Karlsruher Institut für Technologie

Algorithmen für Routenplanung

1. Vorlesung, Sommersemester 2015

Andreas Gemsa | 13. April 2015

Institut für Theoretische Informatik · Algorithmik · Prof. Dr. Dorothea Wagner

Organisatorisches

Prof. Dorothea Wagner

Moritz Baum

Julian Dibbelt

Andreas Gemsa

Ben Strasser

Tobias Zündorf

Vorlesungswebseite: `http://i11www.iti.kit.edu/teaching/sommer2015/routenplanung/index`

Organisatorisches

Vorlesung

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

Prüfung

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

Vorlesungswebseite:
`http://i11www.iti.kit.edu/teaching/sommer2015/routenplanung/index`

Weitere Vorlesungen

Algorithmische Kartografie

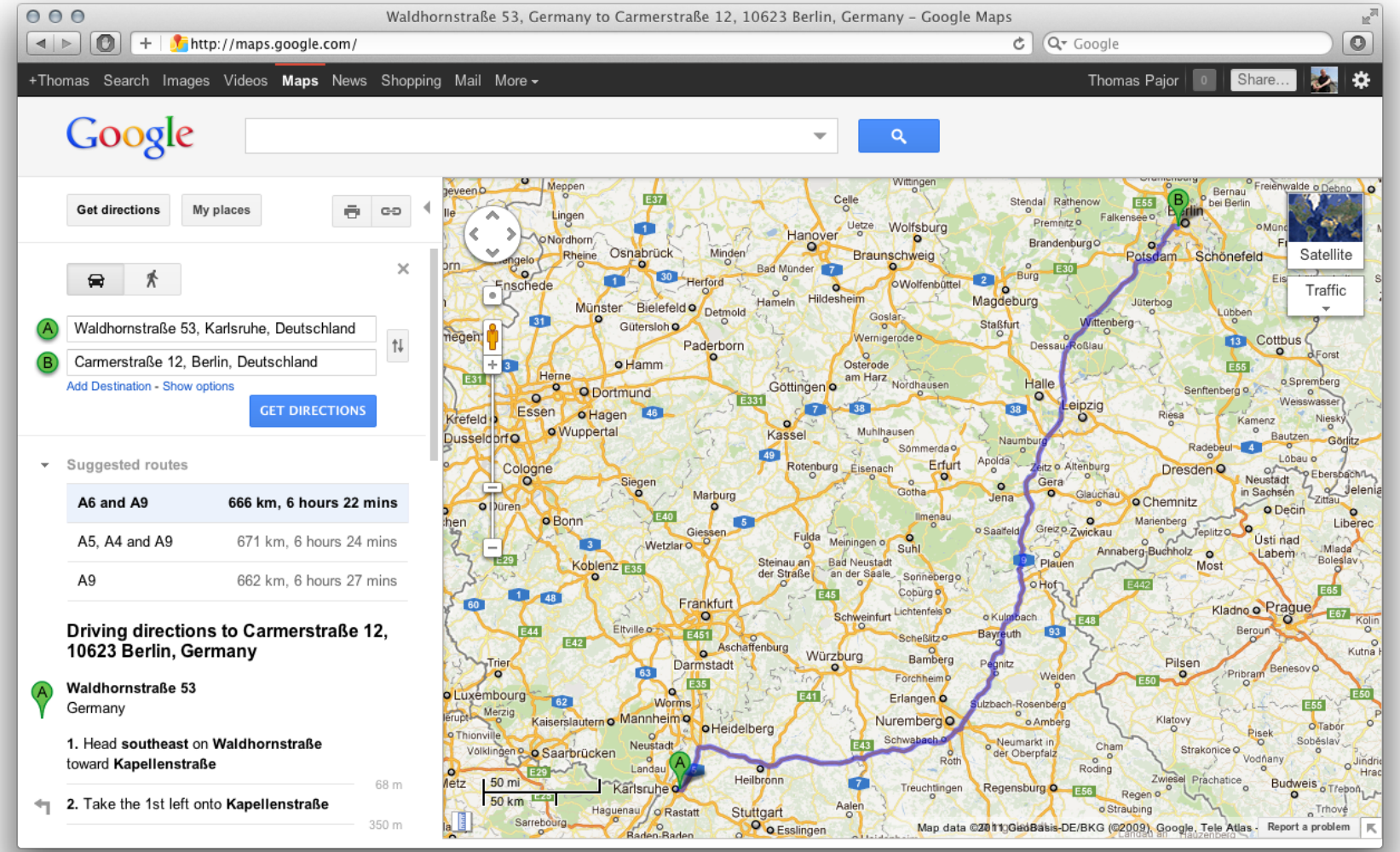
- Dozenten: Dr. Martin Nöllenburg und Benjamin Niedermann
- Vorlesung: dienstags und donnerstags 9:45–11:15 Uhr (SR 301)

Algorithmen für planare Graphen

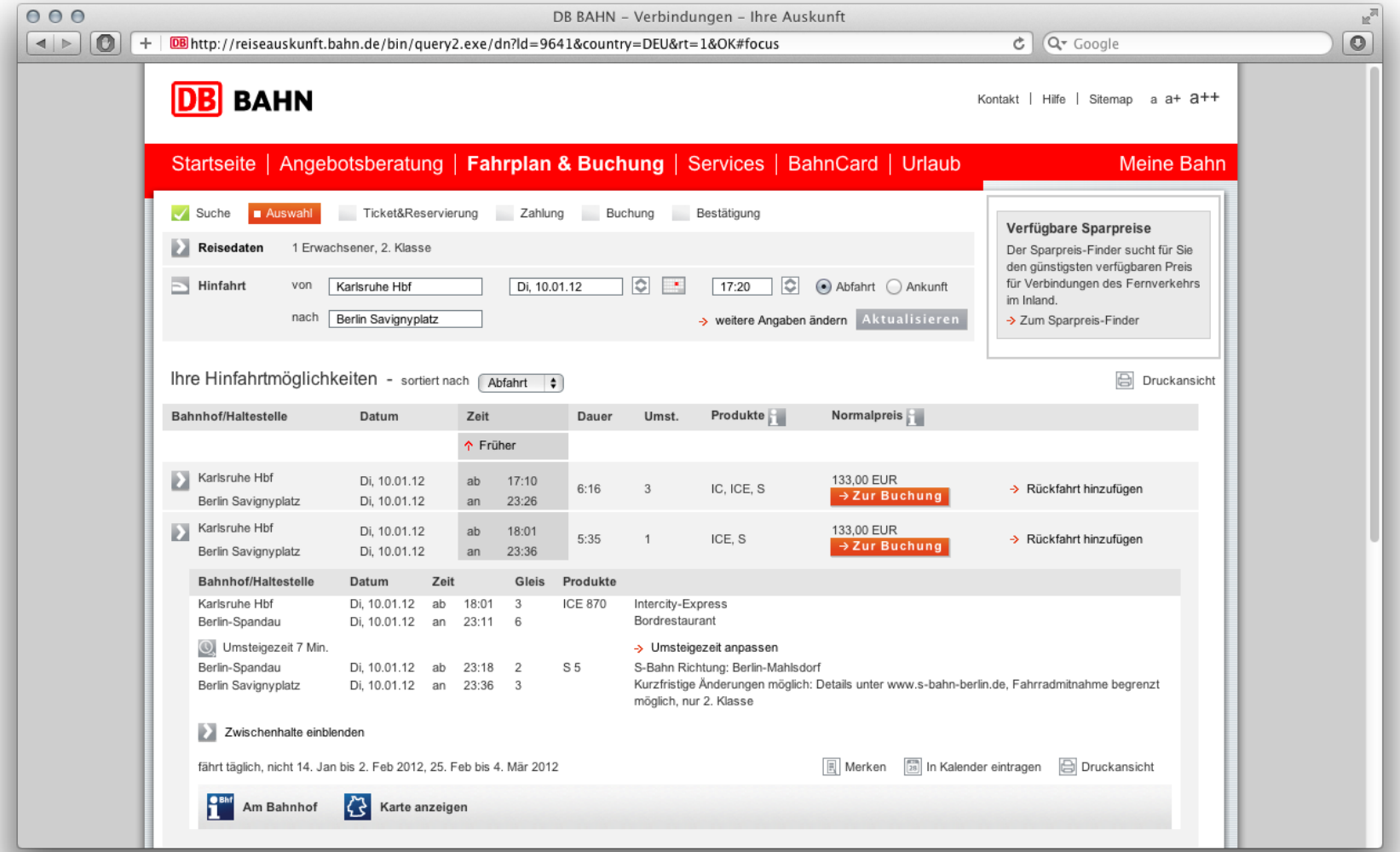
- Dozenten: Prof. Dorothea Wagner
- Vorlesung: dienstags 14:00–15:30 Uhr
  (Hörsaal II - Gebäude 30.41 - Chemie Flachbau)
- Vorlesung: mittwochs 14:00–15:30 Uhr
  (Mittlerer Hörsaal - Gebäude 10.91 - Maschinenbau)
- Übungs: zusätzlich an einigen Vorlesungsterminen

Routenplanung in der Anwendung

Routenplanung in der Anwendung



Routenplanung in der Anwendung



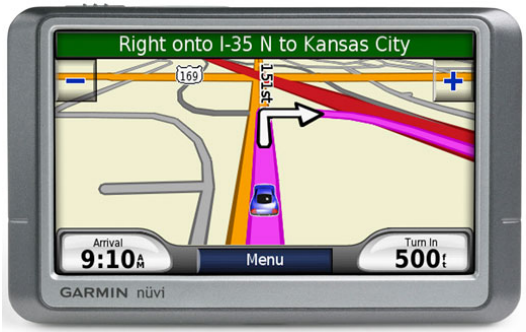
Routenplanung in der Anwendung

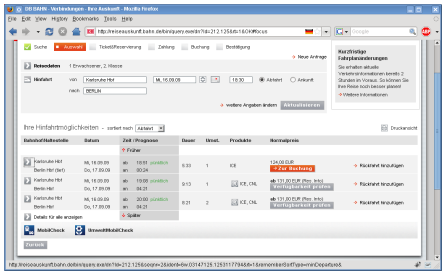
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Motivation Routenplanung

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 [?])

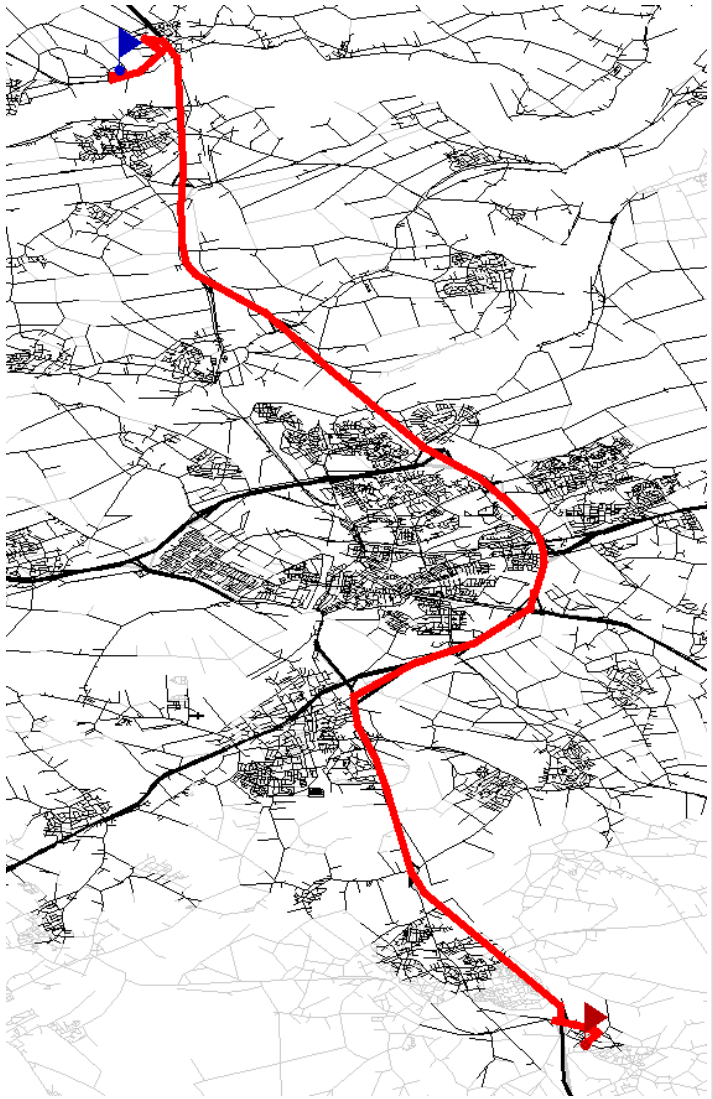
Probleme:

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

Beschleunigungstechniken

Beobachtung:

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



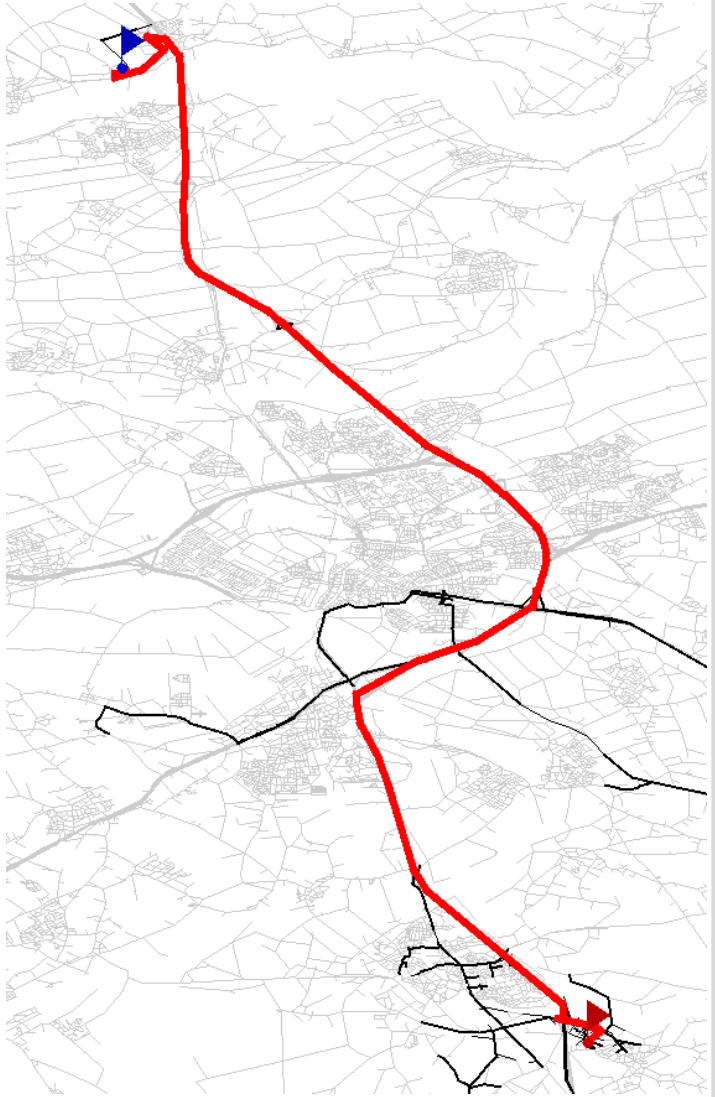
Beschleunigungstechniken

Beobachtung:

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

Idee:

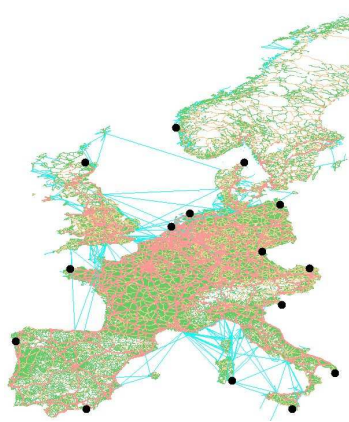
- Zwei Phasen
  - Offline: Generiere Zusatzinformation in Vorberechnung
  - Online: Beschleunige Anfrage mit dieser Information
- Drei Kriterien: Vorberechnungsplatz, Vorberechnungszeit, Beschleunigung



Experimentelle Evaluation

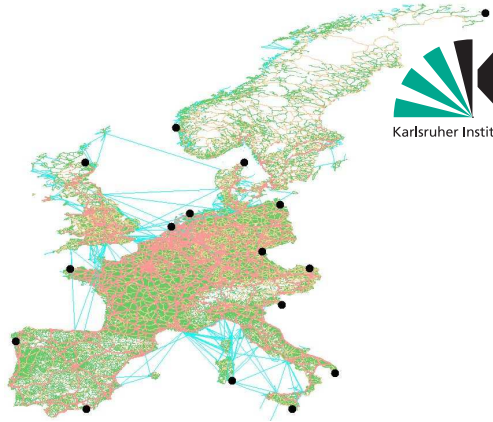
Eingabe: Straßennetzwerk von Europa

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



	Vorberechnung		Anfragen	
Algorithmus	Zeit [h:m]	Platz [GiB]	Zeit [μs]	Beschleun.
Dijkstra [?]	—	—	2 550 000	—
ALT [?, ?]	0:42	2.2	24 521	104
CRP [?]	$\ll$ 0:01	$<$ 0.1	1 650	1 545
Arc-Flags [?]	0:20	0.3	408	6 250
CH [?]	0:05	0.2	110	23 181
TNR [?]	0:20	2.1	1.25	2 040 000
HL [?]	0:37	18.4	0.56	4 553 571

