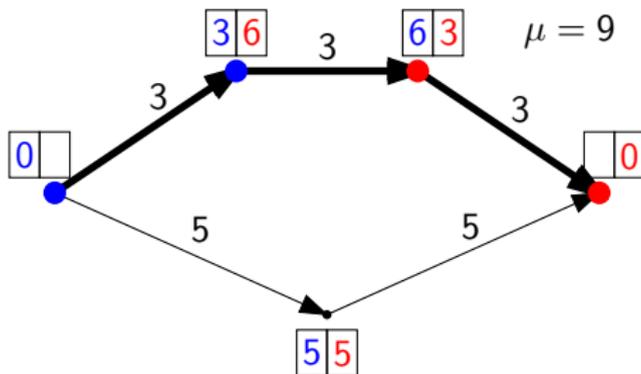
## Anfrage:

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



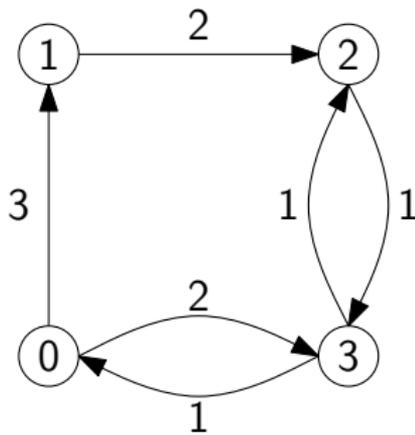
## Anfrage:

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



## Problem:

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



## 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

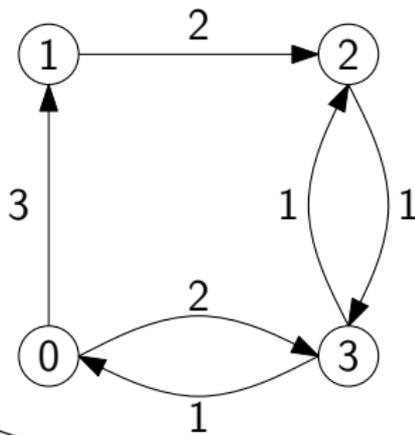
3	2	1	3	2	2	1	1	2	1
---	---	---	---	---	---	---	---	---	---

isIncoming

-	-	✓	✓	-	✓	✓	-	✓	✓
---	---	---	---	---	---	---	---	---	---

isOutgoing

✓	✓	-	-	✓	-	✓	✓	-	✓
---	---	---	---	---	---	---	---	---	---



- 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  $\overleftarrow{d}[]$

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

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

berechnet, sowie der (vorläufige) Mittelknoten  $m$ .

- Dazu wird bei der Relaxierung von Kante  $(u, v)$  zusätzlich

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

ausgeführt und  $m$  ggfs. aktualisiert. (Initial ist  $\mu = \infty$ ).

- Nach Terminierung beinhaltet  $\mu$  die Distanz  $d(s, t)$ ,  
 $P = s, \dots, m, \dots, t$  ist kürzester Pfad.

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

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

berechnet, sowie der (vorläufige) Mittelknoten  $m$ .

- Dazu wird bei der Relaxierung von Kante  $(u, v)$  zusätzlich

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

ausgeführt und  $m$  ggfs. aktualisiert. (Initial ist  $\mu = \infty$ ).

- Nach Terminierung beinhaltet  $\mu$  die Distanz  $d(s, t)$ ,  
 $P = s, \dots, m, \dots, t$  ist kürzester Pfad.

## Was sind gute Abbruchstrategien?

## Abbruchstrategie (1)

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

## Abbruchstrategie (1)

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

**Abbruchstrategie (1) berechnet  $d(s, t)$  korrekt. Beweisskizze:**

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

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

## 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:

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

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

$$\text{dist}(s, v_i) \geq \text{dist}(s, v^*)$$

$$\text{dist}(v_i, t) \geq \text{dist}(v^*, t)$$

- 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)

Abbruch, sobald  $\mu \leq \min\text{Key}(\vec{Q}) + \min\text{Key}(\overleftarrow{Q})$

## Abbruchstrategie (2)

Abbruch, sobald  $\mu \leq \min\text{Key}(\vec{Q}) + \min\text{Key}(\overleftarrow{Q})$

**Abbruchstrategie (2) berechnet  $d(s, t)$  korrekt. Beweisskizze:**

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

## Abbruchstrategie (2)

Abbruch, sobald  $\mu \leq \min\text{Key}(\vec{Q}) + \min\text{Key}(\overleftarrow{Q})$

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

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

$$d(s, u) \leq \min\text{Key}(\vec{Q}) \quad \text{und} \quad d(v, t) \leq \min\text{Key}(\overleftarrow{Q}) \quad \text{und}$$

$$d(s, u) < \min\text{Key}(\vec{Q}) \quad \text{oder} \quad d(v, t) < \min\text{Key}(\overleftarrow{Q}).$$

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

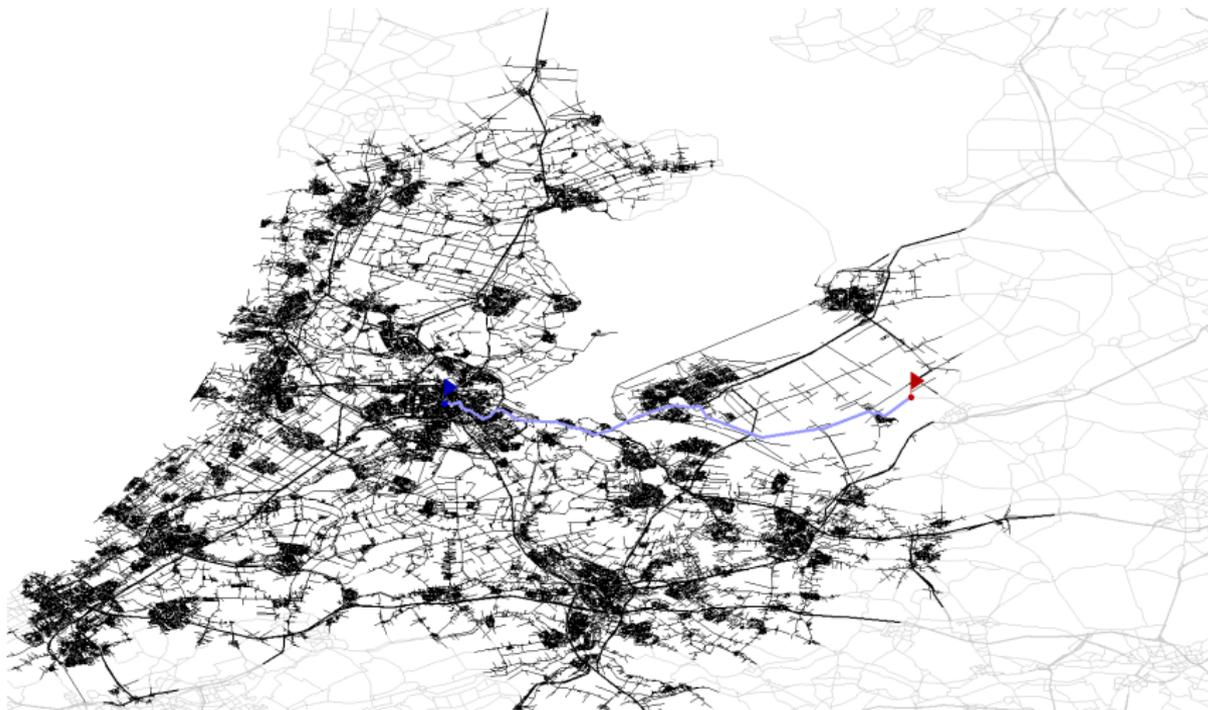
Damit ist  $\mu \leq d(s, t)$ . Widerspruch!

