

# Algorithmen für Routenplanung

2. Sitzung, Sommersemester 2012

Thomas Pajor | 23. April 2012

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

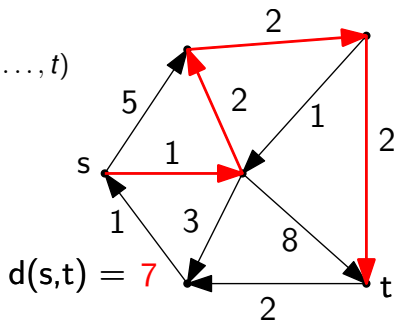


## Gegeben:

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  $P := (s, \dots, t)$



## Azyklität:

- Kürzeste Wege sind zyklenfrei

## Aufspannungseigenschaft:

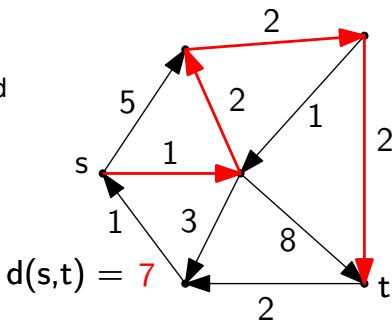
- Alle kürzesten Wege von  $s$  aus bilden DAG bzw. Baum

## Vererbungseigenschaft:

- Subwege von kürzesten Wegen sind kürzeste Wege

## Abstand:

- Steigender Abstand von Wurzel zu Blättern



**Priority Queue:** Datenstruktur die folgende Operationen erfüllt.

Operation	Beschreibung
Empty()	Prüft ob die Queue leer ist
Clear()	Löscht alle Elemente aus der Queue
Insert( $e, k$ )	Fügt Element $e$ mit Key $k$ ein
DecreaseKey( $e, k$ )	Senkt den Key von Element $e$ auf den Wert $k$
DeleteMin()	Löscht das Element mit kleinstem Key und gibt es zurück
MinKey()	Gibt den Key des kleinsten Elements zurück

Näheres in jedem Algorithmik-Standardlehrbuch.

---

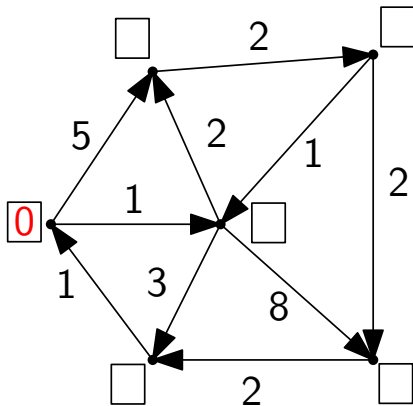
DIJKSTRA( $G = (V, E), s$ )

