

Übung 11 – Algorithmen II

Yaroslav Akhremtsev, Demian Hespe – yaroslav.akhremtsev@kit.edu, hespe@kit.edu

Mit Folien von Michael Axtmann

http://algo2.iti.kit.edu/AlgorithmenII_WS17.php

Institut für Theoretische Informatik - Algorithmik II

```
    result = current_weight;
    return true;
}

for( EdgeID eid = graph.edgeBegin( current ); eid != graph.edgeEnd( current ); ++eid ){
    const Edge & edge = graph.getEdge( eid );
    COUNTING( statistic_data.inc( DijkstraStatisticData::TOUCHED_EDGES ); )
    if( edge.forward ){
        COUNTING( statistic_data.inc( DijkstraStatisticData::RELAXED_EDGES ); )
        weight new_weight = edge.weight + current_weight;
        GUARANTEE( new_weight >= current_weight, std::runtime_error, "Weight overflow detected." );
        if( !priority_queue.isReached( edge.target ) ){
            COUNTING( statistic_data.inc( DijkstraStatisticData::SUCCESSFULLY_RELAXED_EDGES ); )
            COUNTING( statistic_data.inc( DijkstraStatisticData::REACHED_NODES ); )
            priority_queue.push( edge.target, new_weight );
        } else {
            if( priority_queue.getCurrentKey( edge.target ) > new_weight ){
                COUNTING( statistic_data.inc( DijkstraStatisticData::SUCCESSFULLY_RELAXED_NODES ); )
                priority_queue.decreaseKey( edge.target, new_weight );
            }
        }
    }
}
```

- Suche in Graphen
 - Dijkstras Algorithmus
 - Bidirektionale Suche
 - A*-Suche

Eigenschaften

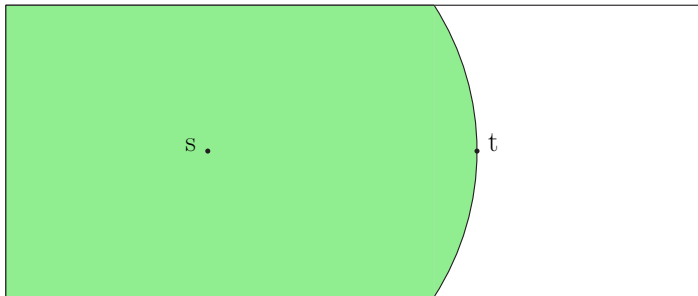
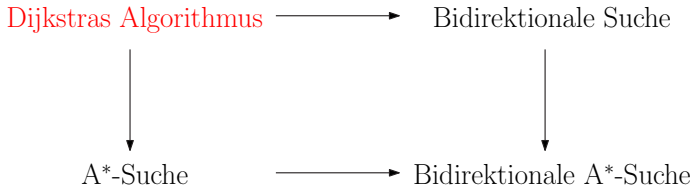
- gegeben
 - Graph $G = (V, E)$
 - Kantengewichte $c(u, v) : E \rightarrow \mathbb{R}_0^+$
- betrachte Suche von Start s nach Ziel t (One-to-One Query)
→ kürzeste Distanz $\mu(s, t)$ bestimmen

Übersicht über verschiedene Varianten

- Dijkstras Algorithmus (auch One-to-All Query)
- Bellmann Ford Algorithmus (auch negative Kantengewichte)
- Bidirektionale Suche (Beschleunigungstechnik)
- A*-Suche (Heuristik, wichtig in KI)
- ...

Suche in Graphen

Übersicht über verschiedene Varianten



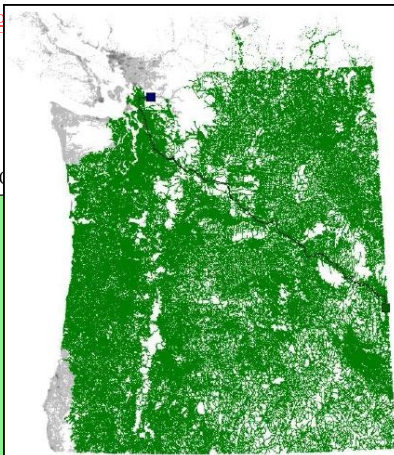
Suche in Graphen

Übersicht über verschiedene Varianten

Dijkstras Alg



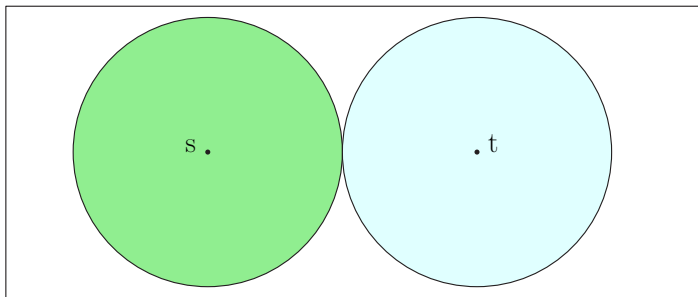
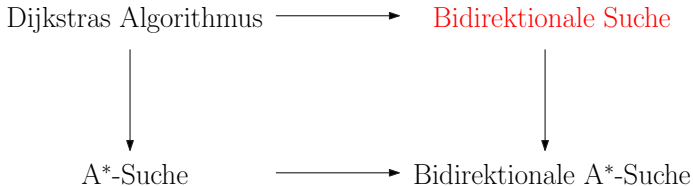
A*-Suche