**Frage: Was sind mögliche Wechselstrategien?**

## 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



## Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius  $r$

⇒ 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

⇒ Gesamtspeedup ca. 4

## Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius  $r$

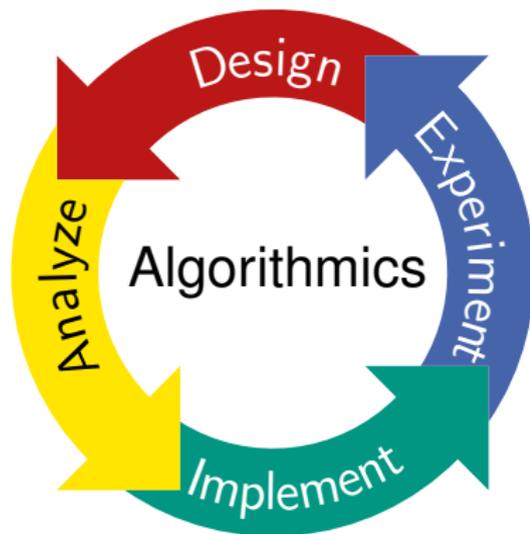
⇒ Speedup bzgl. Suchraum (ca.):

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

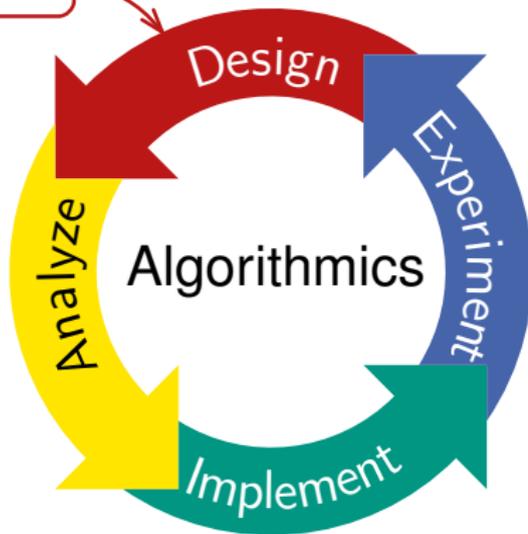
- Führe Suchen *parallel* aus

⇒ Gesamtspeedup ca. 4

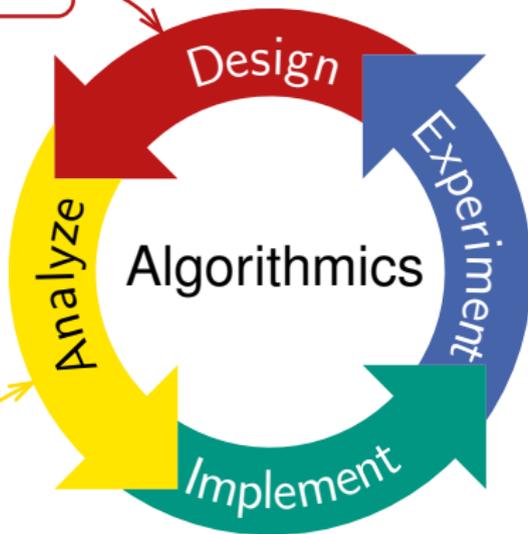
Wichtiger Bestandteil vieler effizienter Techniken!