Experimentelle Evaluation



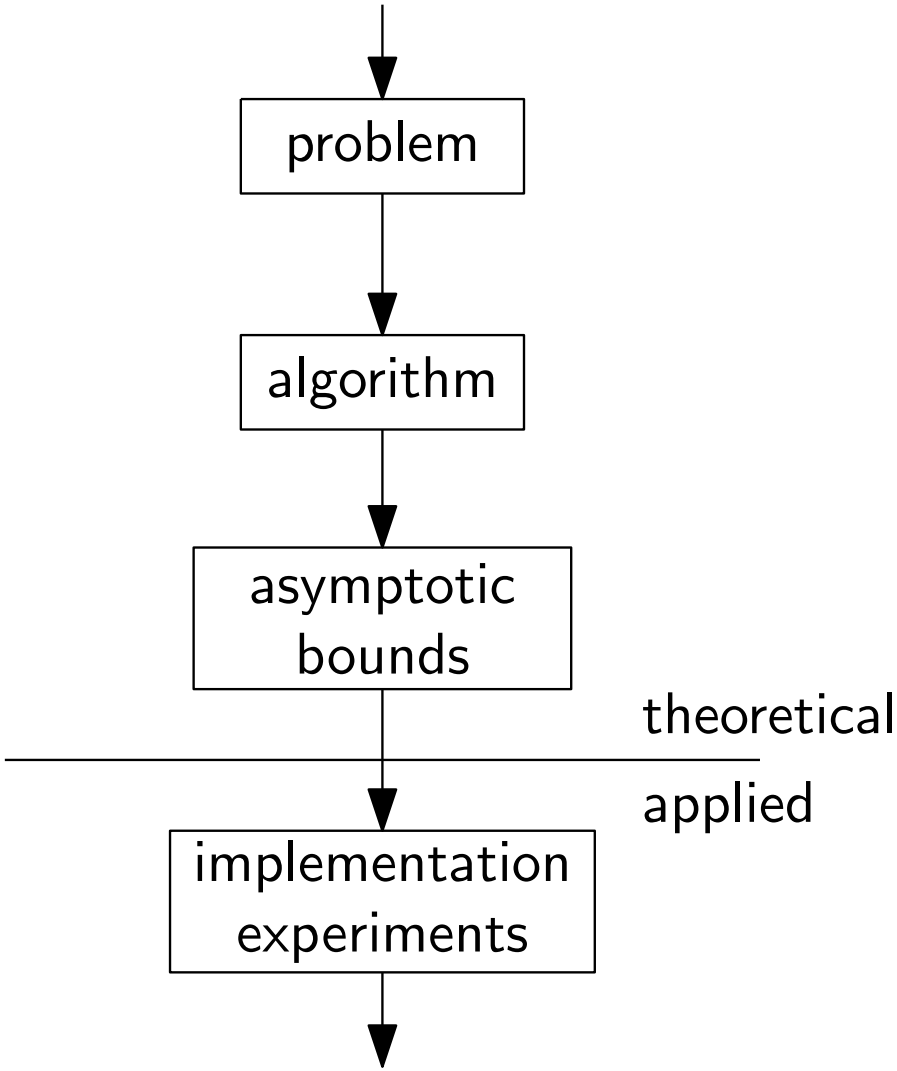
Eingabe: Straßennetzwerk von Europa

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

	Vorberechnung		Anfragen	
Algorithmus	Zeit [h:m]	Platz [GiB]	Zeit [μs]	Beschleun.
Dijkstra [?]	—	—	2 550 000	—
ALT [?, ?]	0:42	2.2	24 521	104
CRP [?]	$\ll$ 0:01	$<$ 0.1	1 650	1 545
Arc-Flags [?]	0:20	0.3	408	6 250
CH [?]	0:05	0.2	110	23 181
TNR [?]	0:20	2.1	1.25	2 040 000
HL [?]	0:37	18.4	0.56	4 553 571

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

Klassischer Algorithmenentwurf



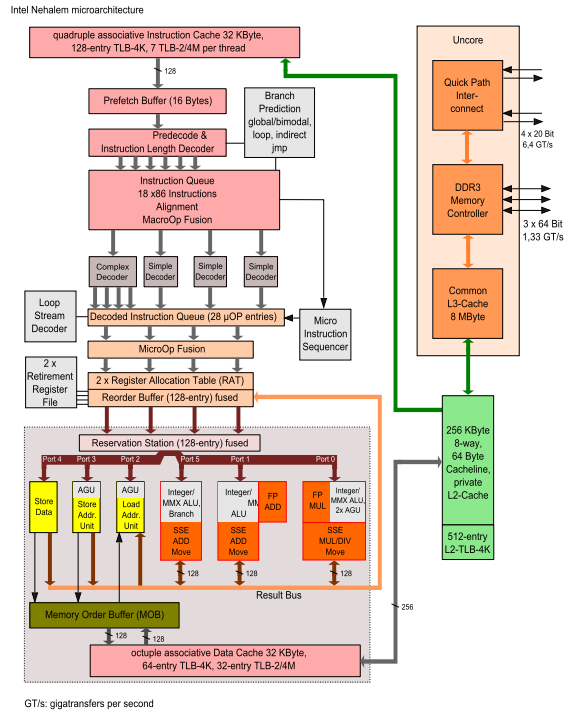
Moderne CPU Architektur

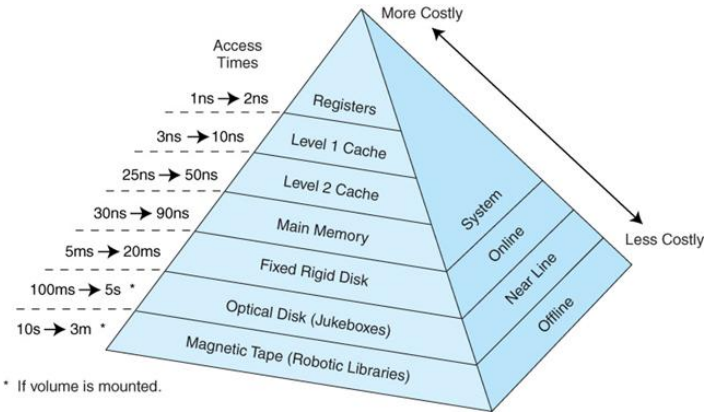
Einige Fakten:

- steile Speicherhierarchie
- viele Kerne
- mehr Kerne als Speicherkontroller
- Cache-Kohärenz
- SIMD (SSE, AVX)

Hauptherausforderungen:

- Speicherzugriff/Datenlokalität
- Parallelisierung
  - Nicht nur race-conditions, etc.
  - auch Speicherzugriff
    z.B. “false-sharing” (Cache-Kohärenz)





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

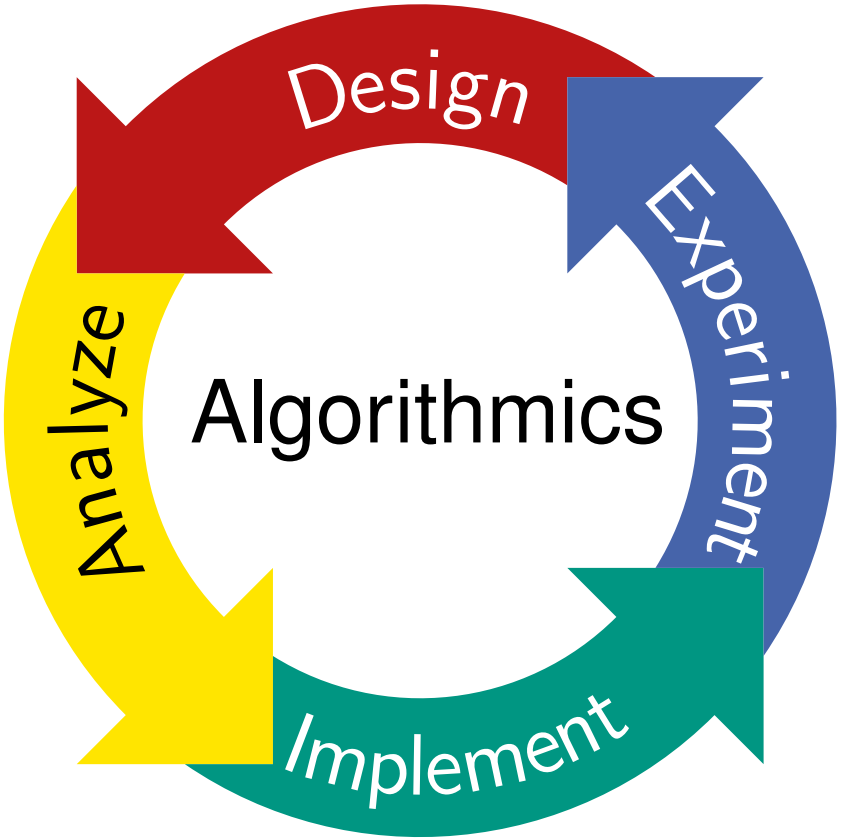
Lücke Theorie vs. Praxis

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

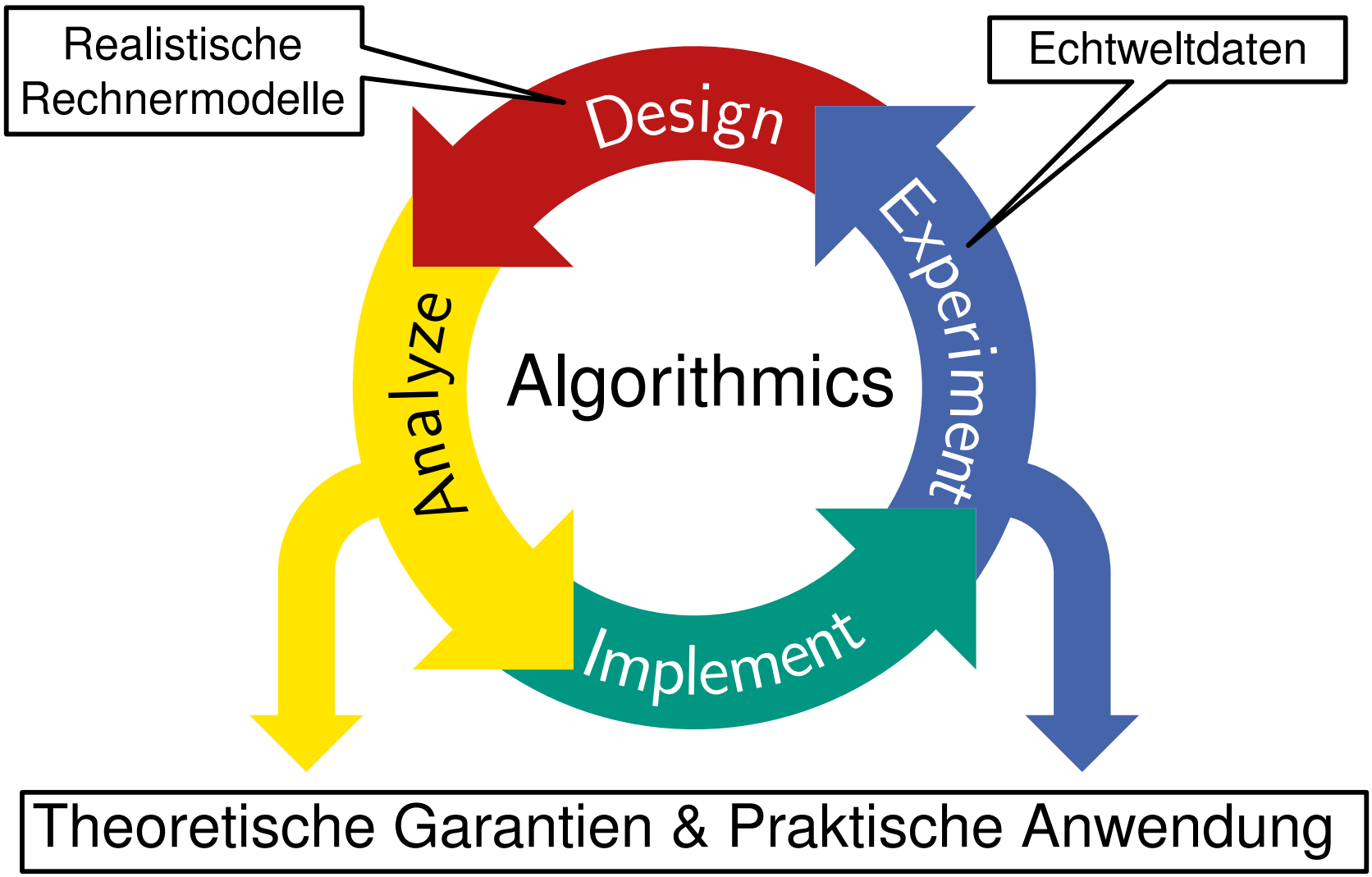
Routenplanung:

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

Algorithm Engineering



Algorithm Engineering



Schwerpunkte der Vorlesung

- Algorithm Engineering + ein bisschen Theorie
- Beschleunigungstechniken
- Implementierungsdetails
- Ergebnisse auf Real-Welt Daten
- aktuellster Stand der Forschung (Veröffentlichungen bis 2014)
- ideale Grundlage für Bachelor, Master und Diplomarbeiten

Was diese Vorlesung nicht ist

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

Inhalt der Vorlesung

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 Distanztabellen
- Kombinationen

Inhalt der Vorlesung

Karlsruher Institut für Technologie

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

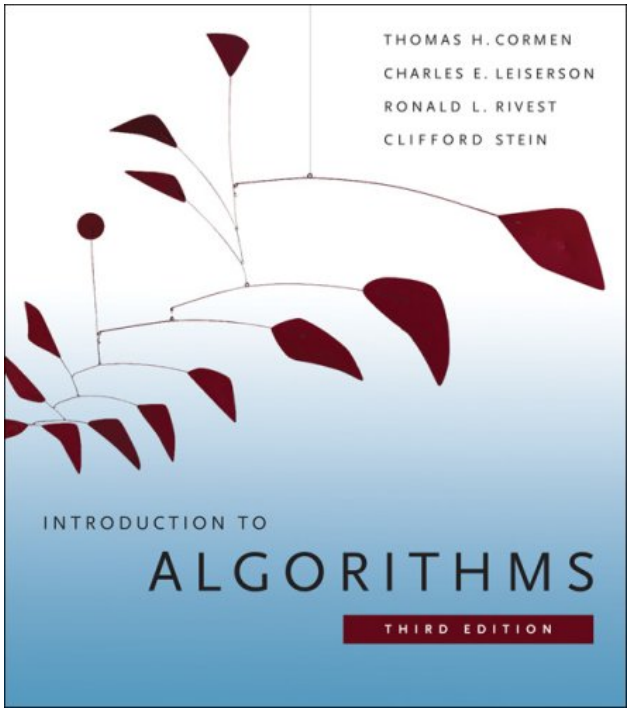
Nützliche Vorkenntnisse

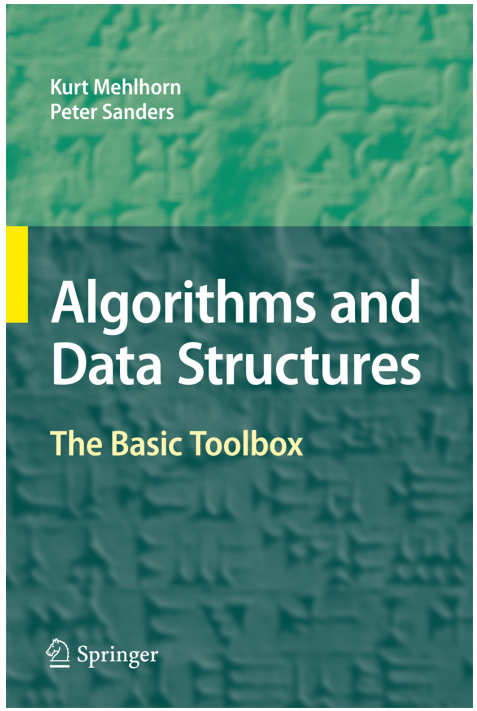
- Informatik I/II oder Algorithmen I
- Algorithmentechnik oder Algorithmen II (muss aber nicht sein)
- ein bisschen Rechnerarchitektur
- passive Kenntnisse von C++/Java

Vertiefungsgebiet: Algorithmentechnik, (Theoretische Grundlagen)

Material

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





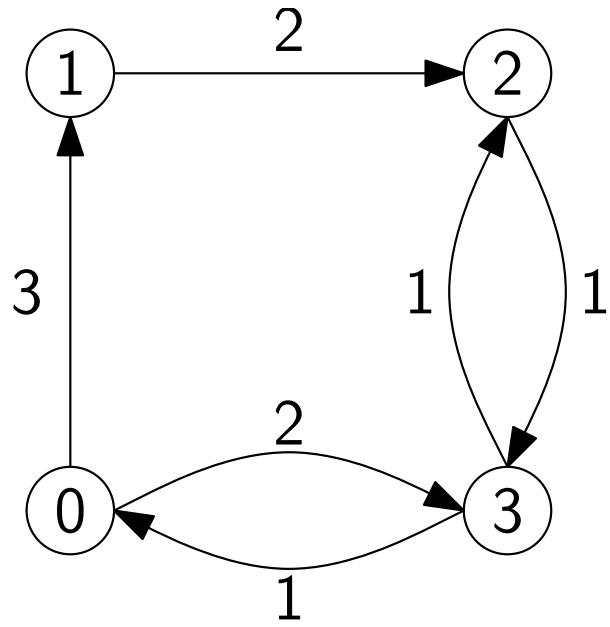
1. Grundlagen

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Graph-Repräsentationen

Drei klassische Ansätze:

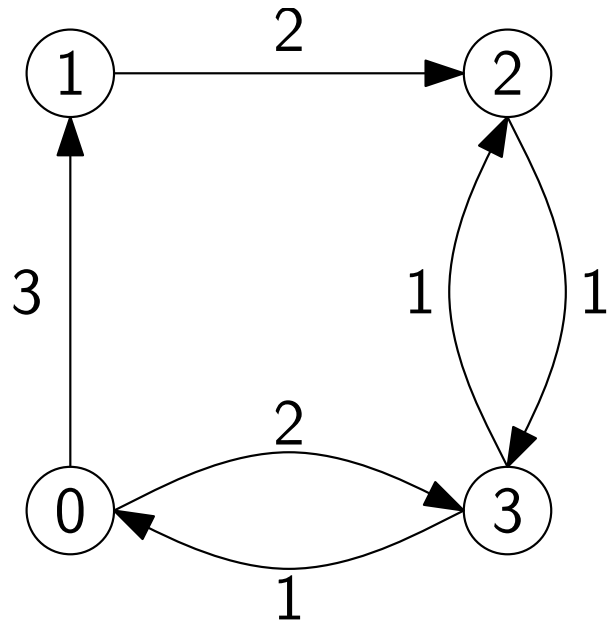
- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray



Graph-Repräsentationen

Drei klassische Ansätze:

- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray

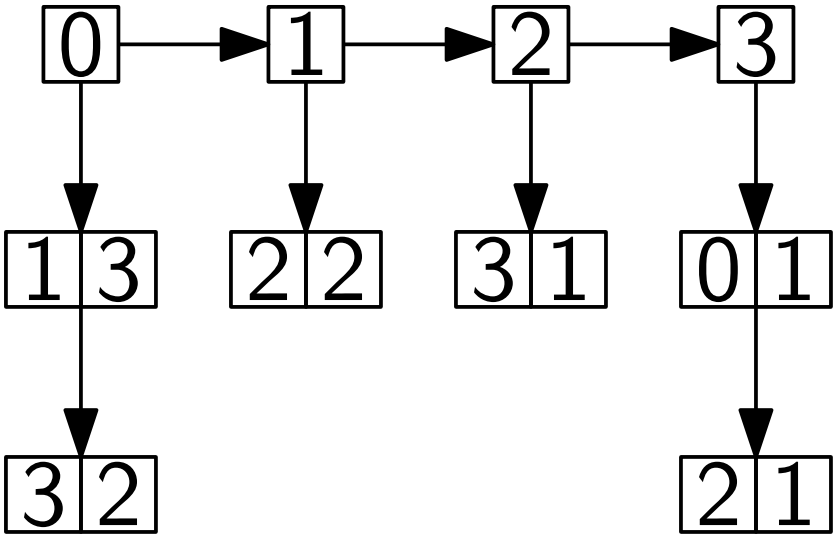


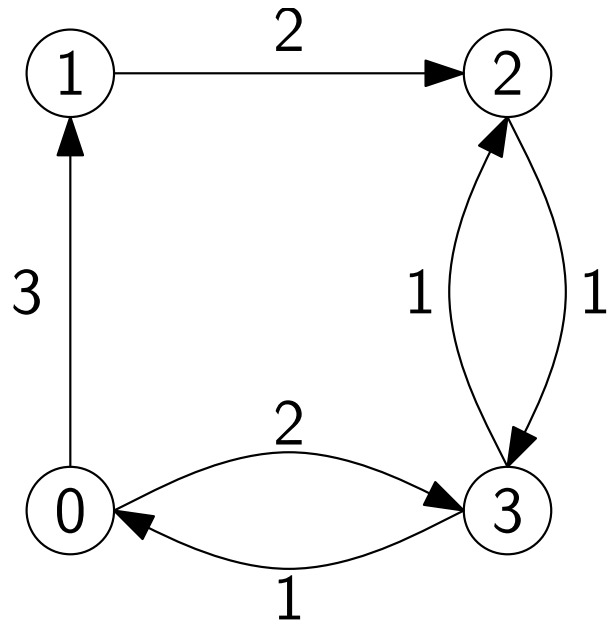
	0	1	2	3
0	–	3	–	2
1	–	–	2	–
2	–	–	–	1
3	1	–	1	–

Graph-Repräsentationen

Drei klassische Ansätze:

- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray



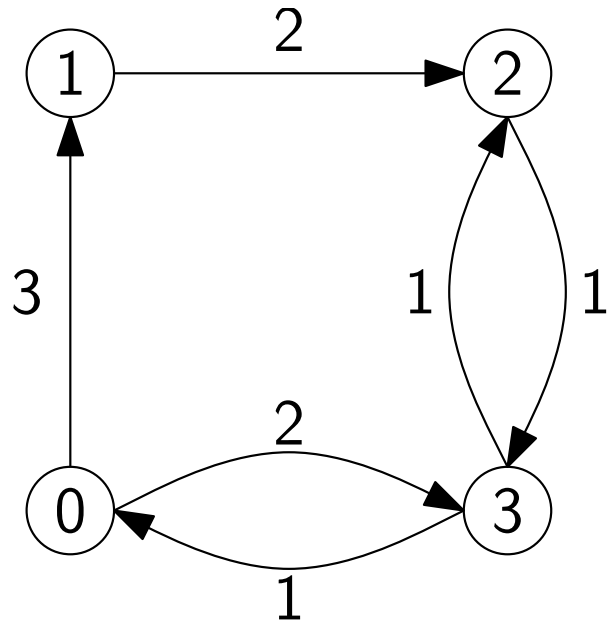


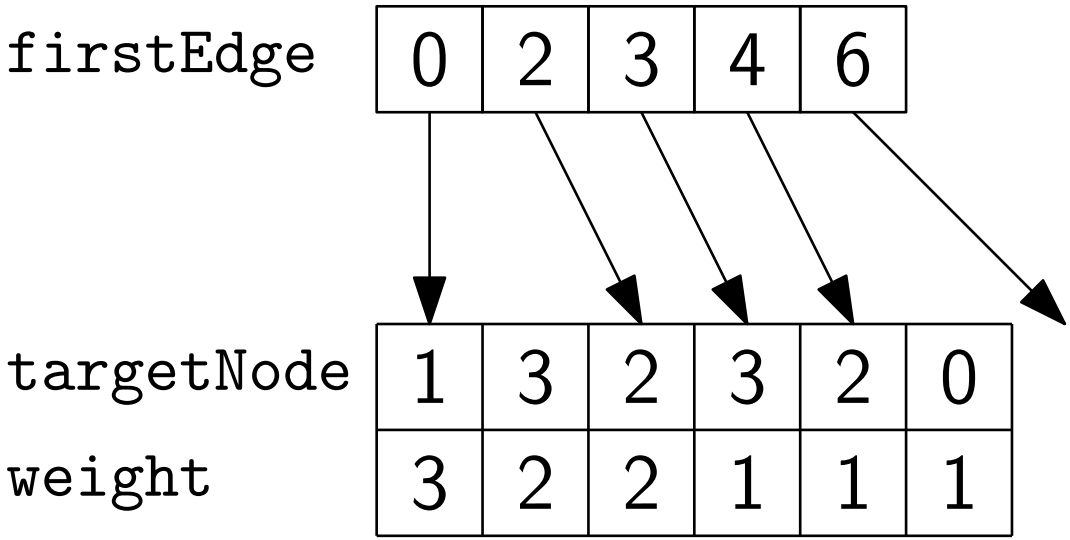
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Graph-Repräsentationen

Drei klassische Ansätze:

- Adjazenzmatrix
- Adjazenzlisten
- Adjazenzarray





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 (topologisch)	$+$	$+$	$-$
Updates (Gewicht)	$+$	$+$	$+$

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 (topologisch)	$+$	$+$	$-$
Updates (Gewicht)	$+$	$+$	$+$

Fragen:

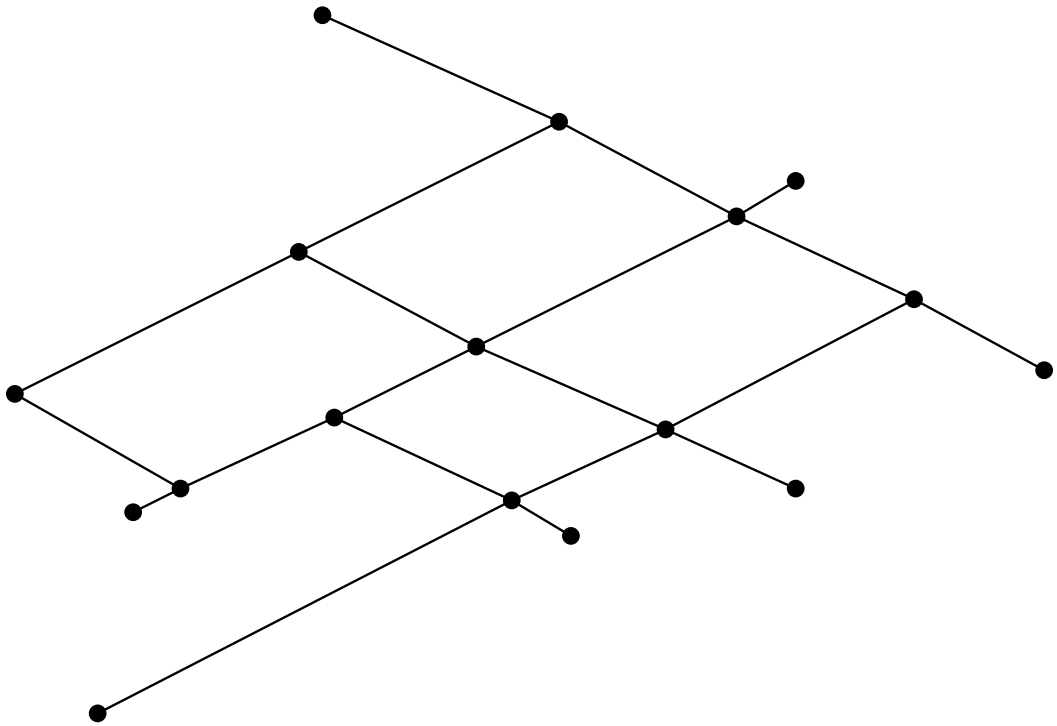
- Was brauchen wir?
- Was muss nicht super effizient sein?
- erstmal Modelle anschauen!

Modellierung (Straßengraphen)

Modellierung (Straßengraphen)

Modellierung (Straßengraphen)

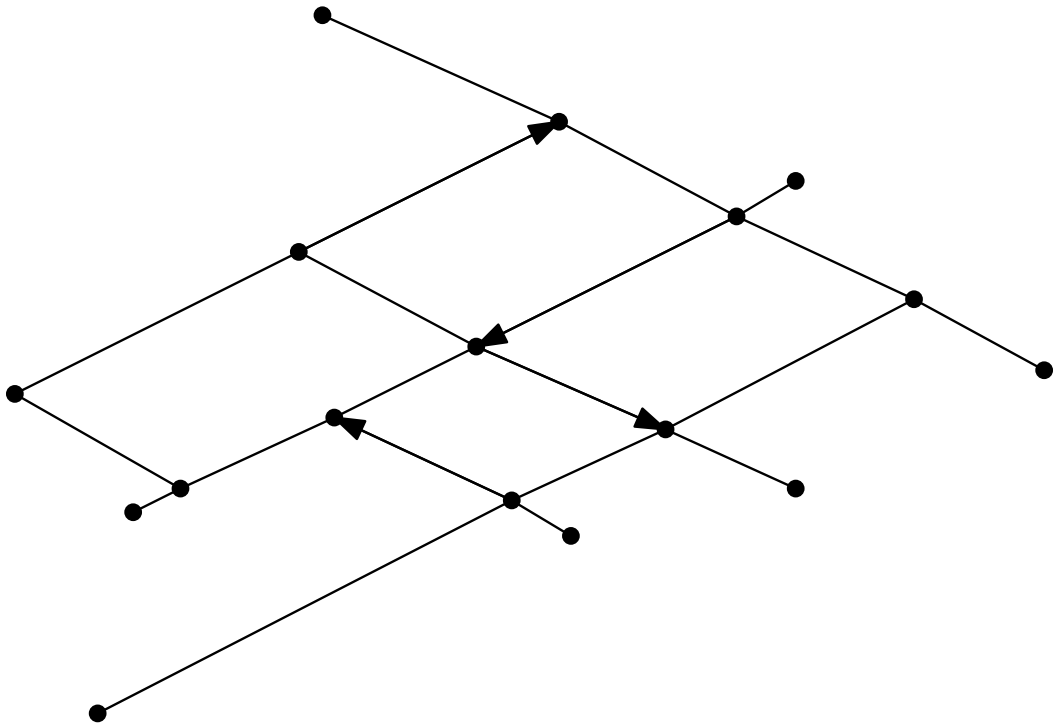
- Knoten sind Kreuzungen
- Kanten sind Straßen



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Modellierung (Straßengraphen)

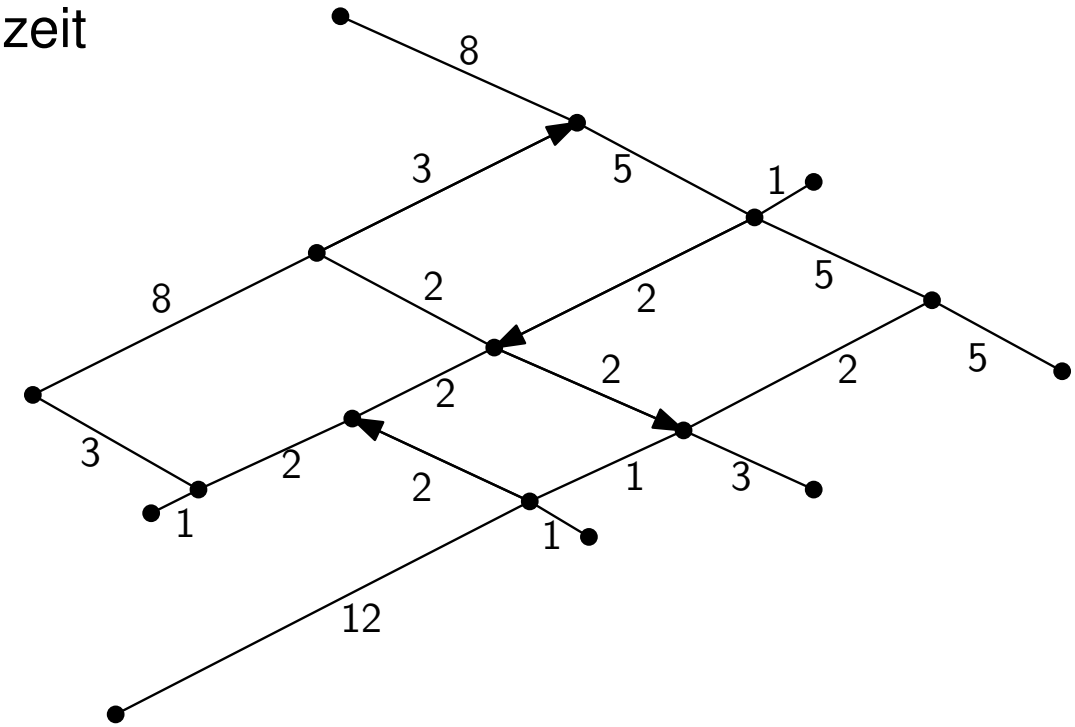
- Knoten sind Kreuzungen
- Kanten sind Straßen
- Einbahnstraßen



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

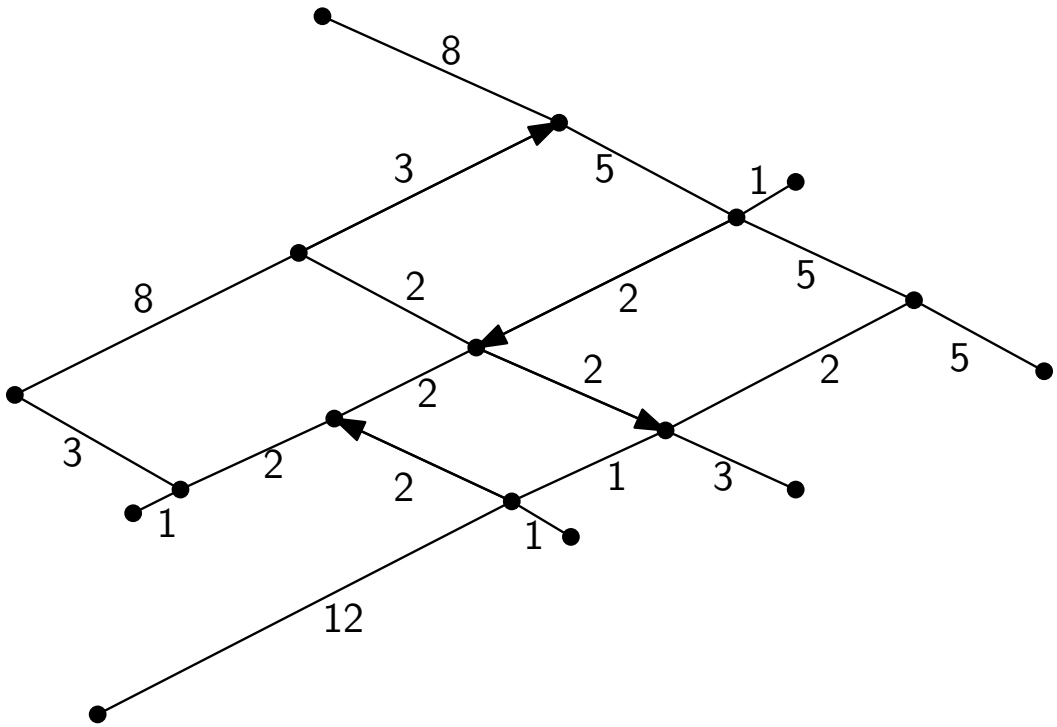
Modellierung (Straßengraphen)

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



Modellierung (Straßengraphen)

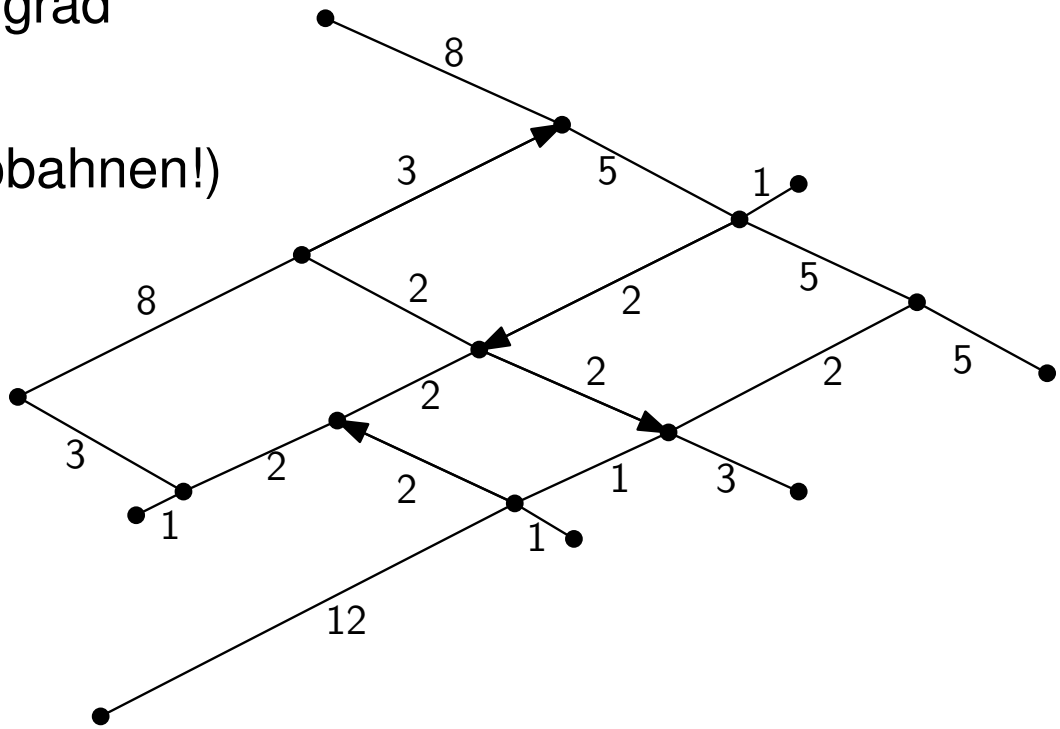
Eigenschaften (sammeln):



Modellierung (Straßengraphen)

Eigenschaften:

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



Modellierung (Eisenbahngraphen)

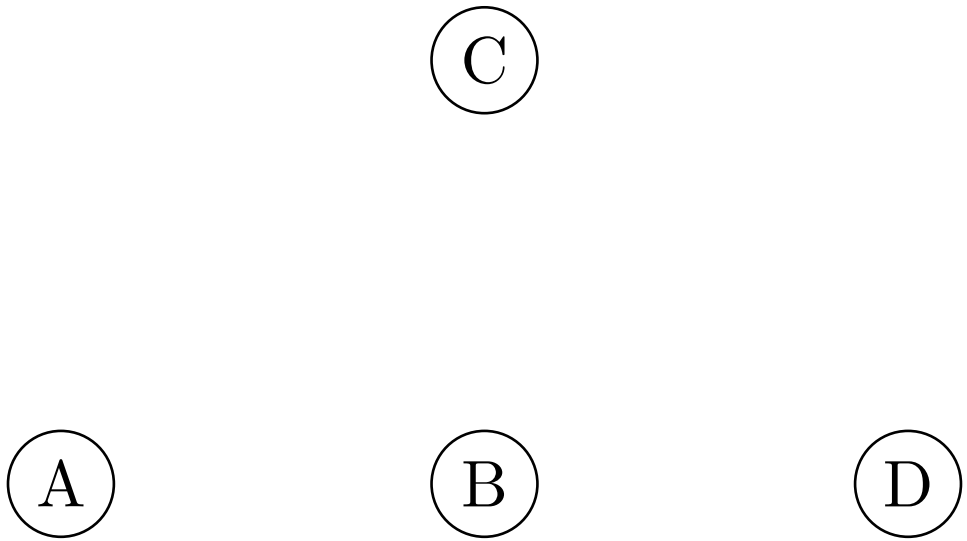
Beispiel:

- 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

Modellierung (Eisenbahngraphen)

Beispiel:

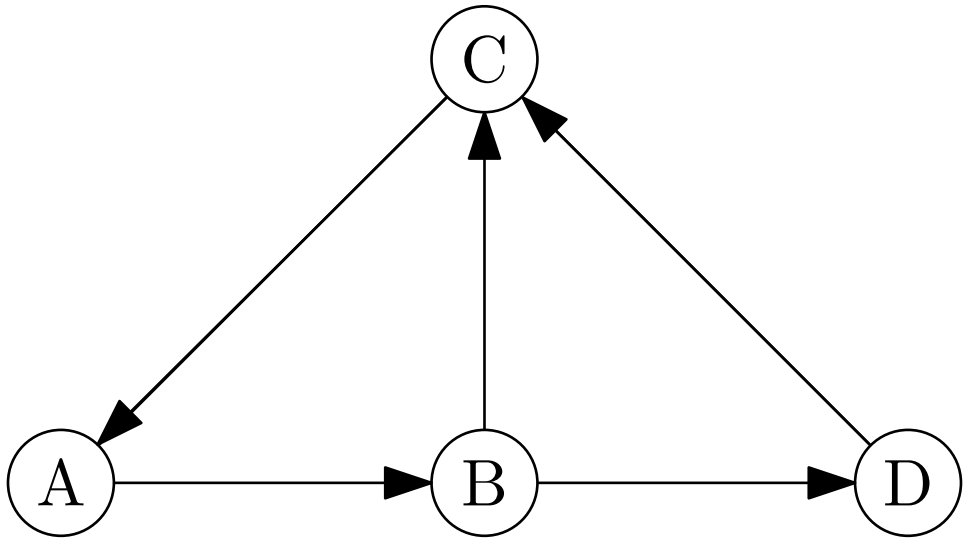
- 4 Stationen (A,B,C,D)
- Zug 1: Station A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A
- Zug 2: Station A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ C $\rightarrow$ A
- Züge operieren alle 10 Minuten



Modellierung (Eisenbahngraphen)

Beispiel:

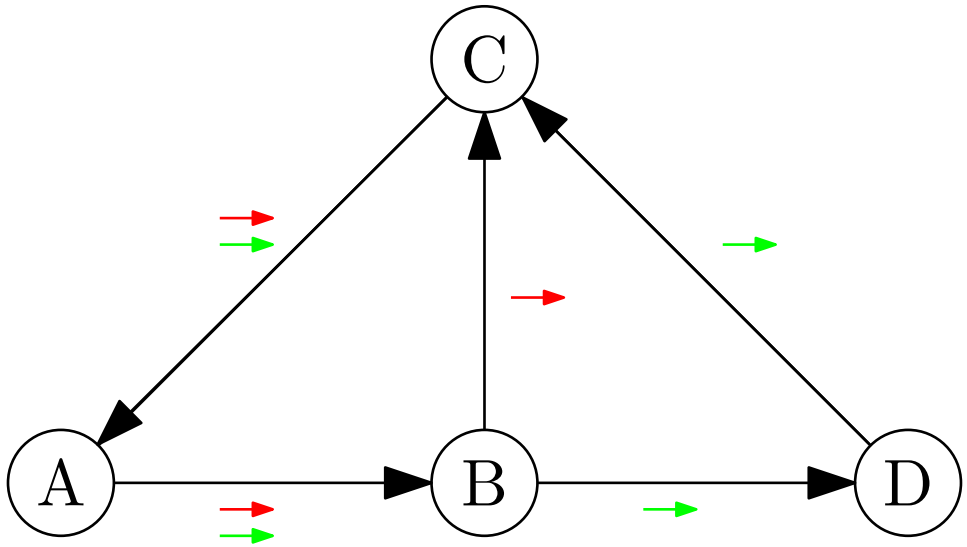
- 4 Stationen (A,B,C,D)
- Zug 1: Station A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A
- Zug 2: Station A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ C $\rightarrow$ A
- Züge operieren alle 10 Minuten



Modellierung (Eisenbahngraphen)

Beispiel:

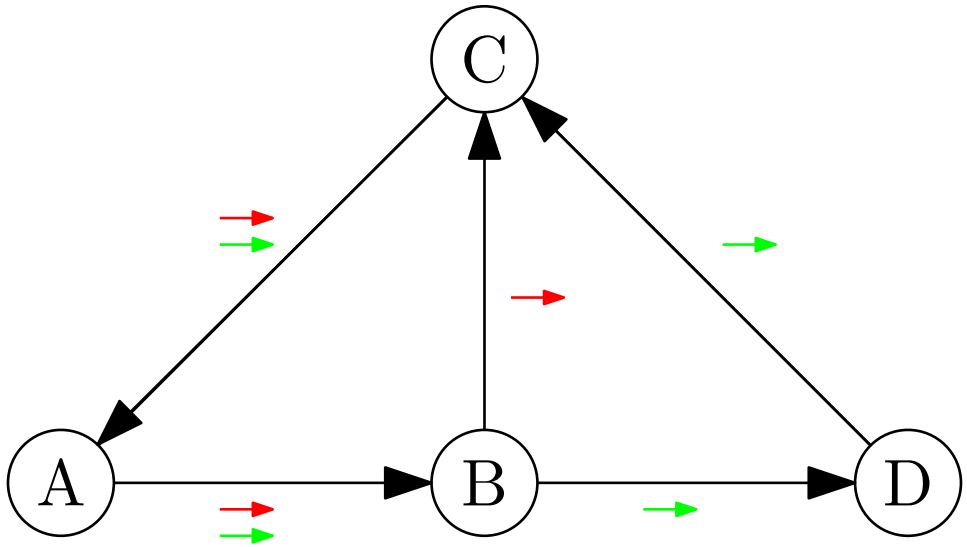
- 4 Stationen (A,B,C,D)
- Zug 1: Station A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A
- Zug 2: Station A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ C $\rightarrow$ A
- Züge operieren alle 10 Minuten



Modellierung (Eisenbahngraphen)

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

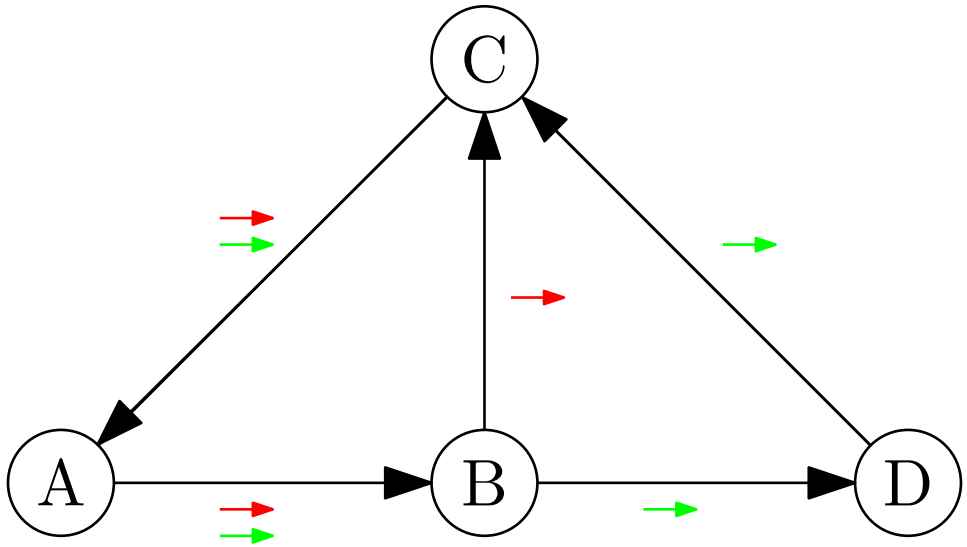


Kanten sind zeitabhängig!
(wann fährt ein Zug wie lange?)

Modellierung (Eisenbahngraphen)

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



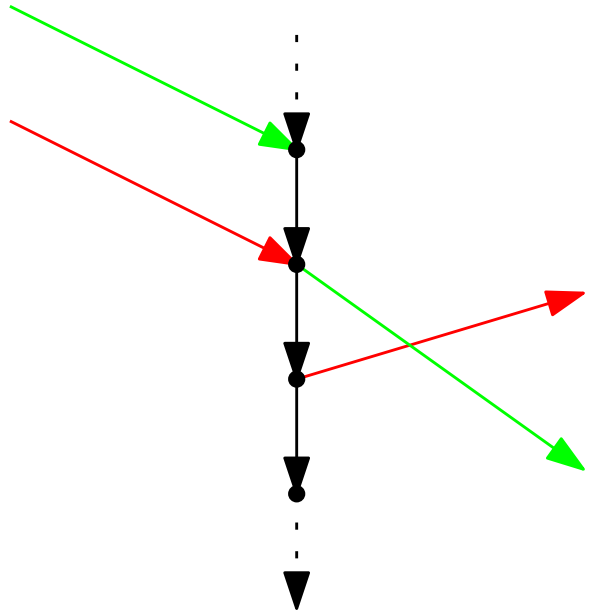
Kanten sind zeitabhängig! oder roll die Zeit aus
(wann fährt ein Zug wie lange?)

Zeitexpandiertes Modell

Vorgehen:

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

Station B:

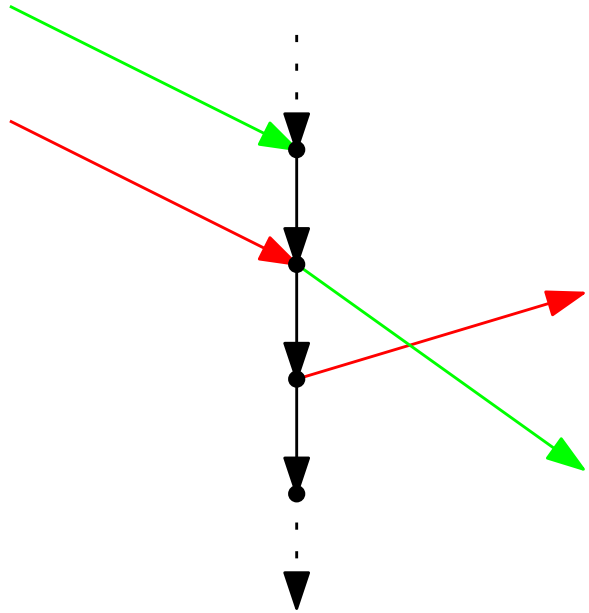


Zeitexpandiertes Modell

Vorgehen:

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

Station B:

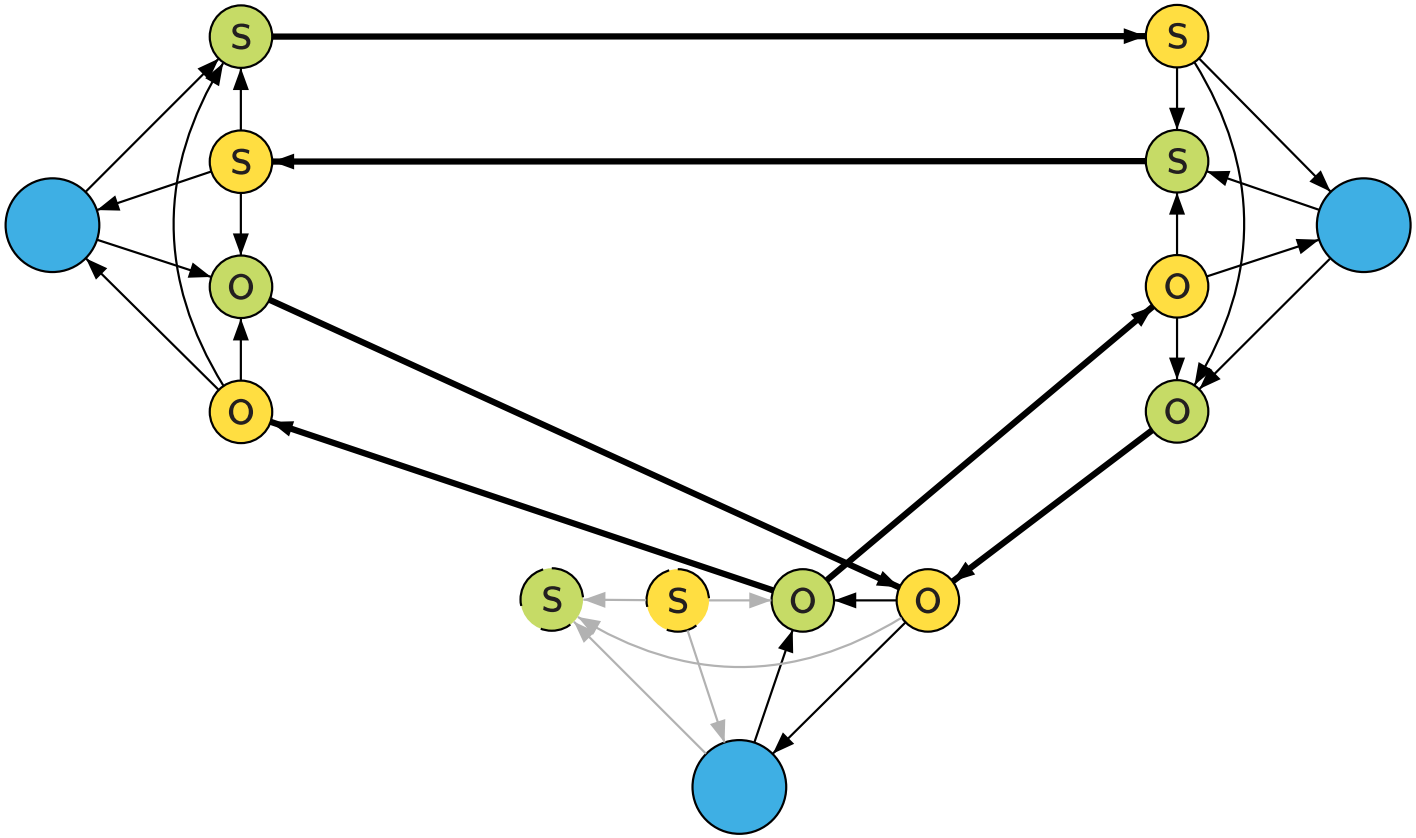


Diskussion:

\+ keine zeitabhängigen Kanten

– Graph größer

Modellierung (Flugdaten)



Eigenschaften der Graphen

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

Eigenschaften der Graphen

- 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

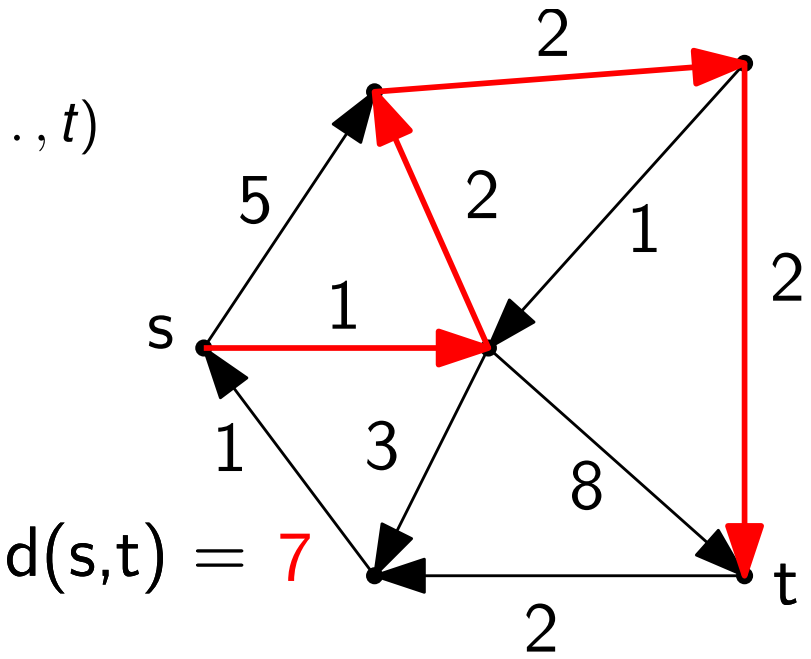
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Problemstellung

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

Mögliche Aufgaben

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



Eigenschaften von kürzesten Wegen

Azyklität:

- Kürzeste Wege sind zyklenfrei

Aufspannungseigenschaft:

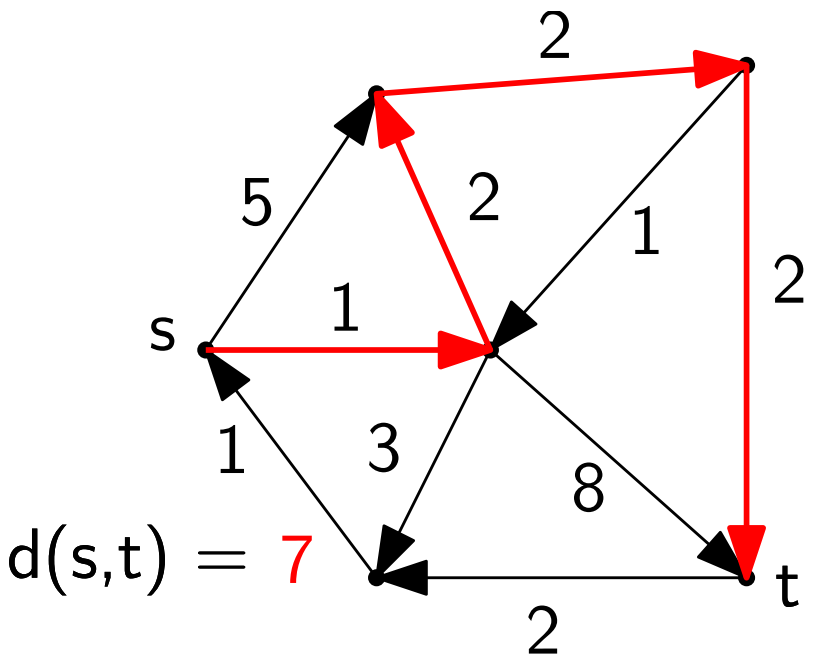
- Alle kürzesten Wege von *s* aus bilden DAG bzw. Baum

Vererbungseigenschaft:

- Teilwege von kürzesten Wegen sind kürzeste Wege

Abstand:

- Steigender Abstand von Wurzel zu Blättern



Berechnung kürzester Wege

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“

Der Bellman-Ford Algorithmus

Bellman-Ford(G, s)

```
1 for v ∈ V do d[v] ← ∞
2
3 d[s] ← 0
4 for i = 1 to |V| − 1 do
5     forall the edges (u, v) ∈ E do
6         if d[u] + len(u, v) < d[v] then
7             d[v] ← d[u] + len(u, v)
```

Der Bellman-Ford Algorithmus

Bellman-Ford(G, s)

```
1  for v ∈ V do d[v] ← ∞
2
3  d[s] ← 0
4  for i = 1 to |V| − 1 do
5      forall the edges (u, v) ∈ E do
6          if d[u] + len(u, v) < d[v] then
7              d[v] ← d[u] + len(u, v)
8  forall the edges (u, v) ∈ E do
9      if d[v] > d[u] + len(u, v) then
10         negative cycle found
```

Problem Longest Path

Problem Longest Path

Gegeben:

- Gerichteter, gewichteter Graph $G = (V, E)$ mit Längenfunktion $\text{len}\colon E \to \mathbb{N}$
- 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])

Dijkstras Algorithmus

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                    // distances, parents
3  d[s] = 0
4  Q.clear(), Q.insert(s, 0)                    // container
5  while !Q.empty() do
6      u ← Q.deleteMin()                        // settling node u
7      forall the edges e = (u, v) ∈ E do
           // relaxing edges
8          if d[u] + len(e) < d[v] then
9              d[v] ← d[u] + len(e)
10             p[v] ← u
11             if v ∈ Q then Q.decreaseKey(v, d[v])
12             else Q.insert(v, d[v])
```

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

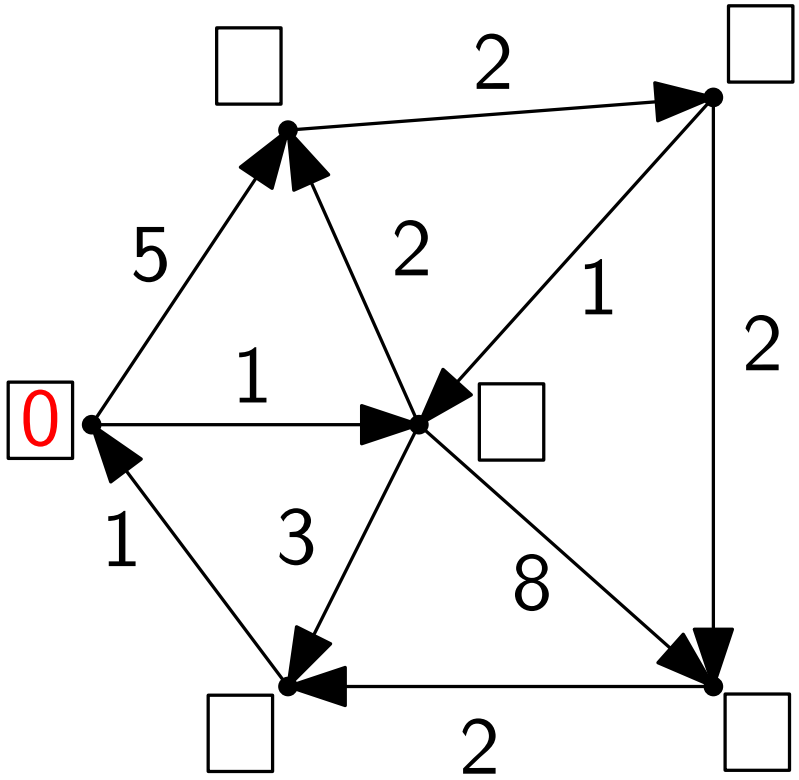
Idee: Suche in G mit zunehmender *Distanz* von s.

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

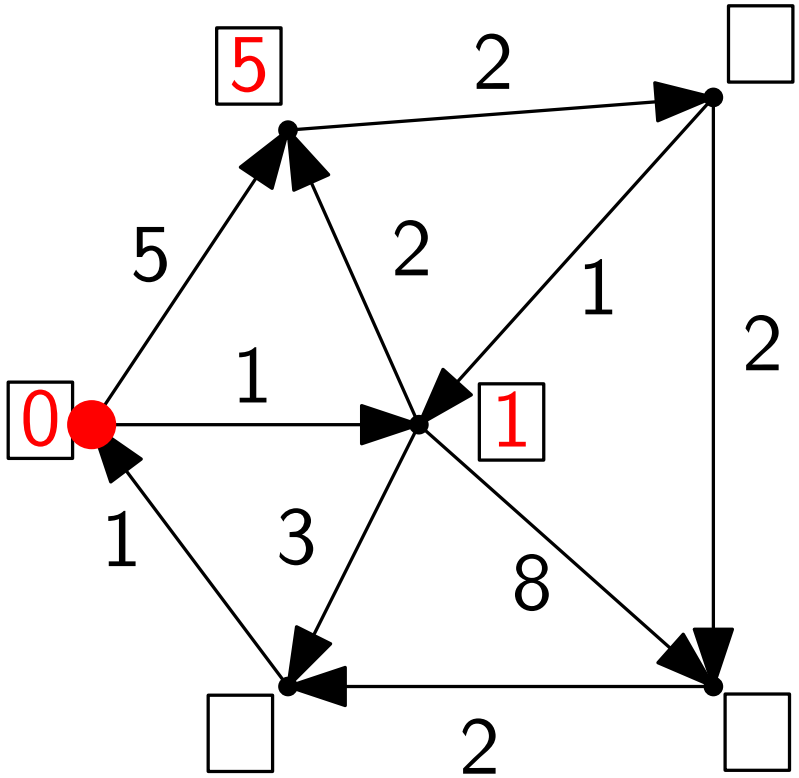


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

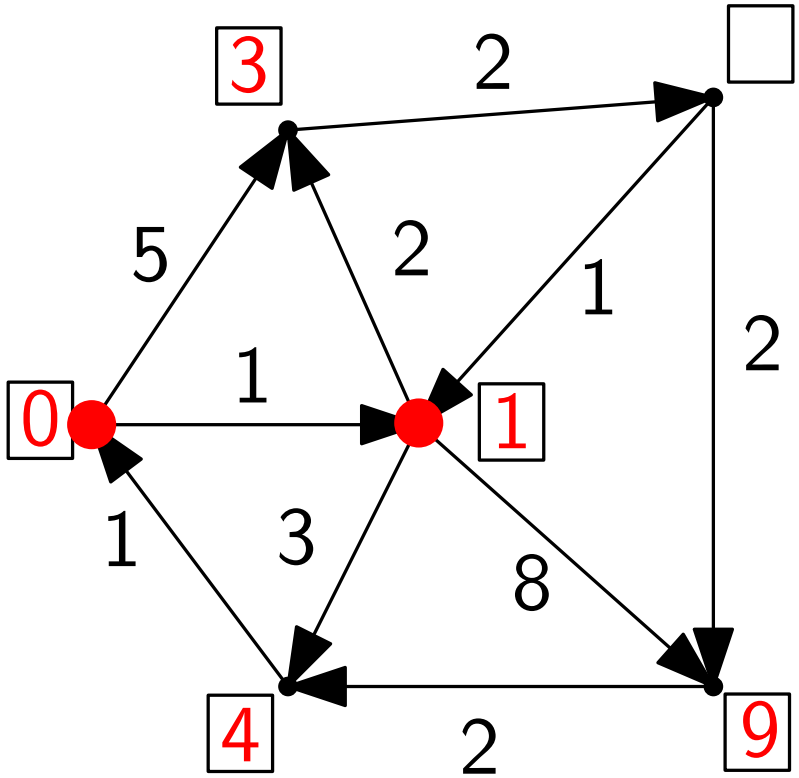


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

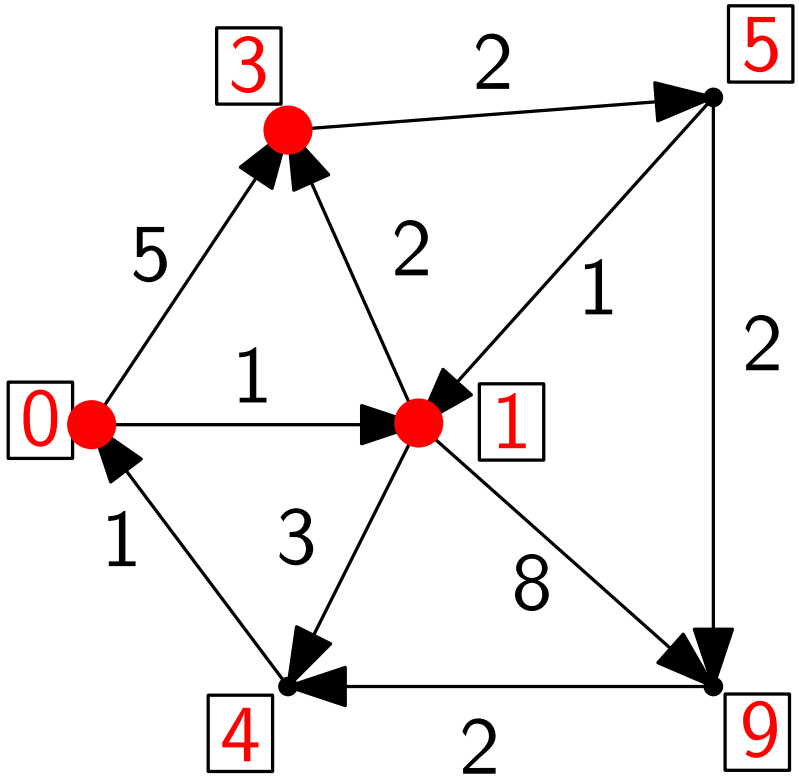


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

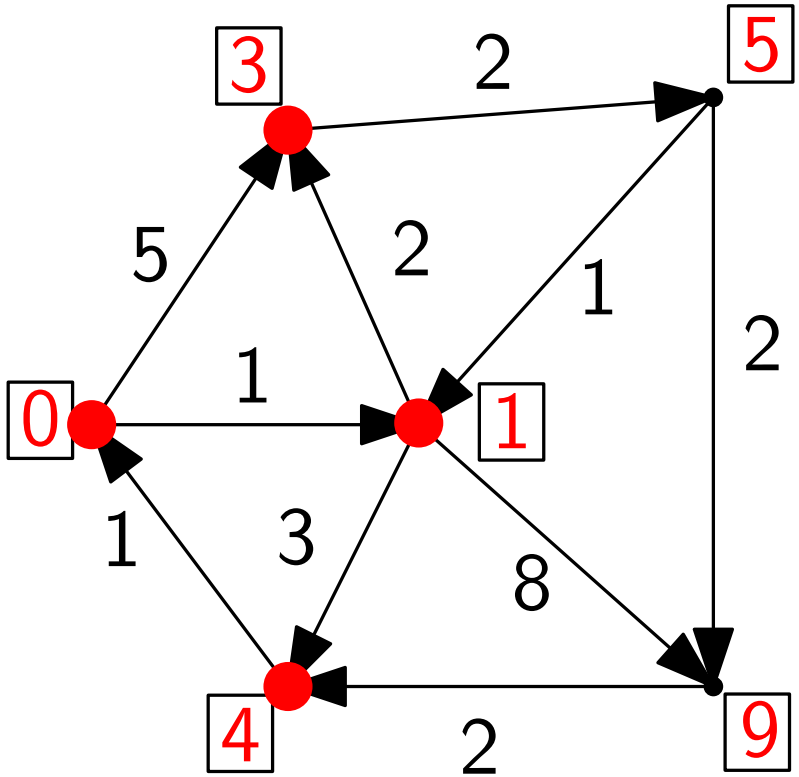


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

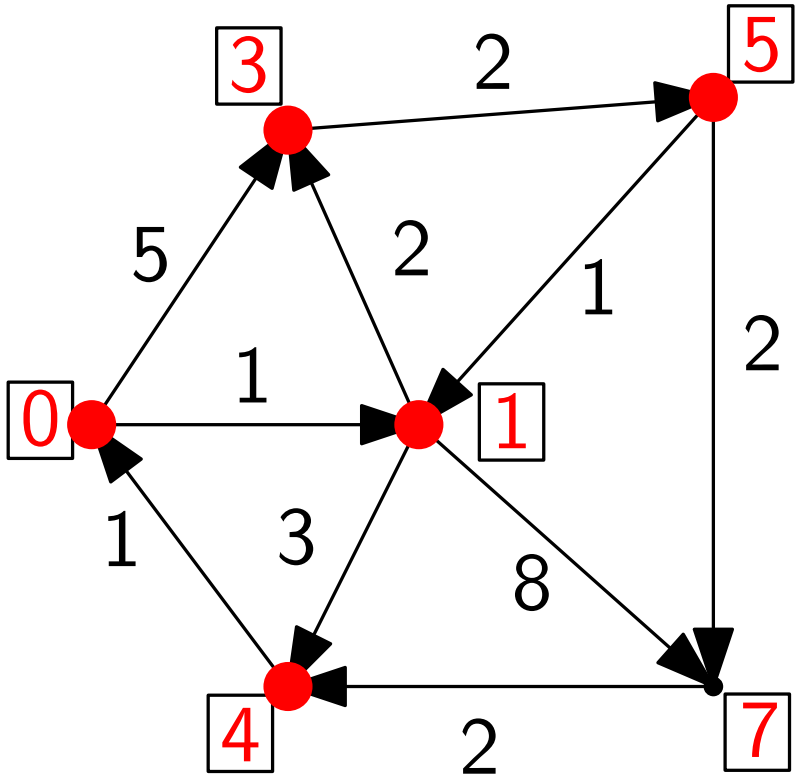


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

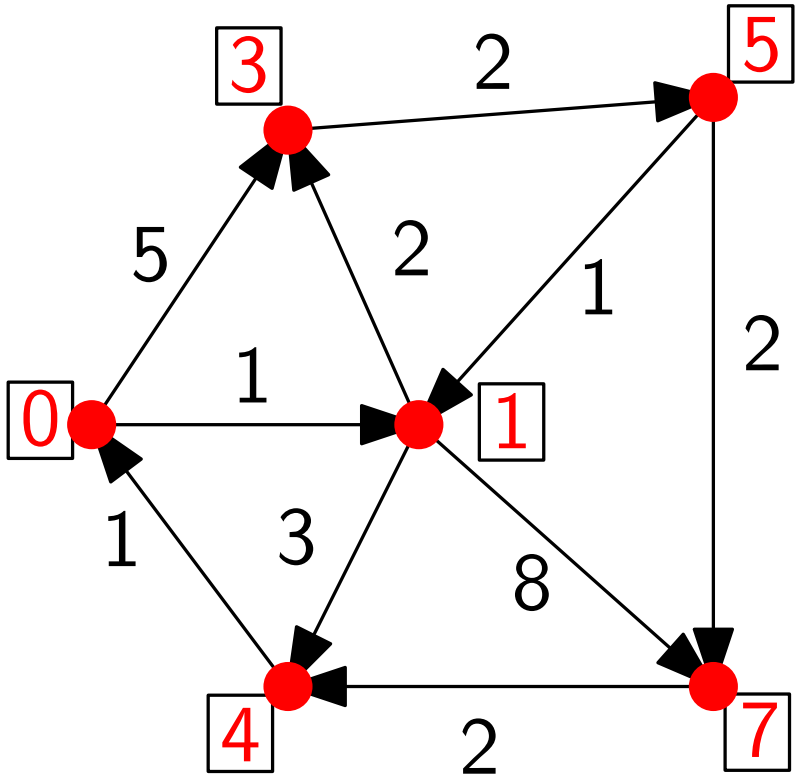


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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

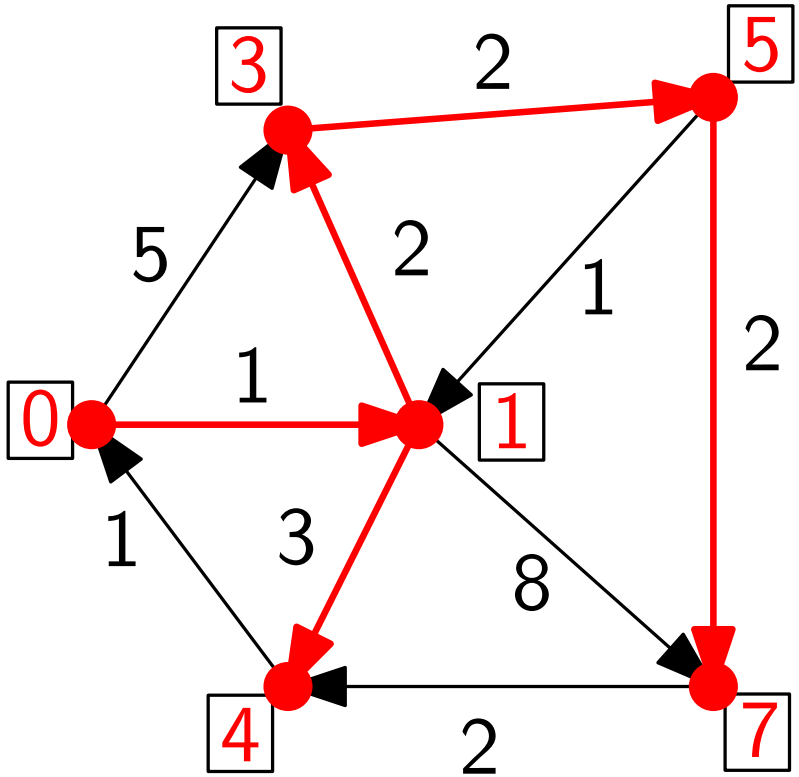


Invariante: Pfade zu *roten* Knoten sind optimal.

Dijkstras Algorithmus

Gegeben: Graph G, Startknoten.

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



Invariante: Pfade zu *roten* Knoten sind optimal.

Kantenrelaxierung

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                    // distances, parents
3  d[s] = 0
4  Q.clear(), Q.insert(s, 0)                    // container
5  while !Q.empty() do
6      u ← Q.deleteMin()                        // settling node u
7      forall the edges e = (u, v) ∈ E do
           // relaxing edges
8          if d[u] + len(e) < d[v] then
9              d[v] ← d[u] + len(e)
10             p[v] ← u
11             if v ∈ Q then Q.decreaseKey(v, d[v])
12
13             else Q.insert(v, d[v])
```

Definitionen

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*.

Definitionen

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).

Definitionen

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/scanned)* wenn er (zu diesem Zeitpunkt) schon in die Queue eingefügt und wieder extrahiert wurde.