nahe Suche

le A*-Suche

Übersicht über verschiedene Varianten



Suche in Graphen

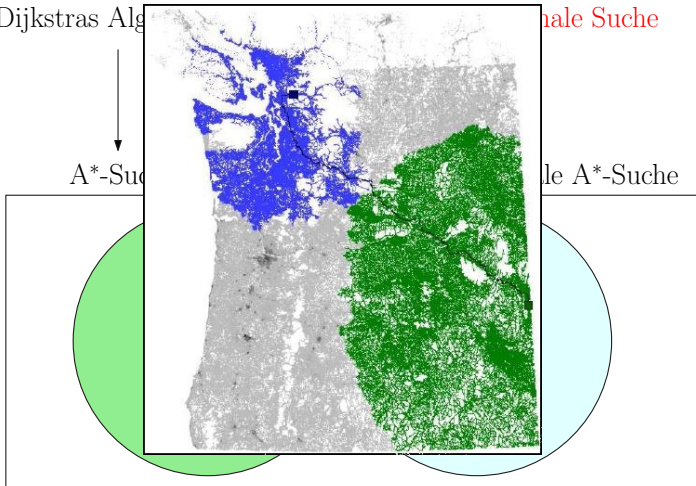
Übersicht über verschiedene Varianten

Dijkstras Alg

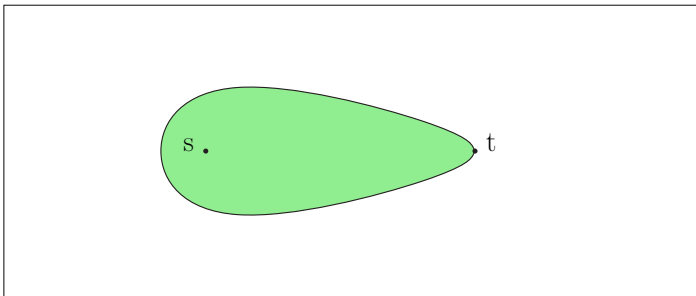
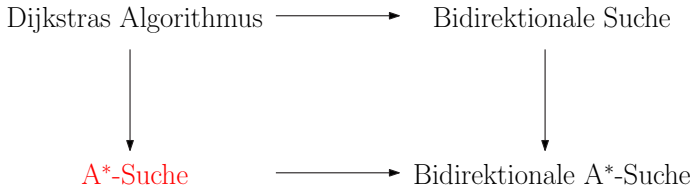
A*-Suche

Heuristische Suche

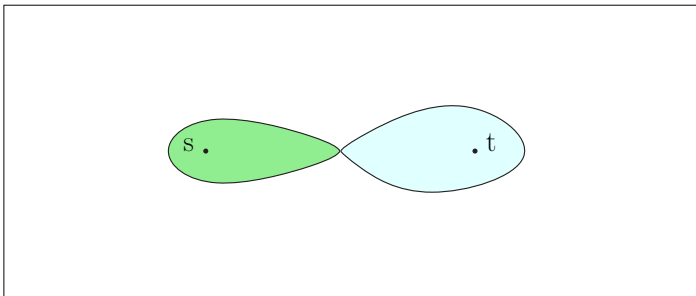
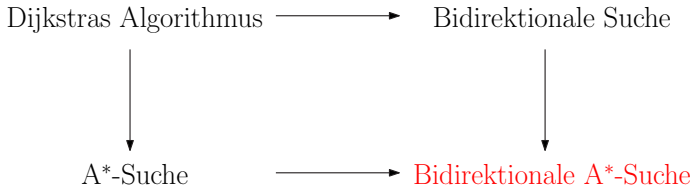
Heuristische A*-Suche



