

# Algorithmen für Routenplanung

1. Vorlesung, Sommersemester 2016

Tobias Zündorf | 18. April 2016

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





Prof. Dorothea  
Wagner



Moritz Baum



Ben Strasser



Tobias Zündorf



Valentin Buchhold

**Vorlesungswebseite:** <http://i11www.itl.kit.edu/teaching/sommer2016/routenplanung/index>

## Vorlesung

- Montags 14:00–15:30 Uhr, SR 301 (hier)
- Mittwochs 11:30–13:00 Uhr, SR 301 (hier)

## Prüfung

- Prüfbar im Master (und Hauptstudium Diplom)
- Im Master: 5 ECTS Kredite
- VF: 1 (Theoretische Grundlagen), 2 (Algorithmentechnik)

### Vorlesungswebseite:

[http://illwww.iti.kit.edu/teaching/sommer2016/  
routenplanung/index](http://illwww.iti.kit.edu/teaching/sommer2016/routenplanung/index)

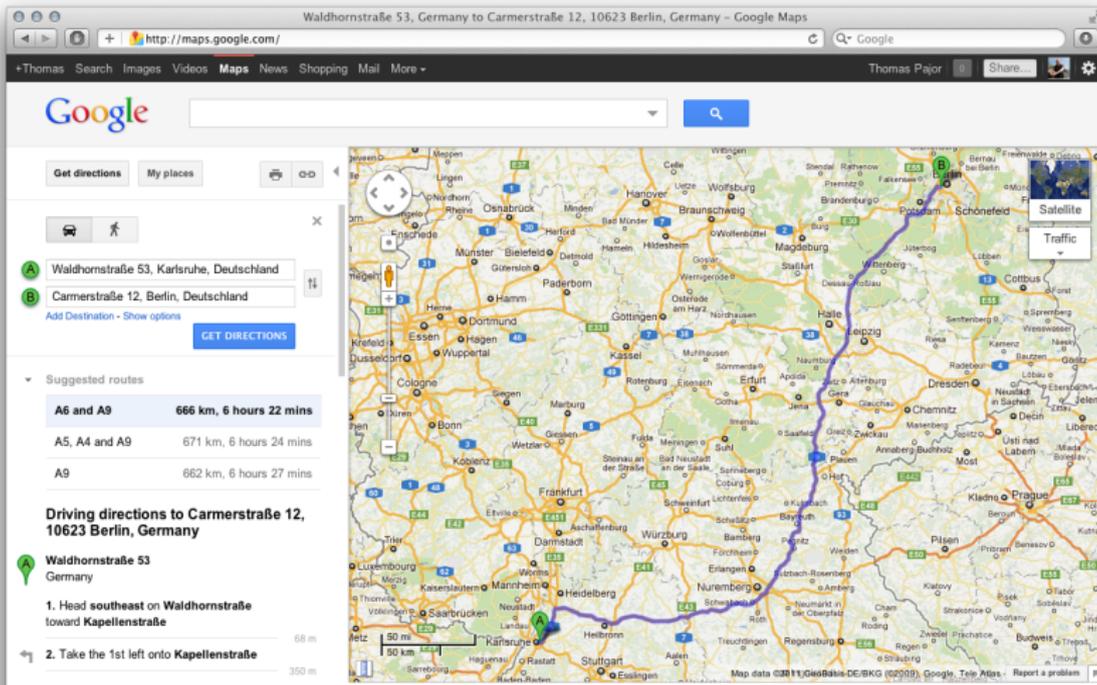
## Algorithmen für planare Graphen

- Dozenten: Prof. Dorothea Wagner
- Vorlesung: Montags 15:45–17:15 Uhr  
(Hörsaal -102 – Gebäude 50.34 – Informatik-Hauptgebäude)
- Vorlesung: Dienstags 14:00–15:30 Uhr  
(Hörsaal II – Gebäude 30.41 – Chemie Flachbau)
- Übung: zusätzlich an einigen Vorlesungsterminen

# Routenplanung in der Anwendung



# Routenplanung in der Anwendung



Waldhornstraße 53, Germany to Carmerstraße 12, 10623 Berlin, Germany - Google Maps

http://maps.google.com/

Thomas Pajor

Google

Get directions My places

Waldhornstraße 53, Karlsruhe, Deutschland

Carmerstraße 12, Berlin, Deutschland

GET DIRECTIONS

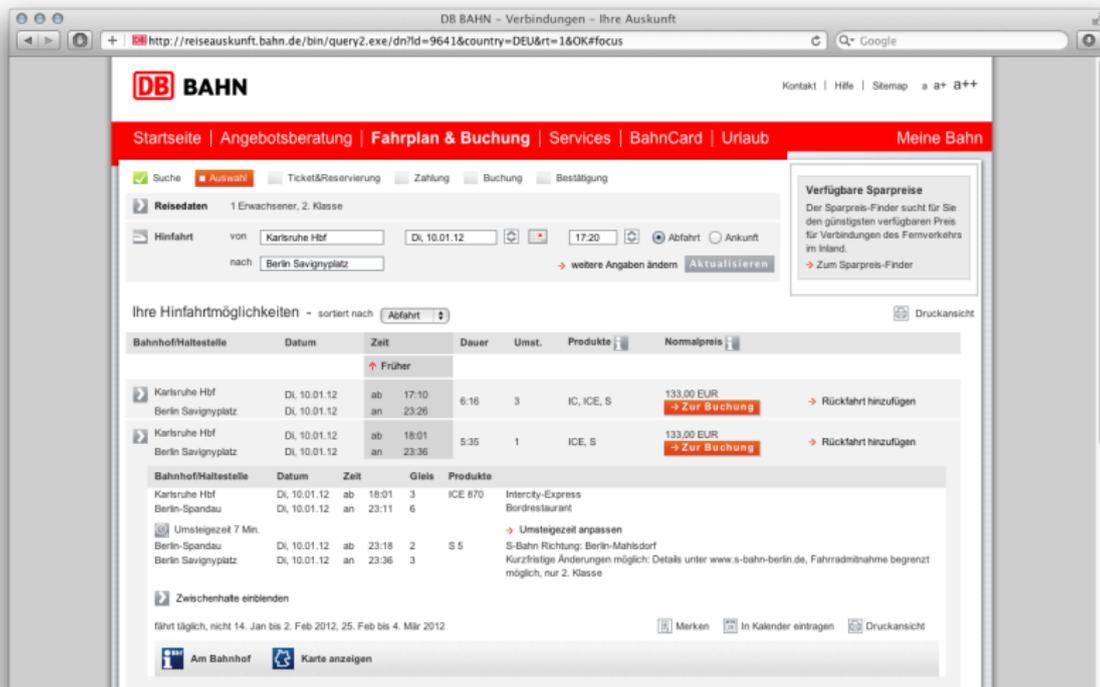
Suggested routes

Route	Distance	Time
A6 and A9	666 km	6 hours 22 mins
A5, A4 and A9	671 km	6 hours 24 mins
A9	662 km	6 hours 27 mins

Driving directions to Carmerstraße 12, 10623 Berlin, Germany

- Head southeast on Waldhornstraße toward Kapellenstraße 68 m
- Take the 1st left onto Kapellenstraße 350 m

# Routenplanung in der Anwendung



DB BAHN

Kontakt | Hilfe | Sitemap | Meine Bahn

Suche **Auswahl** Ticket&Reservierung Zahlung Buchung Bestätigung

**Reisedaten** 1 Erwachsener, 2. Klasse

Hinfahrt von     Abfahrt  Ankunft  
nach  [weitere Angaben ändern](#) [Aktualisieren](#)

**Verfügbare Sparpreise**  
Der Sparpreis-Finder sucht für Sie den günstigsten verfügbaren Preis für Verbindungen des Fernverkehrs im Inland.  
[Zum Sparpreis-Finder](#)

Ihre Hinfahrtmöglichkeiten - sortiert nach **Abfahrt** [Druckansicht](#)

Bahnhof/Haltestelle	Datum	Zeit	Dauer	Ums.	Produkte	Normalpreis	
		<b>↑ Früher</b>					
<input type="checkbox"/> Karlsruhe Hbf	Di, 10.01.12	ab 17:10	6:16	3	IC, ICE, S	133,00 EUR <a href="#">→ Zur Buchung</a>	<a href="#">→ Rückfahrt hinzufügen</a>
<input type="checkbox"/> Berlin Savignyplatz	Di, 10.01.12	an 23:26					
<input type="checkbox"/> Karlsruhe Hbf	Di, 10.01.12	ab 18:01	5:35	1	ICE, S	133,00 EUR <a href="#">→ Zur Buchung</a>	<a href="#">→ Rückfahrt hinzufügen</a>
<input type="checkbox"/> Berlin Savignyplatz	Di, 10.01.12	an 23:36					

Bahnhof/Haltestelle	Datum	Zeit	Gleis	Produkte
Karlsruhe Hbf	Di, 10.01.12	ab 18:01	3	ICE 670
Berlin-Spandau	Di, 10.01.12	an 23:11	6	Intercity-Express Bordrestaurant
<input type="checkbox"/> Umsteigezeit 7 Min.				<a href="#">→ Umsteigezeit anpassen</a>
Berlin-Spandau	Di, 10.01.12	ab 23:18	2	S 5
Berlin Savignyplatz	Di, 10.01.12	an 23:36	3	S-Bahn Richtung: Berlin-Mahlsdorf Kurzfristige Änderungen möglich: Details unter <a href="http://www.s-bahn-berlin.de">www.s-bahn-berlin.de</a> , Fahrradmitnahme begrenzt möglich, nur 2. Klasse

Zwischenhalte einblenden

fährt täglich, nicht 14. Jan bis 2. Feb 2012, 25. Feb bis 4. Mär 2012 [Merken](#) [In Kalender eintragen](#) [Druckansicht](#)

[Am Bahnhof](#) [Karte anzeigen](#)

# Routenplanung in der Anwendung



## Wichtiger Anwendungsbereich

- Navigationssysteme
- Kartendienste: Google Maps, Bing Maps, ...
- Fahrplanauskunftssysteme



## Viele kommerzielle Systeme

- Nutzen **heuristische** Methoden
- Betrachten nur Teile des Transportnetzwerkes
- Geben keine **Qualitätsgarantien**

Wir untersuchen Methoden zur **Routenplanung** in Transportnetzwerken mit **beweisbar optimalen** Lösungen

# Problemstellung

## Anfrage:

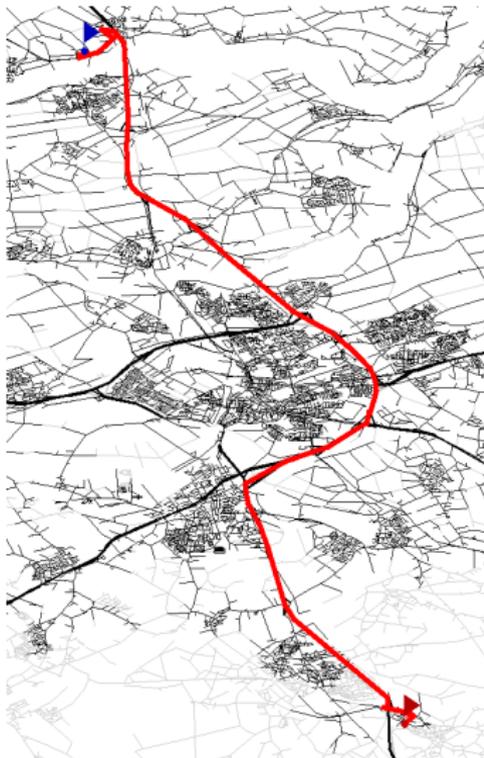
- Finde **beste** Verbindung in Transportnetz

## Modellierung:

- Netzwerk als Graph  $G = (V, E)$
- Kantengewichte sind **Reisezeit**
- **Kürzeste** Wege in  $G$  entsprechen **schnellsten** Verbindungen
- Klassisches Problem (Dijkstra [Dij59])