Dijkstras Algorithmus

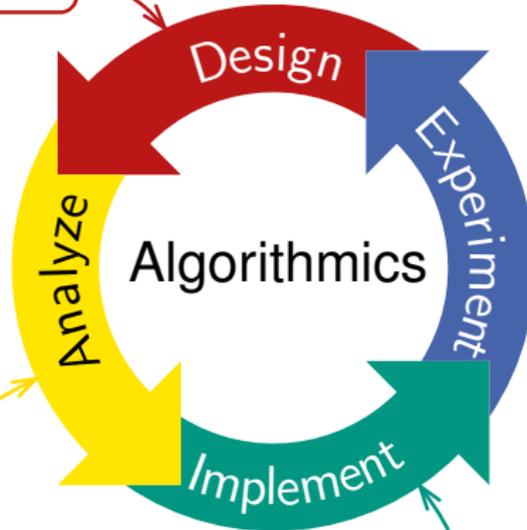


Dijkstras Algorithmus



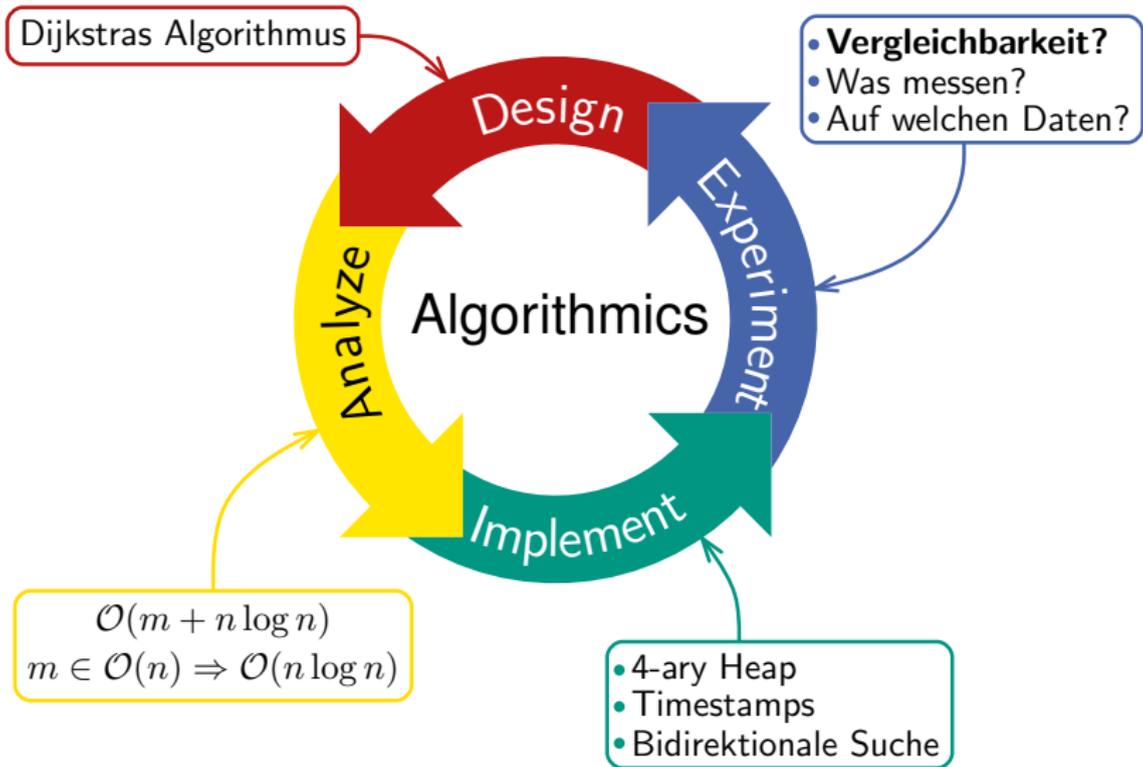
$$\mathcal{O}(m + n \log n)$$
$$m \in \mathcal{O}(n) \Rightarrow \mathcal{O}(n \log n)$$

Dijkstras Algorithmus



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

- 4-ary Heap
- Timestamps
- Bidirektionale Suche



## Problem:

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

## Problem:

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

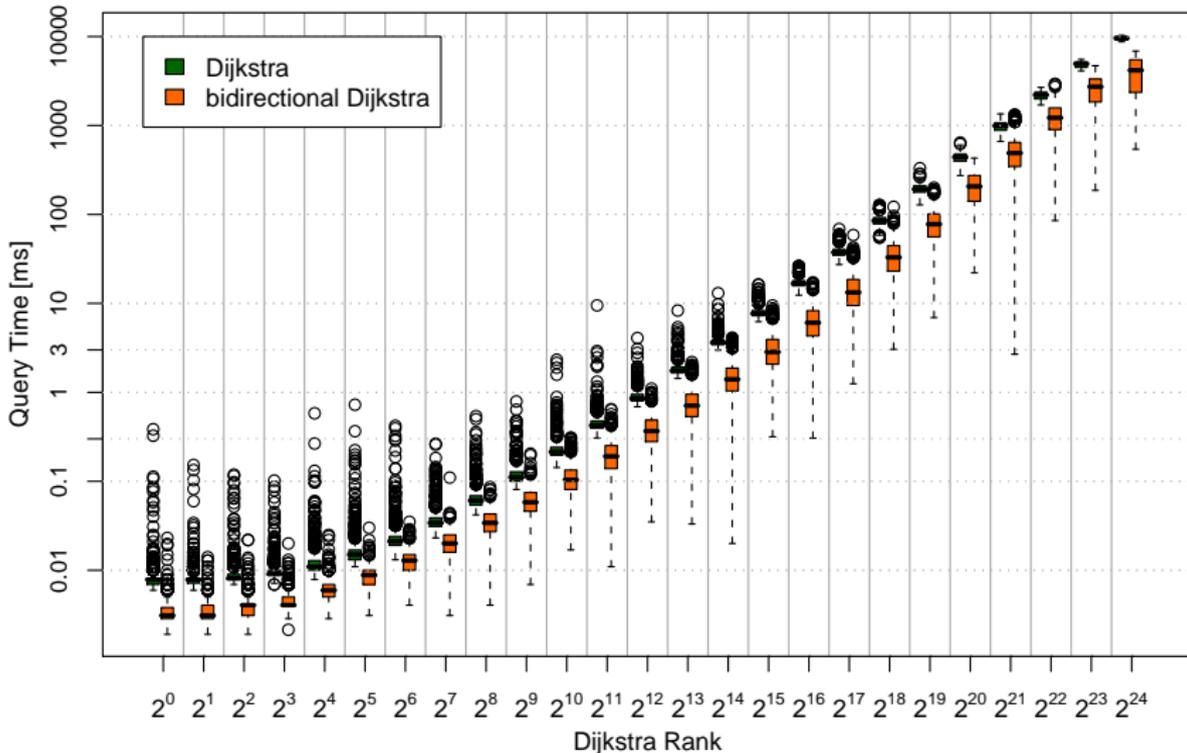
## Idee:

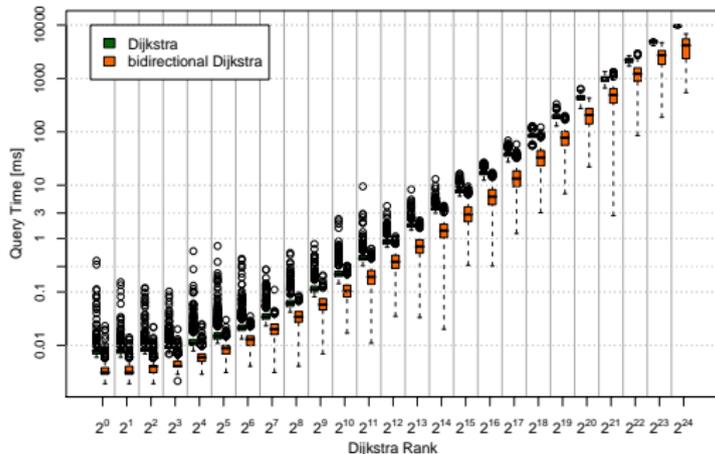
- DIJKSTRA definiert für Startknoten  $s$  Ordnung auf den Knoten
- DIJKSTRA-Rang  $r_s(u)$  eines Knoten  $u$  für gegebenes  $s$
- Wähle 1000 zufällige Startknoten, analysiere jeweils den Suchraum