Übersicht über verschiedene Varianten



Übersicht über verschiedene Varianten



Suche in Graphen

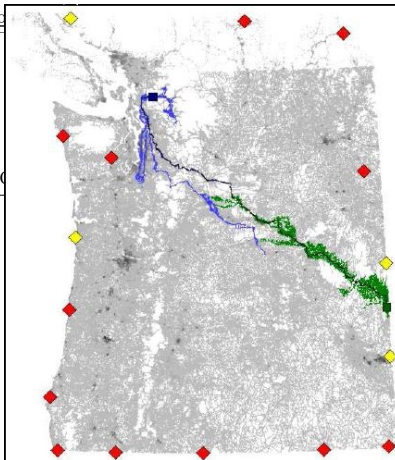
Übersicht über verschiedene Varianten

Dijkstras Alg

A*-Suche

nahe Suche

le A*-Suche



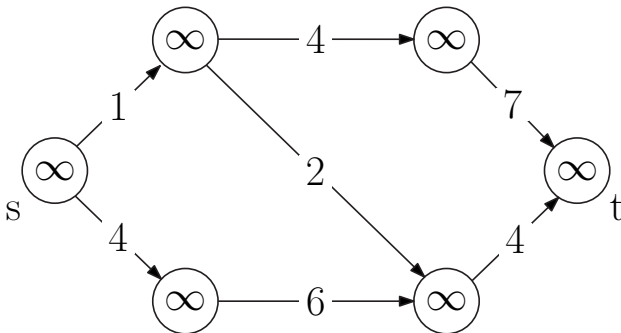
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



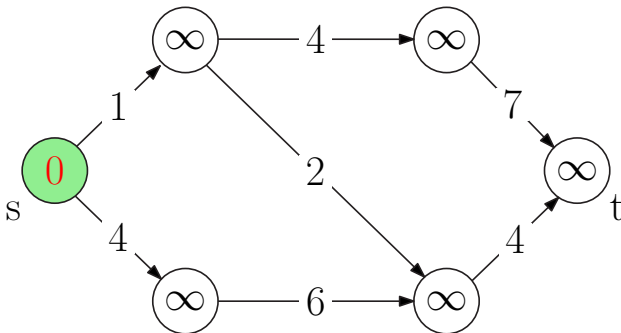
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



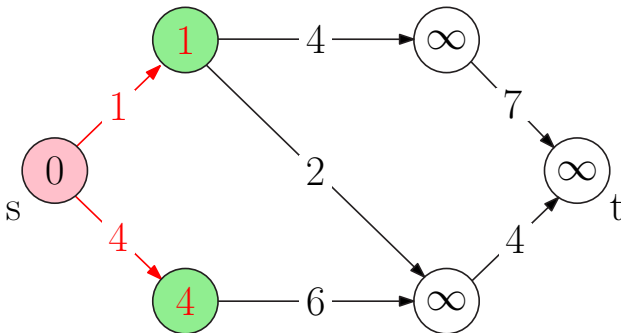
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



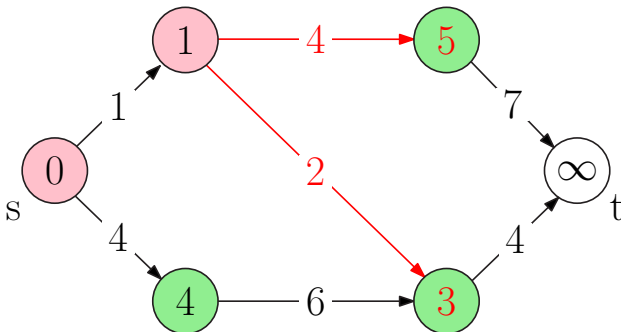
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



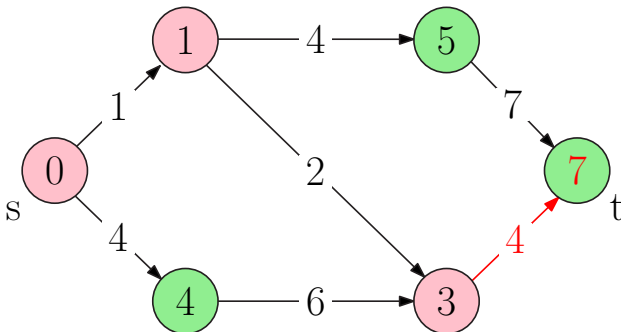
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



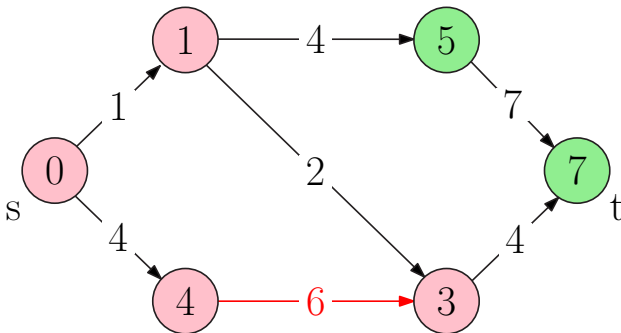
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