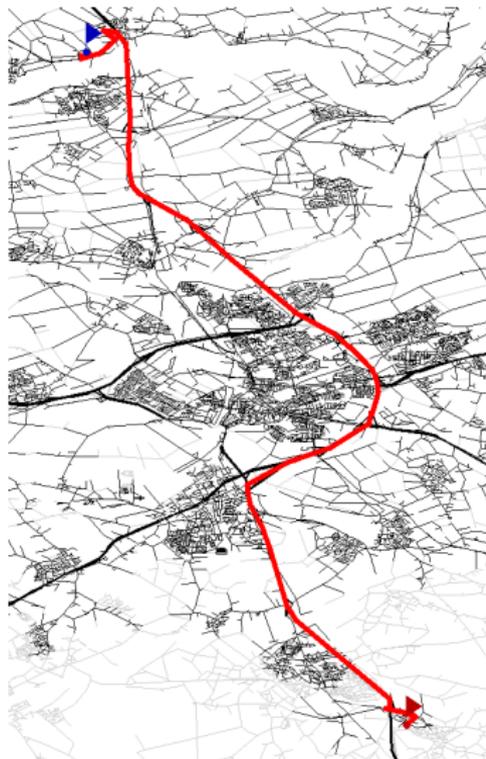
## Probleme:

- Transportnetzwerke sind **riesig**
- Dijkstras Algorithmus zu **langsam**  
( $> 1$  Sekunde)



## Beobachtung:

- Viele Anfragen in (statischem) Netzwerk
- „Unnötige“ Berechnungen

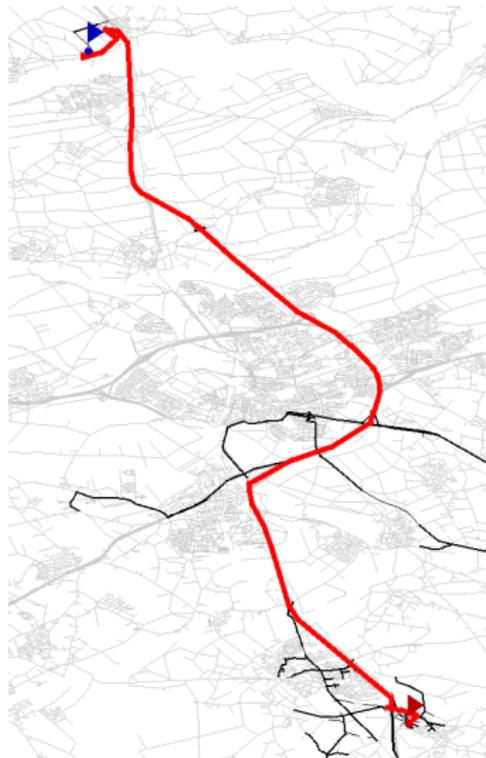


## Beobachtung:

- Viele Anfragen in (statischem) Netzwerk
- „Unnötige“ Berechnungen

## Idee:

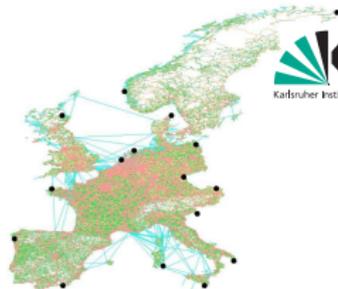
- Zwei Phasen
  - Offline: Generiere Zusatzinformation in **Vorbereitung**
  - Online: **Beschleunige** Anfrage mit dieser Information
- Drei Kriterien: Vorberechnungsplatz, Vorberechnungszeit, Beschleunigung



# Experimentelle Evaluation

**Eingabe:** Straßennetzwerk von Europa

- ca. 18 Mio. Knoten
- ca. 42 Mio. Kanten

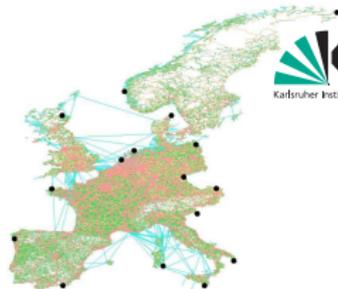


Algorithmus	VORBERECHNUNG		ANFRAGEN	
	Zeit [h:m]	Platz [GiB]	Zeit [ $\mu$ s]	Beschleun.
Dijkstra <small>[Dij59]</small>	—	—	2 550 000	—
ALT <small>[GH05, GW05]</small>	0:42	2.2	24 521	104
CRP <small>[DGPW11]</small>	≪ 0:01	< 0.1	1 650	1 545
Arc-Flags <small>[Lau04]</small>	0:20	0.3	408	6 250
CH <small>[GSSD08]</small>	0:05	0.2	110	23 181
TNR <small>[ALS13]</small>	0:20	2.1	1.25	2 040 000
HL <small>[ADGW12]</small>	0:37	18.4	0.56	4 553 571

# Experimentelle Evaluation

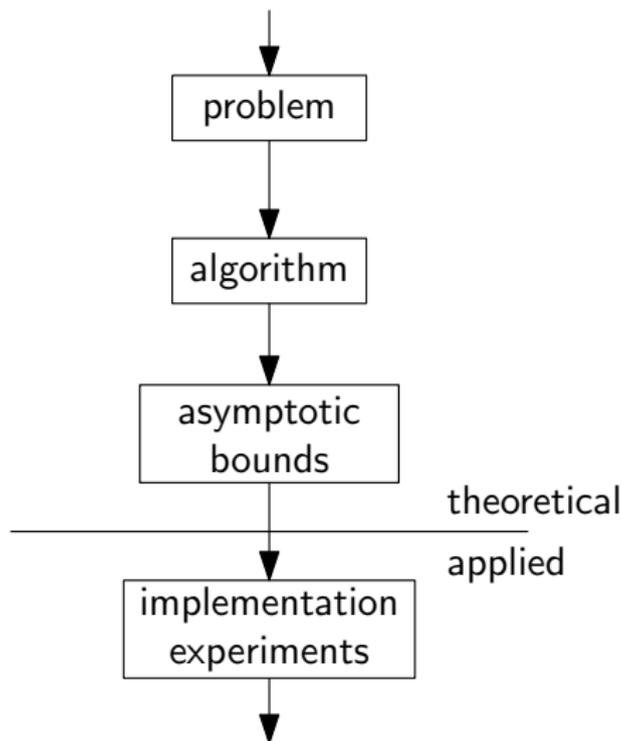
**Eingabe:** Straßennetzwerk von Europa

- ca. 18 Mio. Knoten
- ca. 42 Mio. Kanten



Algorithmus	VORBERECHNUNG		ANFRAGEN	
	Zeit [h:m]	Platz [GiB]	Zeit [ $\mu$ s]	Beschleun.
Dijkstra <small>[Dij59]</small>	—	—	2 550 000	—
ALT <small>[GH05, GW05]</small>	0:42	2.2	24 521	104
CRP <small>[DGPW11]</small>	$\ll$ 0:01	$<$ 0.1	1 650	1 545
Arc-Flags <small>[Lau04]</small>	0:20	0.3	408	6 250
CH <small>[GSSD08]</small>	0:05	0.2	110	23 181
TNR <small>[ALS13]</small>	0:20	2.1	1.25	2 040 000
HL <small>[ADGW12]</small>	0:37	18.4	0.56	4 553 571

Mittlerweile im Einsatz bei Bing, Google, Tomtom, ...





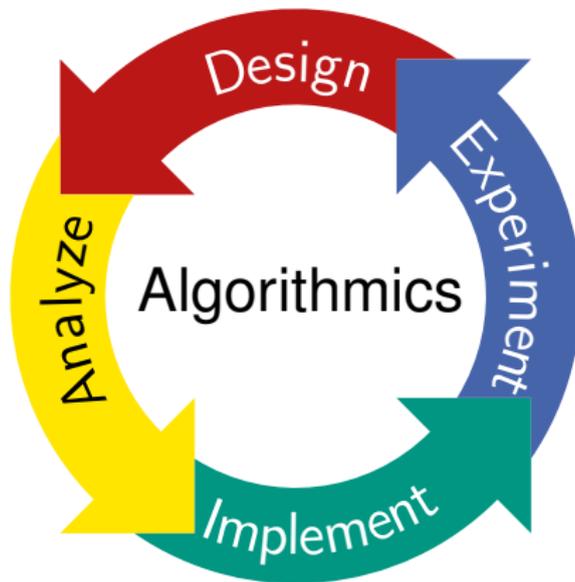
# Lücke Theorie vs. Praxis

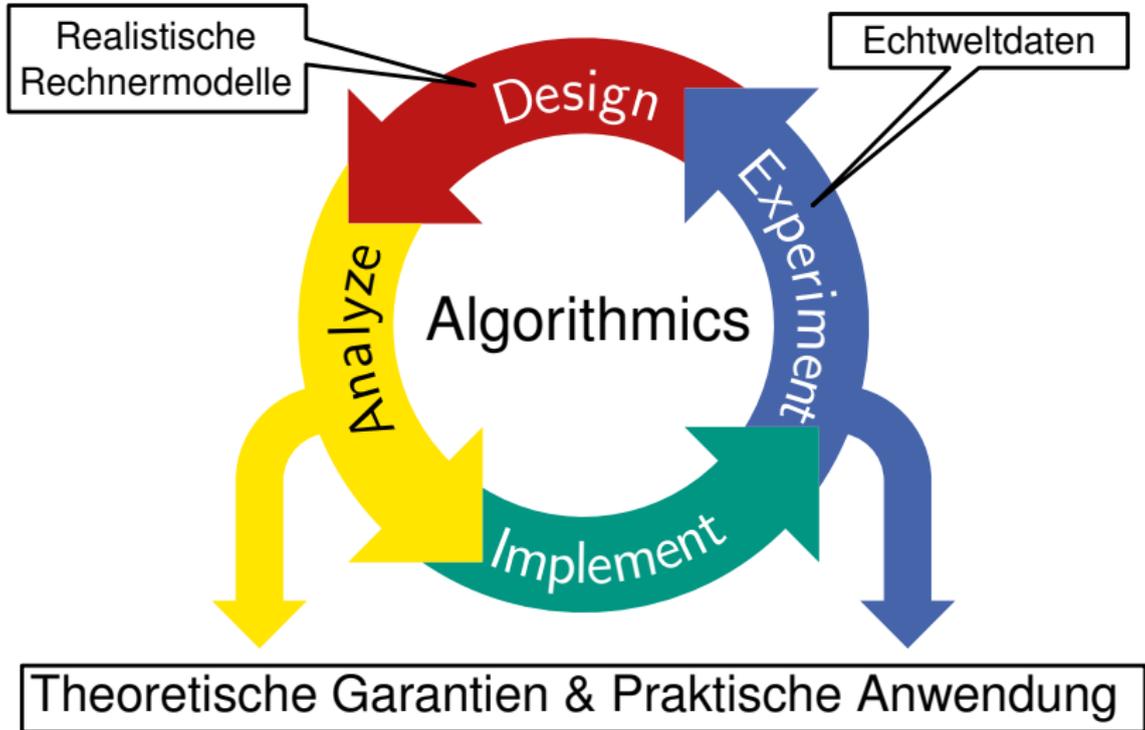
Theorie	vs.	Praxis
einfach einfach	Problem-Modell Maschinenmodell	komplex komplex
komplex fortgeschritten	Algorithmen Datenstrukturen	einfach einfach
worst-case asymptotisch	Komplexitäts-Messung Effizienz	typische Eingaben konstante Faktoren

Theorie	vs.	Praxis
einfach einfach	Problem-Modell Maschinenmodell	komplex komplex
komplex fortgeschritten	Algorithmen Datenstrukturen	einfach einfach
worst-case asymptotisch	Komplexitäts-Messung Effizienz	typische Eingaben konstante Faktoren

## Routenplanung:

- sehr anwendungsnahes Gebiet
- Eingaben sind **echte** Daten
  - Straßengraphen
  - Eisenbahn (Fahrpläne)
  - Flugpläne





- Algorithm Engineering + ein bisschen Theorie
- Beschleunigungstechniken
- Implementierungsdetails
- Ergebnisse auf Real-Welt Daten
- aktuellster Stand der Forschung (Veröffentlichungen bis 2015)
- ideale Grundlage für Bachelor, Master und Diplomarbeiten

## keine Algorithmen III

- Vertiefung von kürzesten Wegen (Dijkstra)
  - Grundlagen sind Stoff von Algo I/II;  
heute nochmal Crashkurs
- Grundvorlesung “vereinfachen” Wahrheit oft
- Implementierung
- Betonung auf Messergebnisse