Zielknoten  $t :=$  Knoten mit Rang  $2^1, \dots, 2^{\log n}$

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

# Dijkstra-Rank – Bidirektionale Suche

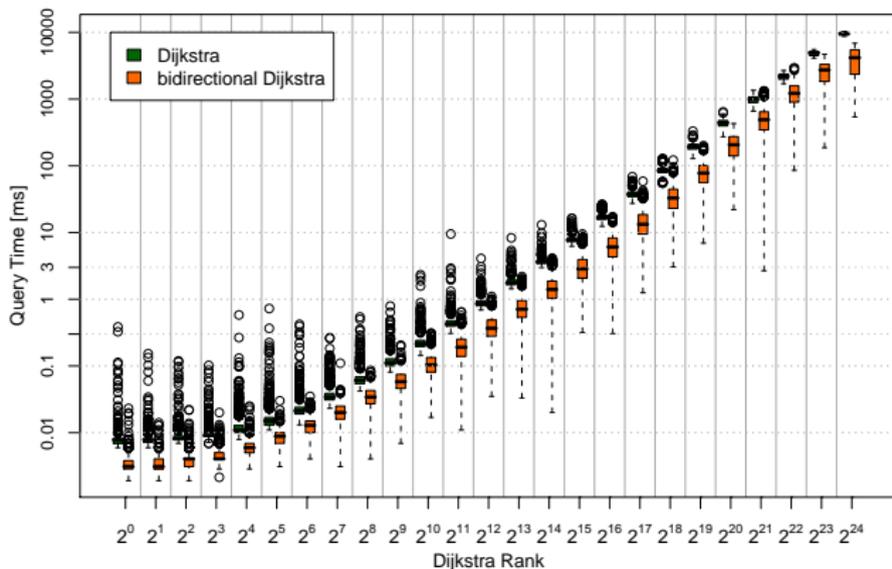




## Boxplots

- 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

## Welche Graphen für Experimente nutzen:

- Reales Straßennetz
- Synthetische Daten
- Zufallsgraphen

## Welche Graphen für Experimente nutzen:

- Reales Straßennetz
- Synthetische Daten
- Zufallsgraphen

## Zielsetzung:

- Anwendungsnahe
- Effizienz auf echten Daten

## Welche Graphen für Experimente nutzen:

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

## Zielsetzung:

- Anwendungsnahe
- Effizienz auf echten Daten

## Welche Graphen für Experimente nutzen:

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

## Zielsetzung:

- Anwendungsnahe
- Effizienz auf echten Daten

## Welchen Einfluss haben die Daten auf die Experimente?

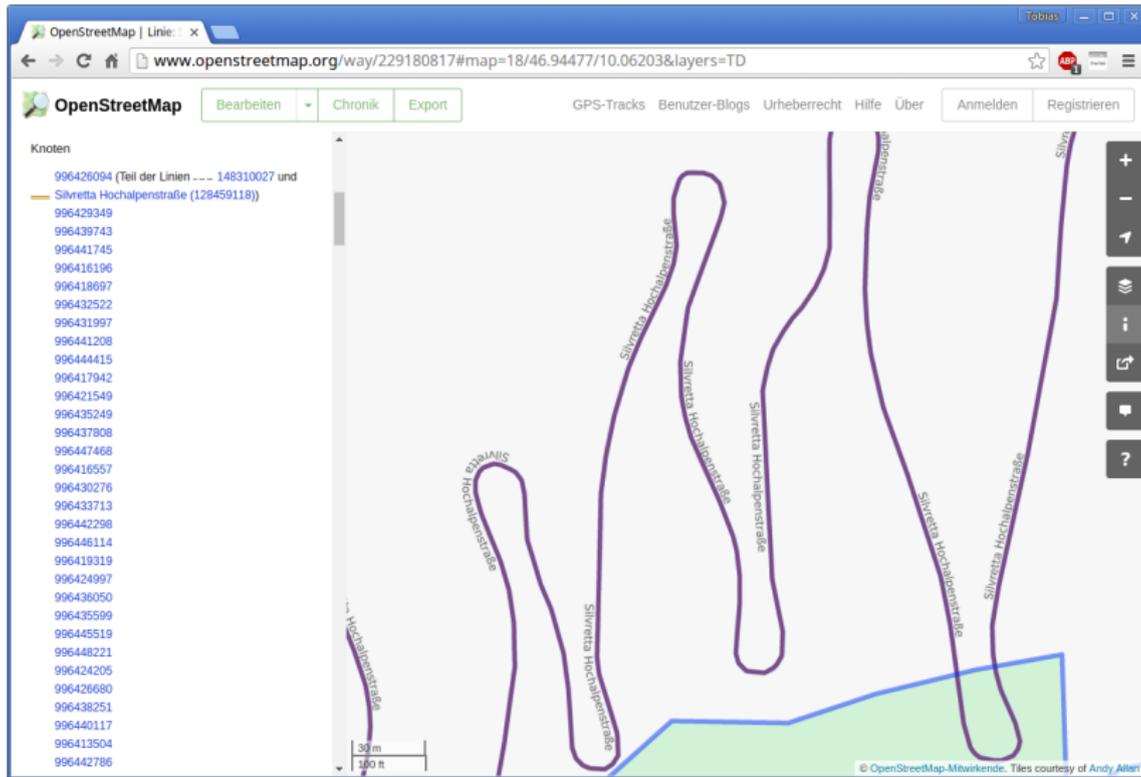
## Straßengraphen extrahiert von OSM:

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

## 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
- ⇒ Experimente schwer vergleichbar





OpenStreetMap | Linie: x

www.openstreetmap.org/way/229180817#map=18/46.94477/10.06203&layers=TD

OpenStreetMap Bearbeiten Chronik Export

GPS-Tracks Benutzer-Blogs Urheberrecht Hilfe Über Anmelden Registrieren

Knoten

- 996426094 (Teil der Linien ---- 148310027 und
- Silvretta Hochalpenstraße (128459118))
- 996429349
- 996439743
- 996441745
- 996416196
- 996418697
- 996432522
- 996431997
- 996441208
- 996444415
- 996417942
- 996421549
- 996435249
- 996437808
- 996447468
- 996416557
- 996430276
- 996433713
- 996442298
- 996446114
- 996419319
- 996424997
- 996436050
- 996435599
- 996445519
- 996448221
- 996424205
- 996426680
- 996438251
- 996440117
- 996413504
- 996442786