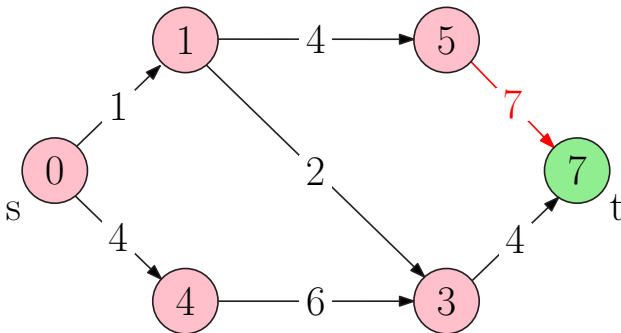
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

Beispiel



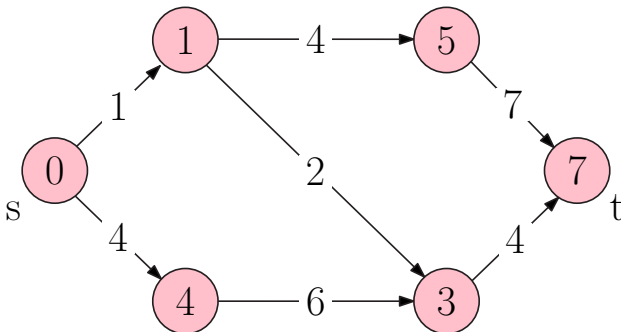
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

- benötigt **nicht-negative** Kantengewichte
- verwaltet vorläufige Distanzen $d[\cdot]$ in *Priority Queue*
→ unterscheide **erreichte** Knoten und **gescannte** Knoten

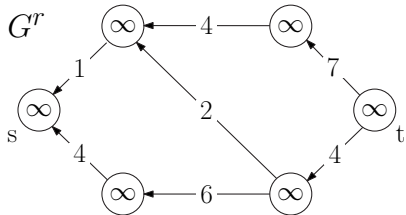
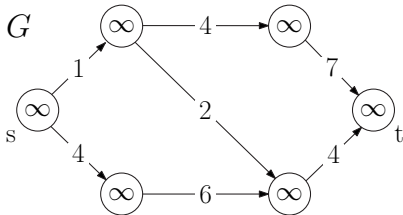
Beispiel



Eigenschaften

- führt zweimal Dijkstras Algorithmus aus
 - Vorwärtssuche ($s \rightarrow t$) auf normalem Graph G ,
 - Rückwärtssuche ($t \rightarrow s$) auf Rückwärtsgraph G^r

Beispiel



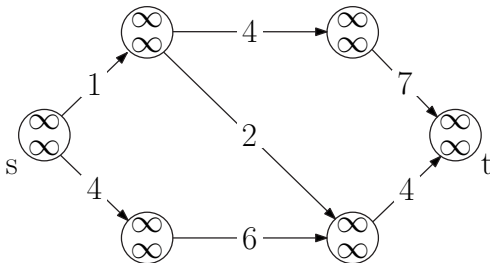
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel!



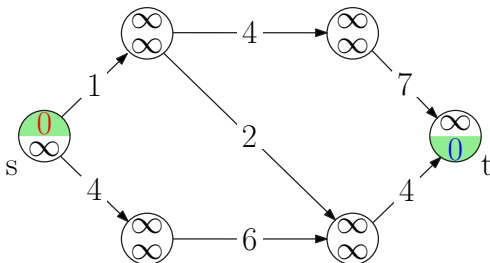
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel



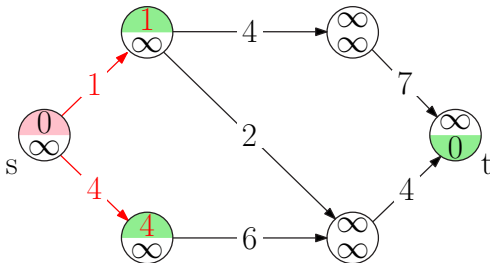
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel



forward

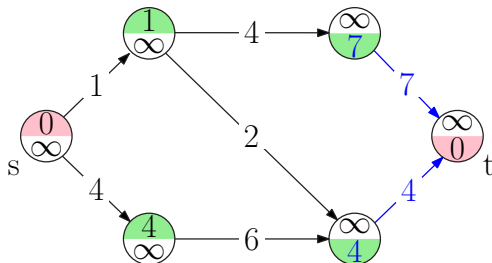
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel



backward

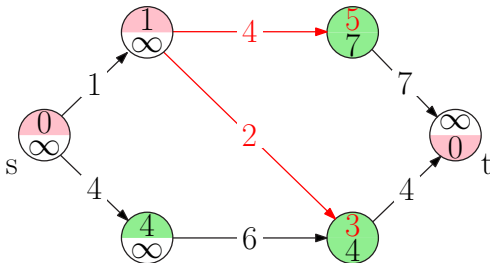
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel



forward

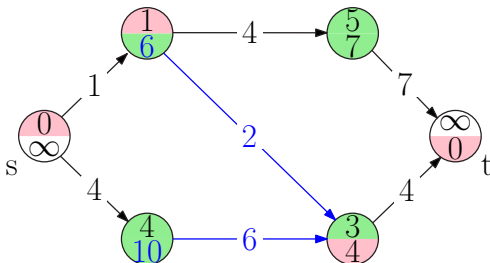
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