---

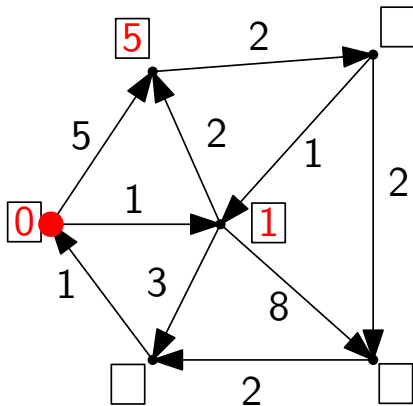
```
1 forall the nodes  $v \in V$  do
2    $d[v] = \infty, p[v] = \text{NULL}$            // distances, parents
3  $d[s] = 0$ 
4  $Q.\text{clear}(), Q.\text{insert}(s, 0)$            // container
5 while  $!Q.\text{empty}()$  do
6    $u \leftarrow Q.\text{deleteMin}()$            // settling node u
7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

---

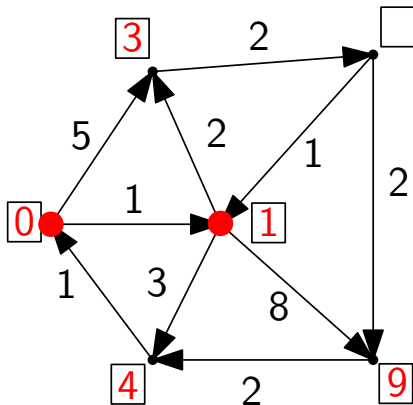
# Beispiel



# Beispiel

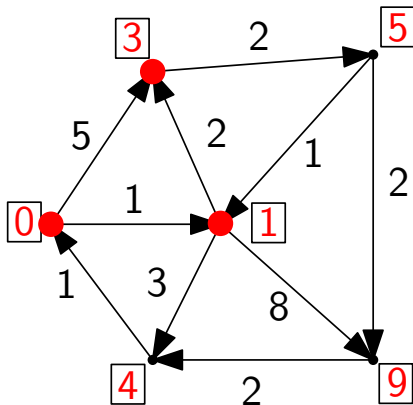


# Beispiel

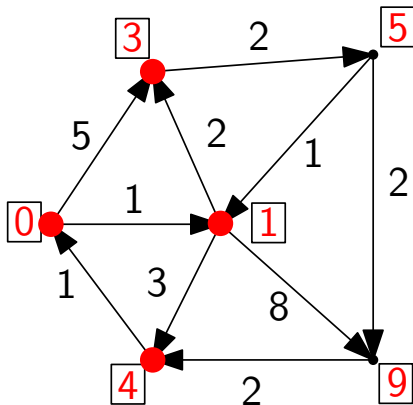




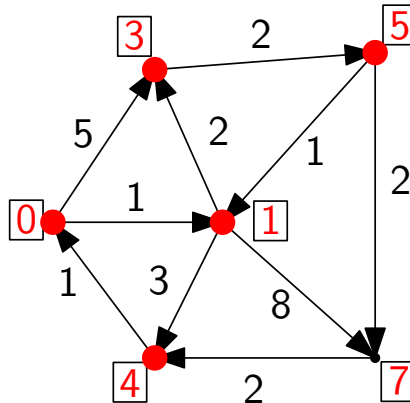
# Beispiel



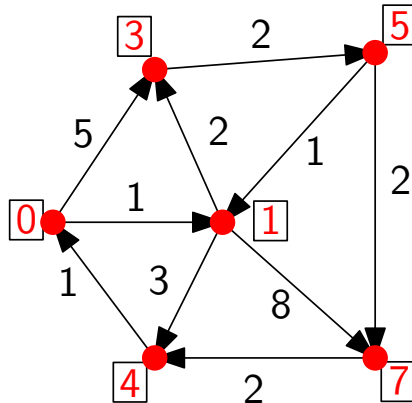
# Beispiel



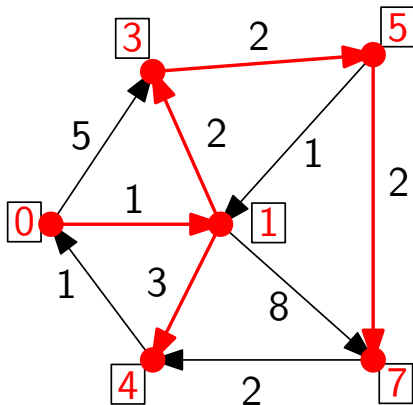
# Beispiel



# Beispiel



# Beispiel



---

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.

**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  $d[v] = d(s, v)$

**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 abgearbeitet, ist aber erreichbar
- es gibt kürzesten  $s$ - $t$ -Weg ( $s = v_1, \dots, v_k = v$ )
- $s$  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

**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

**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  $s = v_1, \dots, v_k = v$  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( $G = (V, E), s$ )

---