## keine reine Theorievorlesung

- relativ wenig Beweise (wenn doch, eher kurz)
- reale Leistung vor Asymptotik
- Vielen vorkommende Optimierungsproblemen sind  $\mathcal{NP}$ -schwer

## 1. Grundlagen

- Algorithm Engineering
- Graphen, Modelle, usw.
- Kürzeste Wege
- Dijkstra's Algorithmus

## 2. Beschleunigung von (statischen) Punkt-zu-Punkt Anfragen

- zielgerichtete Verfahren
- hierarchische Techniken
- many-to-many-Anfragen und Distanztabelle
- Kombinationen

## 3. Theorie

- Theoretische Charakterisierung von Straßennetzwerken
- Highway-Dimension
- Komplexität von Beschleunigungstechniken

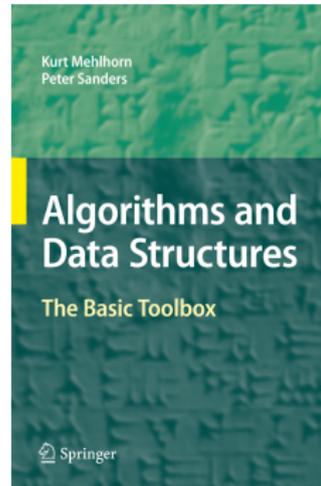
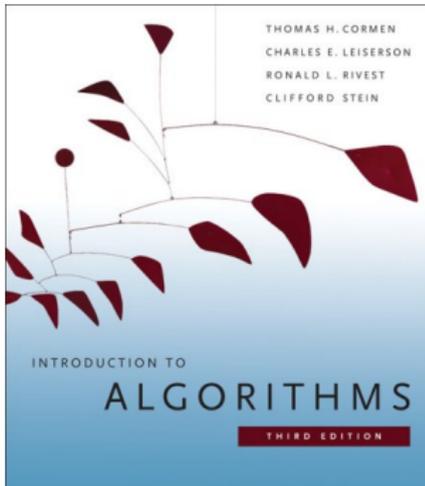
## 4. Fortgeschrittene Szenarien

- schnelle many-to-many und all-pairs shortest-paths
- Alternativrouten
- zeitabhängige Routenplanung
- Fahrplanauskunft
- multi-modale Routenplanung

- Informatik I/II oder Algorithmen I
- Algorithmentechnik oder Algorithmen II (muss aber nicht sein)
- ein bisschen Rechnerarchitektur
- passive Kenntnisse von C++/Java

**Vertiefungsgebiet:** Algorithmentechnik, (Theoretische Grundlagen)

- Folien
- wissenschaftliche Aufsätze (siehe Vorlesunghomepage)
- Basiskenntnisse:

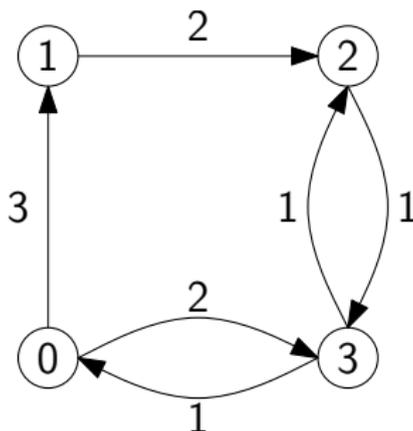


# 1. Grundlagen



## Drei klassische Ansätze:

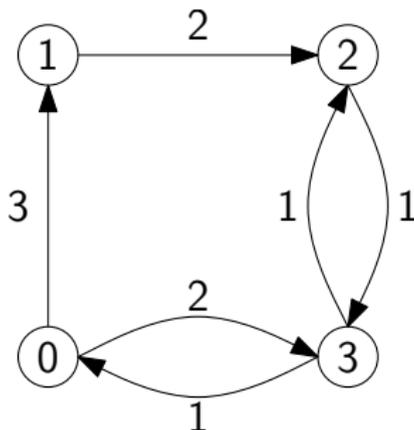
- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray



## Drei klassische Ansätze:

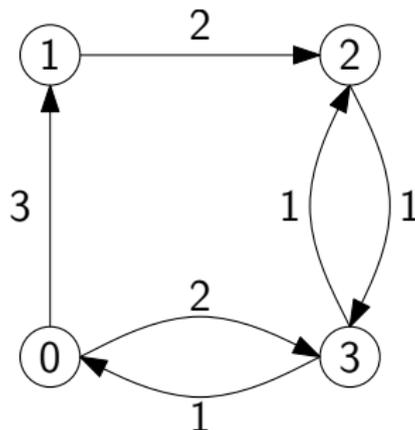
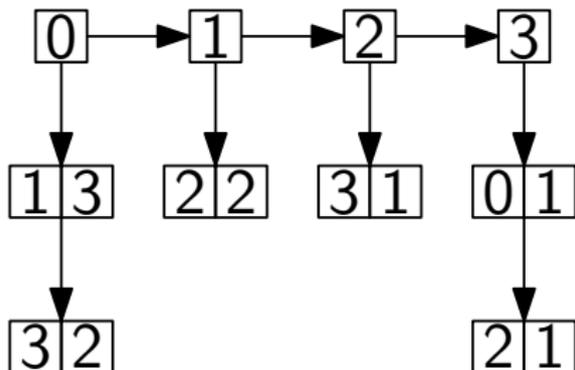
- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray

	0	1	2	3
0	—	3	—	2
1	—	—	2	—
2	—	—	—	1
3	1	—	1	—



## Drei klassische Ansätze:

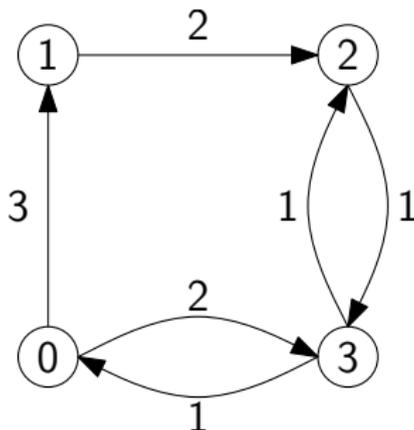
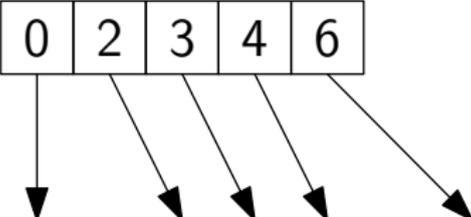
- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray



## Drei klassische Ansätze:

- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray

firstEdge	0	2	3	4	6
targetNode	1	3	2	3	2
weight	3	2	2	1	1



# Was brauchen wir?

Eigenschaften:	Matrix	Liste	Array
Speicher	$\mathcal{O}(n^2)$	$\mathcal{O}(n + m)$	$\mathcal{O}(n + m)$
Ausgehende Kanten iterieren	$\mathcal{O}(n)$	$\mathcal{O}(\deg u)$	$\mathcal{O}(\deg u)$
Kantenzugriff $(u, v)$	$\mathcal{O}(1)$	$\mathcal{O}(\deg u)$	$\mathcal{O}(\deg u)$
Effizienz (Speicherlayout)	+	-	+
Updates (neue Kante)	+	+	-
Updates (Gewicht)	+	+	+

Eigenschaften:	Matrix	Liste	Array
Speicher	$\mathcal{O}(n^2)$	$\mathcal{O}(n + m)$	$\mathcal{O}(n + m)$
Ausgehende Kanten iterieren	$\mathcal{O}(n)$	$\mathcal{O}(\deg u)$	$\mathcal{O}(\deg u)$
Kantenzugriff $(u, v)$	$\mathcal{O}(1)$	$\mathcal{O}(\deg u)$	$\mathcal{O}(\deg u)$
Effizienz (Speicherlayout)	+	-	+
Updates (neue Kante)	+	+	-
Updates (Gewicht)	+	+	+

## Fragen:

- Was brauchen wir?
- Was muss nicht super effizient sein?
- erstmal Modelle anschauen!

# Modellierung (Straßengraphen)

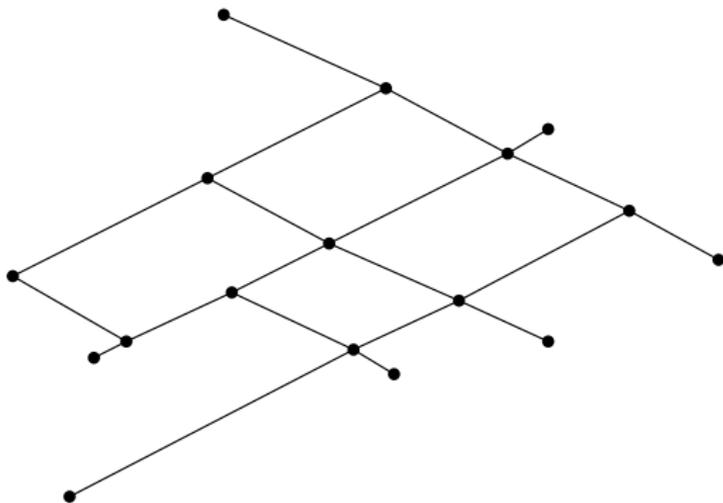


# Modellierung (Straßengraphen)



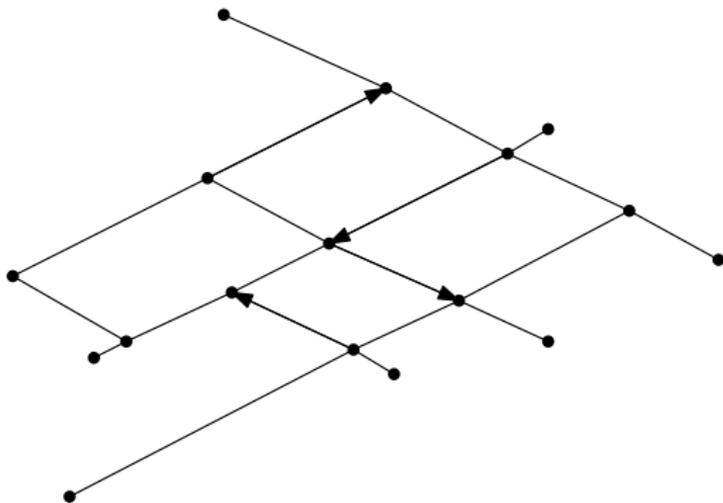
# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen



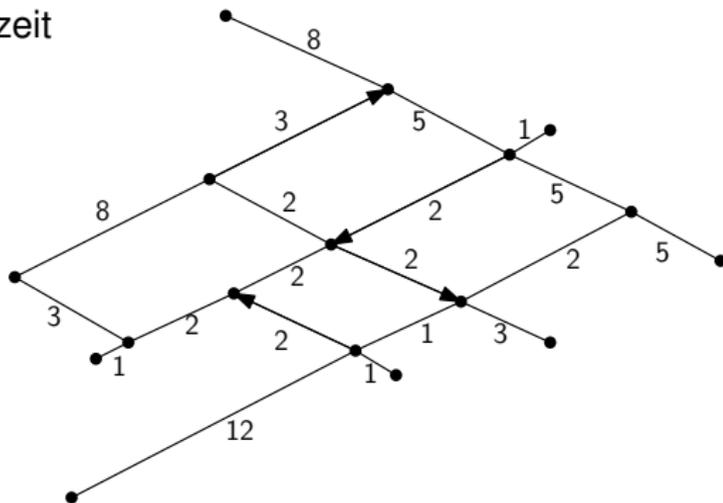
# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen
- Einbahnstraßen