Beispiel



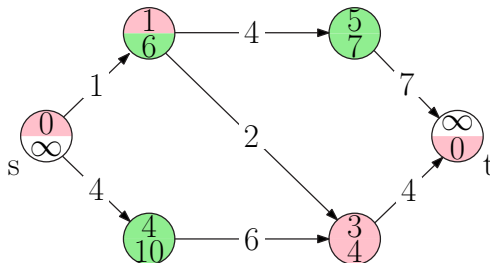
Suche in Graphen

Bidirektionale Suche

Eigenschaften (weiter)

- wechselt Suchrichtung in jedem Schritt
(alternativ: wähle Richtung mit kleinerem $pq.min$)
- Abbruch, wenn ein Knoten in beiden Suchen **gescannt** wurde
(aufpassen bei alternativer Wahl der Suchrichtung!)

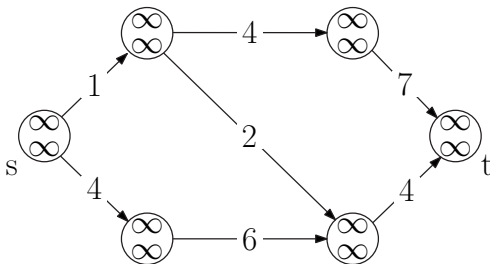
Beispiel



Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

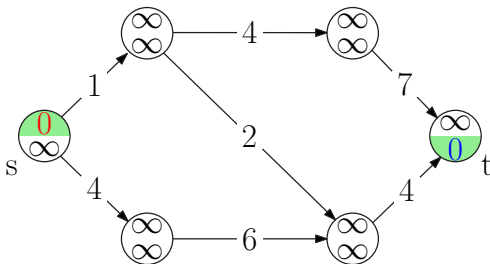
Beispiel



Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere Vorgänger u ,
 - falls vorläufige kürzeste Distanz $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere Treffpunkt v

Beispiel

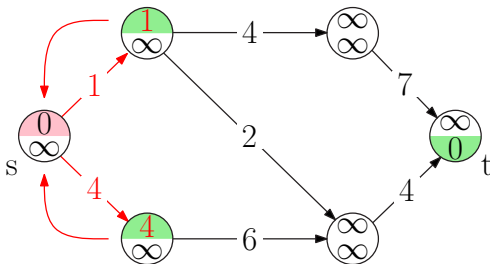


$$d[s, t] = \infty$$

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



forward

$$d[s, t] = \infty$$

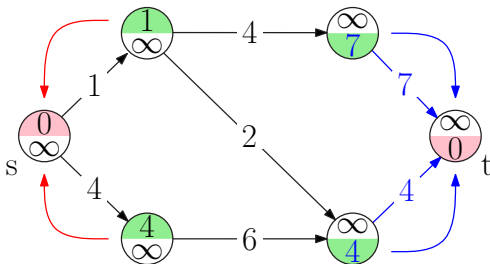
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



backward

$$d[s, t] = \infty$$

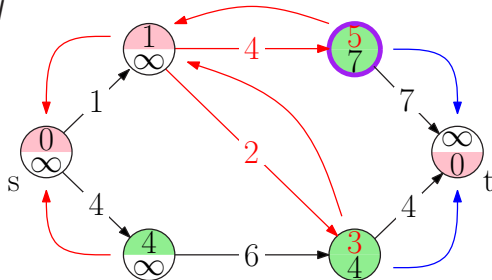
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



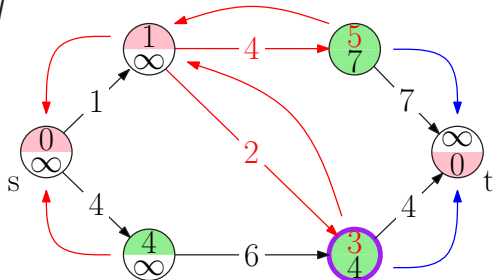
forward

$$d[s, t] = 12$$

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



forward

$$d[s, t] = 7$$

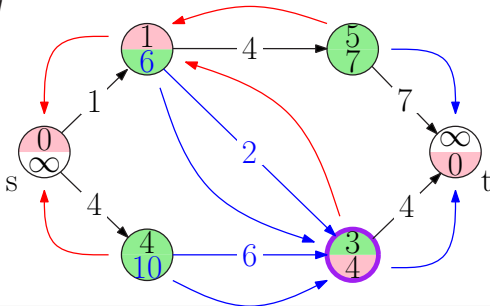
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