```
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    $d[v] = \infty, p[v] = \mathbf{NULL}$  // n Mal
3  $d[s] = 0, Q.clear(), Q.insert(s, 0)$ 
4 while ! $Q.empty()$  do
5    $u \leftarrow Q.deleteMin()$ 
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  $Q.decreaseKey(v, d[v])$ 
11      else  $Q.insert(v, d[v])$ 
```

---

---

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    $d[v] = \infty, p[v] = \mathbf{NULL}$  // n Mal
3  $d[s] = 0, Q.clear(), Q.insert(s, 0)$  // 1 Mal
4 while ! $Q.empty()$  do
5    $u \leftarrow Q.deleteMin()$  // 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  $Q.decreaseKey(v, d[v])$  // m Mal
11      else  $Q.insert(v, d[v])$  // 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>				

# Laufzeit

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

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>	$\mathcal{O}(n^2 + m)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}(m + n \log n)$

# Laufzeit

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

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

# Laufzeit

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

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

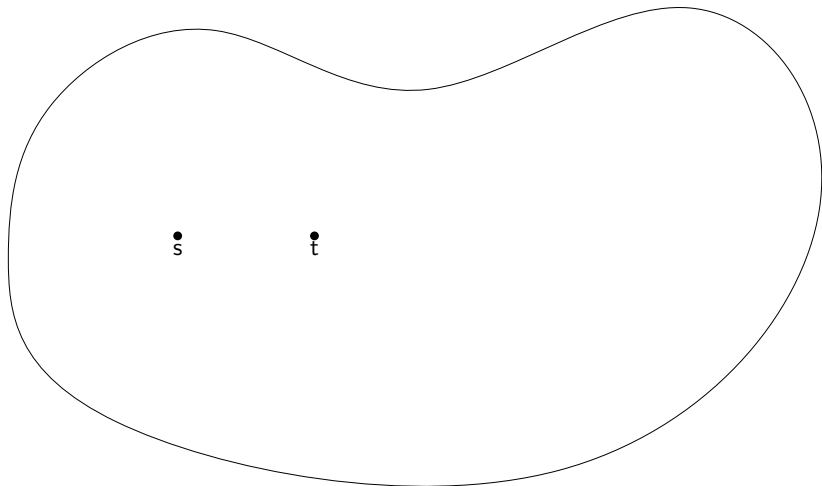


$$T_{\text{DIJKSTRA}} = T_{\text{init}} + n \cdot T_{\text{deleteMin}} + m \cdot T_{\text{decreaseKey}} + n \cdot T_{\text{insert}}$$

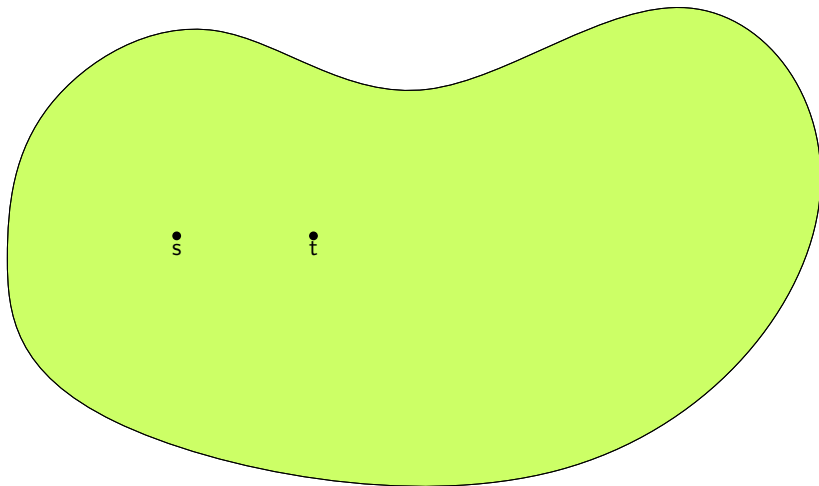
Operation	Liste (worst-case)	Binary Heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Init	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(\log k)$	$\mathcal{O}(\log k)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(1)$	$\mathcal{O}(\log k)$	$\Theta(1)$
DeleteMin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

Transportnetzwerke sind dünn  $\Rightarrow$  Binary Heaps

# 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

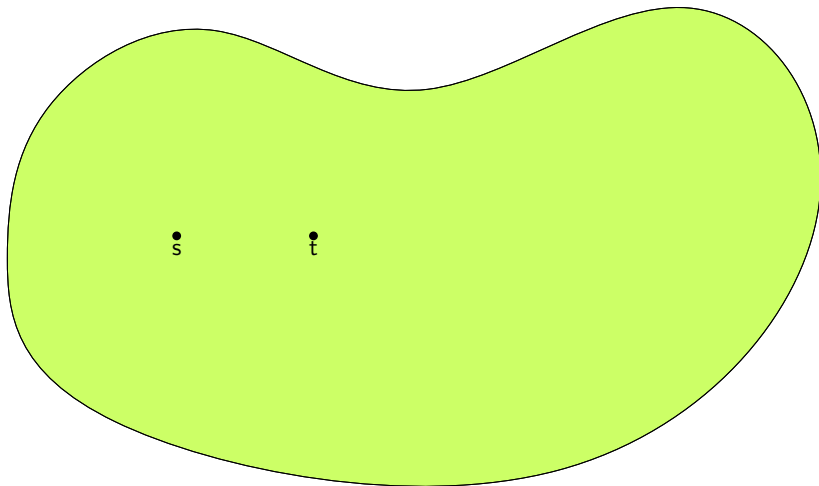
## Idee

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

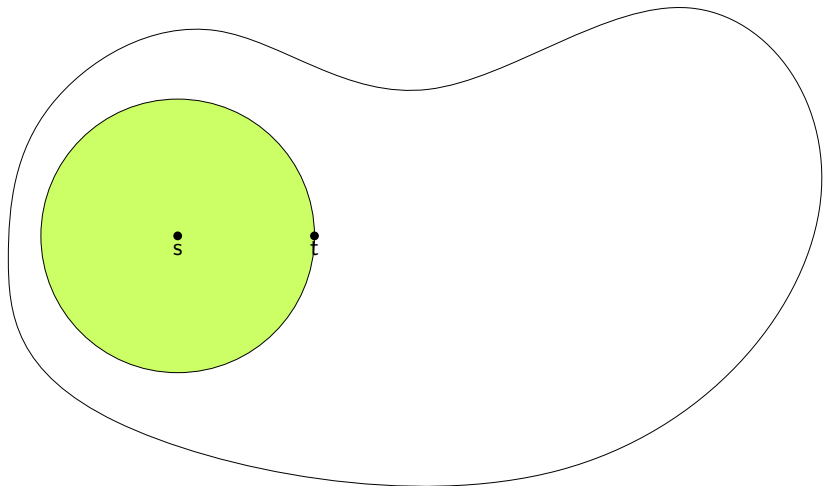
## Korrektheit

- 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 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   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      else  $Q.\text{insert}(v, d[v])$ 
```