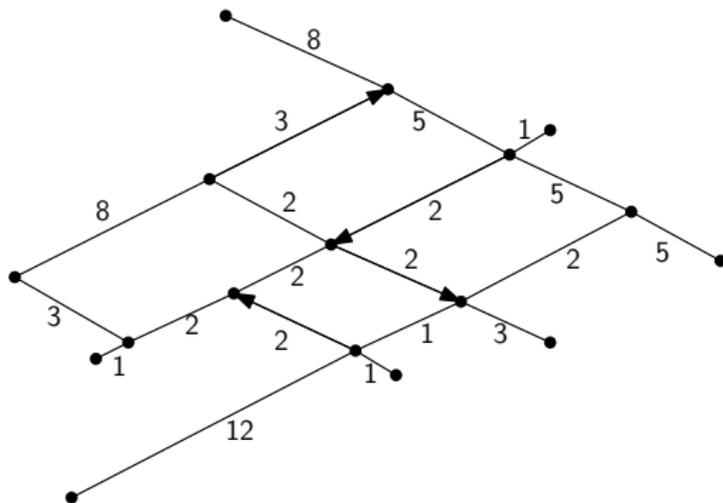


# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen
- Einbahnstraßen
- Metrik ist Reisezeit

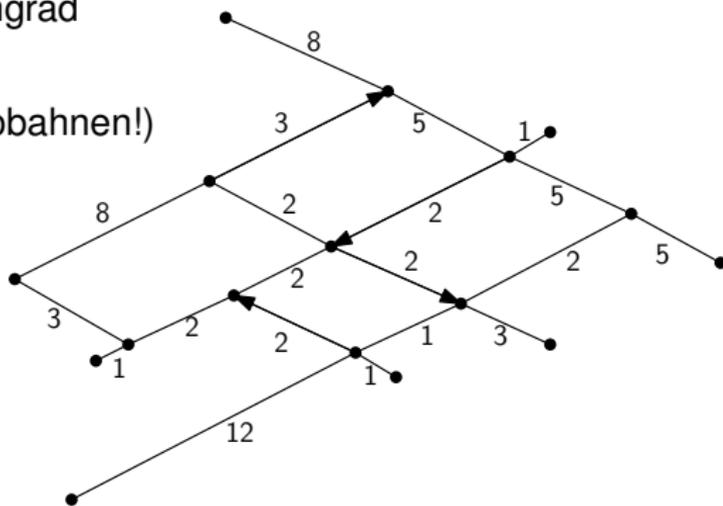


Eigenschaften (sammeln):



## Eigenschaften:

- dünn
- (fast) ungerichtet
- geringer Knotengrad
- Kantenzüge
- Hierarchie (Autobahnen!)

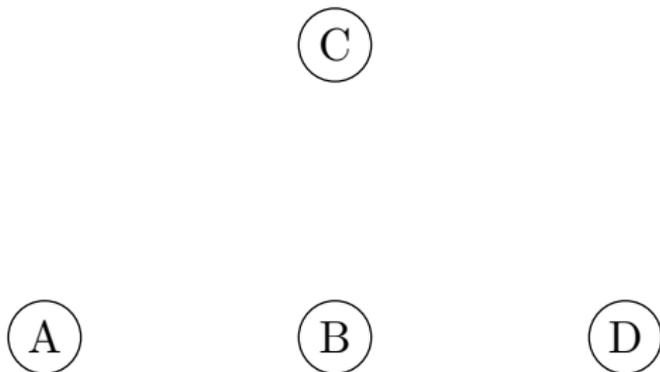


## Beispiel:

- 4 Stationen (A,B,C,D)
- **Zug 1:** Station A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  A
- **Zug 2:** Station A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  C  $\rightarrow$  A
- Züge operieren alle 10 Minuten

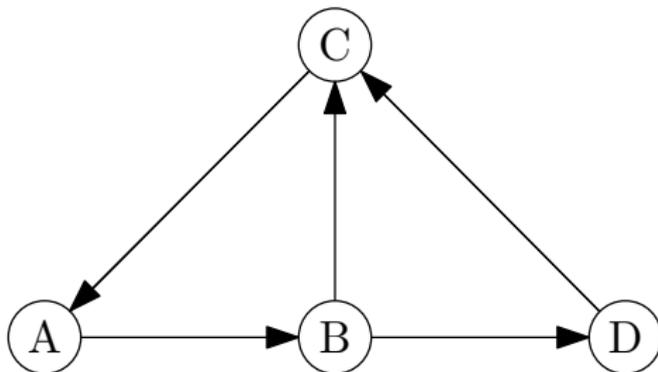
## Beispiel:

- 4 Stationen (A,B,C,D)
- Zug 1: Station A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  A
- Zug 2: Station A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  C  $\rightarrow$  A
- Züge operieren alle 10 Minuten



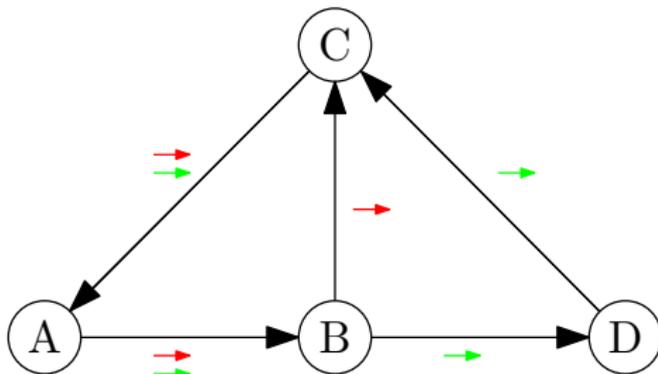
## Beispiel:

- 4 Stationen (A,B,C,D)
- **Zug 1:** Station A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  A
- **Zug 2:** Station A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  C  $\rightarrow$  A
- Züge operieren alle 10 Minuten



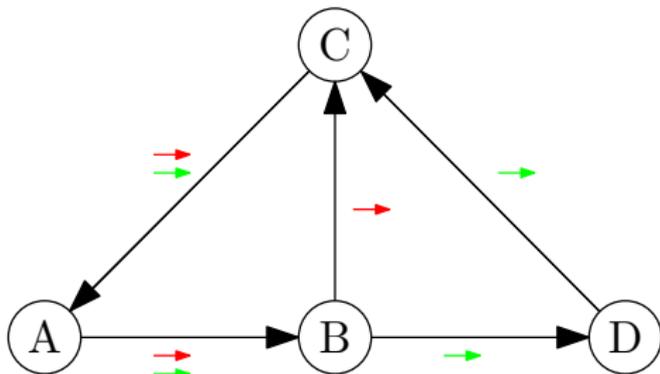
## Beispiel:

- 4 Stationen (A,B,C,D)
- Zug 1: Station A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  A
- Zug 2: Station A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  C  $\rightarrow$  A
- Züge operieren alle 10 Minuten



## Beispiel:

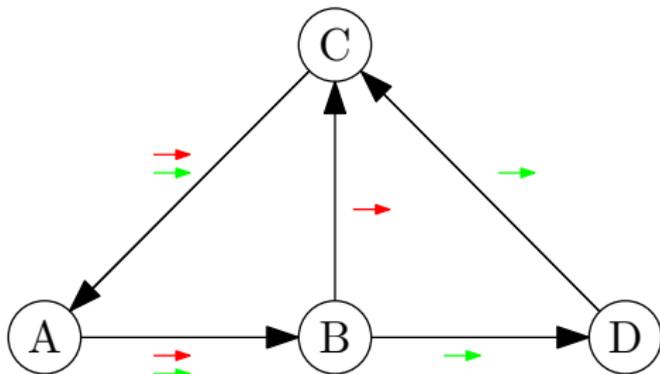
- 4 Stationen (A,B,C,D)
- **Zug 1:** Station A → B → C → A
- **Zug 2:** Station A → B → D → C → A
- Züge operieren alle 10 Minuten



Kanten sind zeitabhängig!  
(wann fährt ein Zug wie lange?)

## Beispiel:

- 4 Stationen (A,B,C,D)
- Zug 1: Station A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  A
- Zug 2: Station A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  C  $\rightarrow$  A
- Züge operieren alle 10 Minuten



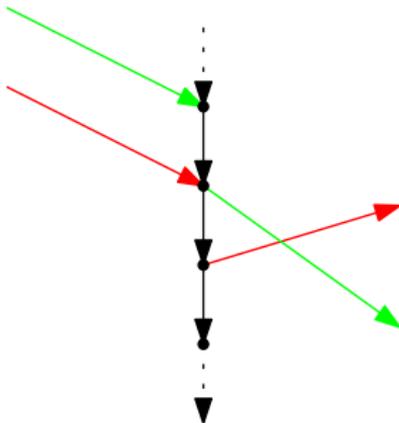
Kanten sind zeitabhängig!  
(wann fährt ein Zug wie lange?)

oder roll die Zeit aus

## Vorgehen:

- Knoten sind Abfahrts-/Ankunftsereignisse
- Kanten für die Fahrt von Station zu Station
- Wartekanten an den Stationen für Umstiege

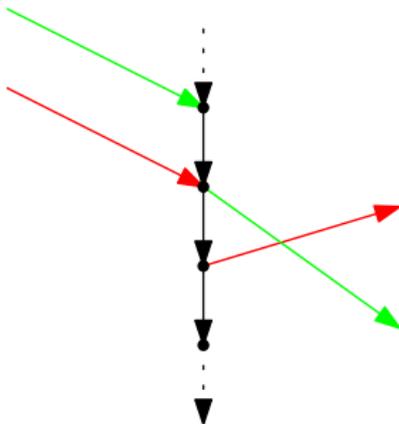
## Station B:



## Vorgehen:

- Knoten sind Abfahrts-/Ankunftsereignisse
- Kanten für die Fahrt von Station zu Station
- Wartekanten an den Stationen für Umstiege

## Station B:



## Diskussion:

- + keine zeitabhängigen Kanten
- Graph größer

- zusammenhängend
- dünn
- gerichtet
- geringer Knotengrad
- meist verborgene Hierarchie (Autobahnen, ICE)
- Einbettung vorhanden (fast planar?)
- Kantengewichte nicht-negativ
- teilweise zeitabhängig
- dünne Separatoren?

- zusammenhängend
- dünn
- gerichtet
- geringer Knotengrad
- meist verborgene Hierarchie (Autobahnen, ICE)
- Einbettung vorhanden (fast planar?)
- Kantengewichte nicht-negativ
- teilweise zeitabhängig
- dünne Separatoren?

## Diskussion:

- berechne beste Verbindungen in solchen Netzwerken
- Adjazenzarray als Graphdatenstruktur

# 2. Kürzeste Wege

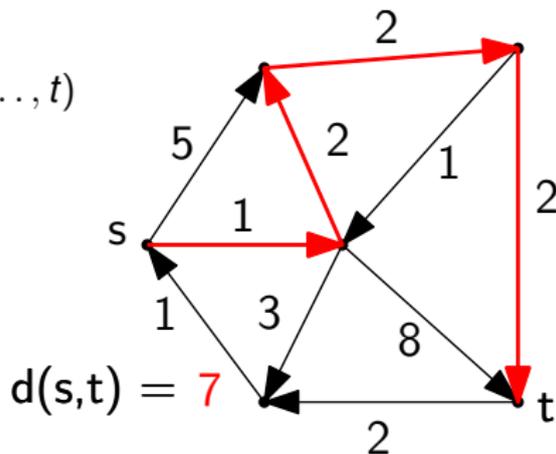


## Gegeben:

Graph  $G = (V, E, \text{len})$  mit positiver Kantenfunktion  $\text{len}: E \rightarrow \mathbb{R}_{\geq 0}$ ,  
Knoten  $s, t \in V$

## Mögliche Aufgaben

- Berechne Distanz  $d(s, t)$
- Finde kürzesten  $s$ - $t$ -Pfad  $P := (s, \dots, t)$



## Azyklichkeit:

- Kürzeste Wege sind zyklenfrei

## Aufspannungseigenschaft:

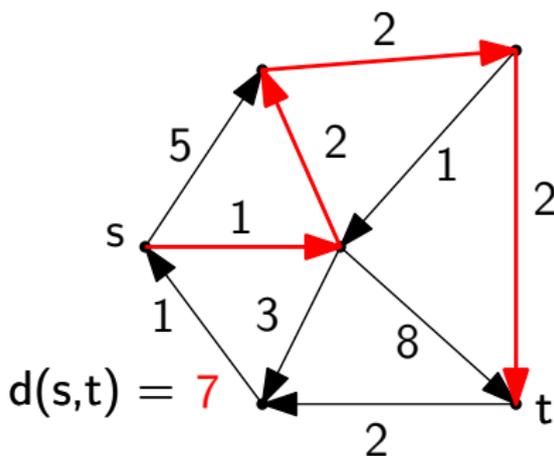
- Alle kürzesten Wege von  $s$  aus bilden DAG bzw. Baum

## Vererbungseigenschaft:

- Teilwege von kürzesten Wegen sind kürzeste Wege

## Abstand:

- Steigender Abstand von Wurzel zu Blättern



Komplexität von **Single-Source Shortest Paths** abhängig vom Eingabegraphen.

