Karlsruher Institut für Technologie

Algorithmen für Routenplanung

2. Termin, Sommersemester 2011

Reinhard Bauer | 18. April 2011

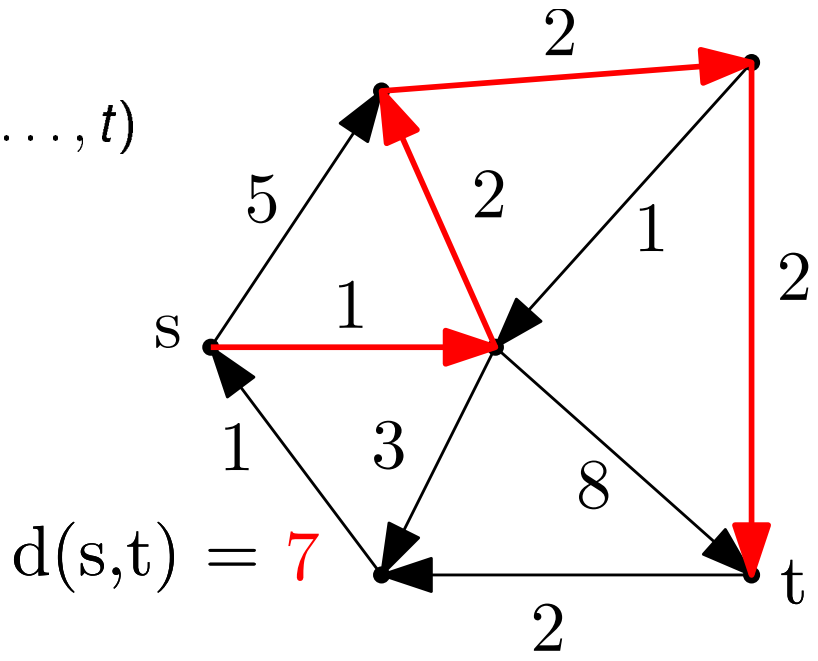
Institut für Theoretische Informatik · Algorithmik I · Prof. Dr. Dorothea Wagner

Problemstellung

Gegeben:
Graph $G = (V, E, \text{len})$ mit positiver Kantenfunktion $\text{len} : E \to \mathbb{R}_{>0}$,
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:

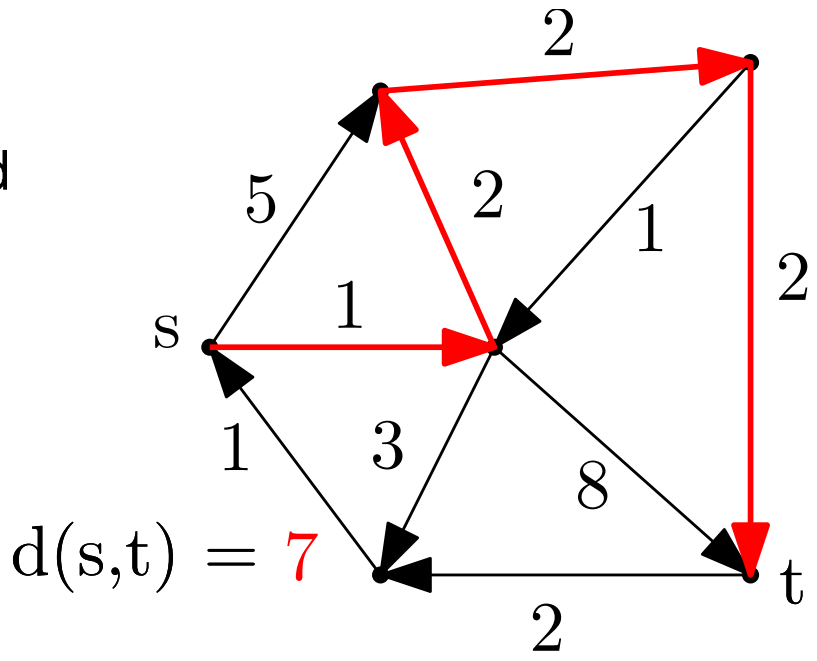
- Alle kürzesten Wege von *s* aus bilden DAG bzw. Baum

Vererbungseigenschaft:

- Subwege von kürzesten Wegen sind kürzeste Wege

Abstand:

- Steigender Abstand von Wurzel zu Blättern



Wiederholung: Priority Queues

- Datenstruktur die folgende Operationen erfüllt.
- Näheres in jedem Algorithmik-Standardlehrbuch.

Wiederholung: Priority Queues

isEmpty	
description:	checks if S is empty
output:	`true` if S is empty and `false` otherwise

insert(u, key)	
description:	inserts (u, key) in S
precondition:	S does not contain an element (u, key') for any value key$'$
action:	$S := S \cup \{(u, \text{key})\}$

contains(u)	
description:	checks if S contains an element (u, key) for some value key
output:	`true` if there is a value key such that $(u, \text{key}) \in S$, `false` otherwise

extractMin	
description:	finds and outputs an element with minimum key, removes it from S
precondition:	S is not the empty set
action:	find arbitrary element (u, key) with $\text{key} = \min\{\text{key}' \mid (v, \text{key}') \in S\}$ $S := S \setminus \{(u, \text{key})\}$
output:	u

Wiederholung: Priority Queues

DECREASEKEY(u, key)	
description:	decreases the key of element u
precondition:	S contains exactly one element (u, key') for some key$'$ and key $\leq$ key$'$ holds
action:	$S := (S \setminus \{(u, \text{key}')\}) \cup \{(u, \text{key})\}$

INSERTORUPDATE(u, key)	
description:	inserts u into the queue or updates its priority
precondition:	if S contains an element (u, key') for a value key$'$ it is key $\leq$ key$'$
action:	if CONTAINS(u) is `true` then DECREASEKEY(u, key) otherwise INSERT(u, key)

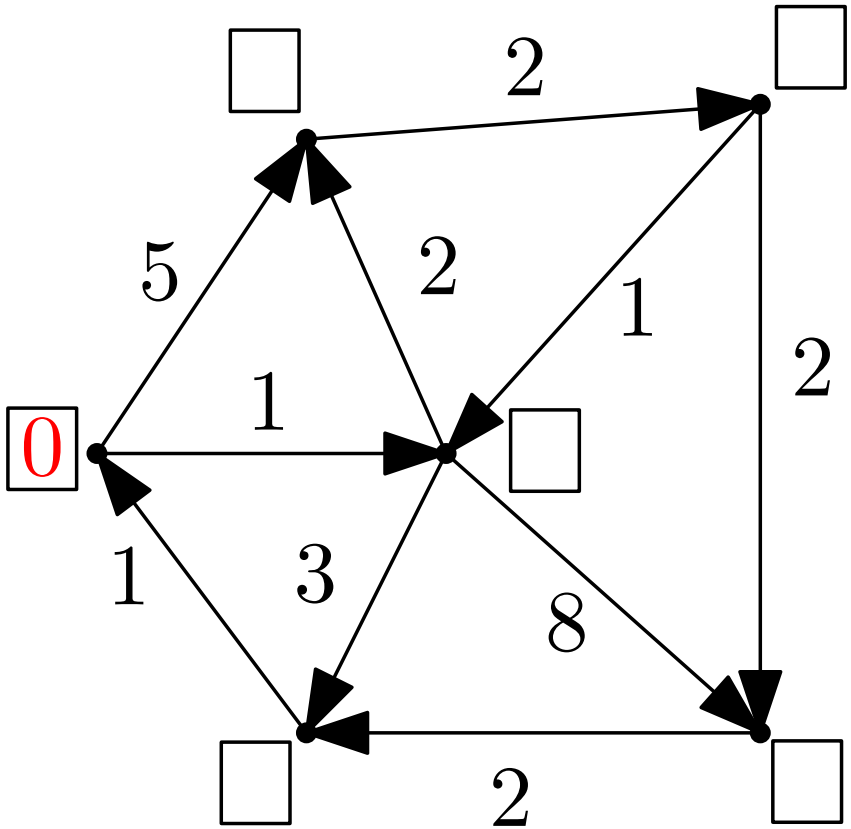
REMOVE(u)	
description:	removes u from the queue
precondition:	S contains exactly one element (u, key') for some key$'$
action:	$S := S \setminus \{(u, \text{key}')\}$

Dijkstra

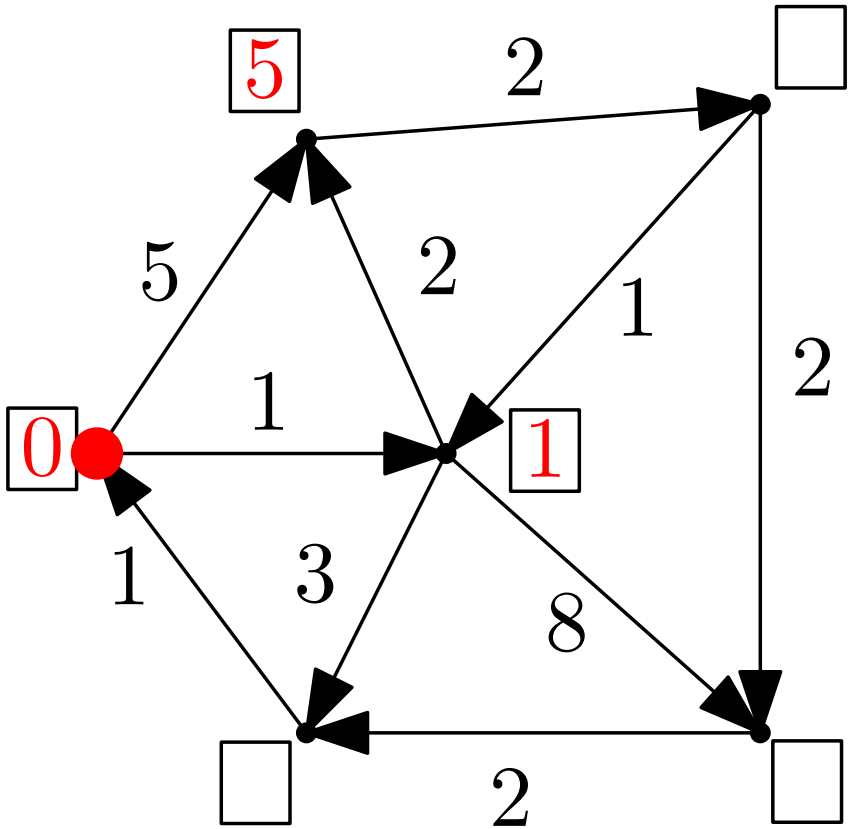
DIJKSTRA($G = (V, E)$, s)