Komplexität von Dijkstra

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL
3  d[s] = 0, Q.clear(), Q.insert(s, 0)
4  while !Q.empty() do
5      u ← Q.deleteMin()
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11
12             else Q.insert(v, d[v])
13
```

Komplexität von Dijkstra

Karlsruher Institut für Technologie

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)
4  while !Q.empty() do
5      u ← Q.deleteMin()
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11
12             else Q.insert(v, d[v])
13
```

Komplexität von Dijkstra

Karlsruher Institut für Technologie

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11
12             else Q.insert(v, d[v])
13
```

Komplexität von Dijkstra

Karlsruher Institut für Technologie

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11
12             else Q.insert(v, d[v])
13
```


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Komplexität von Dijkstra

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])   // m Mal
11
12             else Q.insert(v, d[v])
13
```

Komplexität von Dijkstra

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])   // m Mal
11
12             else Q.insert(v, d[v])                 // n Mal
13
```

Komplexität von Dijkstra

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

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.insert(s, 0)                // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              p[v] ← u
10             if v ∈ Q then Q.decreaseKey(v, d[v])   // m Mal
11
12             else Q.insert(v, d[v])                 // n Mal
13
```

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

Laufzeit Dijkstra

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

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

Laufzeit Dijkstra

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

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

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Laufzeit Dijkstra Experimente

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

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

Graph: 18M Knoten, 42M Kanten

Laufzeit Dijkstra Experimente

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

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

Graph: 18M Knoten, 42M Kanten

Dijkstra: $\approx 1.5\,\text{s}$ $\Rightarrow$ nicht interaktiv

Laufzeit Dijkstra Experimente

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

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

Graph: 18M Knoten, 42M Kanten

Dijkstra:	$\approx 1.5\,\text{s}$	$\Rightarrow$ nicht interaktiv
$n + m$ CPU clock cycles:	$\approx 30\,\text{ms}$	$\Rightarrow$ viel schneller

Laufzeit Dijkstra Experimente

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

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

Graph: 18M Knoten, 42M Kanten

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

Laufzeit Dijkstra Experimente

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

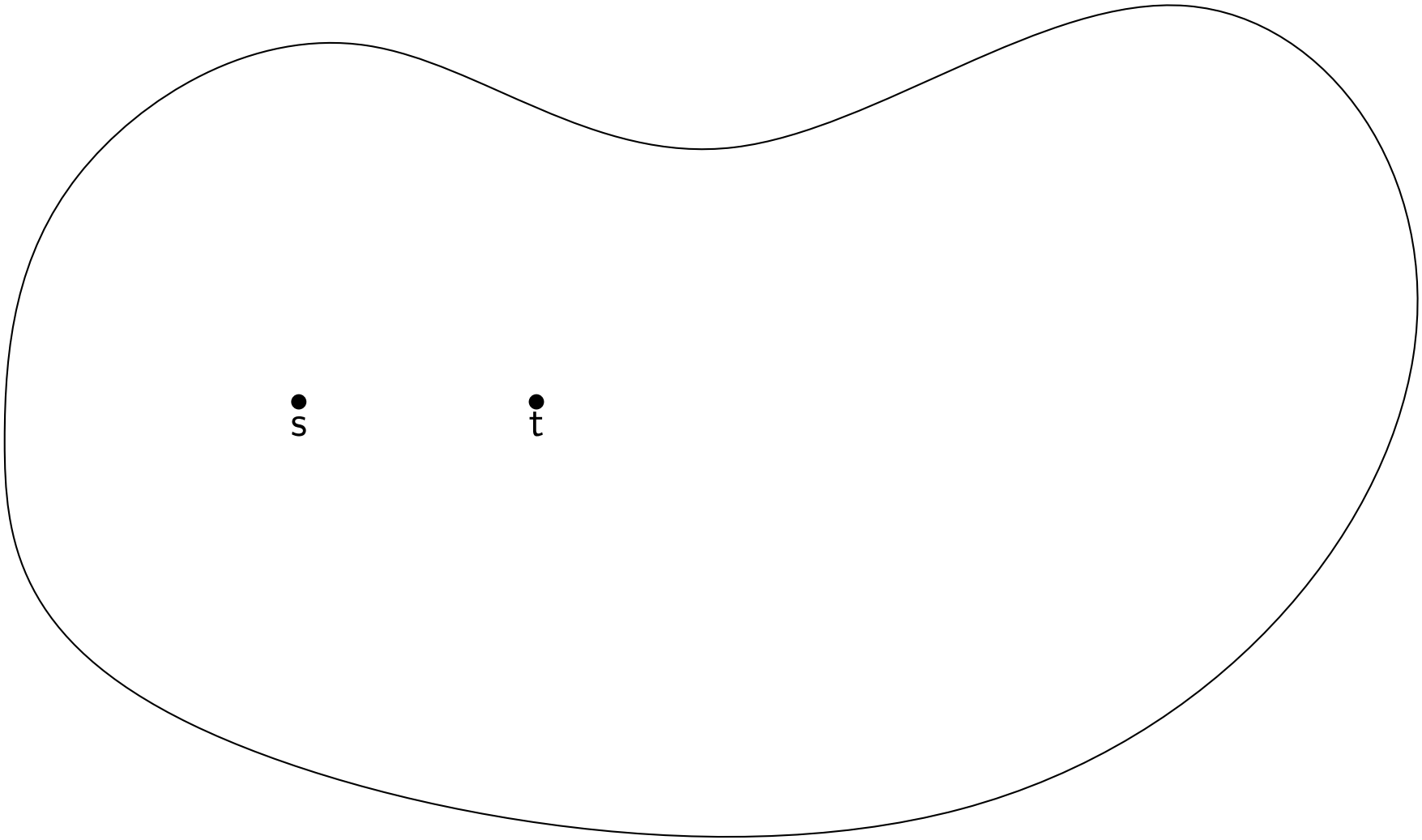
k	Query [sec]
2	1.834
3	1.595
4	1.507
5	1.525
8	1.561

Graph: 18M Knoten, 42M Kanten

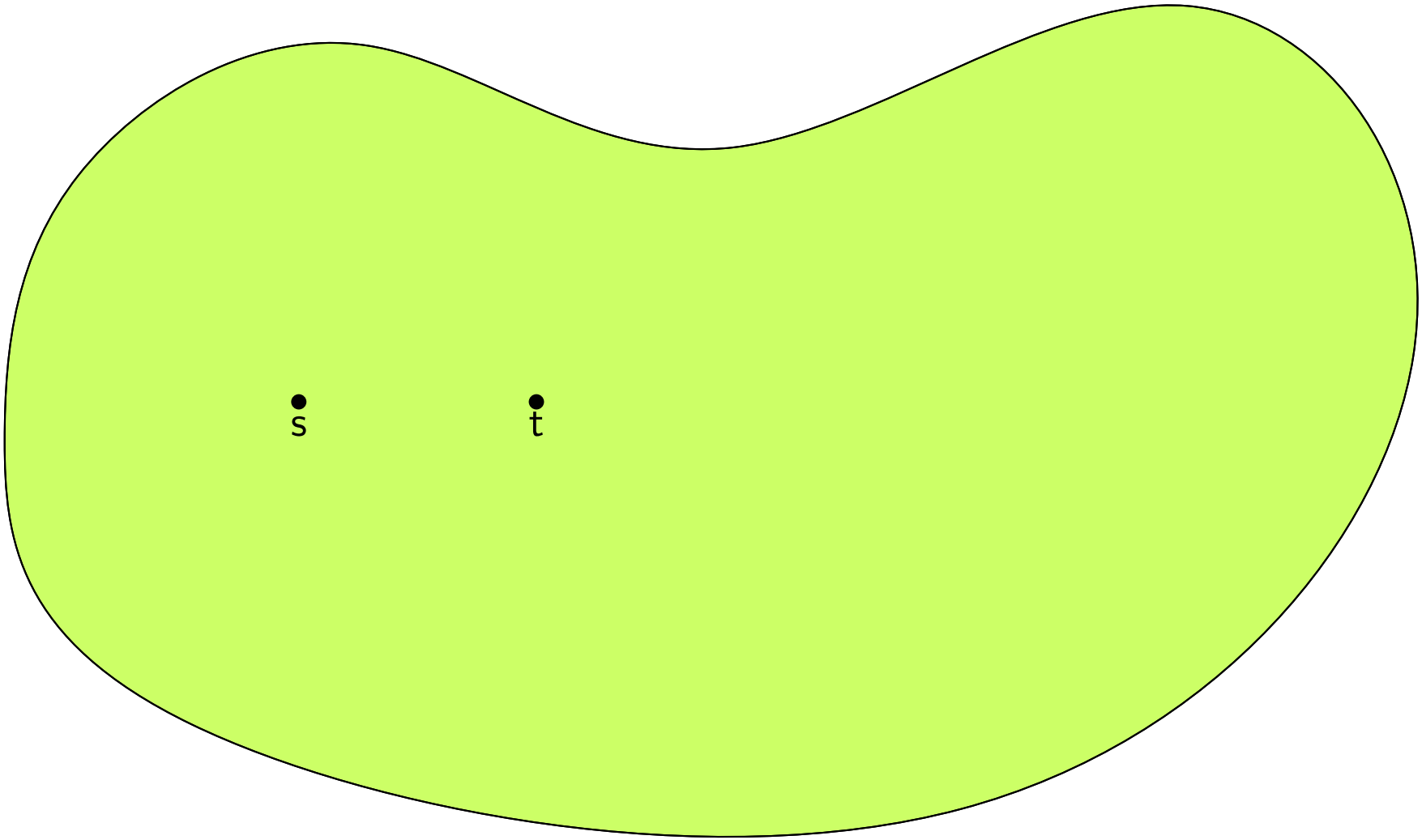
Dijkstra:	$\approx 1.5\,s$	$\Rightarrow$ nicht interaktiv
$n + m$ CPU clock cycles:	$\approx 30\,ms$	$\Rightarrow$ viel schneller
BFS:	$\approx 1.2\,s$	$\Rightarrow$ an der Queue liegt's nicht

Performanz von Graphsuchen ist speicher-begrenzt

Schematischer Suchraum, Dijkstra



Schematischer Suchraum, Dijkstra



Abbruch

Beobachtung

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

Abbruch

Beobachtung

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

Idee

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

Abbruch

Beobachtung

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

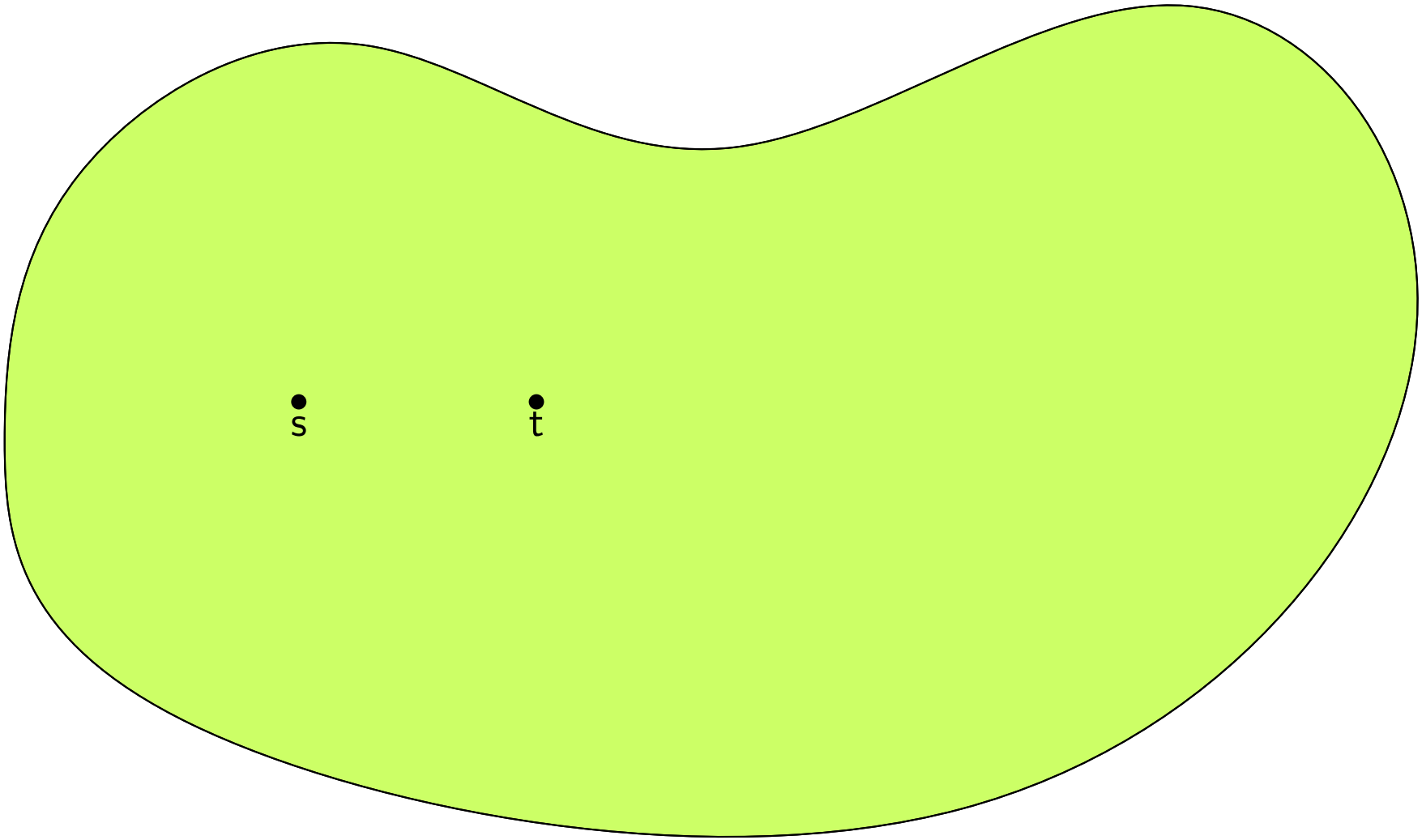
Idee

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

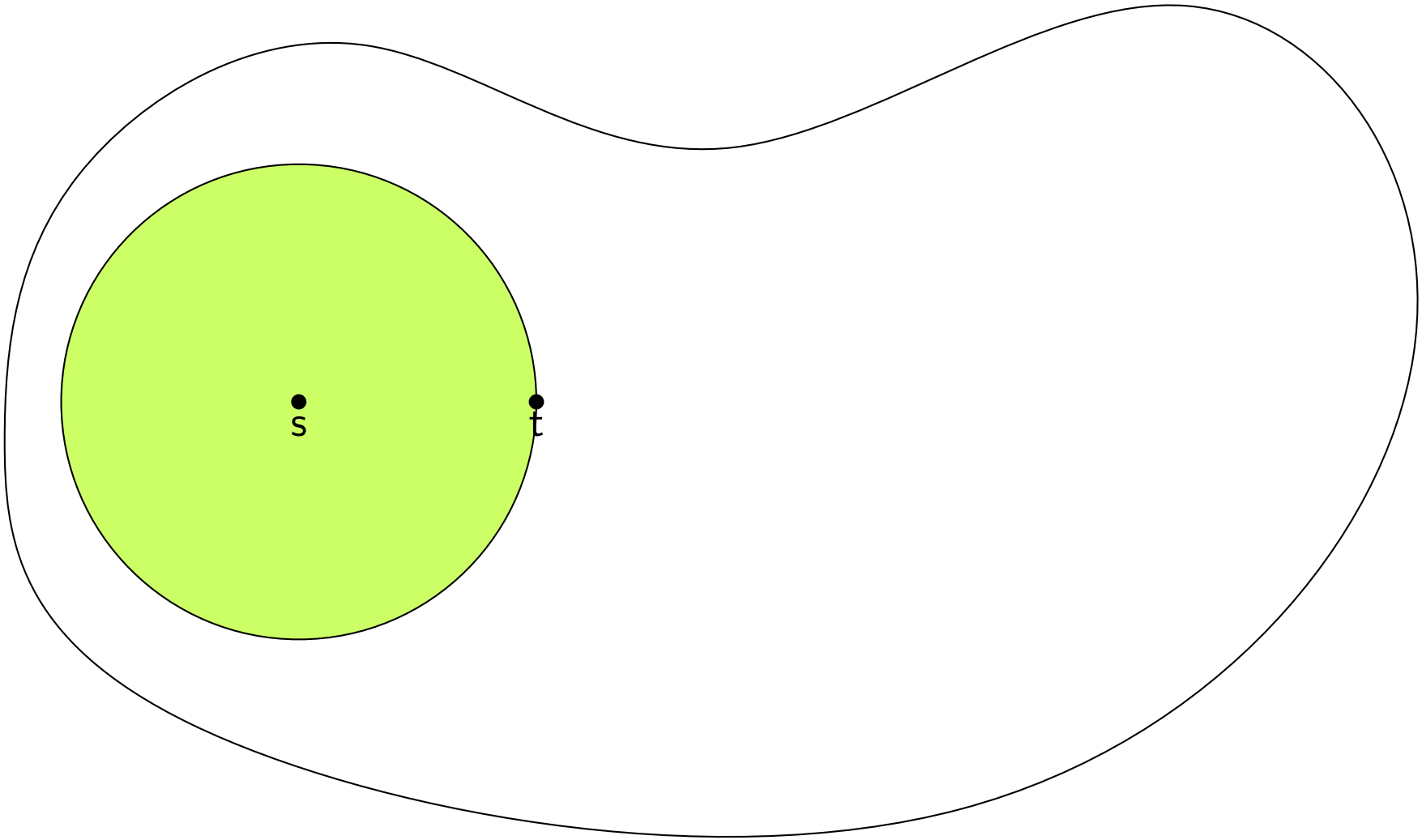
Korrektheit

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

Schematischer Suchraum, Dijkstra



Schematischer Suchraum, Dijkstra



Dijkstra mit Abbruchkriterium

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

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

Frage

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

Dijkstra mit Abbruchkriterium

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

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

Dijkstra mit Abbruchkriterium

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

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

Viele Anfragen

Problem

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

Viele Anfragen

Problem

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

Idee

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

Dijkstra mit Timestamps

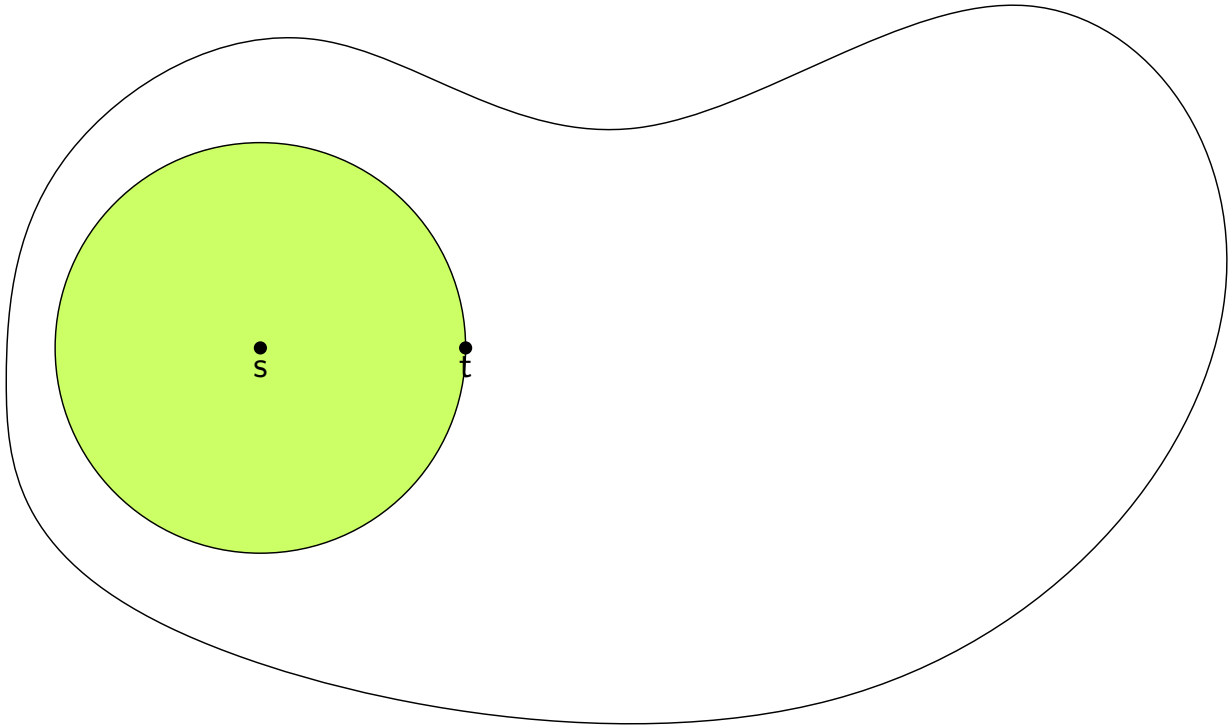
```
1  count ← count + 1                                   // Überlauf
2  d[s] = 0
3  Q.clear(), Q.add(s, 0)
4  while !Q.empty() do
5      u ← Q.deleteMin()
6      if u = t then return
7      forall the edges e = (u, v) ∈ E do
8          if run[v] ≠ count then
9              d[v] ← d[u] + len(e)
10 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