backward

$$d[s, t] = 7$$

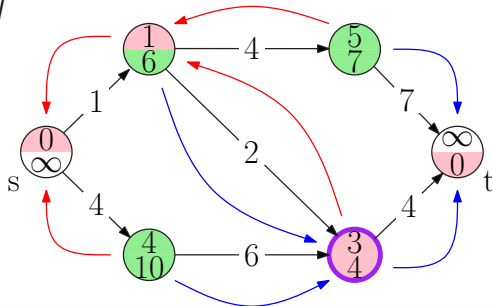
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



forward

$$d[s, t] = 7$$

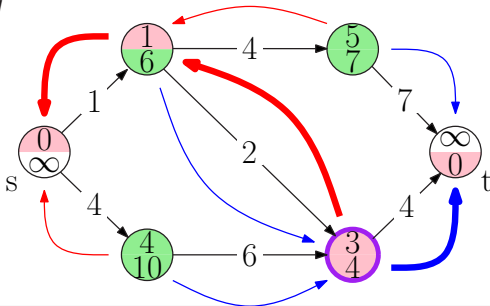
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere **Vorgänger** u ,
 - falls **vorläufige kürzeste Distanz** $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere **Treffpunkt** v

Beispiel



forward

$$d[s, t] = 7$$

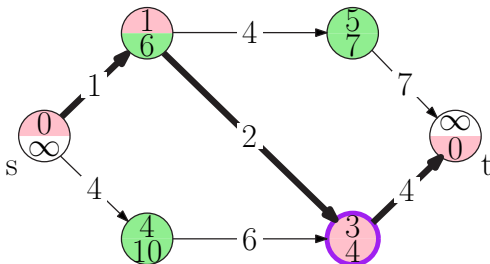
Suche in Graphen

Bidirektionale Suche

Wie wissen wir kürzeste Distanz und Weg nach Abbruch?

- wenn sich $d_{fwd}[v]$ oder $d_{bwd}[v]$ ändert,
 - aktualisiere Vorgänger u ,
 - falls vorläufige kürzeste Distanz $d[s, t] > d_{fwd}[v] + d_{bwd}[v]$
 - aktualisiere $d[s, t]$,
 - aktualisiere Treffpunkt v

Beispiel



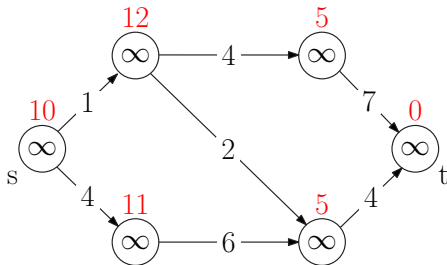
forward

$$d[s, t] = 7$$

Eigenschaften

- zielgerichtete Suche
 - benötigt **Potentialfunktion** $\text{pot}(\cdot) : V \longrightarrow \mathbb{R}$
 - **reduzierte Kantengewichte** $\bar{c}(u, v) := c(u, v) + \text{pot}(v) - \text{pot}(u)$ bzw.
 - **modifizierte Schlüssel** $\bar{d}[v] := d[v] + \text{pot}(v)$
- Abarbeitung der Knoten wird geändert (verbessert?)
→ kürzeste Pfade bleiben erhalten

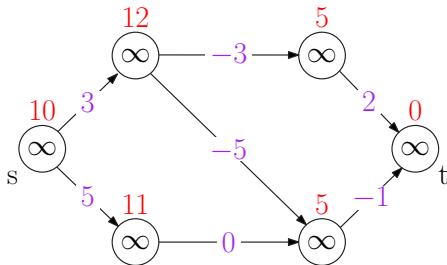
Beispiel



Eigenschaften

- zielgerichtete Suche
 - benötigt **Potentialfunktion** $\text{pot}(\cdot) : V \longrightarrow \mathbb{R}$
 - **reduzierte Kantengewichte** $\bar{c}(u, v) := c(u, v) + \text{pot}(v) - \text{pot}(u)$ bzw.
 - **modifizierte Schlüssel** $\bar{d}[v] := d[v] + \text{pot}(v)$
- Abarbeitung der Knoten wird geändert (verbessert?)
→ kürzeste Pfade bleiben erhalten