30 m  
100 ft

© OpenStreetMap-Mitwirkende. Tiles courtesy of Andy Allan

## OSM Graphen:

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

## OSM Graphen:

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

## Wie Graphen vereinheitlichen?

## 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?

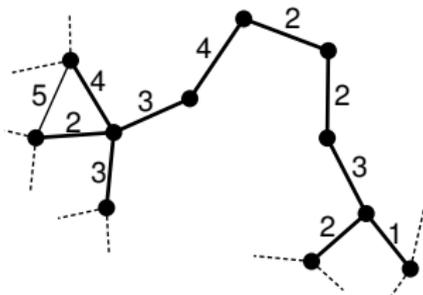
## OSM Graphen:

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

## Wie Graphen vereinheitlichen?

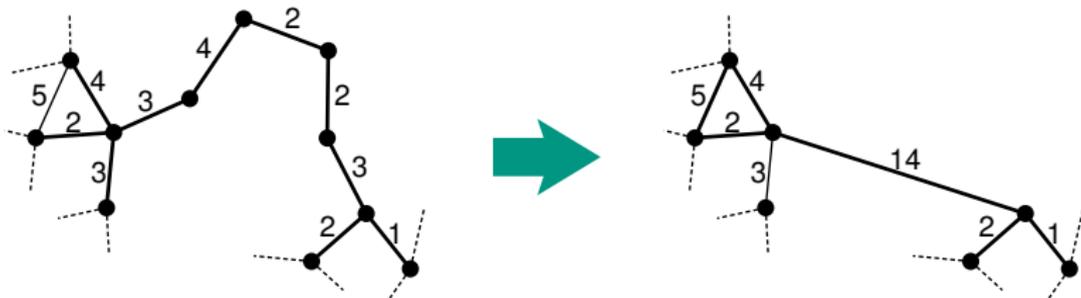
- Grad zwei Knoten löschen?
- Knoten kontrahieren und Overlay Graphen!

## Beispiel: Kontraktion eines Pfades



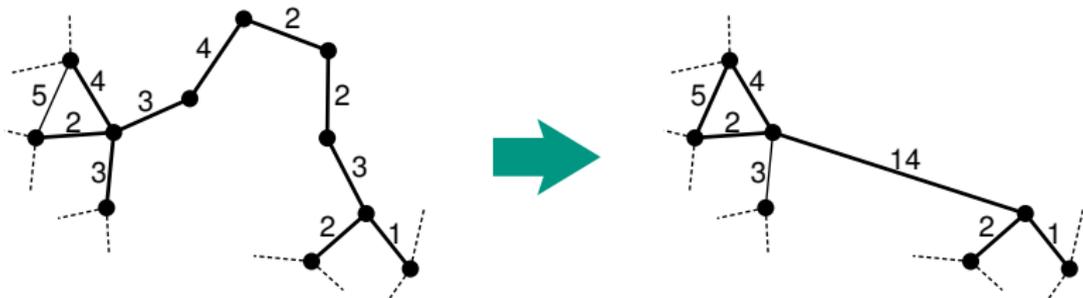
- Finde Pfade im Graphen

## Beispiel: Kontraktion eines Pfades



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

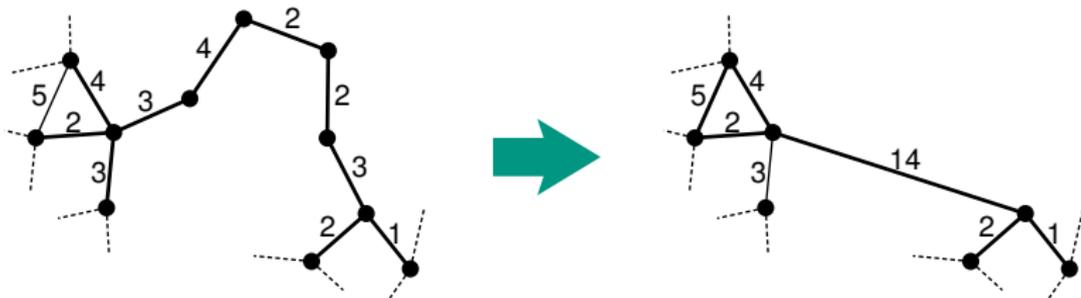
## Beispiel: Kontraktion eines Pfades



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

## Probleme?

## Beispiel: Kontraktion eines Pfades

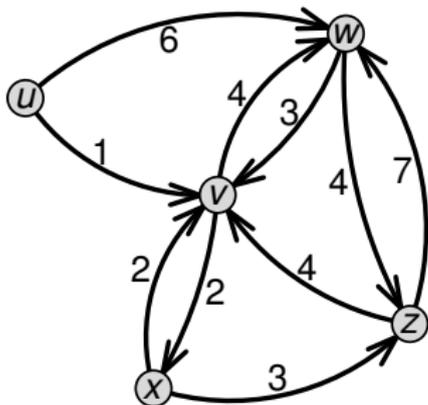


- Finde Pfade im Graphen
- Ersetze jeden Pfad durch einen **Shortcut**
- Länge des Shortcut = Länge des Pfades

## Probleme:

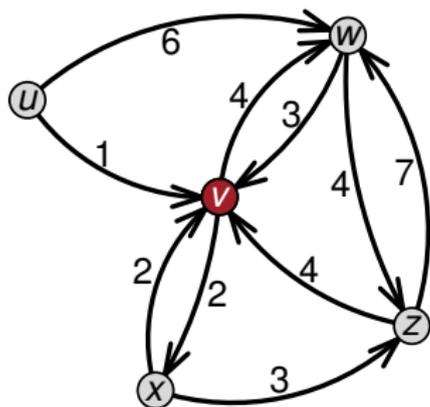
- Gelöschte Knoten können nicht mehr Start/Ziel sein
- Entfernt nur Grad 2 Knoten – Gibt es weitere unnötige Strukturen?