- In dieser Vorlesung: Kantengewichte (*fast*) immer nicht-negativ.
- Ein *negativer Zyklus* ist ein Kreis mit negativem Gesamtgewicht.
- Ein *einfacher Pfad* ist ein Pfad bei dem sich kein Knoten wiederholt.

## Problemvarianten

- Kantengewichte alle positiv:  
Dijkstra's Algorithmus anwendbar (Laufzeit  $|V| \log |V| + |E|$ )
- Kantengewichte auch negativ, aber kein negativer Zyklus:  
Algorithmus von Bellmann-Ford anwendbar (Laufzeit  $|V| \cdot |E|$ )
- Kantengewichte auch negativ, suche kürzesten einfachen Pfad:  
 $\mathcal{NP}$ -schwer, Reduktion von „Problem Longest Path“

---

Bellman-Ford( $G, s$ )

---

```
1 for  $v \in V$  do  $d[v] \leftarrow \infty$ 
2  $d[s] \leftarrow 0$ 
3 for  $i = 1$  to  $|V| - 1$  do
4   forall the edges  $(u, v) \in E$  do
5     if  $d[u] + \text{len}(u, v) < d[v]$  then
6        $d[v] \leftarrow d[u] + \text{len}(u, v)$ 
```

---

Bellman-Ford( $G, s$ )

---

```
1 for  $v \in V$  do  $d[v] \leftarrow \infty$ 
2  $d[s] \leftarrow 0$ 
3 for  $i = 1$  to  $|V| - 1$  do
4   forall the edges  $(u, v) \in E$  do
5     if  $d[u] + \text{len}(u, v) < d[v]$  then
6        $d[v] \leftarrow d[u] + \text{len}(u, v)$ 
7 forall the edges  $(u, v) \in E$  do
8   if  $d[v] > d[u] + \text{len}(u, v)$  then
9     negative cycle found
```

---

## Problem Longest Path

### Gegeben:

- Gerichteter, gewichteter Graph  $G = (V, E)$  mit gewicht  $len: E \rightarrow \mathbb{N}$
- Zahl  $K \in \mathbb{N}$
- Knoten  $s, t \in V$

### Frage:

- Gibt es einen einfachen  $s$ - $t$ -Pfad der Länge mindestens  $K$ ?

Problem Longest Path ist  $\mathcal{NP}$ -schwer (siehe [Garey & Johnson 79])

---

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

---

```
1 forall the nodes  $v \in V$  do
2    $d[v] = \infty, p[v] = \text{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.\text{clear}(), Q.\text{insert}(s, 0)$            // container
5 while  $!Q.\text{empty}()$  do
6    $u \leftarrow Q.\text{deleteMin}()$            // settling node u
7   forall the edges  $e = (u, v) \in E$  do
8     // relaxing edges
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      else  $Q.\text{insert}(v, d[v])$ 
```

---

# Dijkstras Algorithmus

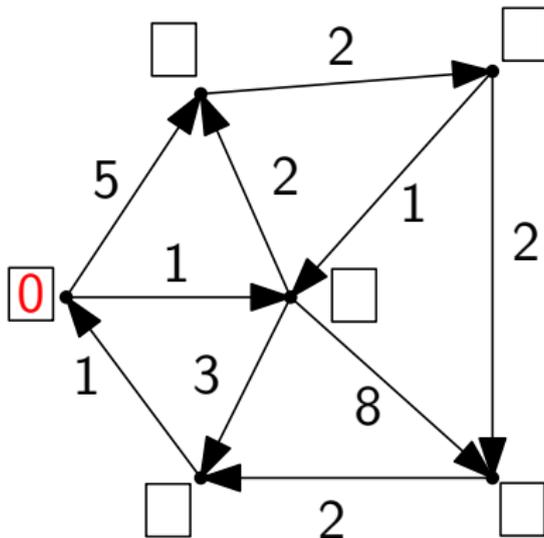
**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

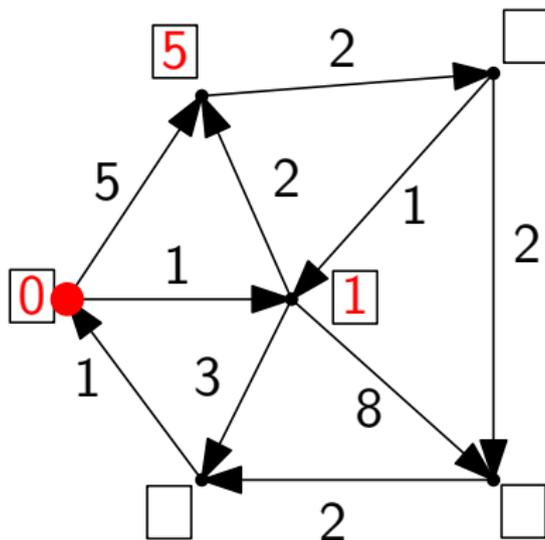


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

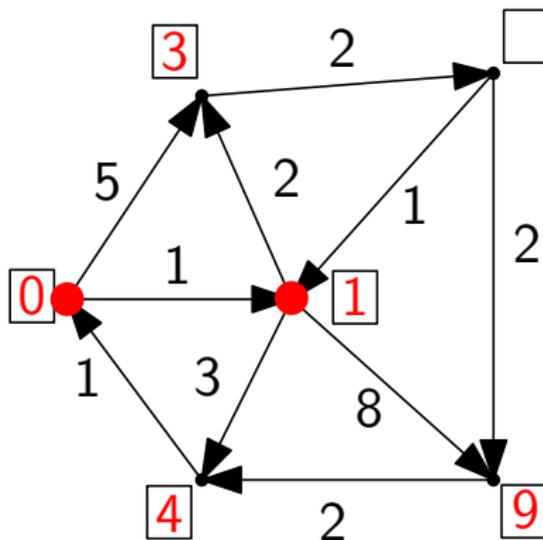


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

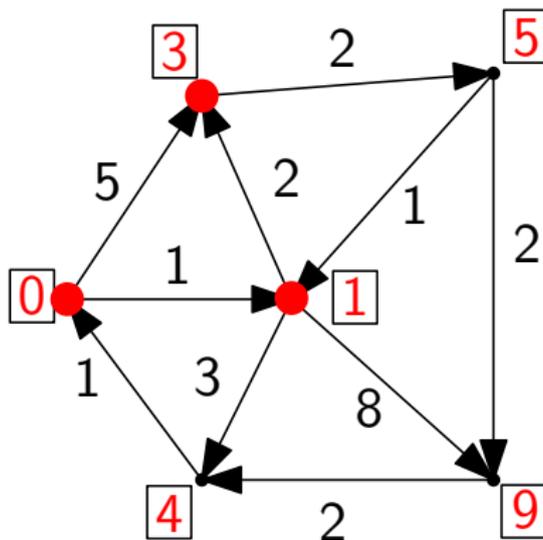


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

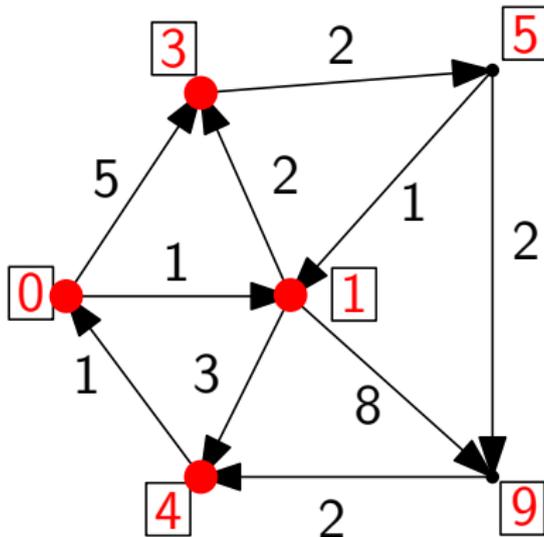


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

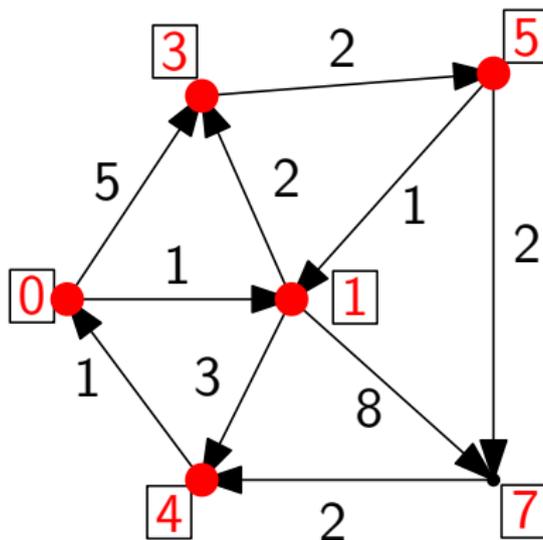


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

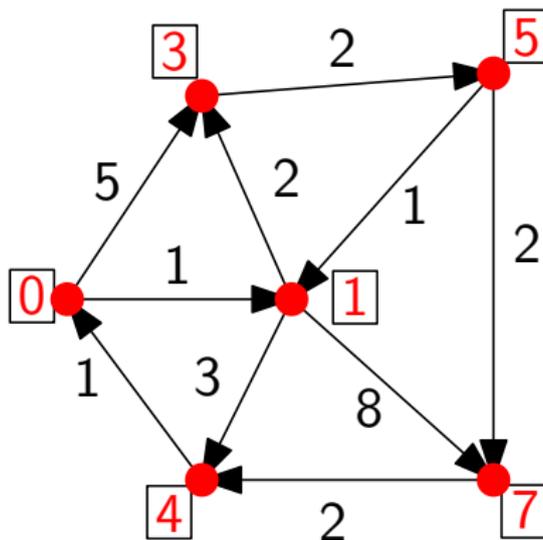


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .

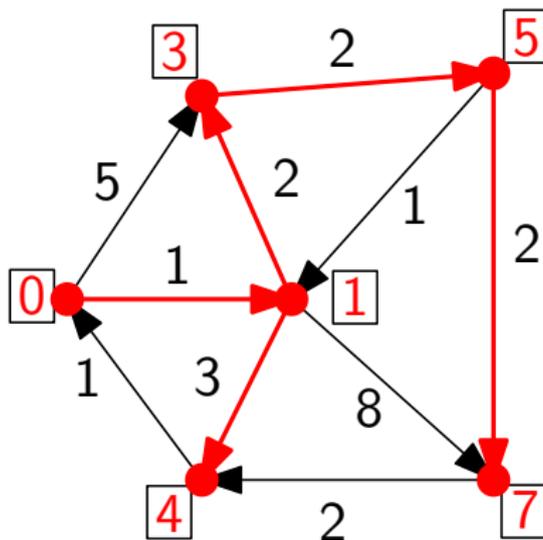


**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

# Dijkstras Algorithmus

**Gegeben:** Graph  $G$ , Startknoten.

**Idee:** Suche in  $G$  mit zunehmender *Distanz* von  $s$ .



**Invariante:** Pfade zu *roten* Knoten sind **optimal**.

---

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

---

```
1 forall the nodes  $v \in V$  do
2    $d[v] = \infty$ ,  $p[v] = \text{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.\text{clear}()$ ,  $Q.\text{insert}(s, 0)$            // container
5 while  $!Q.\text{empty}()$  do
6    $u \leftarrow Q.\text{deleteMin}()$            // settling node  $u$ 
7   forall the edges  $e = (u, v) \in E$  do
8     // relaxing edges
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      else  $Q.\text{insert}(v, d[v])$ 
```

**Kantenrelaxierung:** Der Vorgang

1 **if**  $d[u] + \text{len}(u, v) < d[v]$  **then**  $d[v] \leftarrow d[u] + \text{len}(u, v)$

heißt *Kantenrelaxierung*.