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                    // distances, parents
3  d[s] = 0
4  Q.clear(), Q.add(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])
```

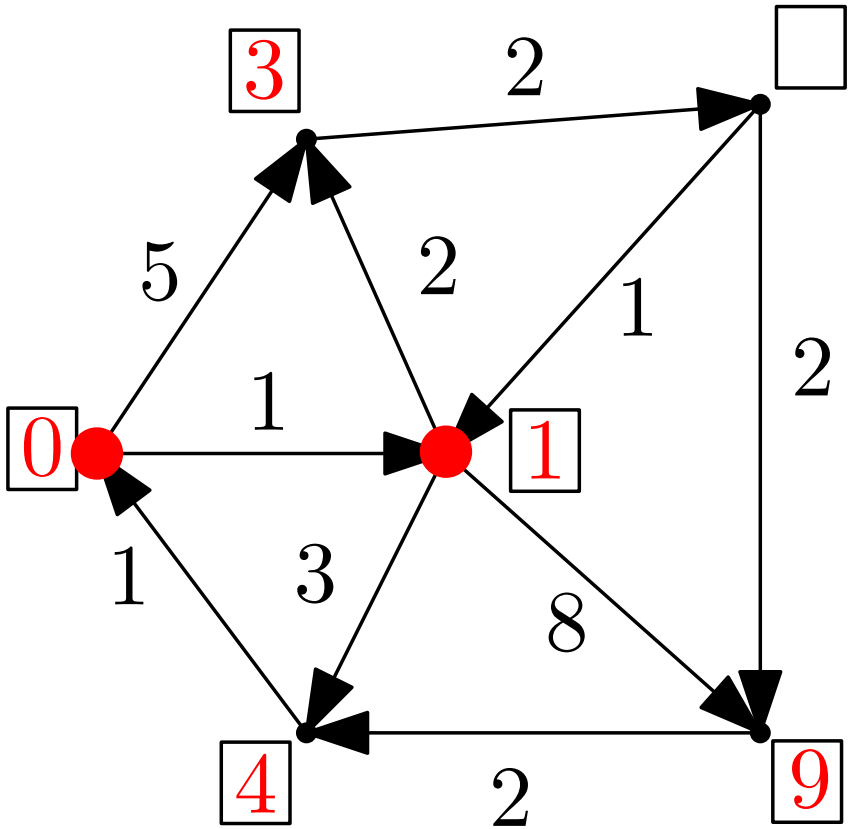
Beispiel



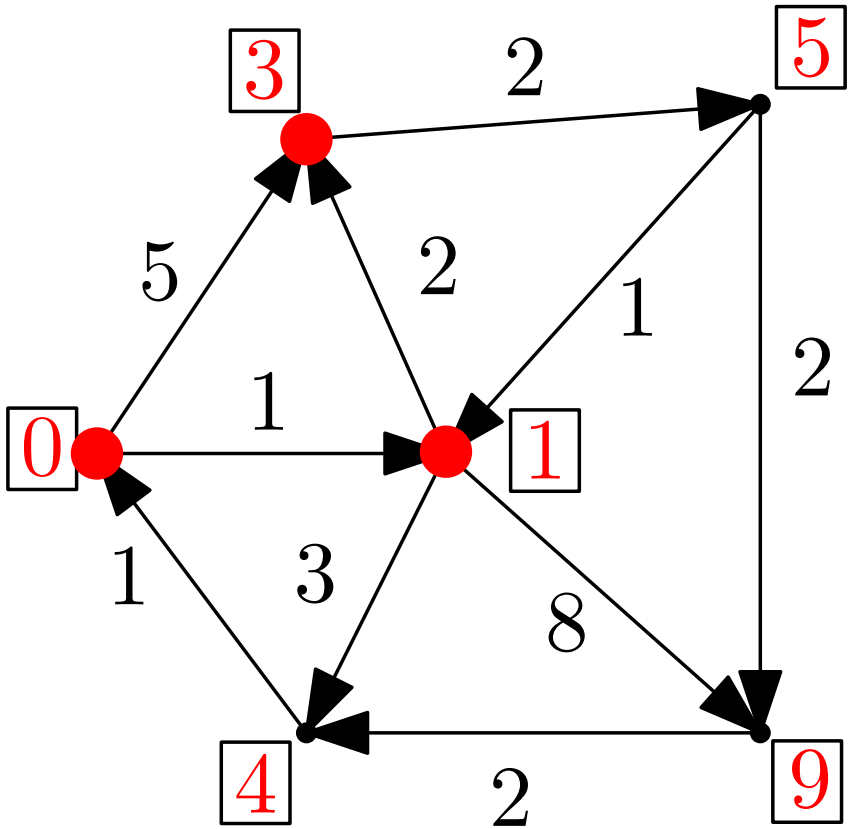
Beispiel



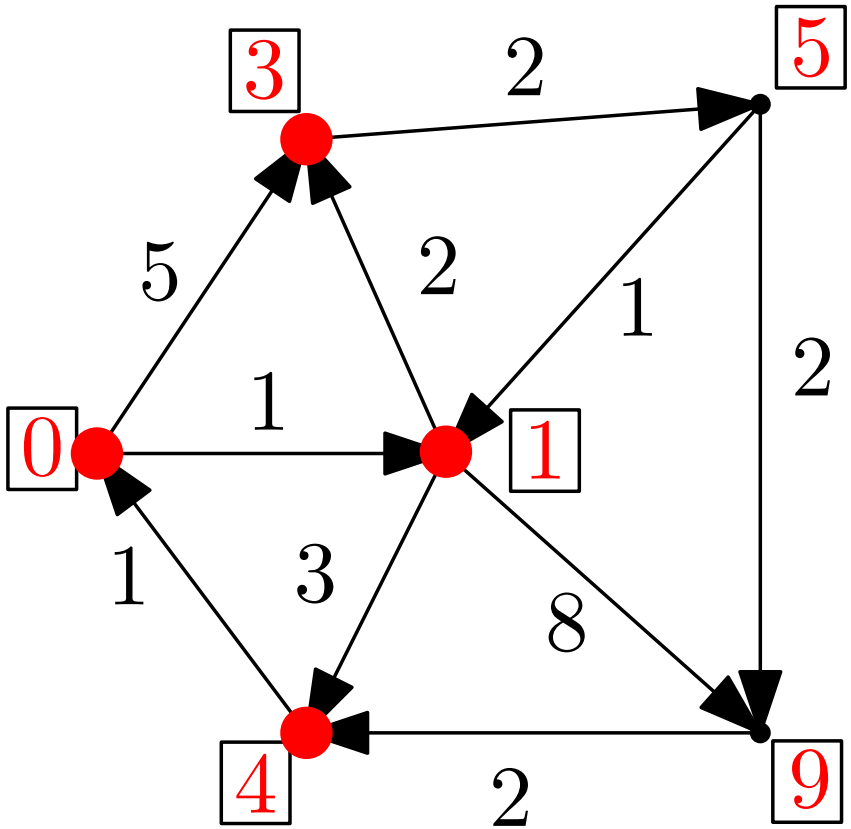
Beispiel



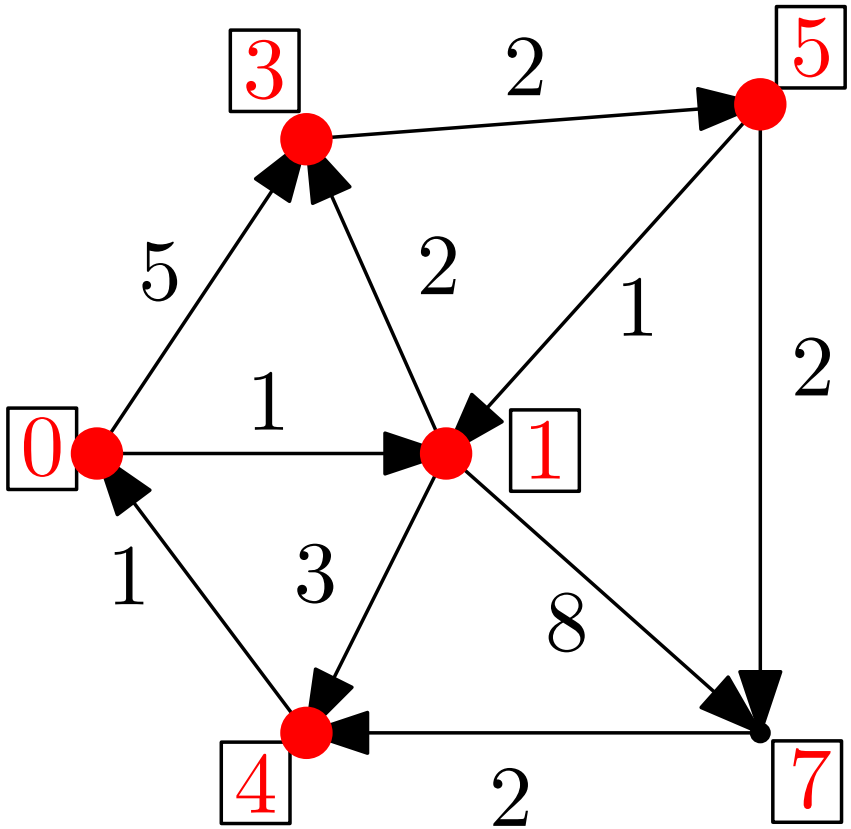
Beispiel



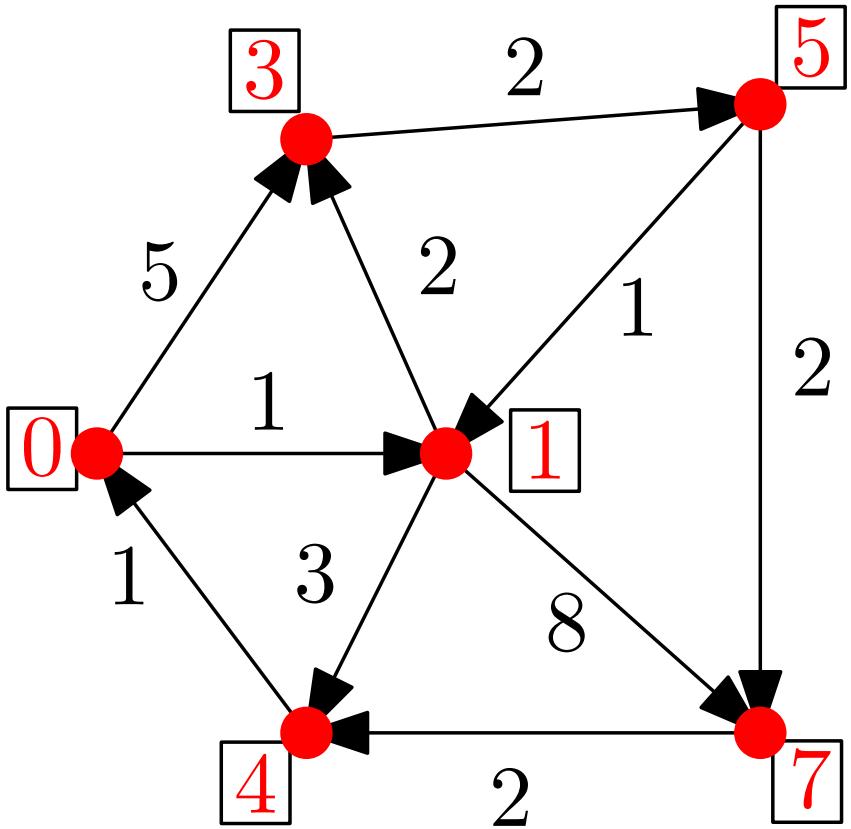
Beispiel



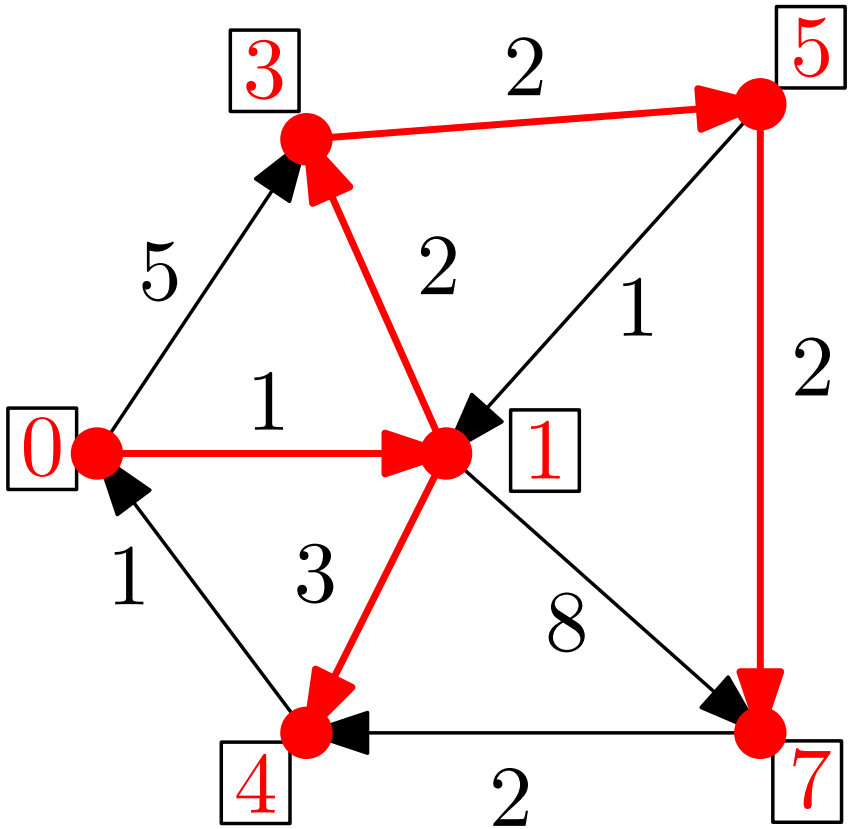
Beispiel



Beispiel



Beispiel



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.add(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])
```