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Bidirektionale Suche

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

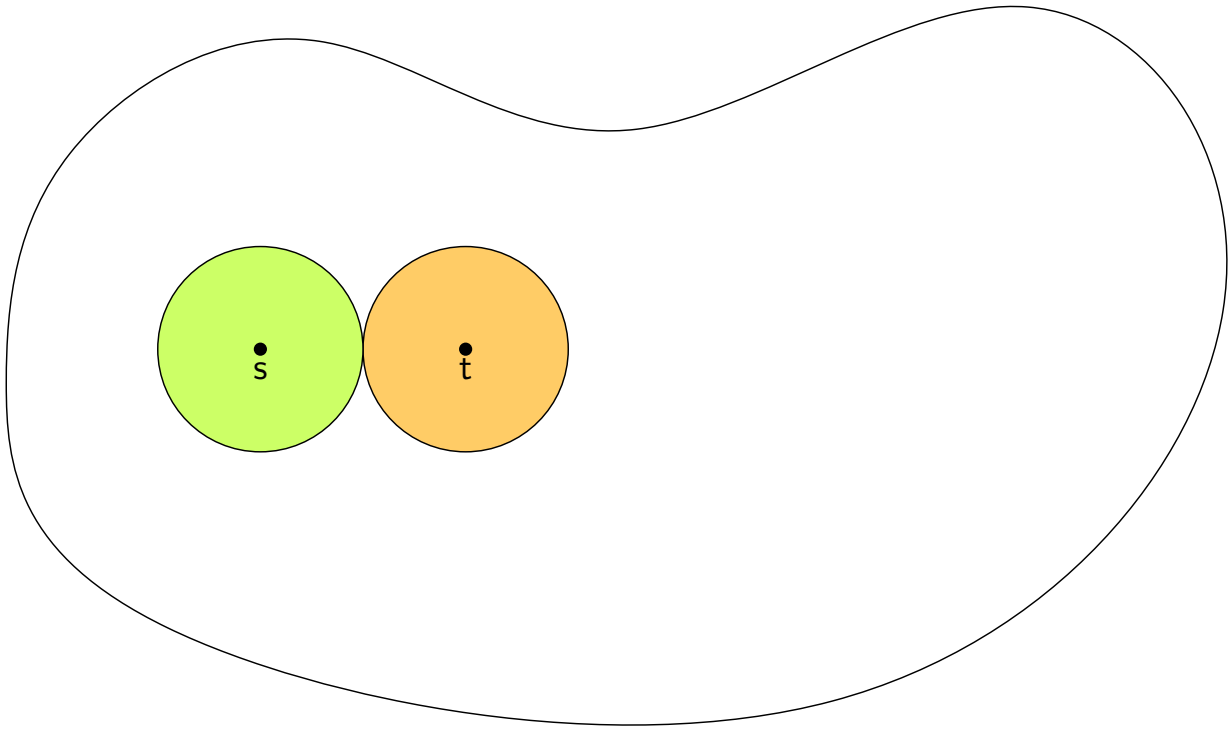
Bidirektionale Suche



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

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

Bidirektionale Suche



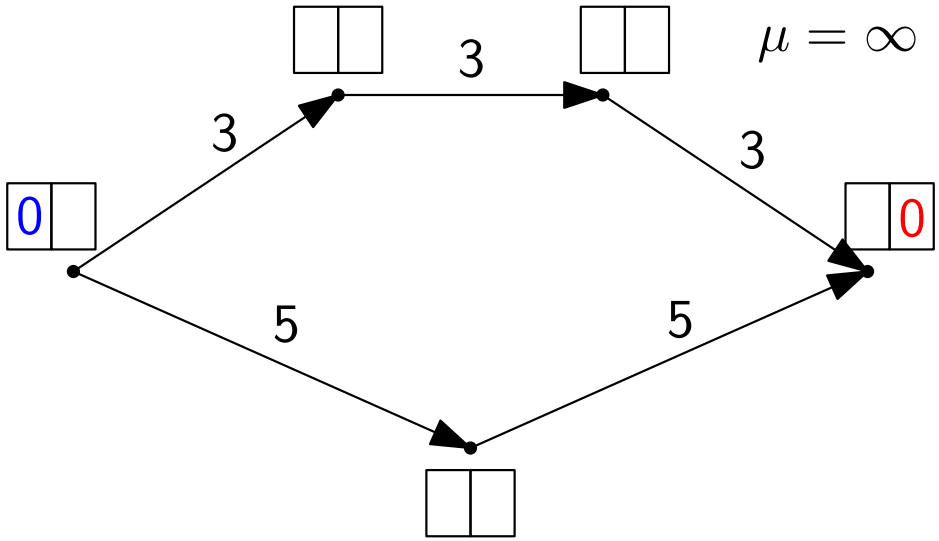
Idee: Kombiniere beide Suchen

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

Bidirektionale Suche

Anfrage:

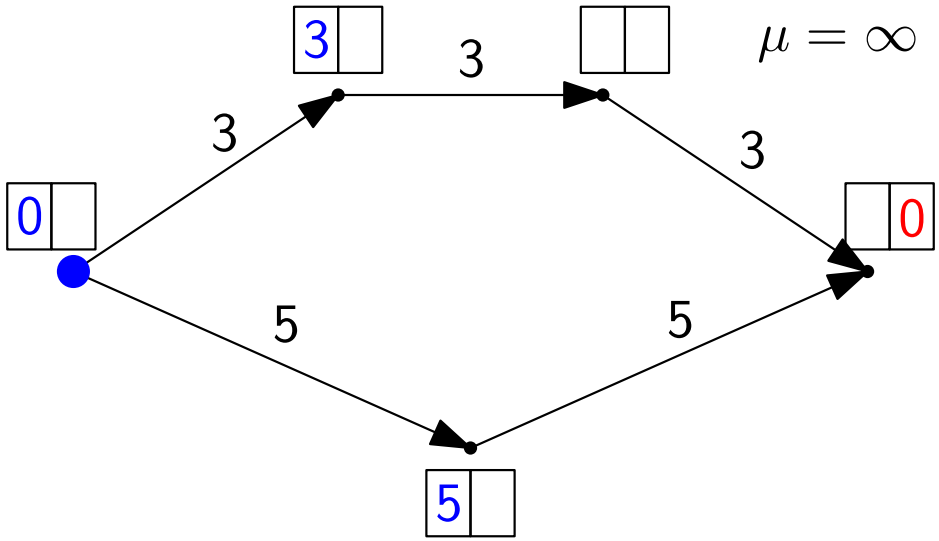
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

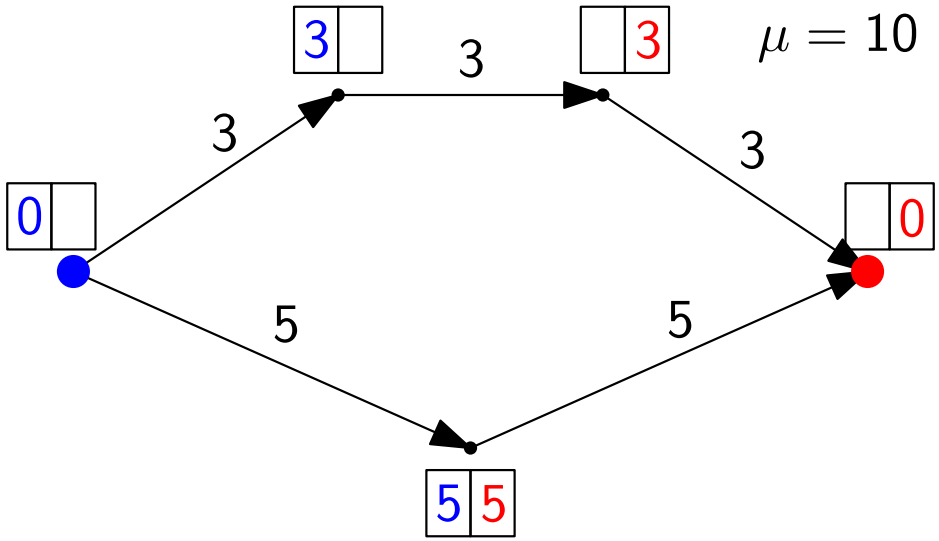
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

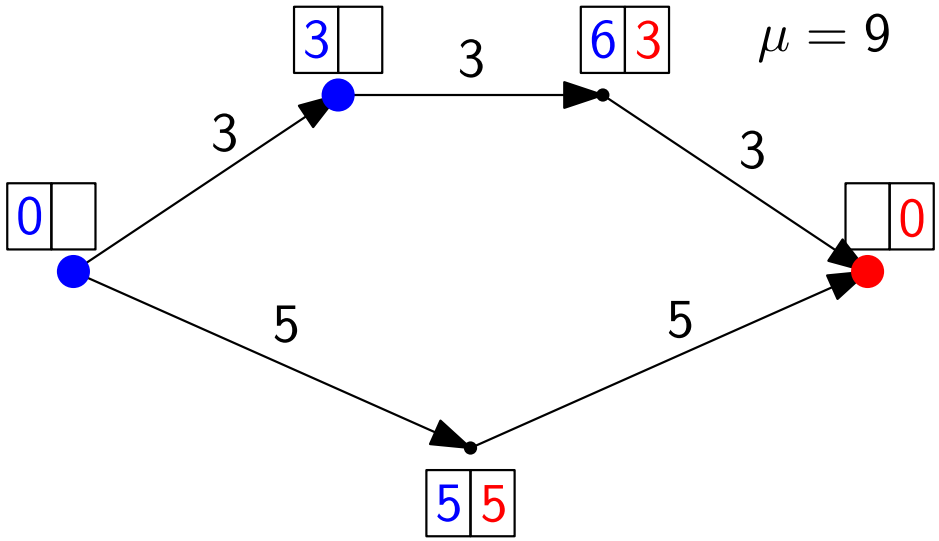
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

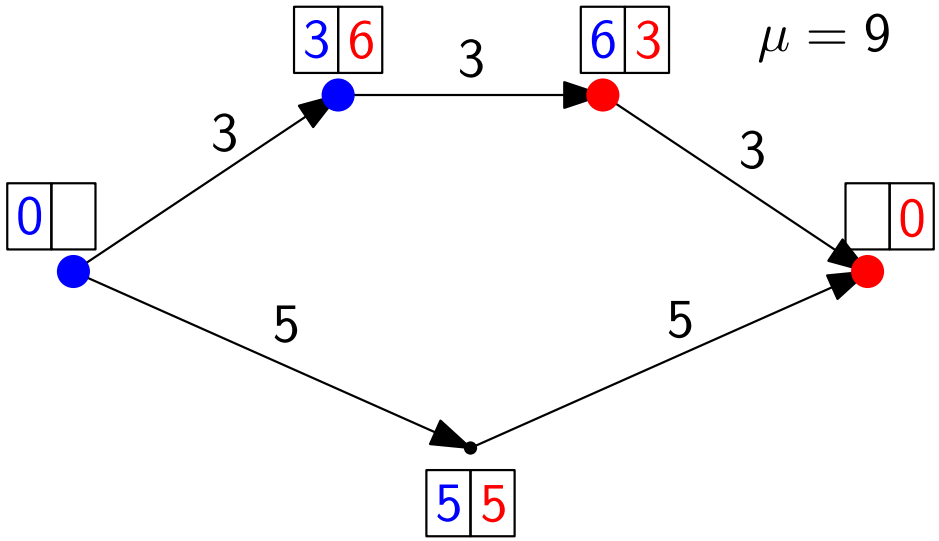
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

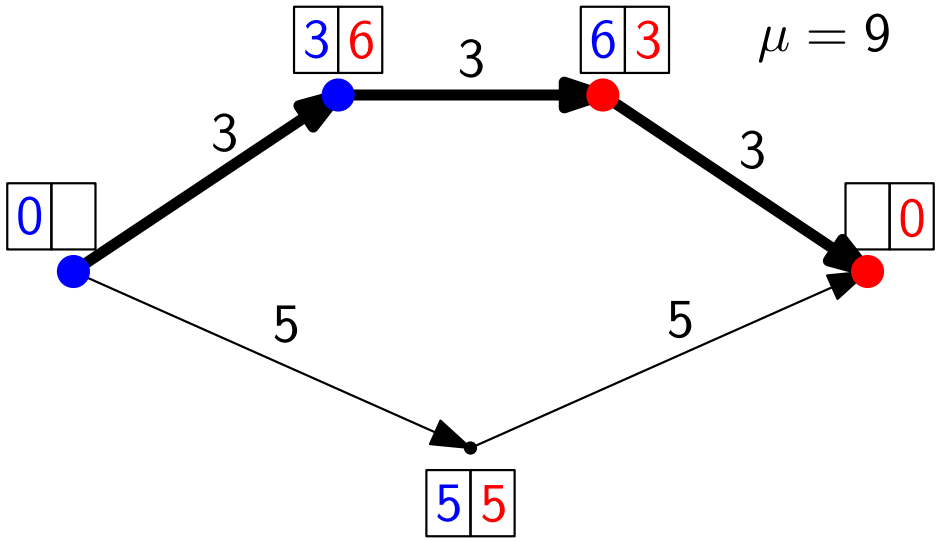
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

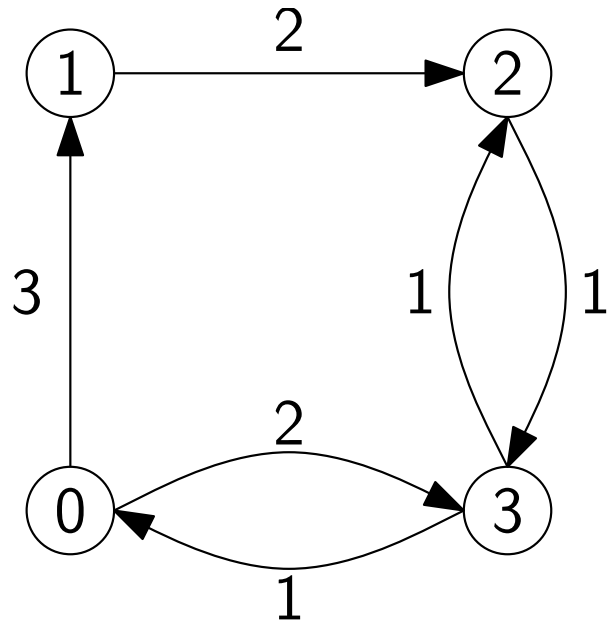
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Graph-Datenstruktur

Problem:

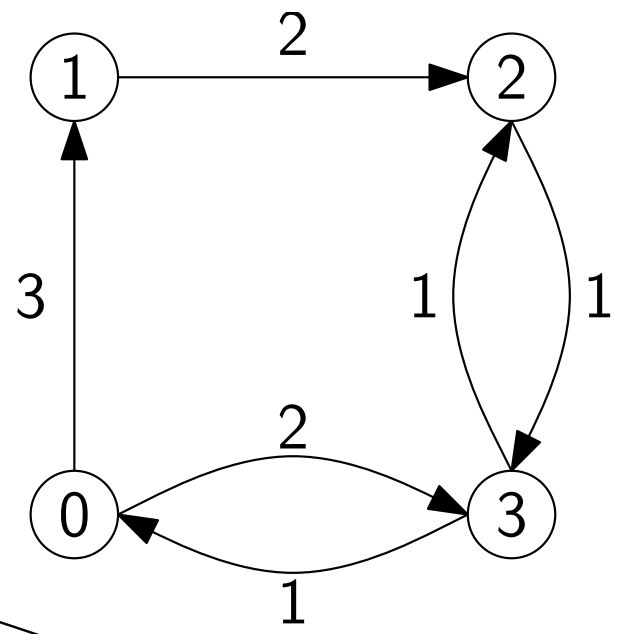
- ein- und ausgehende Kanten-Inzidenz benötigt
- Graph (fast) ungerichtet



Graph-Datenstruktur

Problem:

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



`firstEdge`	0	3	5	7	10					
`targetNode`	1	3	3	0	2	1	3	0	0	2
`weight`	3	2	1	3	2	2	1	1	2	1
`isIncoming`	—	—	√	√	—	√	√	—	√	√
`isOutgoing`	√	√	—	—	√	—	√	√	—	√

Bidirektionale Suche - Definitionen

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

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

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

Bidirektionale Suche - Vorgehen

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

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

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

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

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

Bidirektionale Suche - Vorgehen

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

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

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

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

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

Was sind gute Abbruchstrategien?

Abbruchstrategie (1)

Abbruchstrategie (1)

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

Abbruchstrategie (1)

Abbruchstrategie (1)

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

Frage: Ist m dann auf einem kürzesten s-t-Weg enthalten?