**Kantenrelaxierung:** Der Vorgang

1 **if**  $d[u] + \text{len}(u, v) < d[v]$  **then**  $d[v] \leftarrow d[u] + \text{len}(u, v)$

heißt *Kantenrelaxierung*.

**Besuchte Knoten:** Ein Knoten heißt (zu einem Zeitpunkt) *besucht* (*visited*) wenn er (zu diesem Zeitpunkt) schon in die Queue eingefügt wurde (unabhängig davon, ob er noch in der Queue ist).

**Kantenrelaxierung:** Der Vorgang

1 **if**  $d[u] + \text{len}(u, v) < d[v]$  **then**  $d[v] \leftarrow d[u] + \text{len}(u, v)$

heißt *Kantenrelaxierung*.

**Besuchte Knoten:** Ein Knoten heißt (zu einem Zeitpunkt) *besucht* (*visited*) wenn er (zu diesem Zeitpunkt) schon in die Queue eingefügt wurde (unabhängig davon, ob er noch in der Queue ist).

**Abgearbeitete Knoten:** Ein Knoten heißt (zu einem Zeitpunkt) *abgearbeitet* (*settled*) wenn er (zu diesem Zeitpunkt) schon in die Queue eingefügt und wieder extrahiert wurde.

---

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

---

```
1 forall the nodes  $v \in V$  do
2    $d[v] = \infty, p[v] = \text{NULL}$ 
3  $d[s] = 0, Q.\text{clear}(), Q.\text{insert}(s, 0)$ 
4 while ! $Q.\text{empty}()$  do
5    $u \leftarrow Q.\text{deleteMin}()$ 
6   forall the 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  $Q.\text{decreaseKey}(v, d[v])$ 
11      else  $Q.\text{insert}(v, d[v])$ 
```

---

---

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

---

```
1 forall the nodes  $v \in V$  do
2    $d[v] = \infty, p[v] = \text{NULL}$  // n Mal
3  $d[s] = 0, Q.\text{clear}(), Q.\text{insert}(s, 0)$ 
4 while ! $Q.\text{empty}()$  do
5    $u \leftarrow Q.\text{deleteMin}()$ 
6   forall the 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  $Q.\text{decreaseKey}(v, d[v])$ 
11      else  $Q.\text{insert}(v, d[v])$ 
```

---

---

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

---

```
1 forall the 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    $u \leftarrow Q.\text{deleteMin}()$ 
6   forall the 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  $Q.\text{decreaseKey}(v, d[v])$ 
11      else  $Q.\text{insert}(v, d[v])$ 
```

---

---

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

---

```
1 forall the 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 the 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  $Q.\text{decreaseKey}(v, d[v])$ 
11      else  $Q.\text{insert}(v, d[v])$ 
```

---

---

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

---

```
1 forall the 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 the 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  $Q.\text{insert}(v, d[v])$ 
```

---

---

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

---

```
1 forall the 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 the 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
```

---

---

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

---

```
1 forall the 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 the 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>	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}(m + n \log n)$

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

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

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

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:  $\approx 30$  ms  $\Rightarrow$  viel schneller

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:	$\approx 30$ ms	$\Rightarrow$ viel schneller
BFS:	$\approx 1.2$ s	$\Rightarrow$ an der Queue liegt's nicht

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

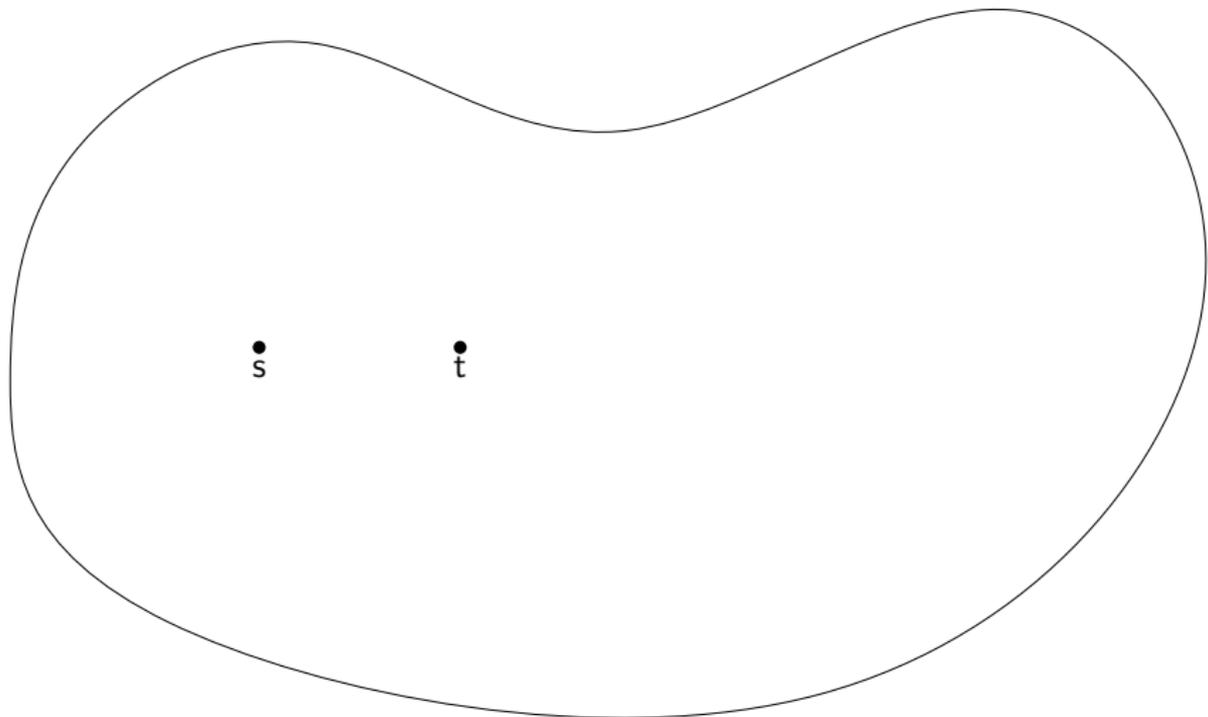
k	Query [sec]
2	1.834
3	1.595
4	1.507
5	1.525
8	1.561

Graph: 18M Knoten, 42M Kanten

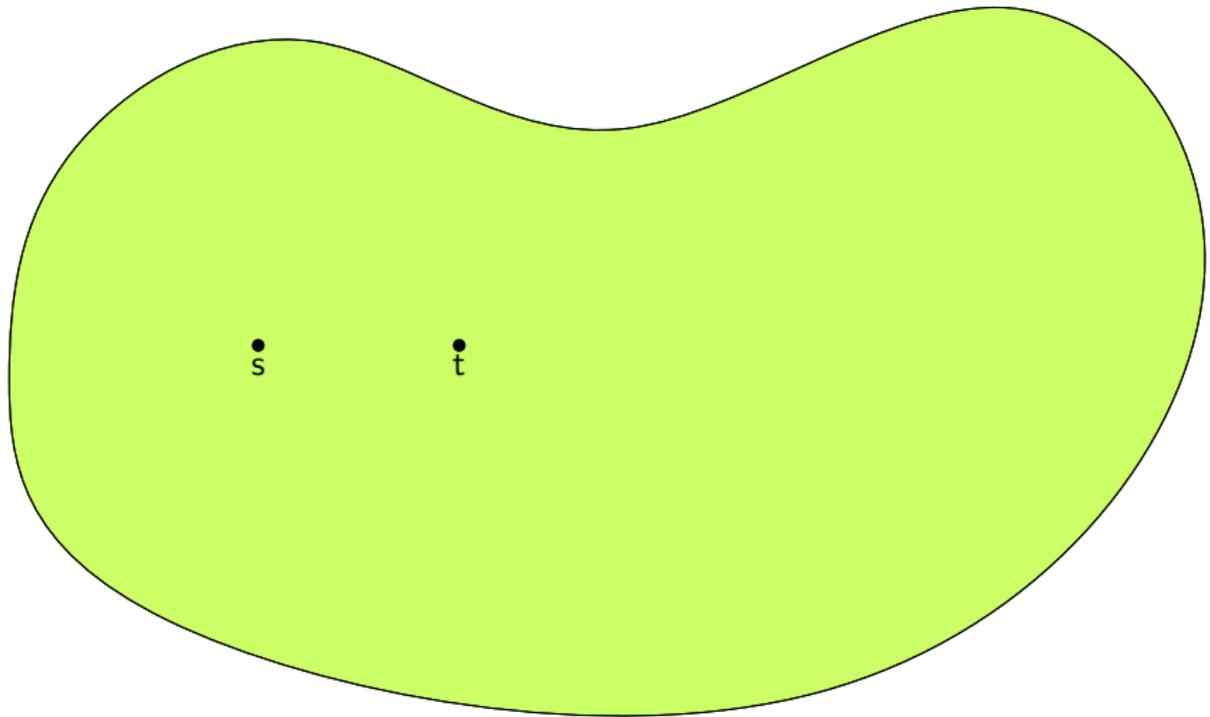
Dijkstra:	$\approx 1.5$ s	$\Rightarrow$ nicht interaktiv
$n + m$ CPU clock cycles:	$\approx 30$ ms	$\Rightarrow$ viel schneller
BFS:	$\approx 1.2$ s	$\Rightarrow$ an der Queue liegt's nicht

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

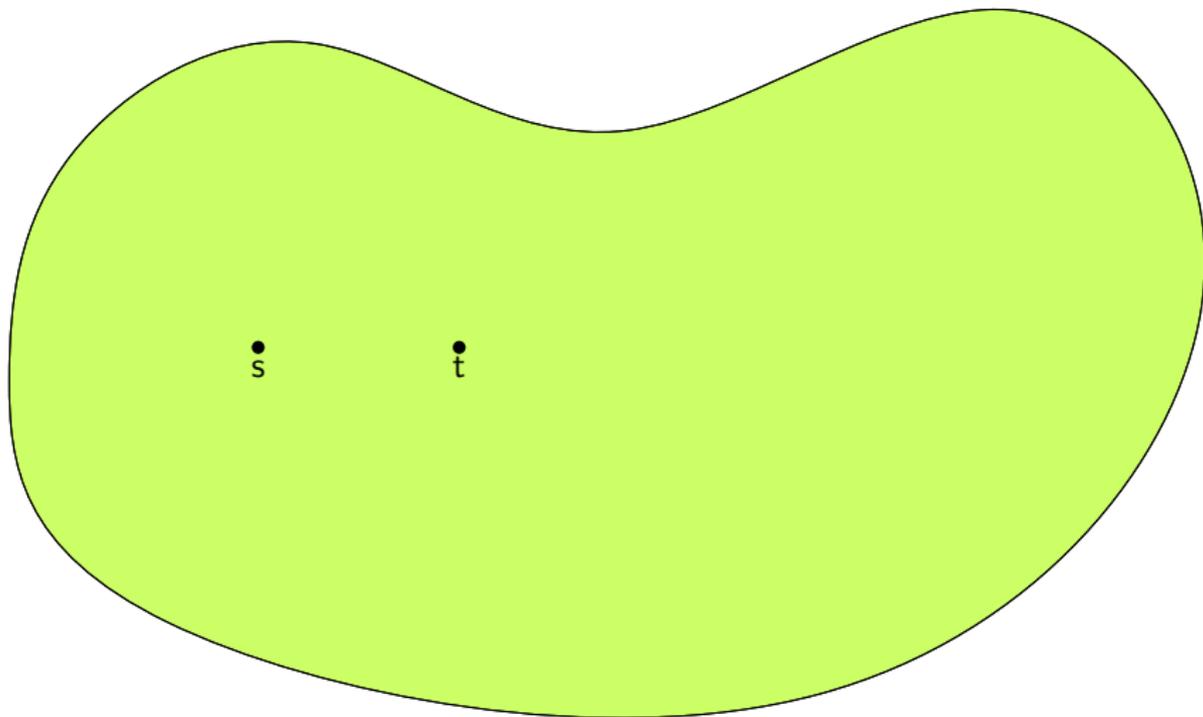
## Idee

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

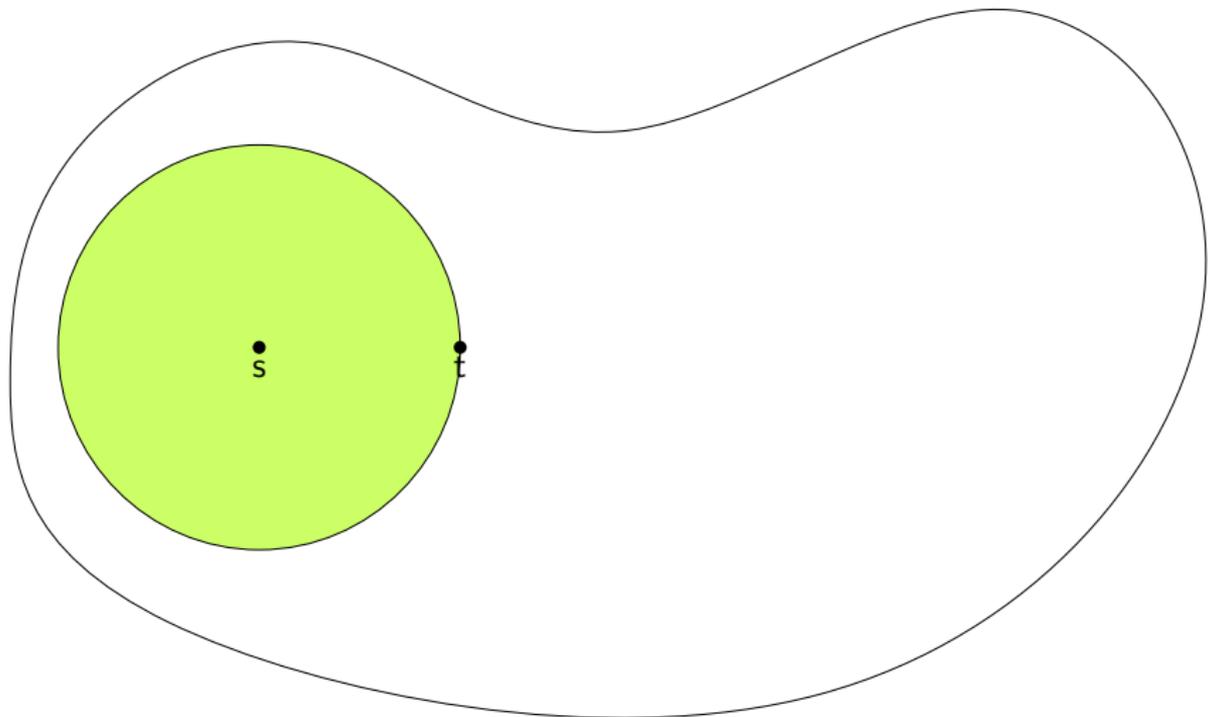
## Korrektheit

- 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 the 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 the 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 the nodes  $v \in V$  do
2    $d[v] = \infty$ ,  $p[v] = \mathbf{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.clear()$ ,  $Q.add(s, 0)$                  // container
5 while  $!Q.empty()$  do
6    $u \leftarrow Q.deleteMin()$            // settling node u
7   break if  $u = t$ 
8   forall the 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  $Q.decreaseKey(v, d[v])$ 
14    else  $Q.insert(v, d[v])$ 
```