Kantenrelaxierung

Der Vorgang

```
1 if d[u] + len(u, v) < d[v] then
2   d[v] ← d[u] + len(u, v)
```

heißt Kantenrelaxierung.

Sprechweisen

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.

Korrektheit

Beh: Dijkstra terminiert mit $d[v] = d(s, v)$ für alle $v \in V$

drei Schritte:

(i) alle erreichbaren Knoten werden abgearbeitet

(ii) es ist immer $d[v] \geq d(s, v)$

(iii) wenn v abgearbeitet wird, ist $d[v] = d(s, v)$

Korrektheit

Beh: Dijkstra terminiert mit $d[v] = d(s, v)$ für alle $v \in V$

drei Schritte:

(i) alle erreichbaren Knoten werden abgearbeitet

(ii) es ist immer $d[v] \geq d(s, v)$

(iii) wenn v abgearbeitet wird, ist $d[v] = d(s, v)$

Beweis (i):

- Beh: v wird nicht bearbeitet, ist aber erreichbar
- es gibt kürzesten s-t-Weg $(s = v_1, \ldots, v_k = v)$
- s wird abgearbeitet
- also gibt es v_i mit v_{i-1} abgearbeitet und v_i nicht
- also wird v_i in die Queue eingefügt
- Widerspruch zu v_i wird nicht abgearbeitet

Korrektheit

Beh: Dijkstra terminiert mit $d[v] = d(s, v)$ für alle $v \in V$

drei Schritte:

(i) alle erreichbaren Knoten werden abgearbeitet

(ii) es ist immer $d[v] \geq d(s, v)$

(iii) wenn v abgearbeitet wird, ist $d[v] = d(s, v)$

Beweis (ii):

- Für jeden Knoten $v \neq s$ gilt:
- Initial ist $d[v] = \infty$
- Alle folgenden Änderungen korrespondieren zu Längen von s-v-Wegen im Graphen
- Die Länge eines s-v-Weges ist immer mindestens so lang, wie die länge eines kürzesten s-v-Weges

Korrektheit

Beweis (iii): wenn v abgearbeitet wird, ist $d[v] = d(s, v)$

- Beh: Es gibt v, so dass v wird abgearbeitet mit $d[v] > d(s, v)$
- Sei v der erste solche Knoten (in Abarbeitungsreihenfolge)
- Sei S die Menge aller Knoten, die vor v abgearbeitet werden.
- Sei $s = v_1, \ldots, v_k = v$ ein kürzester s-v-Weg
- Falls alle $v_i \in S$, so ist $d[v] = d(s, v)$ wenn v besucht wird. Widerspruch.
- Sei also v_i der Knoten mit dem kleinsten i, so dass $v_i \notin S$.
- Wegen $v_{i-1} \in S$ ist $d[v_i] = d(s, v_i) < d[v]$ zu dem Zeitpunkt an dem v bearbeitet wird.
- Also hätte v_i vor v abgearbeitet werden müssen.
- Widerspruch zu $v_i \notin S$

Dijkstra

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.add(s, 0)                   // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
           // relaxing edges
7
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])   // m Mal
12             else Q.insert(v, d[v])                 // n Mal
```

Dijkstra