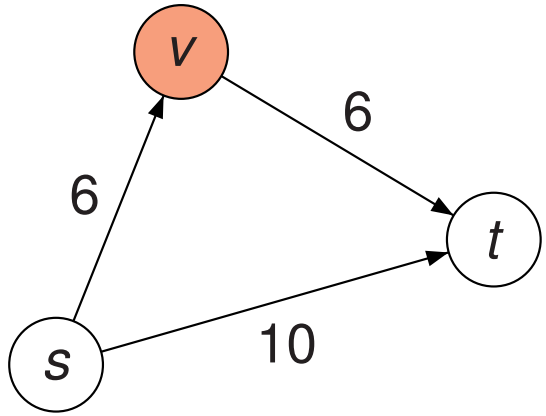
Abbruchstrategie (1)

Abbruchstrategie (1)

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

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

Nein, Gegenbeispiel:



Abbruchstrategie (1)

Abbruchstrategie (1)

Abbruch, sobald ein Knoten m 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: $\overrightarrow{d}[v] + \overleftarrow{d}[v] \geq \text{dist}(s,t)$.
- Seien $\overrightarrow{S}, \overleftarrow{S}$ die abgearbeiteten Knoten von Vor- und Rückwärtssuche nach Terminierung.
- Sei $P = (v_1, \ldots v_k)$ ein kürzester s-t-Weg. Wir zeigen: $\{v_1, \ldots v_k\} \subseteq \overrightarrow{S} \cup \overleftarrow{S}$ oder $\text{len}(P) = \text{dist}(s,m) + \text{dist}(m,t)$.

Abbruchstrategie (1)

Abbruchstrategie (1)

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

- Sei $P = (v_1, \ldots v_k)$ ein kürzester s-t-Weg. Wir zeigen: $\{v_1, \ldots v_k\} \subseteq \overrightarrow{S} \cup \overleftarrow{S}$ oder $\text{len}(P) = \text{dist}(s, m) + \text{dist}(m, t)$.
- Angenommen es gibt $v_i \notin \overrightarrow{S} \cup \overleftarrow{S}$.
- Dann gilt

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

Also

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

Abbruchstrategie (2)

Abbruchstrategie (2)

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

Abbruchstrategie (2)

Abbruchstrategie (2)

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

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

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

Abbruchstrategie (2)

Abbruchstrategie (2)

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

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

- Dann gibt es einen s-t Pfad P der kürzer als μ ist.
- Auf P gibt es eine Kante (u,v) mit $d(s,u) \leq \mathit{minKey}(\overrightarrow{Q})$ und $d(v,t) \leq \mathit{minKey}(\overleftarrow{Q})$.
- 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 μ aktualisiert

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

Wechselstrategien

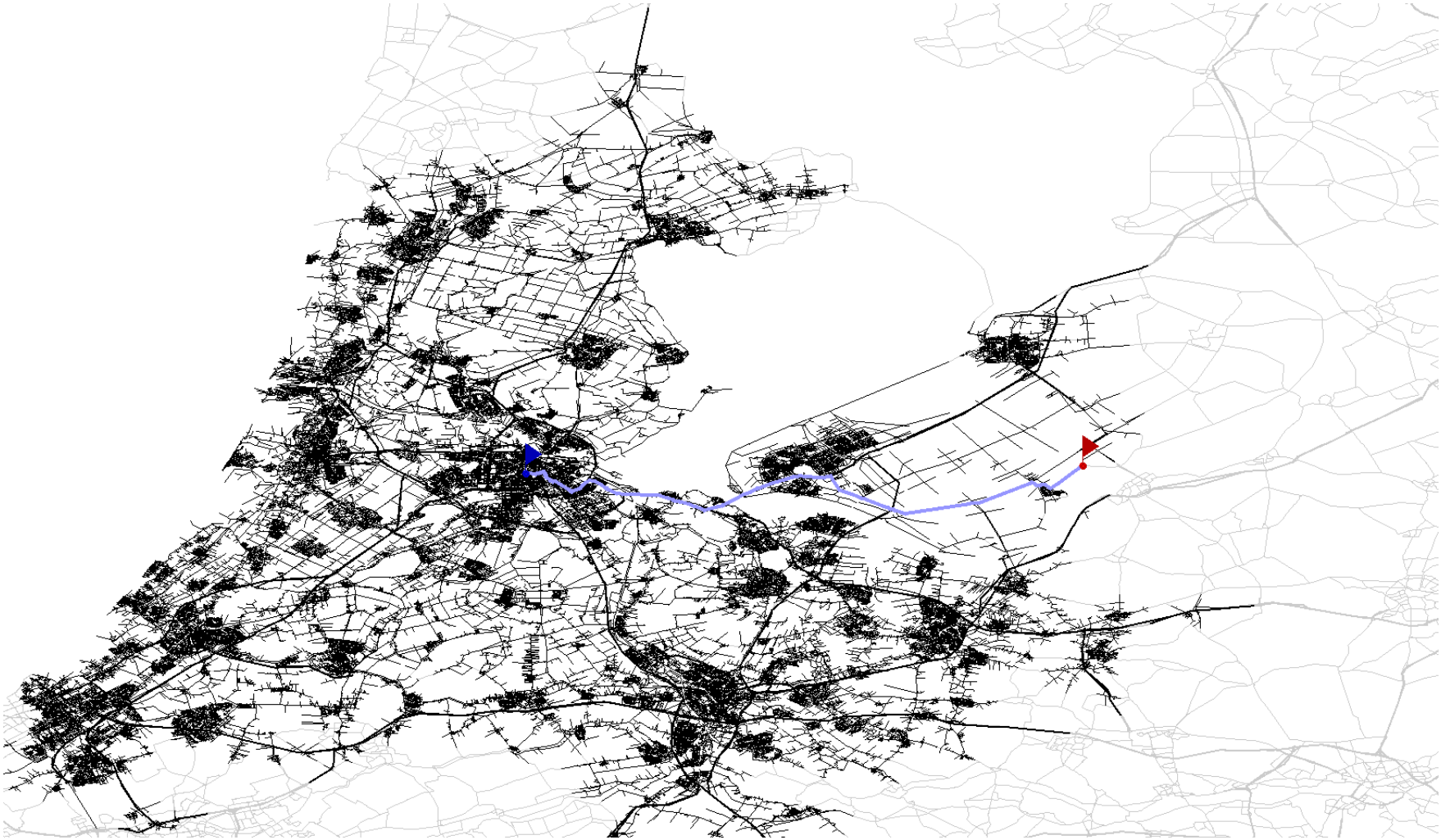
Frage: Was sind mögliche Wechselstrategien?

Wechselstrategien

Mögliche Wechselstrategien

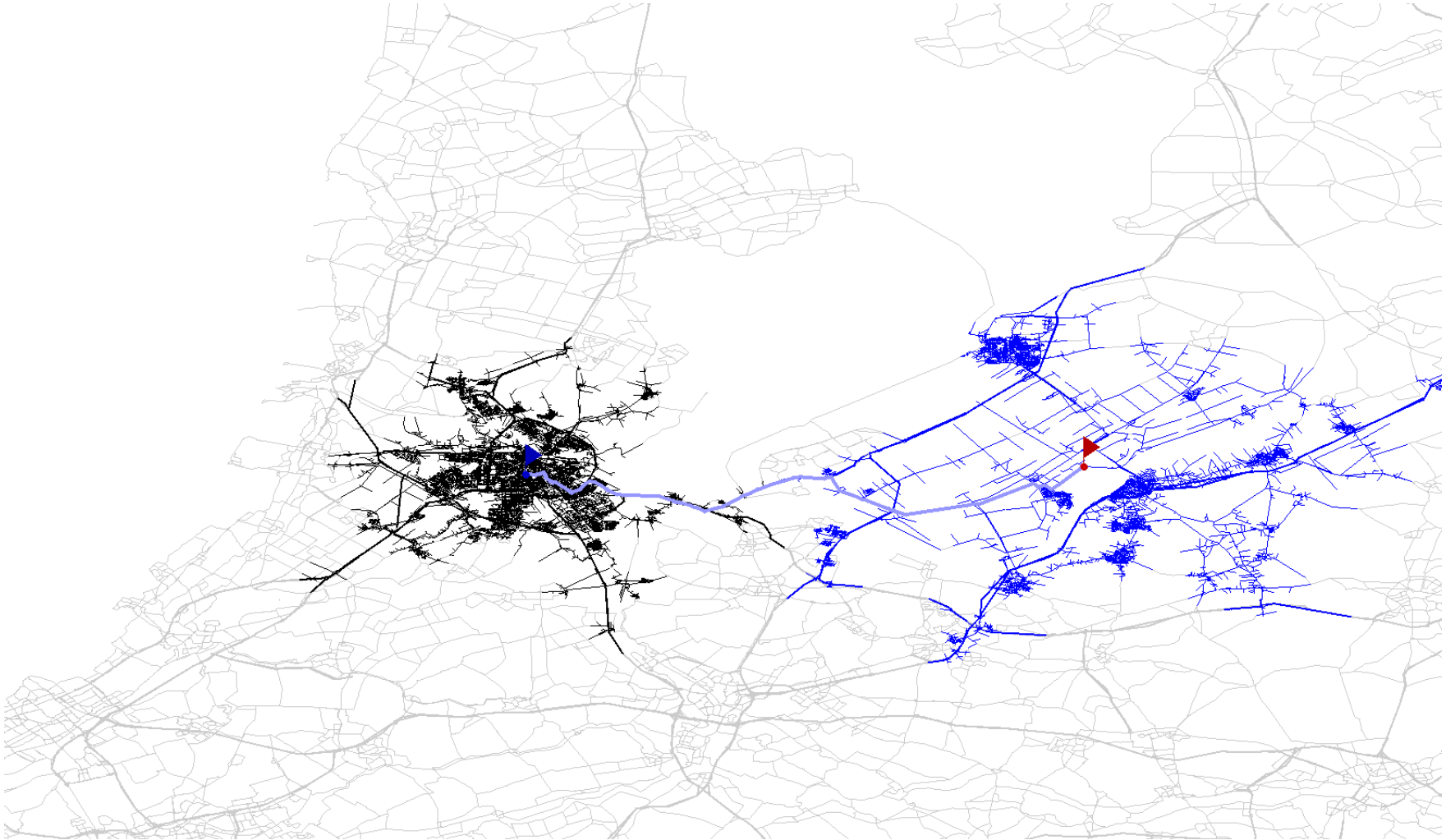
- Prinzipiell jede Wechselstrategie möglich
- Wechsle nach jedem Schritt zur entgegengesetzten Suche
- Führe immer die Suche mit der kleineren Anzahl Elemente in Queue aus
- Führe immer die Suche mit dem kleineren minimalen Queueelement aus
- Oder: Parallele Ausführung auf zwei Kernen

Beispiel



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Beispiel



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Bemerkungen

Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius r.

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

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

- Führe Suchen *parallel* aus.

$\Rightarrow$ Gesamtspeedup ca. 4.

Bemerkungen

Beschleunigung

- Annahme: Suchraum ist Kreisscheibe mit Radius r.

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

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

- Führe Suchen *parallel* aus.

$\Rightarrow$ Gesamtspeedup ca. 4.

Wichtiger Bestandteil vieler effizienter Techniken!

Dijkstra-Rank

Problem:

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

Dijkstra-Rank

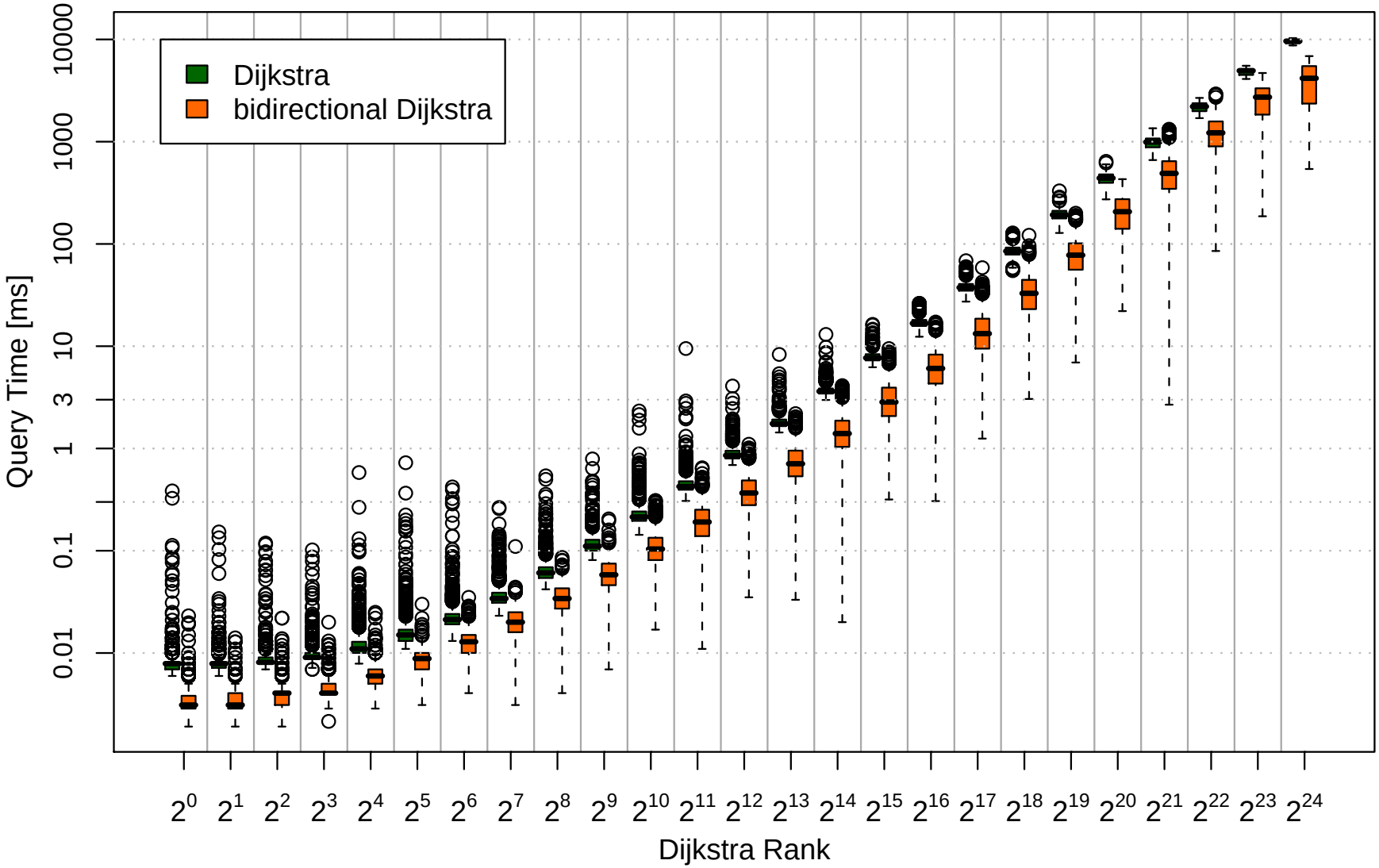
Problem:

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

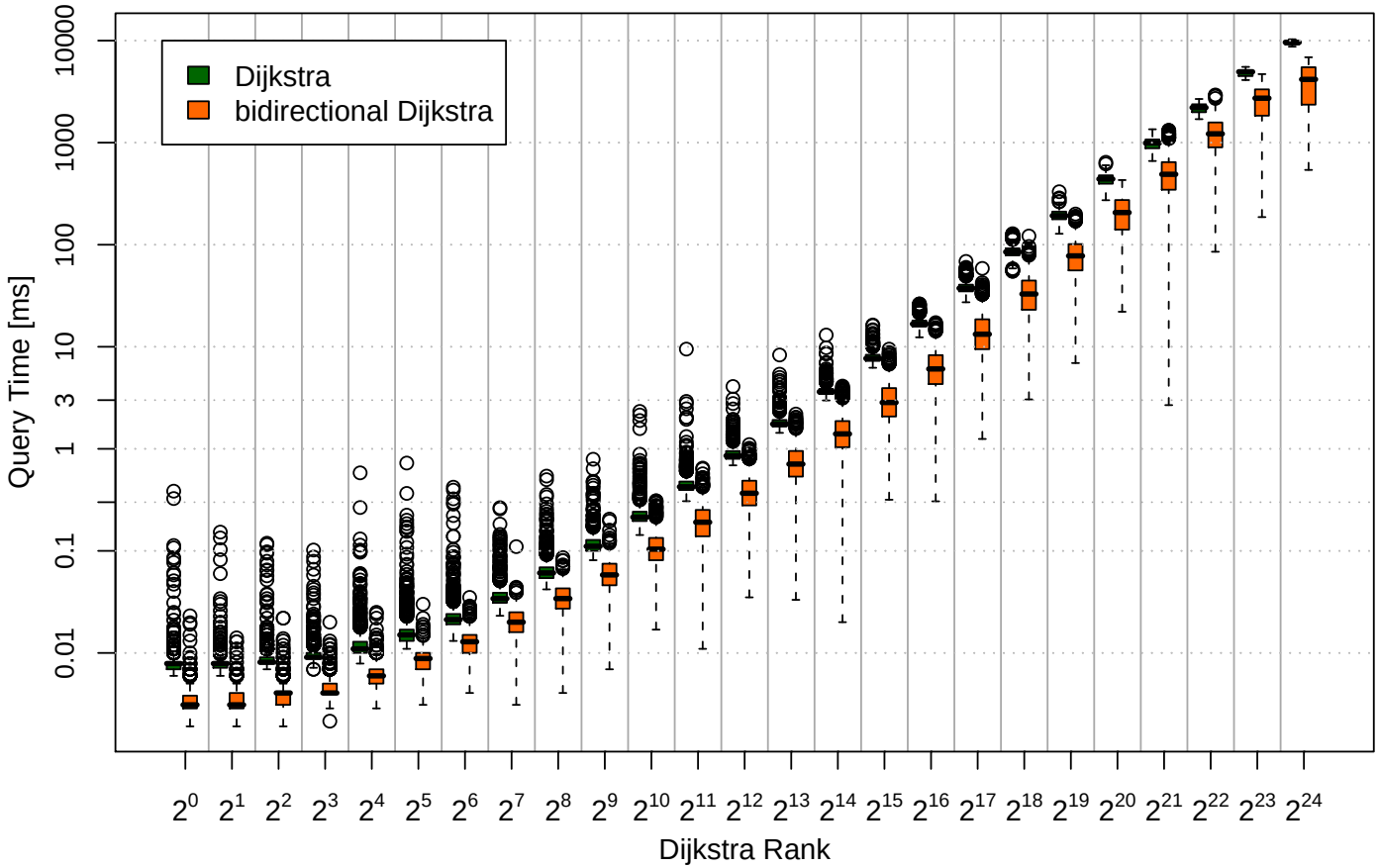
Idee:

- 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 $2^1, \ldots, 2^{\log n}$ 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

Nächster Termin

Karlsruher Institut für Technologie

Montag, 20. April 2014

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Bibliographie I

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.

Bibliographie II

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.