# Dijkstra mit Abbruchkriterium

---

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

---

```
1 forall the 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 the 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.

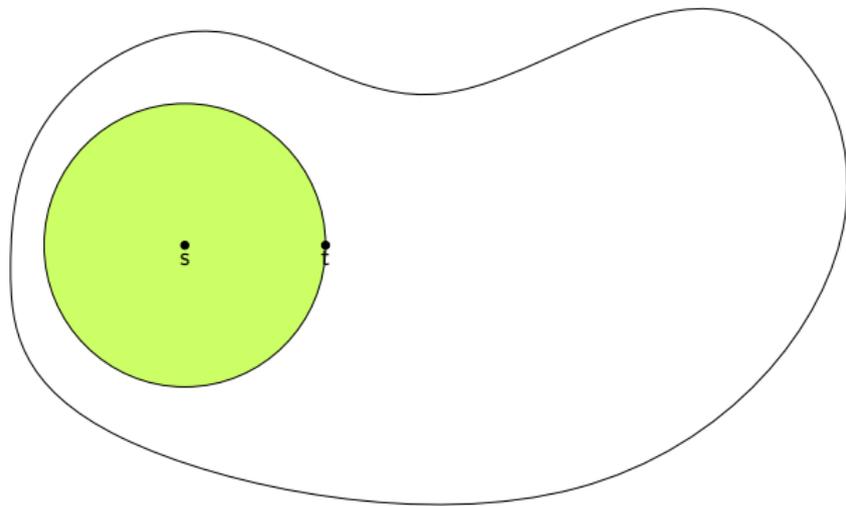
# Dijkstra mit Timestamps

// 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 the 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])
```

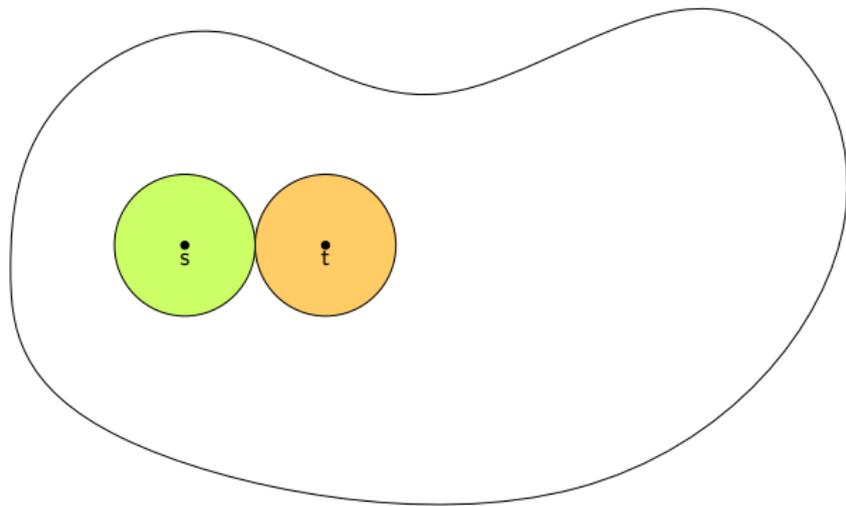
# Bidirektionale Suche





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

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

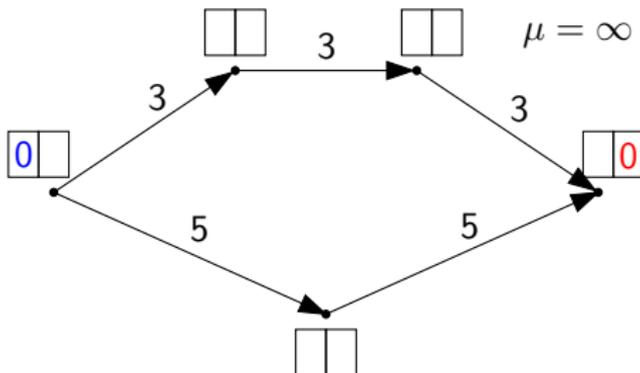


## Idee: Kombiniere beide Suchen

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

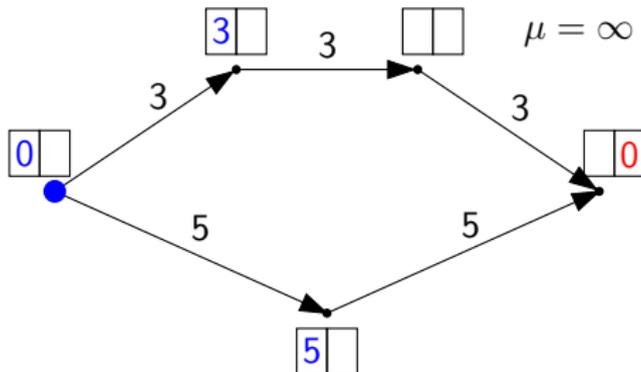
## Anfrage:

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



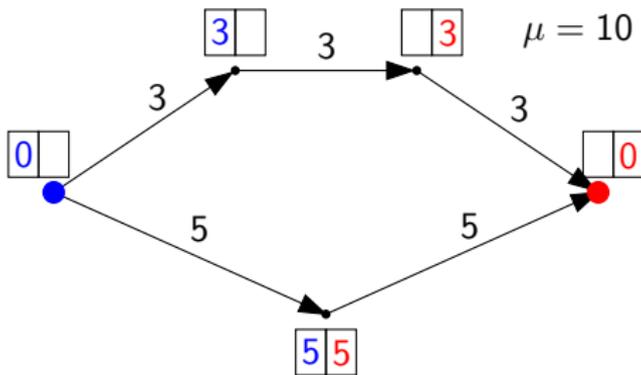
## Anfrage:

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



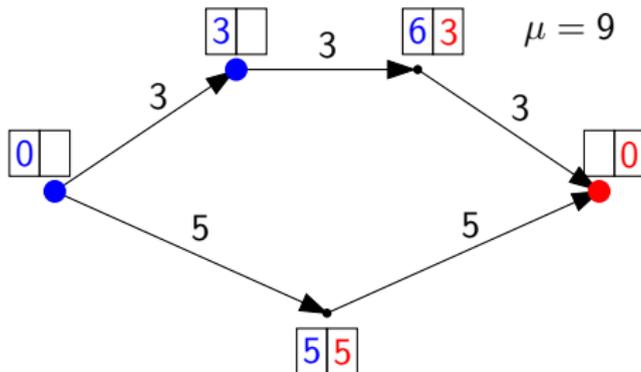
## Anfrage:

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



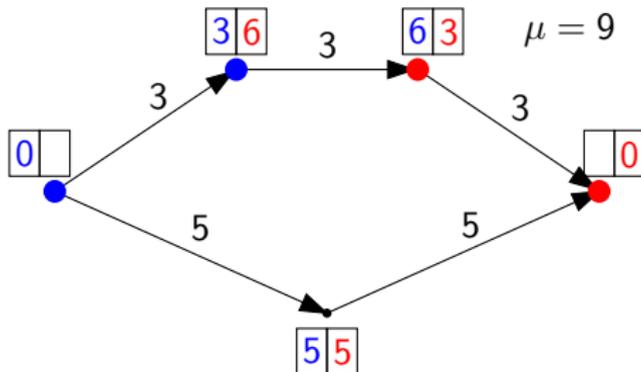
## Anfrage:

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



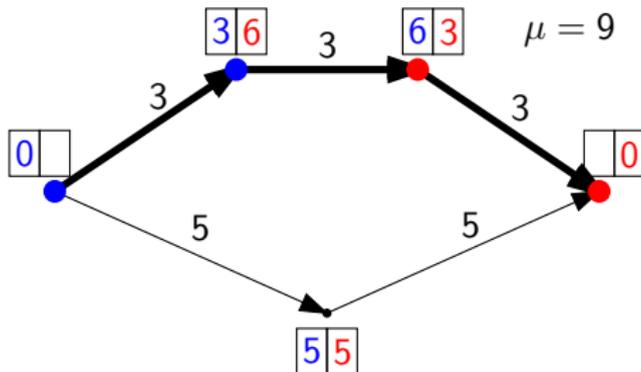
## Anfrage:

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



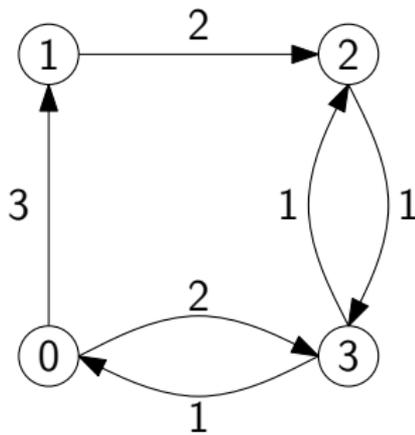
## Anfrage:

- 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

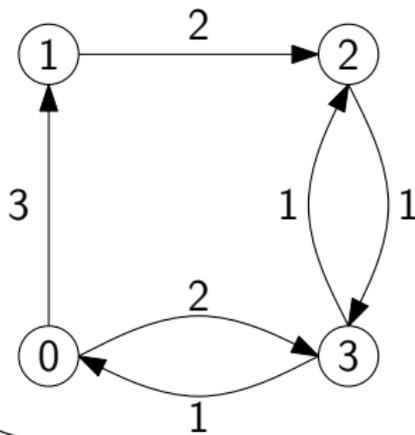
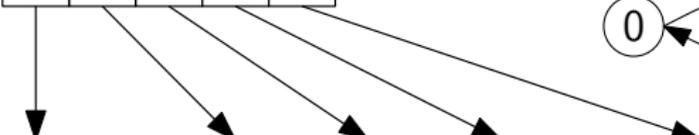
3	2	1	3	2	2	1	1	2	1
---	---	---	---	---	---	---	---	---	---

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.