```
1  forall the nodes v ∈ V do
2      d[v] = ∞, p[v] = NULL                          // n Mal
3  d[s] = 0, Q.clear(), Q.add(s, 0)                   // 1 Mal
4  while !Q.empty() do
5      u ← Q.deleteMin()                              // n Mal
6      forall the edges e = (u, v) ∈ E do
           // relaxing edges
7
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])   // m Mal
12             else Q.insert(v, d[v])                 // n Mal
```

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

Laufzeit

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

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	$\Theta(1)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DeleteMin	$\Theta(n)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Union	$\Theta(1)$	$\Theta(k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Dijkstra				

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Laufzeit

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

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	$\Theta(1)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DeleteMin	$\Theta(n)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Union	$\Theta(1)$	$\Theta(k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Dijkstra	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}(m + n\lg n)$

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Laufzeit

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

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	$\Theta(1)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DeleteMin	$\Theta(n)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Union	$\Theta(1)$	$\Theta(k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Dijkstra	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}(m + n\lg n)$
Dij ($m \in \Theta(n^2)$)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \lg n)$	$\mathcal{O}(n^2 \lg n)$	$\mathcal{O}(n^2)$

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Laufzeit

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

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	$\Theta(1)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DeleteMin	$\Theta(n)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Union	$\Theta(1)$	$\Theta(k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DecreaseKey	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\Theta(1)$