Beispiel



Eigenschaften

- Potentialfunktion $\text{pot}(\cdot) : V \longrightarrow \mathbb{R}$
- heuristische Funktion
 - modelliert **vorhandenes Wissen** über Graph
 - **schätzt Distanz zum Ziel** $\rightarrow \bar{d}[v]$ schätzt $\mu(s, t)$

gültige Potentialfunktion $\text{pot}(\cdot)$

- untere Schranke für Distanz zum Ziel t
 $\rightarrow \text{pot}(u) \leq \mu(u, t) \quad \forall u \in V$
(beendet Suche sobald t gescannt wurde)
- nicht-negative reduzierte Kantengewichte
 $\rightarrow \bar{c}(u, v) := c(u, v) + \text{pot}(v) - \text{pot}(u) \geq 0 \quad \forall (u, v) \in E$
(für Dijkstras Algorithmus)

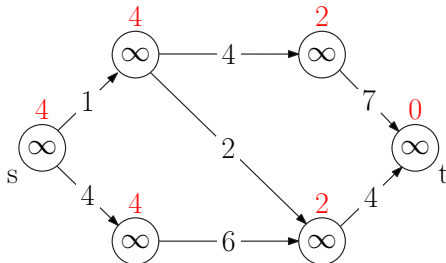
Suche in Graphen

A*-Suche – Potentialfunktionen

Woher nehmen?

- Manhattan-Distanz, euklidischer Abstand, ...
(benötigt geometrische Einbettung des Graphen)
- Distanzen zu Landmarken und Dreiecksungleichung
(funktioniert ohne Einbettung)
- Erfahrungswerte
(z.B. Bewertung von Spielsituationen)

Beispiel



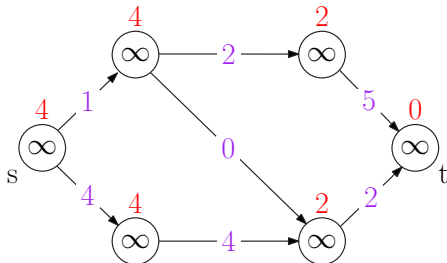
Suche in Graphen

A*-Suche – Potentialfunktionen

Woher nehmen?

- Manhattan-Distanz, euklidischer Abstand, ...
(benötigt geometrische Einbettung des Graphen)
- Distanzen zu Landmarken und Dreiecksungleichung
(funktioniert ohne Einbettung)
- Erfahrungswerte
(z.B. Bewertung von Spielsituationen)