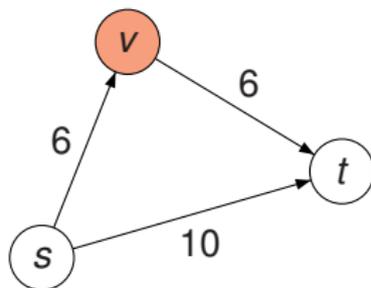
**Frage:** Ist  $v$  dann auf einem kürzesten  $s$ - $t$ -Weg enthalten?

## Abbruchstrategie (1)

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

**Frage:** Ist  $v$  dann auf einem kürzesten  $s$ - $t$ -Weg enthalten?

Nein, Gegenbeispiel:



## 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:  $\vec{d}[v] + \overleftarrow{d}[v] \geq \text{dist}(s, t)$ .
- 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:  
 $\{v_1, \dots, v_k\} \subseteq \vec{S} \cup \overleftarrow{S}$  oder  $\text{len}(P) = \text{dist}(s, v) + \text{dist}(v, t)$ .

## 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:  
 $\{v_1, \dots, v_k\} \subseteq \vec{S} \cup \overleftarrow{S}$  oder  $\text{len}(P) = \text{dist}(s, v) + \text{dist}(v, t)$ .
- Angenommen es gibt  $v_i \notin \vec{S} \cup \overleftarrow{S}$ .
- Dann gilt

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

Also

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

## 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:  $\vec{d}[v] + \overleftarrow{d}[v] \geq \text{dist}(s, t)$ .

## 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 kürzer als  $\mu$  ist.
- Auf  $P$  gibt es eine Kante  $(u, v)$  mit  $d(s, u) \leq \min\text{Key}(\vec{Q})$  und  $d(v, t) \leq \min\text{Key}(\overleftarrow{Q})$ .
- Also müssen  $u$  und  $v$  schon abgearbeitet worden sein (o.B.d.A.  $u$  vor  $v$ )
- Beim relaxieren von  $(u, v)$  wäre  $P$  entdeckt worden und  $\mu$  aktualisiert

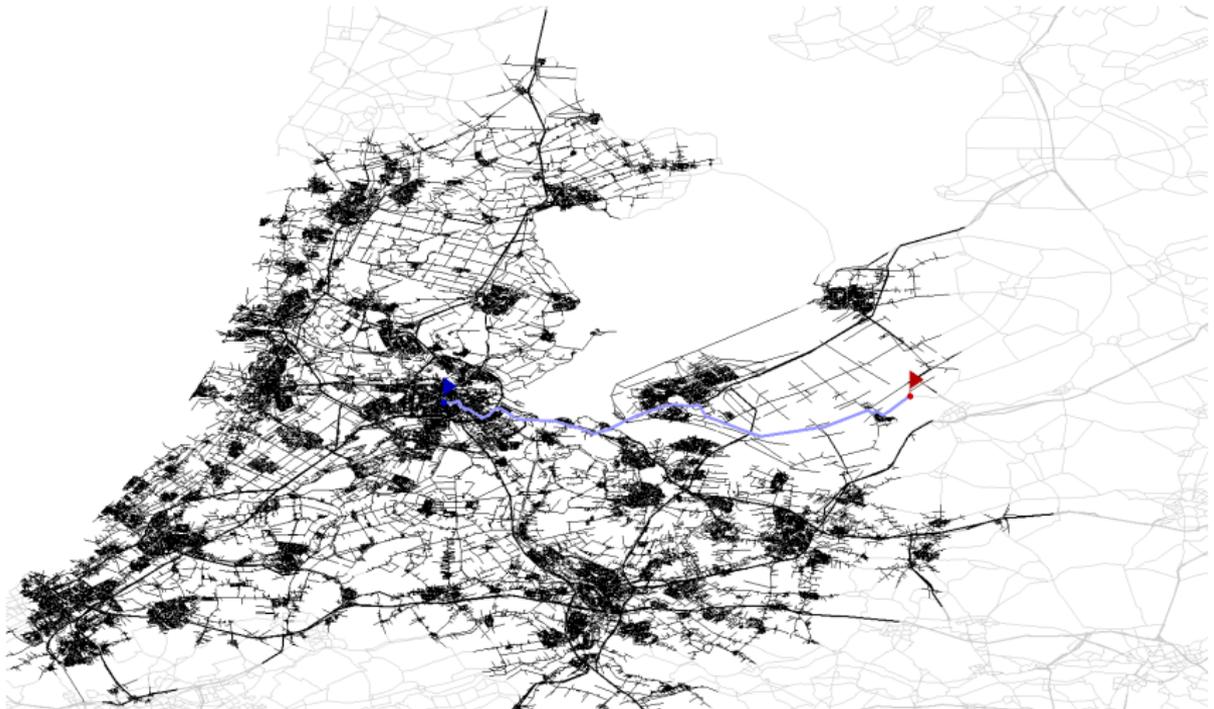
Damit ist  $\mu < d(s, t)$ . Widerspruch!

Frage: Was sind mögliche Wechselstrategien?

## Mögliche Wechselstrategien

- Prinzipiell jede Wechselstrategie möglich
- Wechsle nach jedem Schritt zur entgegengesetzten Suche
- Führe immer die Suche mit der kleineren Anzahl Elemente in Queue aus
- Führe immer die Suche mit dem kleineren minimalen Queueelement aus
- 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!

## Problem:

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

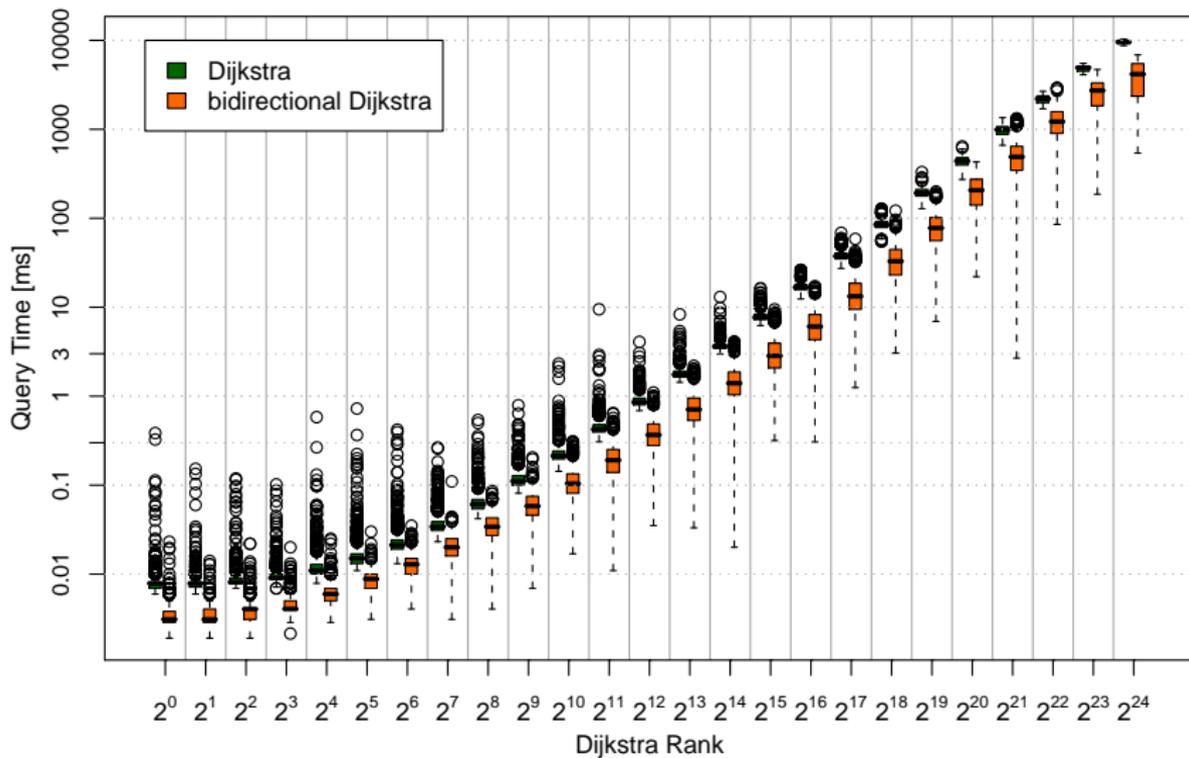
## Problem:

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

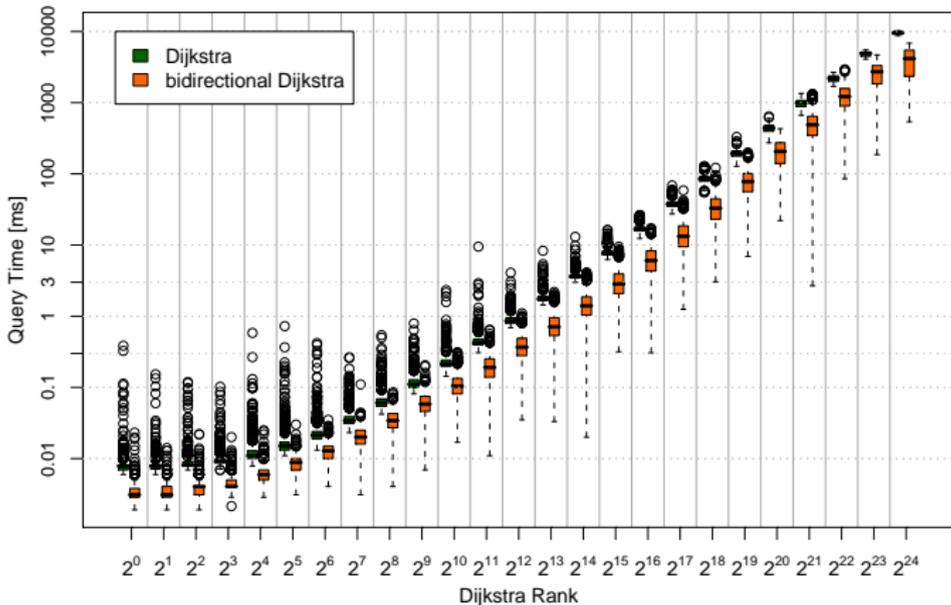
## Idee:

- DIJKSTRA definiert für gegebenen Startknoten Ordnung für auf den Knoten
- DIJKSTRA-Rang  $r_s(u)$  eines Knoten  $u$  für gegebenes  $s$
- wähle 1000 Startknoten und analysiere jeweils die Suchzeiten um die Knoten mit Rang  $2^1, \dots, 2^{\log n}$  zu finden
- zeichne Plot

# Rank Bidirektionale Suche



# 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

# Montag, 25. April 2016

-  Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck.  
**Hierarchical hub labelings for shortest paths.**  
In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.
-  Julian Arz, Dennis Luxen, and Peter Sanders.  
**Transit node routing reconsidered.**  
In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 55–66. Springer, 2013.
-  Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck.  
**Customizable route planning.**  
In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.
-  Edsger W. Dijkstra.  
**A note on two problems in connexion with graphs.**  
*Numerische Mathematik*, 1:269–271, 1959.
-  Andrew V. Goldberg and Chris Harrelson.  
**Computing the shortest path: A\* search meets graph theory.**  
In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.
-  Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling.  
**Contraction hierarchies: Faster and simpler hierarchical routing in road networks.**  
In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.



Andrew V. Goldberg and Renato F. Werneck.

Computing point-to-point shortest paths from external memory.

In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.



Ulrich Lauther.

An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background.

In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.