Delete	$\Theta(1)$	$\Theta(\lg k)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$
Dijkstra	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}((n+m)\lg n)$	$\mathcal{O}(m + n\lg n)$
Dij ($m \in \Theta(n^2)$)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \lg n)$	$\mathcal{O}(n^2 \lg n)$	$\mathcal{O}(n^2)$
Dij ($m \in \mathcal{O}(n)$)	$\mathcal{O}(n^2)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(n \lg n)$	$\mathcal{O}(n \lg n)$

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

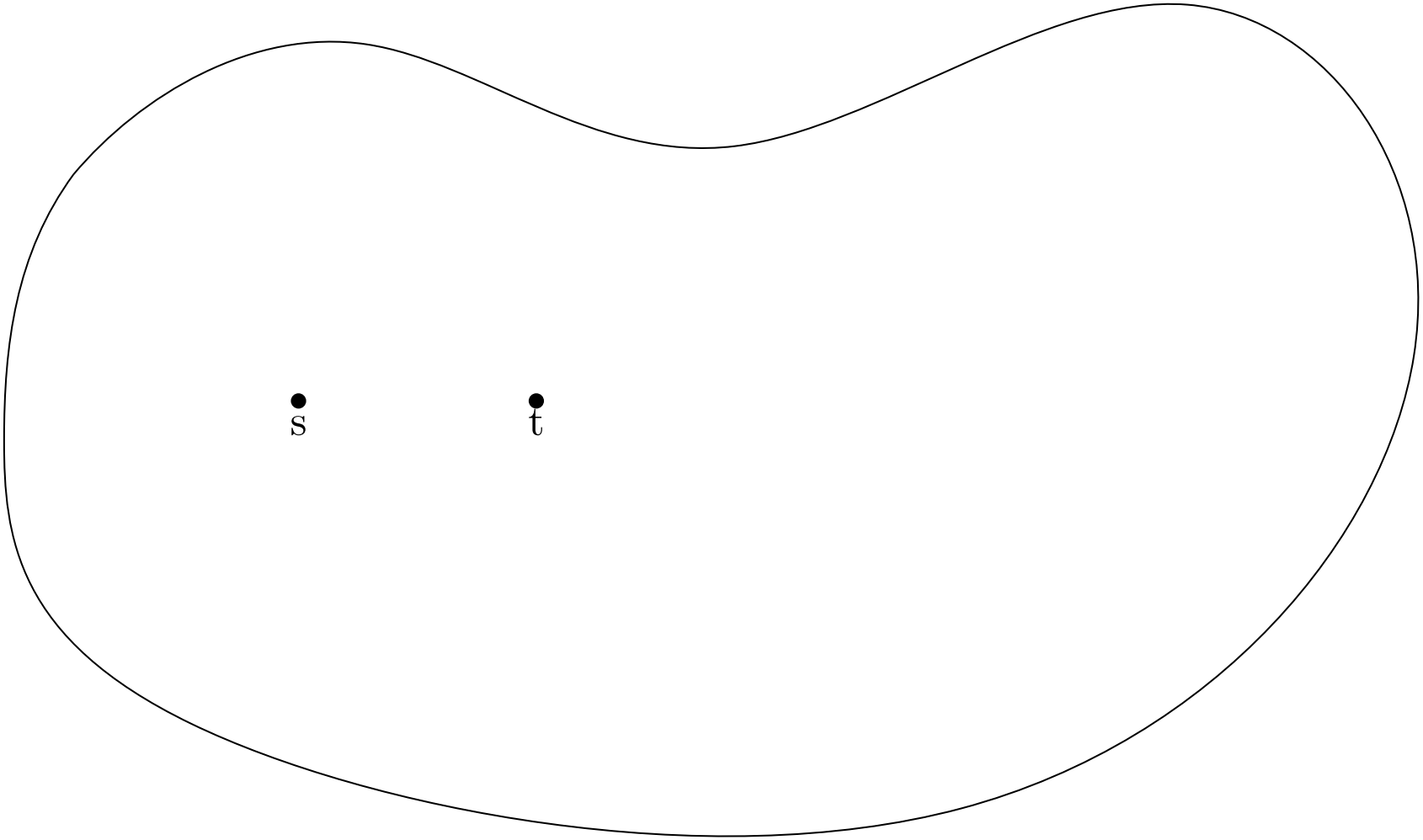
Laufzeit

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

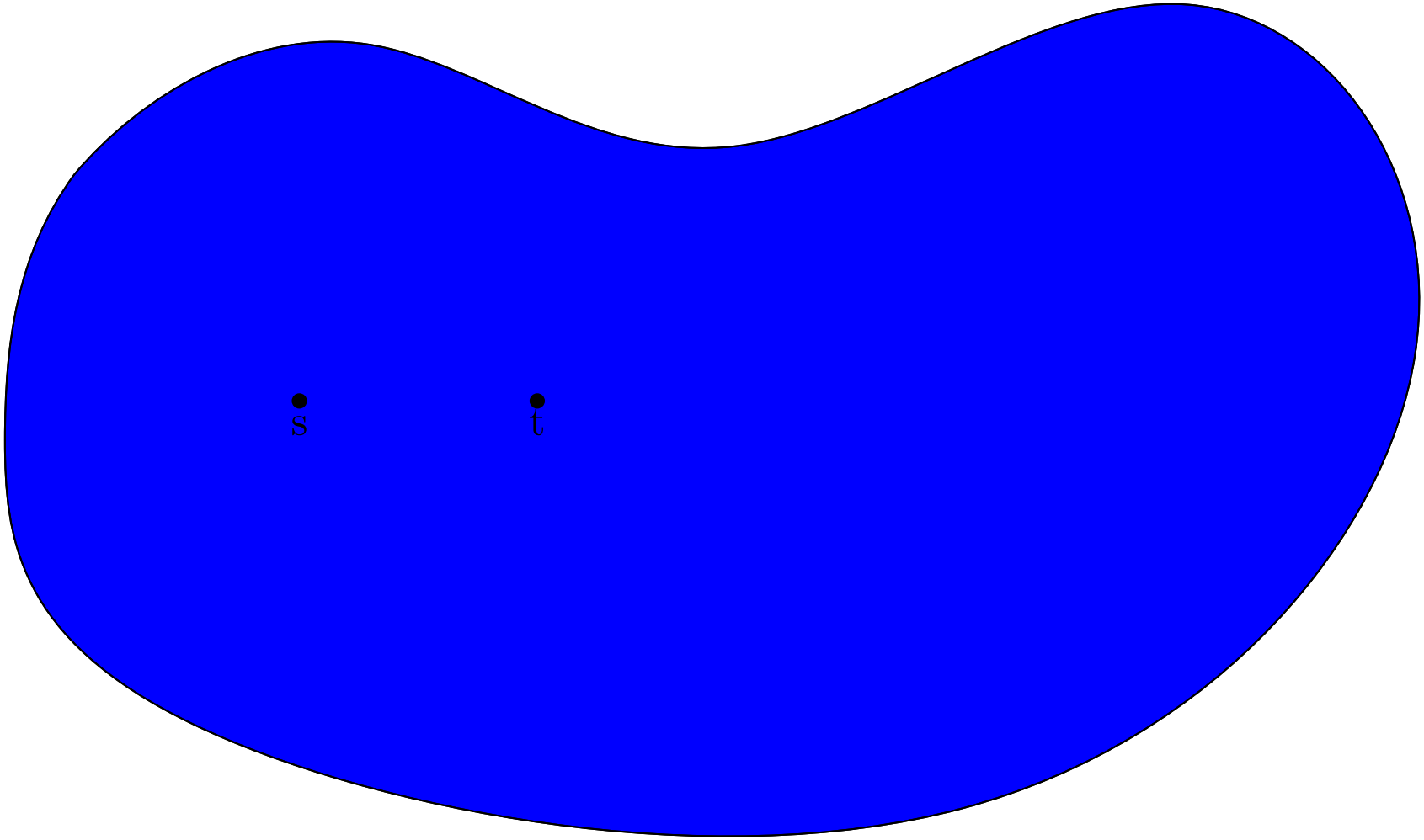
Operation	Liste (worst-case)	Binary Heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Init	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\lg k)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\lg k)$	$\Theta(1)$
DeleteMin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

Transportnetzwerke sind dünn $\Rightarrow$ Binary Heaps

Schematischer Suchraum, Dijkstra



Schematischer Suchraum, Dijkstra



Abbruch

Beobachtung

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

Idee

- stoppe die Anfrage, sobald t aus der Queue entfernt wurde
- Ein Knoten heißt **bearbeitet**, wenn er aus der Queue extrahiert wurde
- **Suchraum**: Menge der bearbeiteten Knoten
- Distanzlabel $d[v]$ ändert sich nicht mehr, sobald v besucht wurde
- Korrektheit des Vorgehens bleibt also erhalten
- Reduziert durchschnittlichen Suchraum von n auf $(n+1)/2$

Abbruch

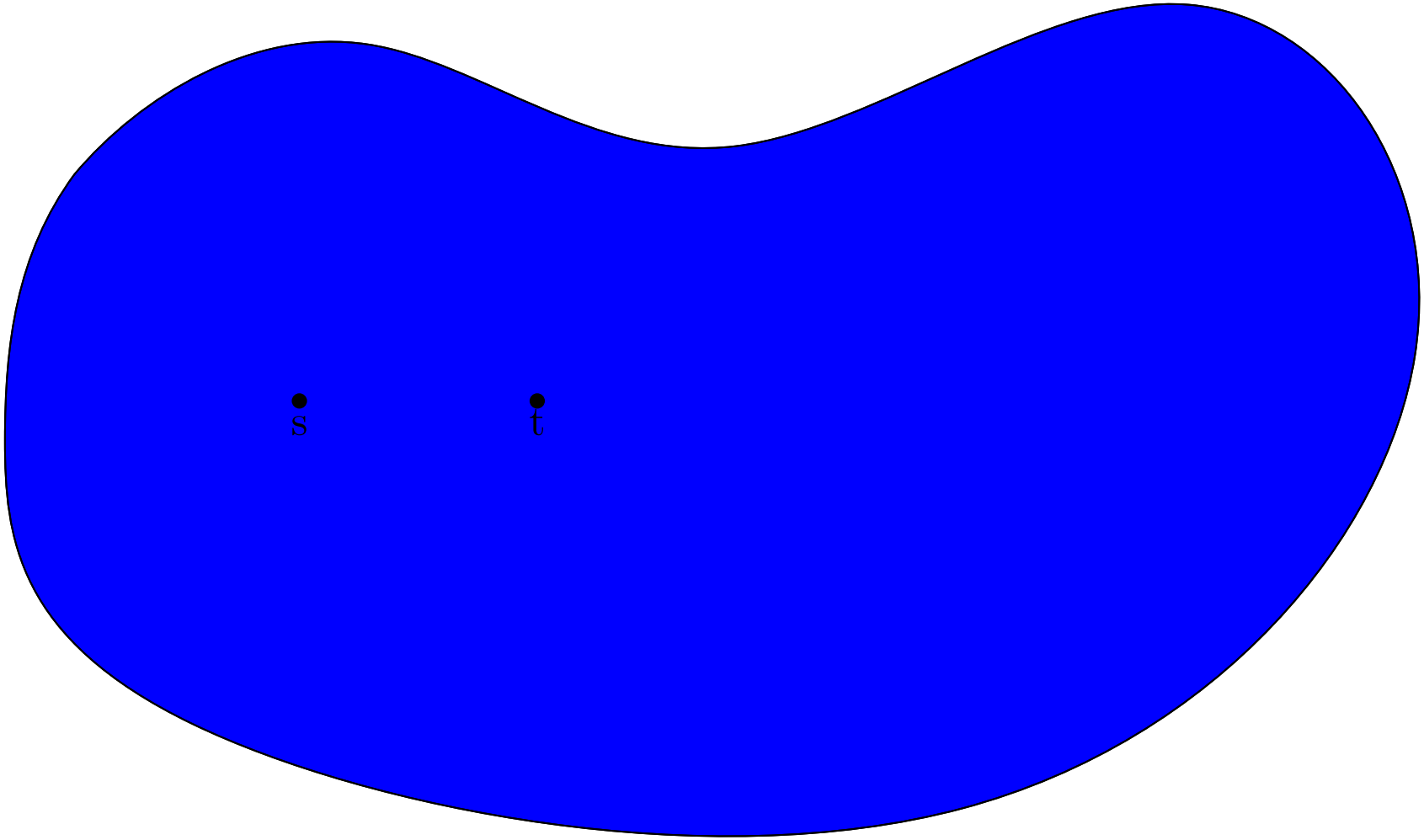
Beobachtung

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

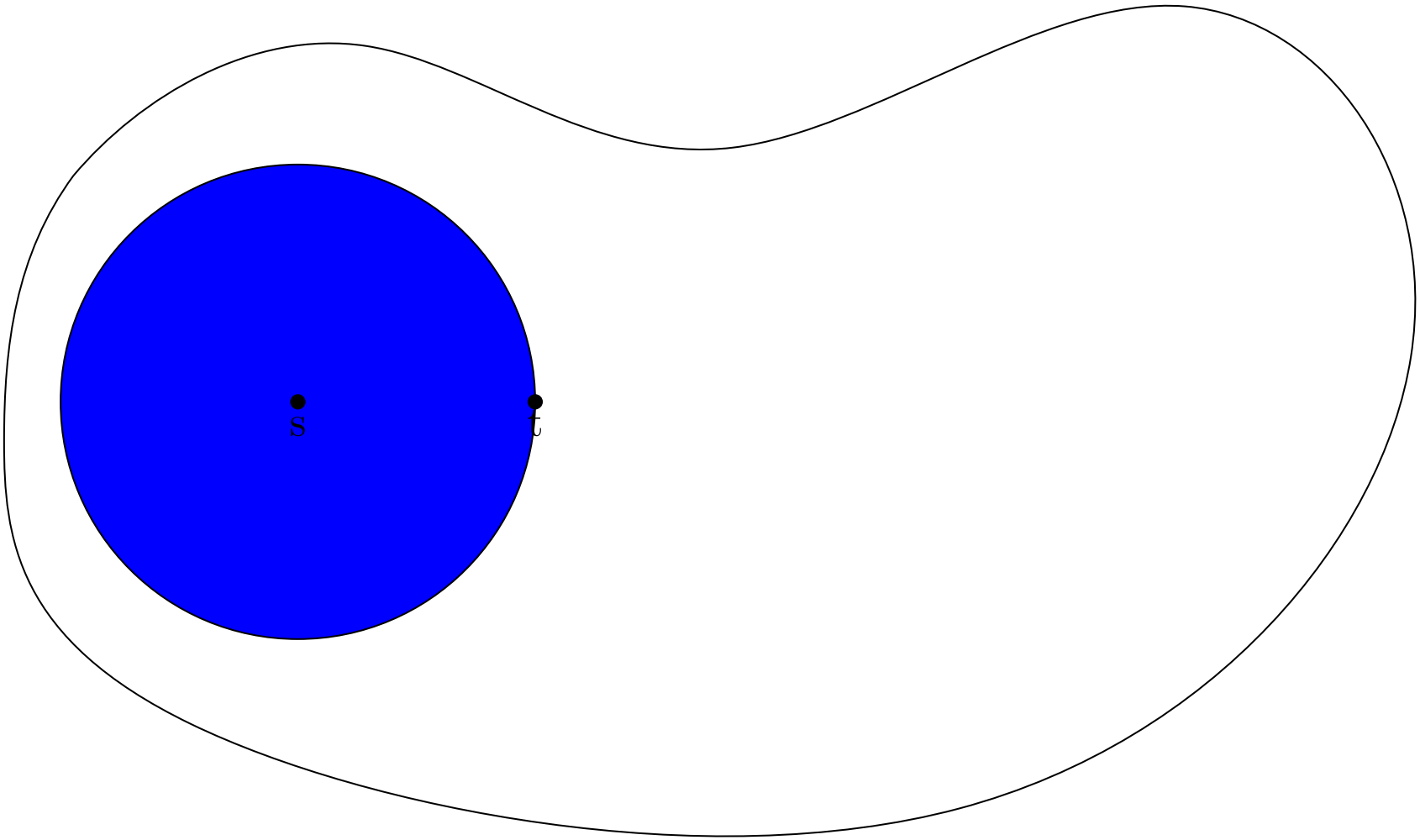
Idee

- stoppe die Anfrage, sobald t aus der Queue entfernt wurde
- Ein Knoten heißt **bearbeitet**, wenn er aus der Queue extrahiert wurde
- **Suchraum**: Menge der bearbeiteten Knoten
- Distanzlabel $d[v]$ ändert sich nicht mehr, sobald v besucht wurde
- Korrektheit des Vorgehens bleibt also erhalten
- Reduziert durchschnittlichen Suchraum von n auf $(n+1)/2$

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      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             else Q.insert(v, d[v])
```