## Allgemeine Knoten Kontraktion:



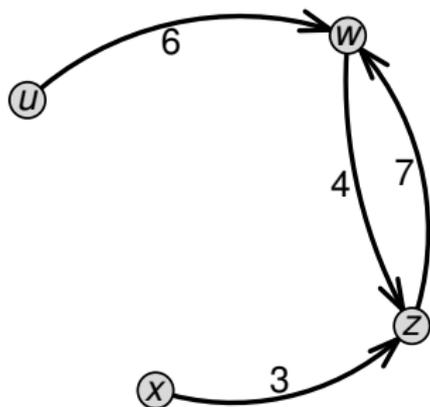
## Allgemeine Knoten Kontraktion:

- Knoten  $v$  soll kontrahiert werden



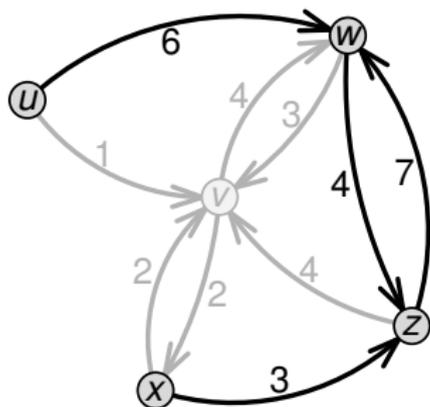
## Allgemeine Knoten Kontraktion:

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



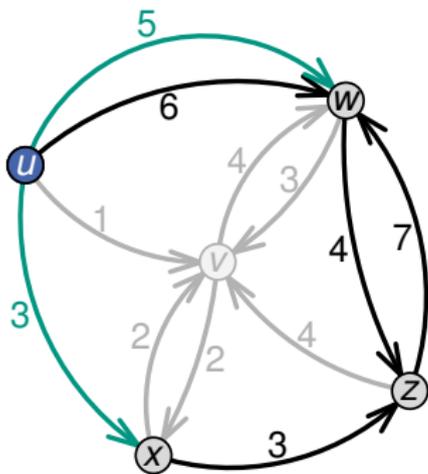
## Allgemeine Knoten Kontraktion:

- 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  $(a, b)$



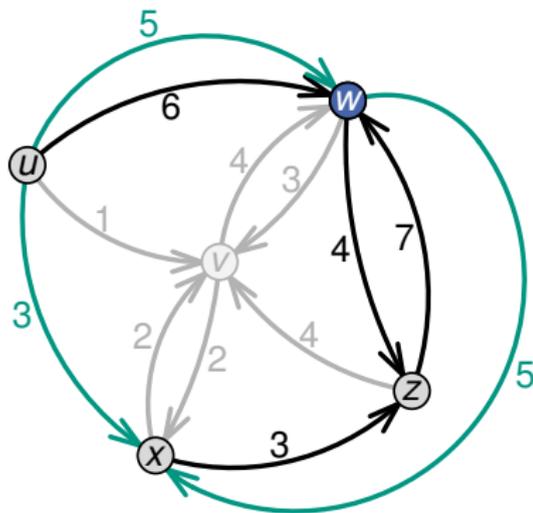
## Allgemeine Knoten Kontraktion:

- 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**  $(a, b)$



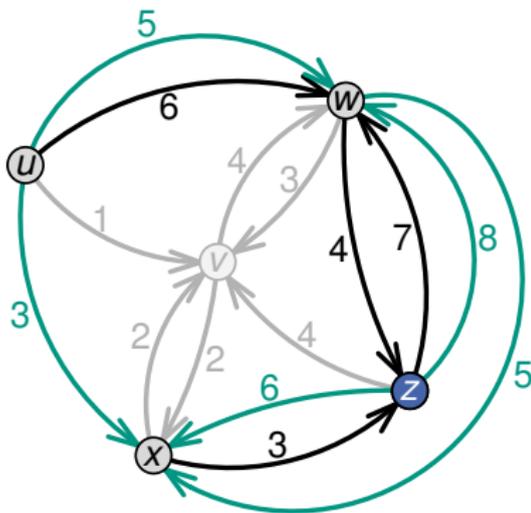
## Allgemeine Knoten Kontraktion:

- 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**  $(a, b)$



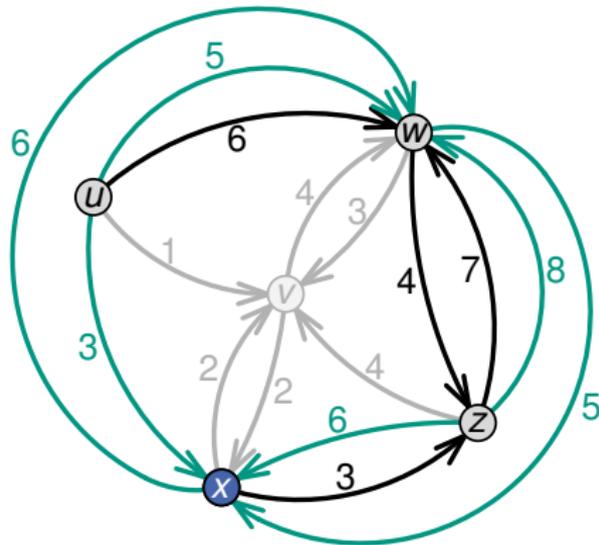
## Allgemeine Knoten Kontraktion:

- 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**  $(a, b)$



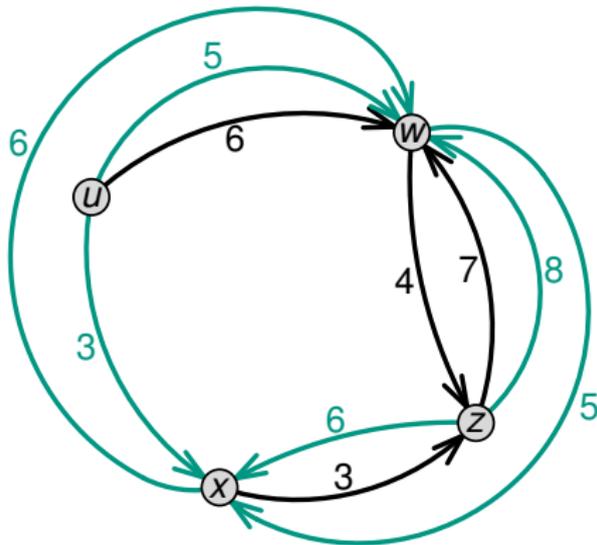
## Allgemeine Knoten Kontraktion:

- 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**  $(a, b)$



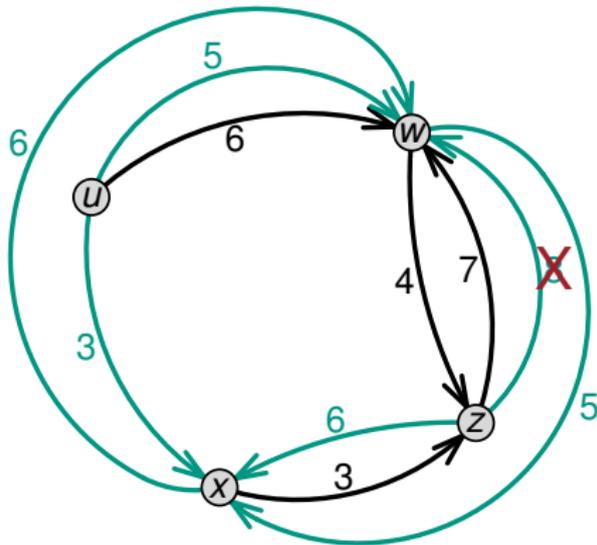
## Allgemeine Knoten Kontraktion:

- 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  $v$



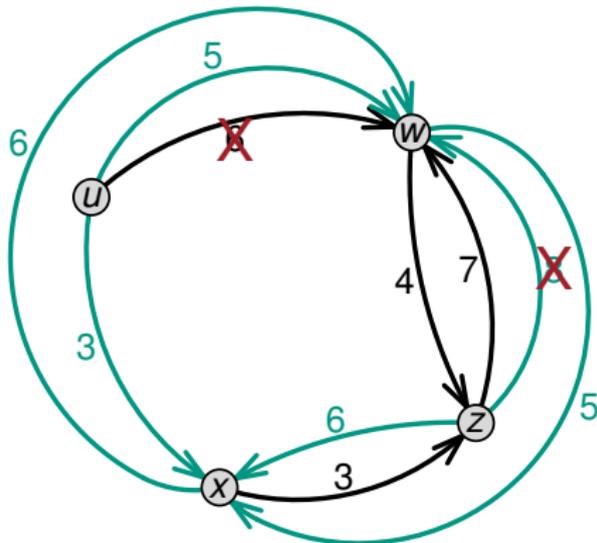
## Allgemeine Knoten Kontraktion:

- 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**



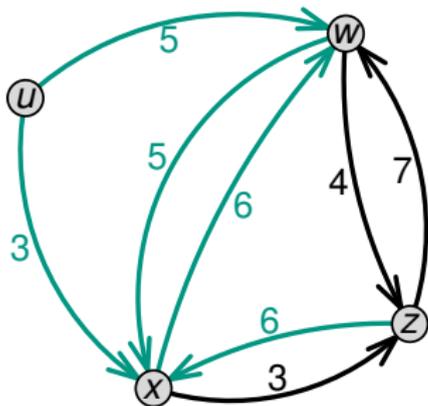
## Allgemeine Knoten Kontraktion:

- 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**



## Allgemeine Knoten Kontraktion:

- 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

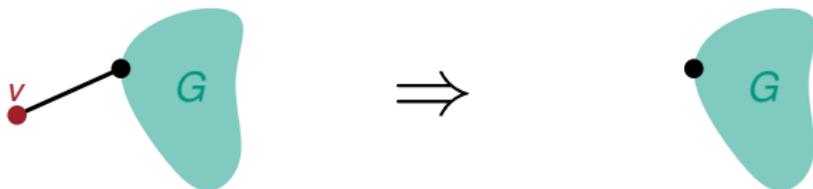


**Grundstein vieler weiterer Beschleunigungstechniken**

**Welche Kontraktionen reduzieren die Komplexität des Graphen?**

## Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad 1 Knoten:



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

## Welche Kontraktionen reduzieren die Komplexität des Graphen?

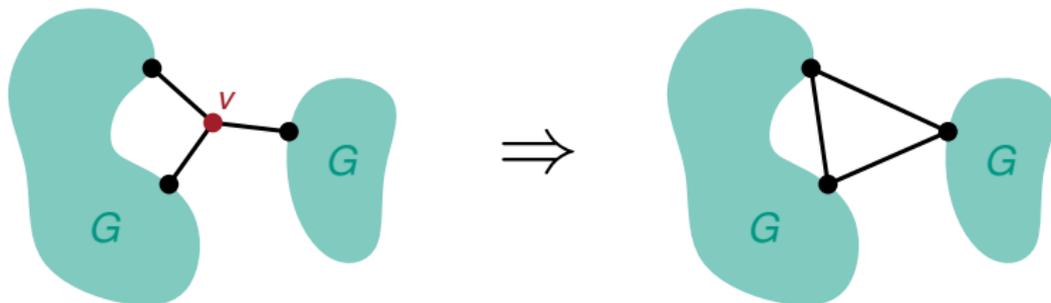
- Grad 2 Knoten:



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

## Welche Kontraktionen reduzieren die Komplexität des Graphen?

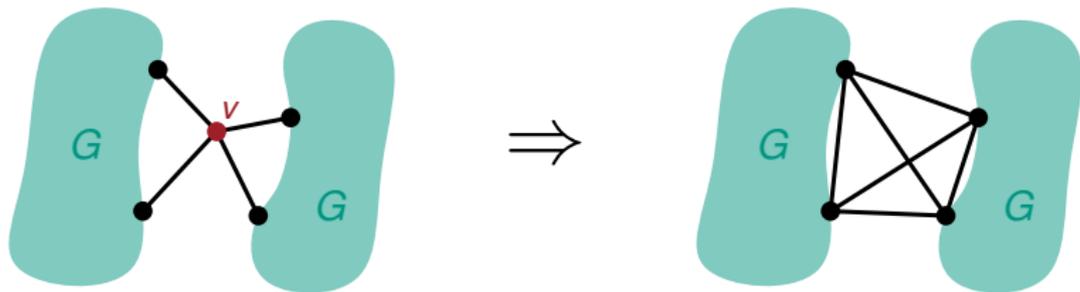
- Grad 3 Knoten:



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

## Welche Kontraktionen reduzieren die Komplexität des Graphen?

- Grad  $\geq 4$  Knoten:



- Struktur **nicht** mehr ausnutzbar
- Kontraktion erhöht Komplexität des Graphen
- Nachbarknoten von  $v$  bilden Clique

## Bisher:

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

## 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

## 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**

## Definition:

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

Der kürzeste  $s$ - $t$ -Pfad enthält nur Knoten aus  $O$

## Definition:

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

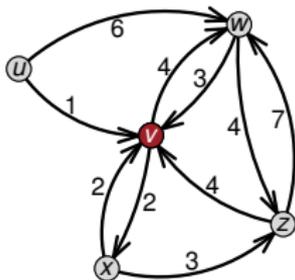
Der kürzeste  $s$ - $t$ -Pfad enthält nur Knoten aus  $O$

## 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  $v \notin O$

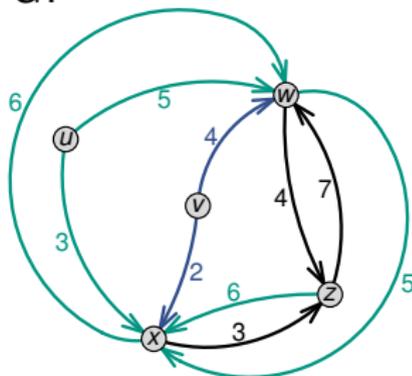
## Beispiel:



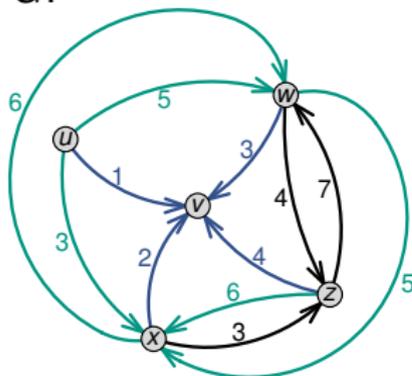
- Nutze speziellen Vorwärtsgraph  $\vec{G}$  und Rückwärtsgraph  $\overleftarrow{G}$
- $\vec{G}$  enthält Kanten ausgehend von kontrahierten Knoten
- $\overleftarrow{G}$  enthält Kanten zu kontrahierten Knoten

## Beispiel:

$\vec{G}$ :



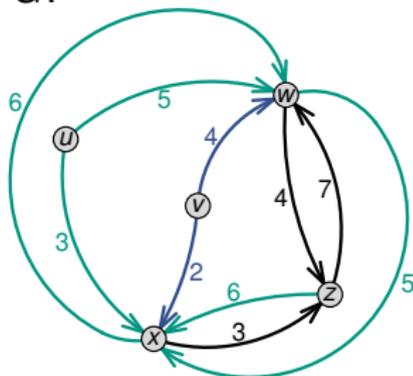
$\overleftarrow{G}$ :



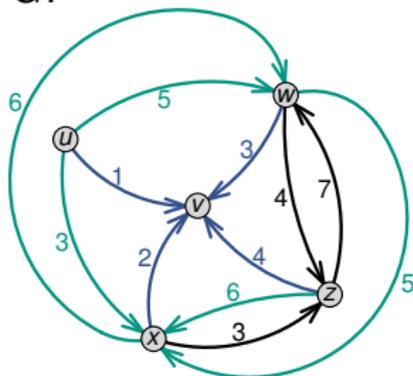
- Nutze speziellen Vorwärtsgraph  $\vec{G}$  und Rückwärtsgraph  $\overleftarrow{G}$
- $\vec{G}$  enthält Kanten ausgehend von kontrahierten Knoten
- $\overleftarrow{G}$  enthält Kanten zu kontrahierten Knoten

## Beispiel:

$\vec{G}$ :



$\overleftarrow{G}$ :



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

**Danach:** Bidirektionaler Dijkstra unverändert benutzbar

## ■ 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

OSM-Eur:



DIMACS-Eur:



OSM-Ger:



## ■ 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	$\geq 4$
OSM-Ger	14%	71%	14%	1%
OSM-Eur	12%	77%	10%	1%
DIMACS-Eur	26%	19%	49%	6%

OSM-Eur:



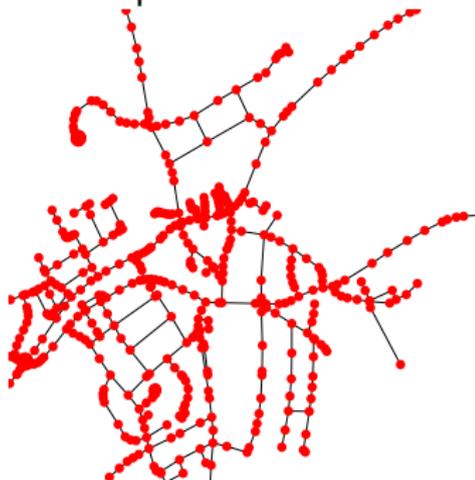
DIMACS-Eur:



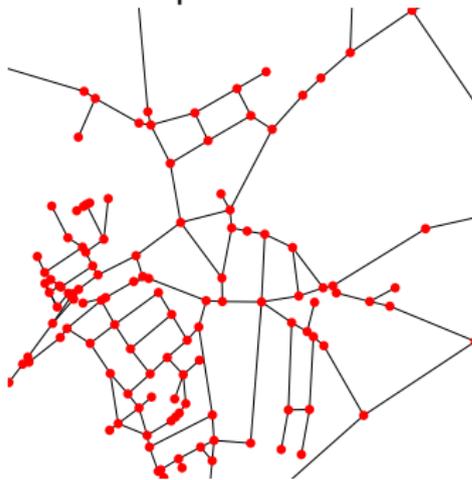
OSM-Ger:



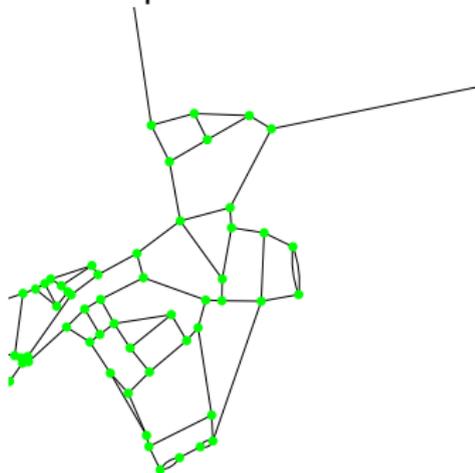
OSM Input:



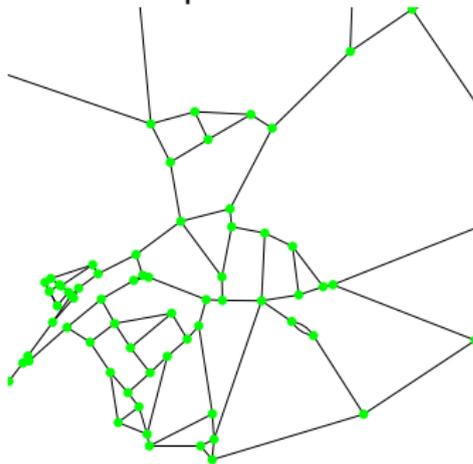
Dimacs Input:



OSM TopoCore:



Dimacs 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	<b>OSM-Ger</b>	17.7	36.1	<b>2:29</b>	<b>249</b>
PRP [FS'15]	10	<b>OSM-Ger</b>	21.7	44.1	<b>n/r</b>	<b>128</b>
TopoCore [DSW'15]	8	OSM-BaWü	3.1	6.2	3	9
TopoCore [DSW'15]	8	<b>OSM-Ger</b>	20.7	41.8	<b>35</b>	<b>86</b>
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

# Montag, 6. Mai 2019



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.