---

- 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  $v \in V$  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

```
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])
```

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

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

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

- In dieser Vorlesung Kantengewichte: 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        $d[v] \leftarrow d[u] + \text{len}(u, v)$ 
7 forall the edges  $(u, v) \in E$  do
8   if  $d[v] > d[u] + \text{len}(u, v)$  then
9     negative cycle found
```

---

## Bemerkungen

- 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

### Gegeben:

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

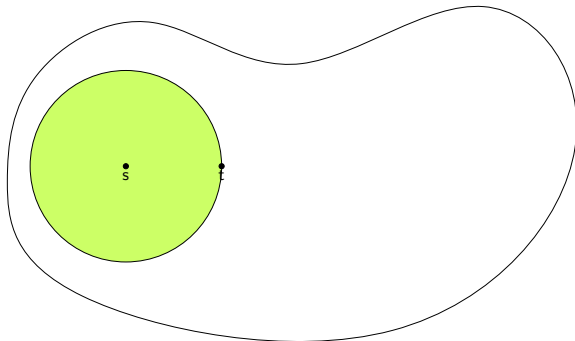
### Frage:

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

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

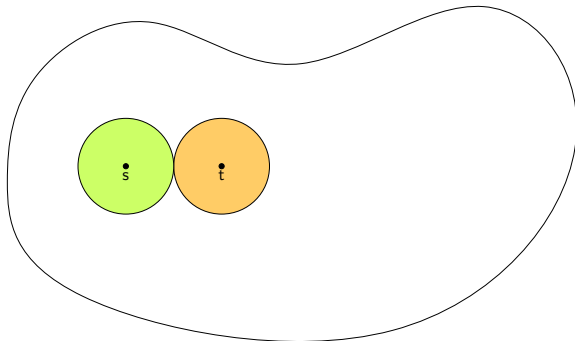
# Bidirektionale Suche





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

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

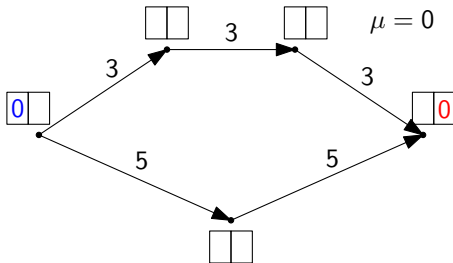


## Idee: Kombiniere beide Suchen

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

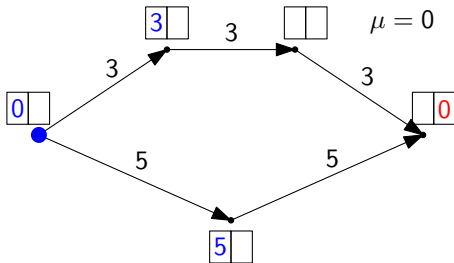
## Anfrage:

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



## Anfrage:

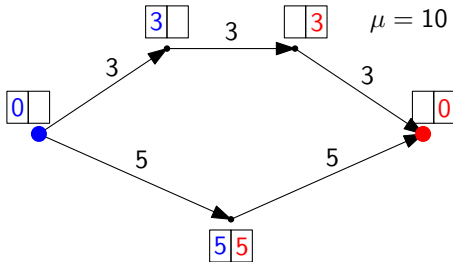
- alterniere Vorwärts- und Rückwärtsuche
  - vorwärts: relaxiere ausgehende Kanten
  - rückwärts: relaxiere eingehende Kanten





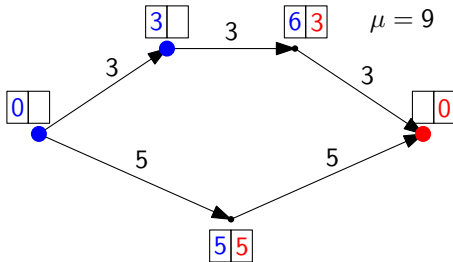
## Anfrage:

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



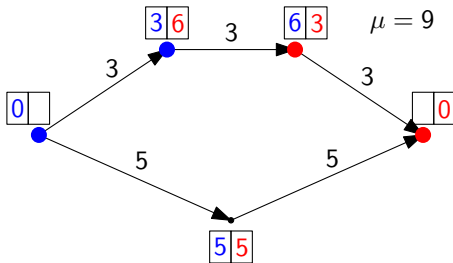
## Anfrage:

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



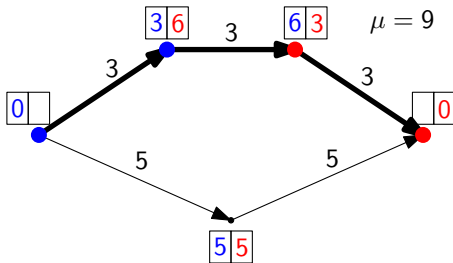
## Anfrage:

- 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



- 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.