Frage

- Häufig werden viele Anfragen auf gleichem Netzwerk gestellt.
- Wo könnte ein Problem bzgl 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             else Q.insert(v, d[v])
```

Frage

- Häufig viele Anfragen
- Problem: Die Initalisierung muss immer für alle Knoten neu ausgeführt werden.
- Wie könnte das Problem gelöst werden?

Lösung

- 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 Abbruchkriterium und Timestamp

```
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])
```

Kleiner Exkurs

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

- In Routenplanungsvorlesung Kantengewichte immer **echt** positiv.
- 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:
  NP-schwer, Reduktion von „Problem Longest Path“, siehe [Garey&Johnson 79]

Der Bellman-Ford Algorithmus

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

Der Bellman-Ford Algorithmus

- Der BF-Algorithmus erkennt, falls ein Graph einen negativen Zyklus enthält
- Enthält ein Graph keinen negativen Zyklus, so enthält nach Terminierung $d[v]$ die Distanz $d(s, v)$.
- Die Laufzeit des BF-Algorithmus' liegt in $O(|V| \cdot |E|)$.

Beweise siehe „Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms."

Problem Longest Path

Problem Longest Path

Gegeben:

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

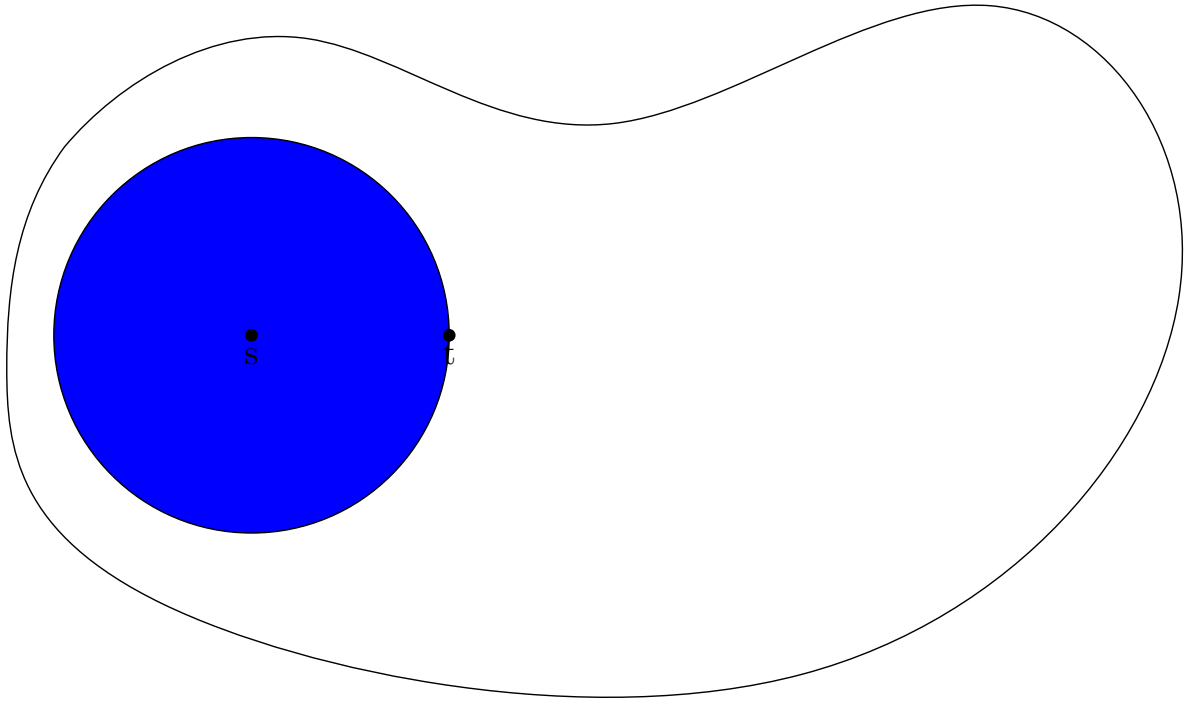
Frage

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

Problem Longest Path ist NP-schwer (siehe [Garey& Johnson 79])

Bidirektionale Suche

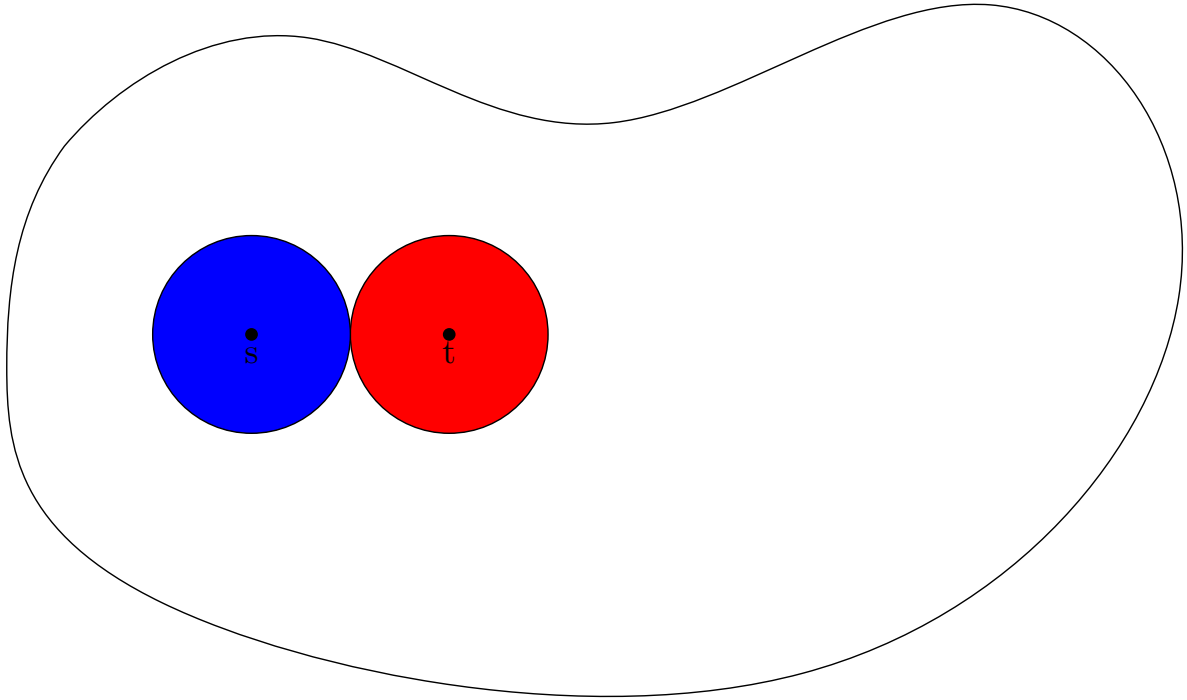
Bidirektionale Suche



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

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

Bidirektionale Suche



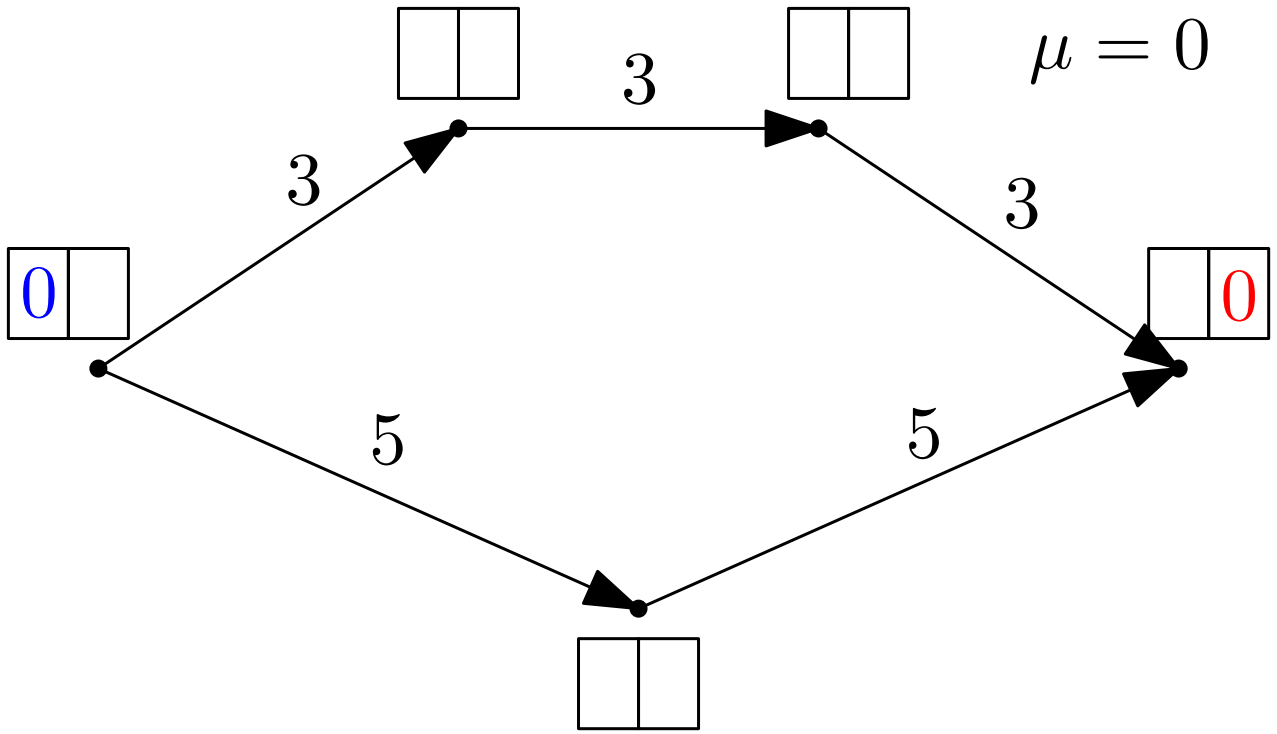
Idee: Kombiniere beide Suchen

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

Bidirektionale Suche

Anfrage:

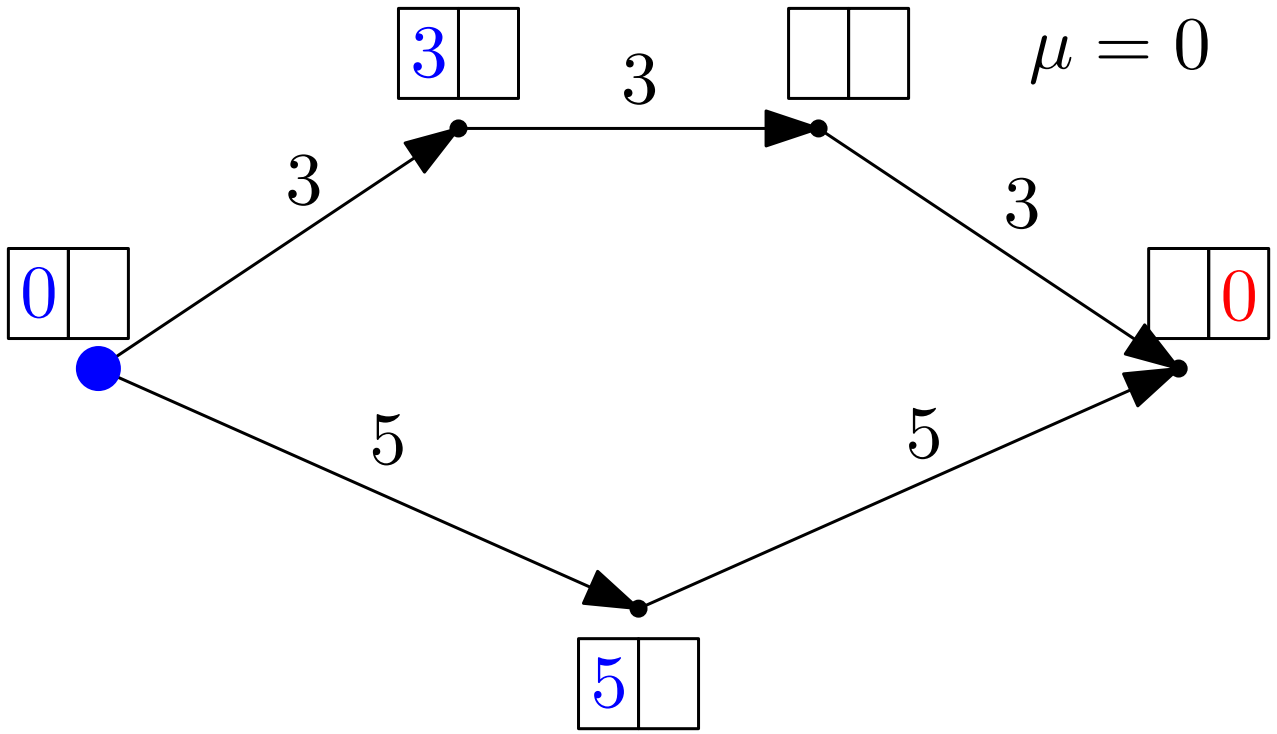
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

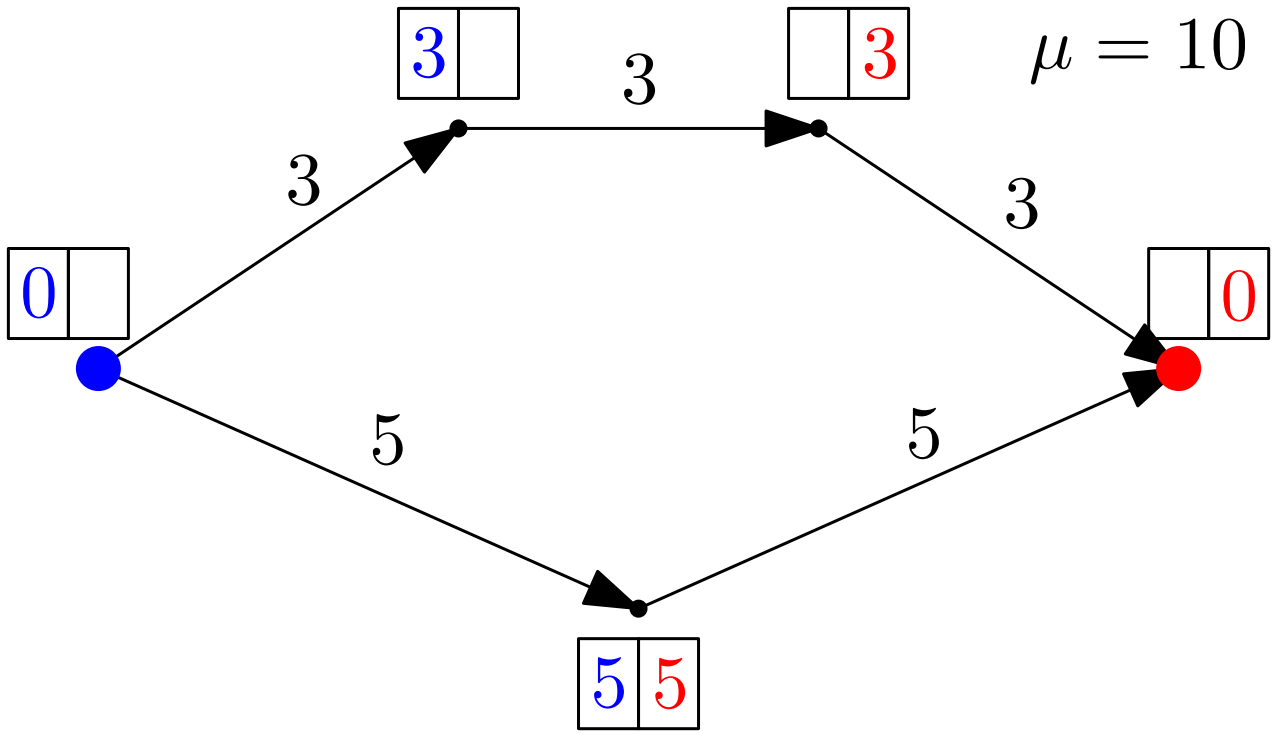
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

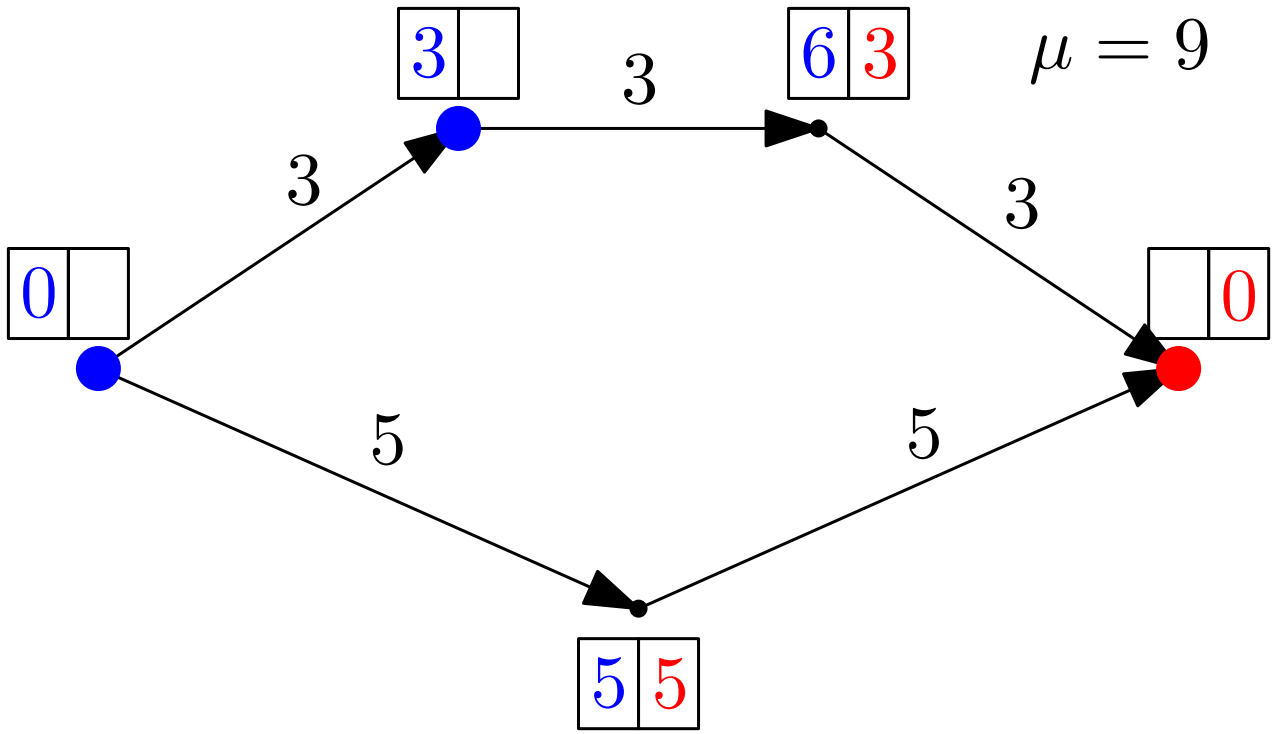
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

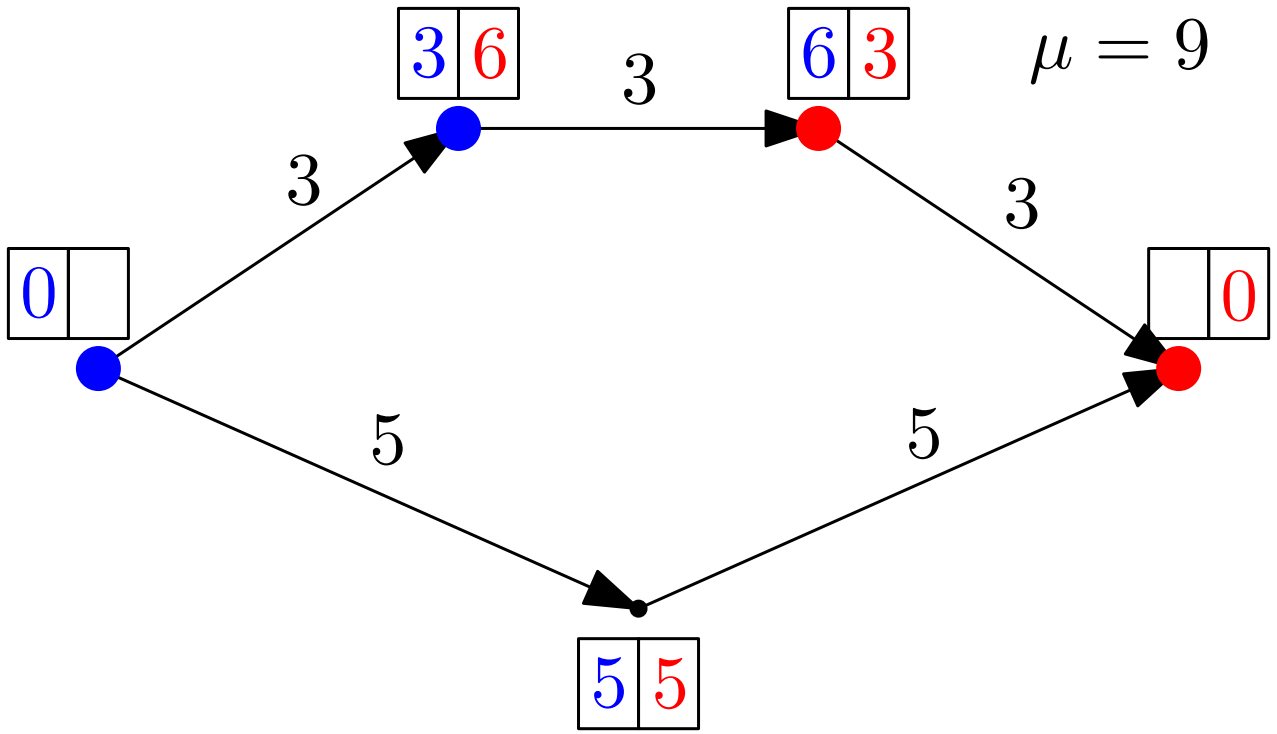
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

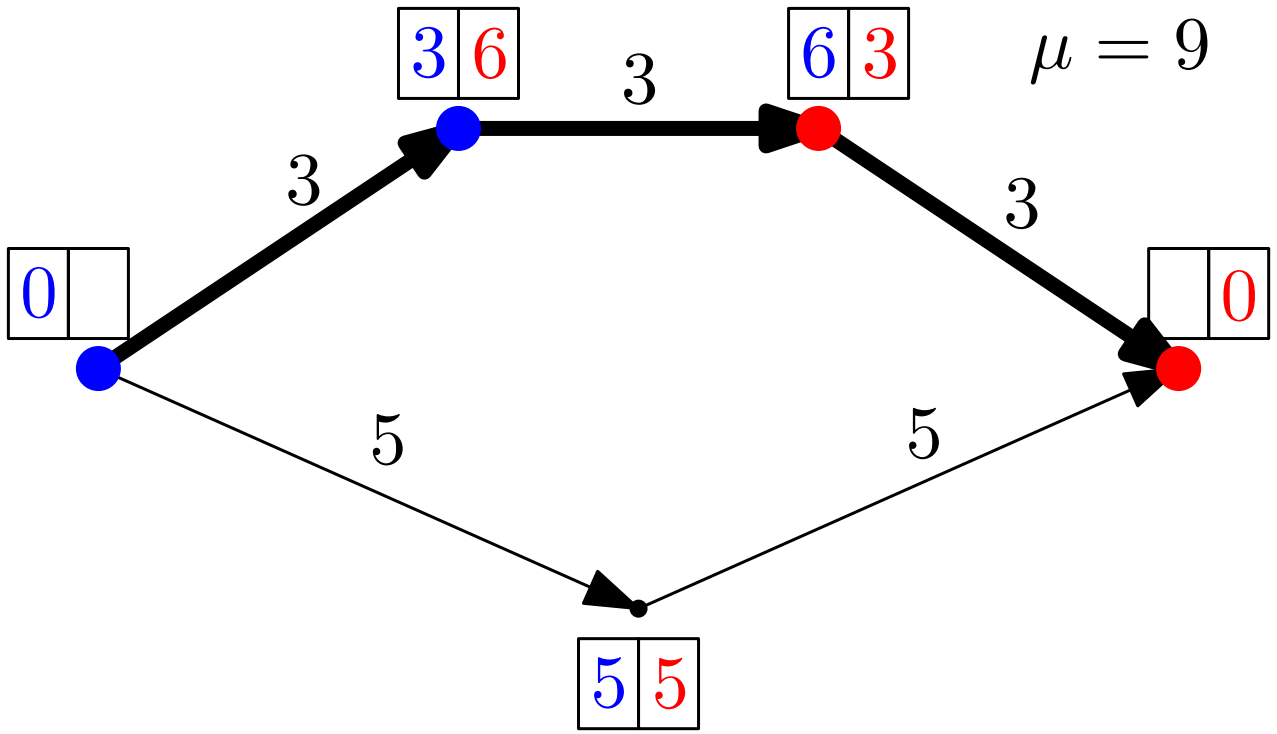
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten



Bidirektionale Suche

Anfrage:

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



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

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

  ausgeführt. (Initial ist $\mu = \infty$).
- Nach Terminierung beinhaltet μ die Distanz $d(s, t)$.

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

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

  ausgeführt. (Initial ist $\mu = \infty$).
- Nach Terminierung beinhaltet μ die Distanz $d(s, t)$.

Was sind gute Abbruchstrategien?

Abbruchstrategie (1)

Abbruchstrategie (1)

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

Abbruchstrategie (1)

Abbruchstrategie (1)

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

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