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

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

ausgeführt. (Initial ist  $\mu = \infty$ ).

- Nach Terminierung beinhaltet  $\mu$  die Distanz  $d(s, t)$ .

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

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

berechnet.

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

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

ausgeführt. (Initial ist  $\mu = \infty$ ).

- Nach Terminierung beinhaltet  $\mu$  die Distanz  $d(s, t)$ .

## Was sind gute Abbruchstrategien?

## Abbruchstrategie (1)

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



## Abbruchstrategie (1)

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

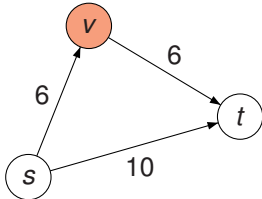
**Frage:** Ist  $m$  dann auf einem kürzesten  $s$ - $t$ -Weg enthalten?

## Abbruchstrategie (1)

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)

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:  $\vec{d}[v] + \overleftarrow{d}[v] \geq \text{dist}(s, t)$ .
- 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  $\text{len}(P) = \text{dist}(s, m) + \text{dist}(m, t)$ .

## Abbruchstrategie (1)

Abbruch, sobald ein Knoten  $m$  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  $\text{len}(P) = \text{dist}(s, m) + \text{dist}(m, t)$ .
- Angenommen es gibt  $v_i \notin \vec{S} \cup \overleftarrow{S}$ .
- Dann gilt

$$\text{dist}(s, v_i) \geq \text{dist}(s, m)$$

$$\text{dist}(v_i, t) \geq \text{dist}(m, t)$$

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)

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!

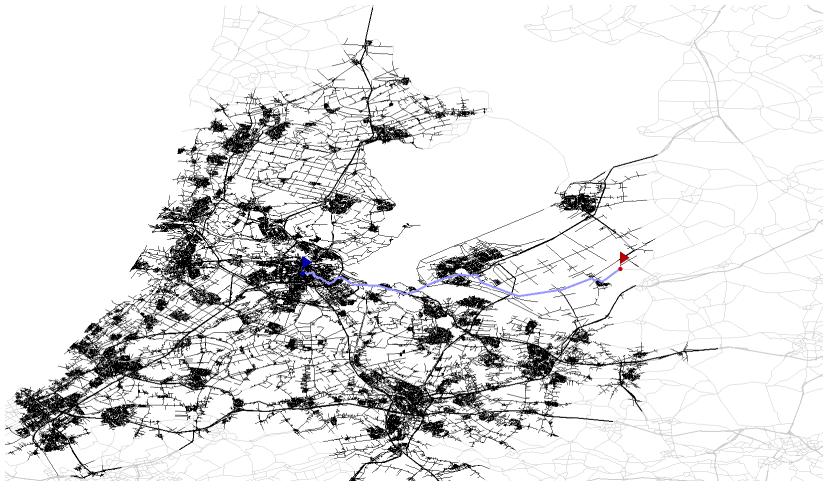
Frage: Was sind mögliche Wechselstrategien?



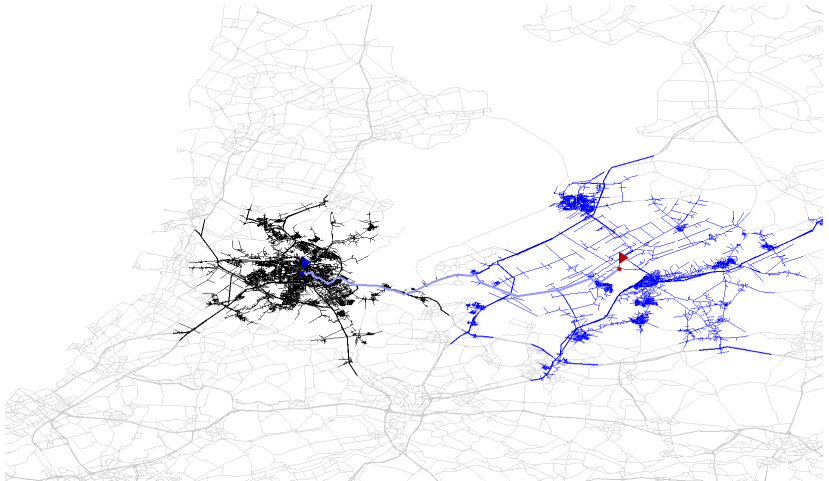
## Mögliche Wechselstrategien

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

# 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!