Nein, Gegenbeispiel: $G = (\{s, t, v\}, \{\{s, t\}, \{s, v\}, \{t, v\}\}$ mit Längen $\text{len}(\{s, t\}) = 10$, $\text{len}(\{s, t\}) = 6$, $\text{len}(\{s, t\}) = 6$.

Abbruchstrategie (1)

Abbruchstrategie (1)

Abbruch, sobald ein Knoten m existiert, der von beiden Suchen bearbeitet 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 bearbeitet 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 $d(v,t) \leq \mathit{minKey}(\overleftarrow{Q})$.
  - O.B.d.A wird t nicht von der Vorwärtssuche abgearbeitet.
  - Sei $(u,v) \in P$, so dass

    $$\text{dist}(s,u) \leq \mathit{minKey}(\overrightarrow{Q}) \text{ und } \text{dist}(s,u) + \text{len}(u,v) \geq \mathit{minKey}(\overrightarrow{Q}) \; .$$

  - Annahme: $\text{dist}(v,t) > \mathit{minKey}(\overleftarrow{Q})$.
  - Dann $\mu > \text{len}(P) = \text{dist}(s,u) + \text{len}(u,v) + \text{dist}(v,t) > \mu$.
  - Widerspruch.

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!

Bidirektionale Suche - Wechselstrategien

Frage: Was sind mögliche Wechselstrategien?

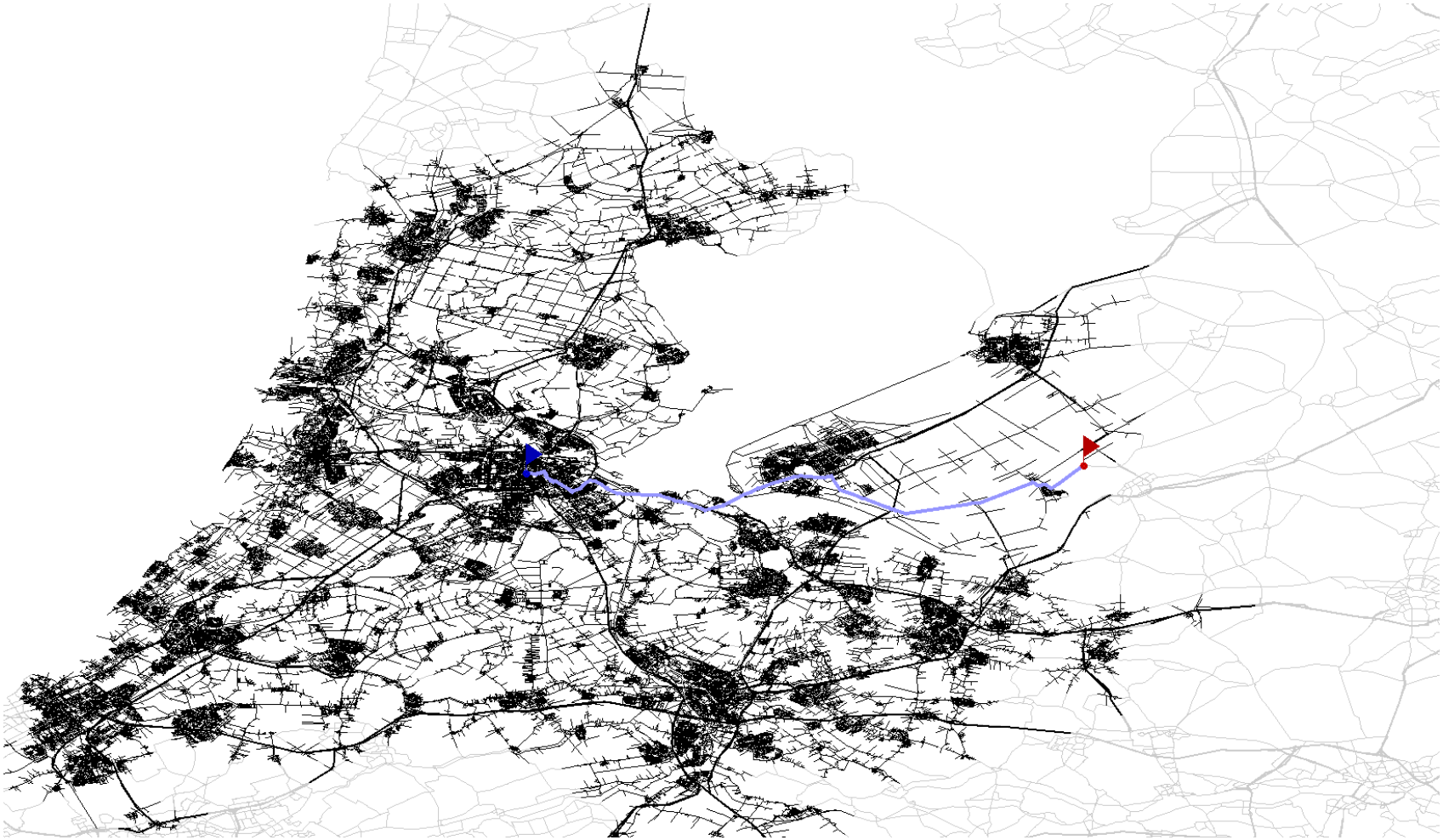
Bidirektionale Suche - Wechselstrategien

Karlsruher Institut für Technologie

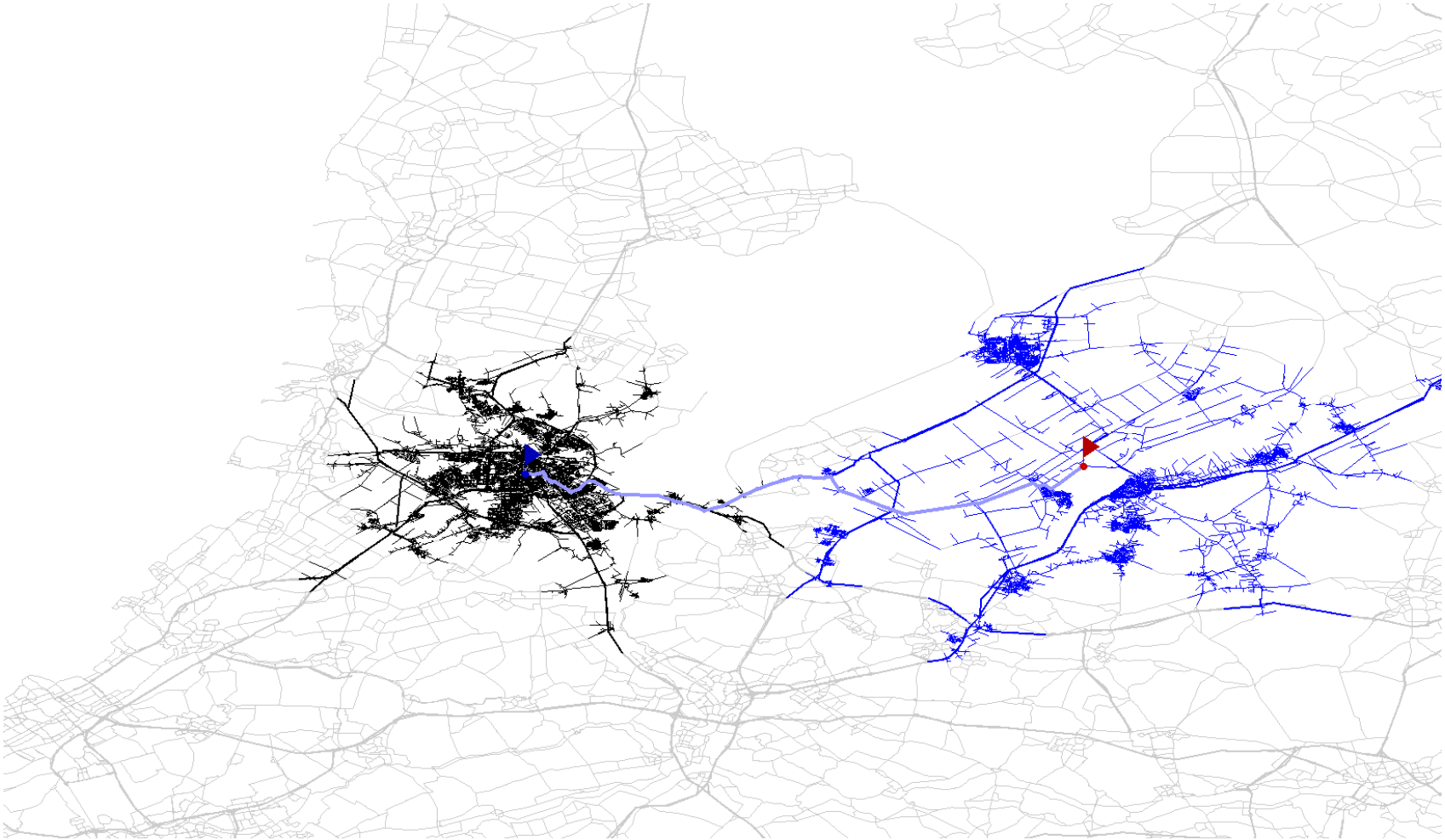
Mögliche Wechselstrategien

- Prinzipiell jede Wechselstrategie möglich
- Parallele Ausführung auf zwei Kernen
- Wechsle nach jedem Schritt zur entgegengesetzten Suche
- Führe immer die Suche mit dem kleineren minimalen Queueelement aus

Beispiel



Beispiel



Bemerkung

- Die Beschleunigung von bidirektionaler Suche ist gering.
- Bidirektionale Suche ist allerdings ein wichtiger Bestandteil von vielen sehr effizienten Beschleunigungstechniken.