Beispiel

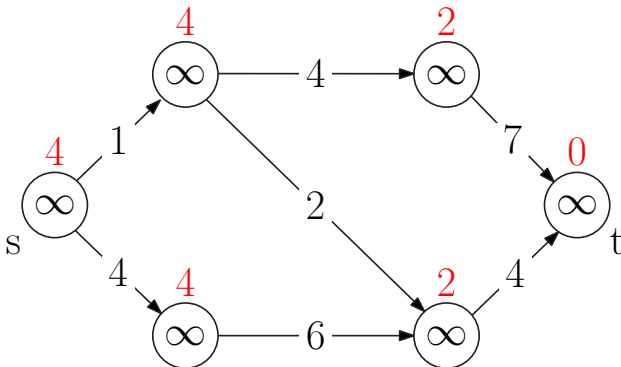


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

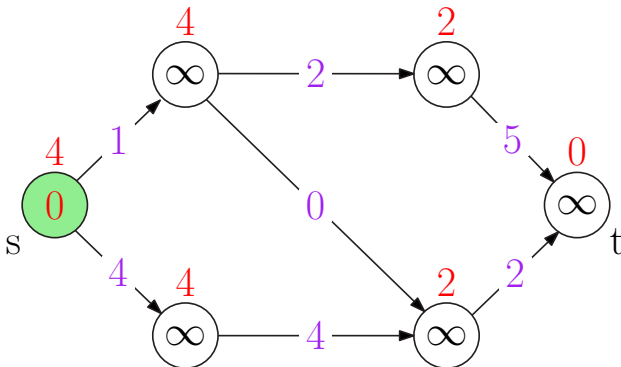


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

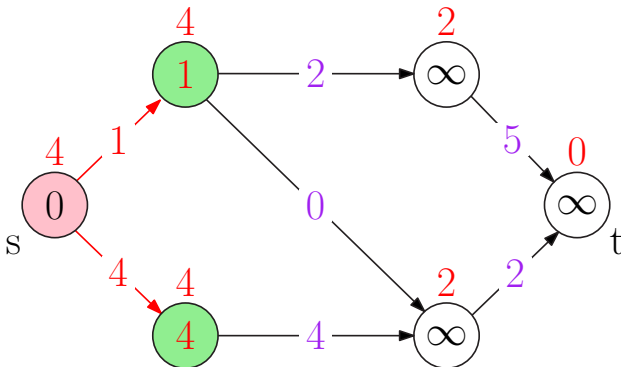


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

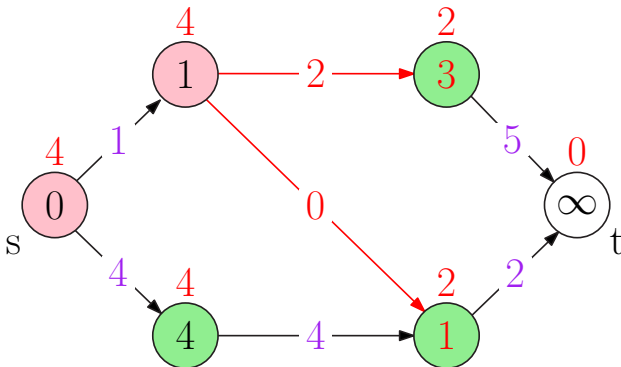


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

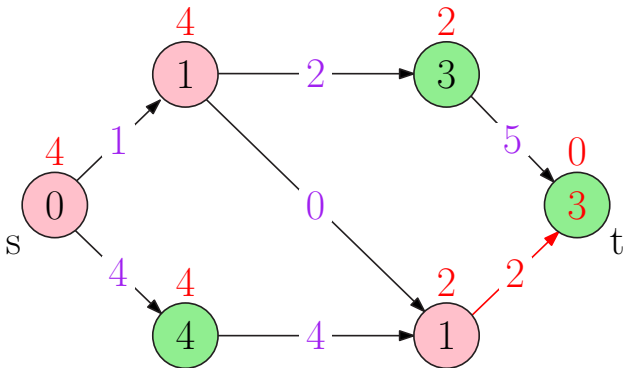


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

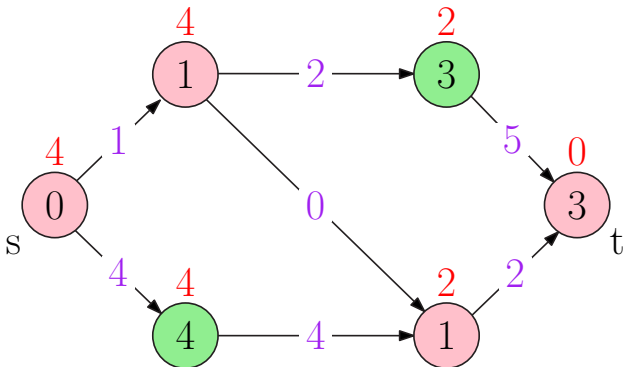


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

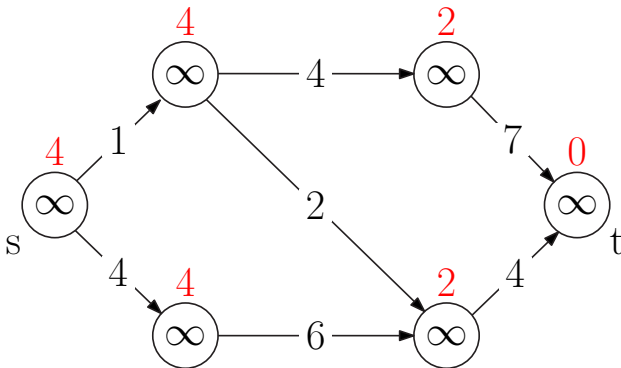


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

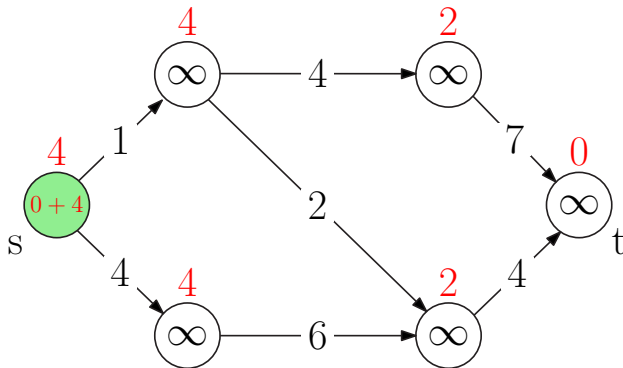


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

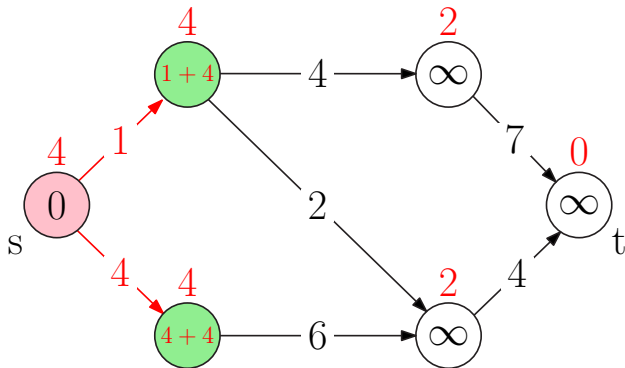


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

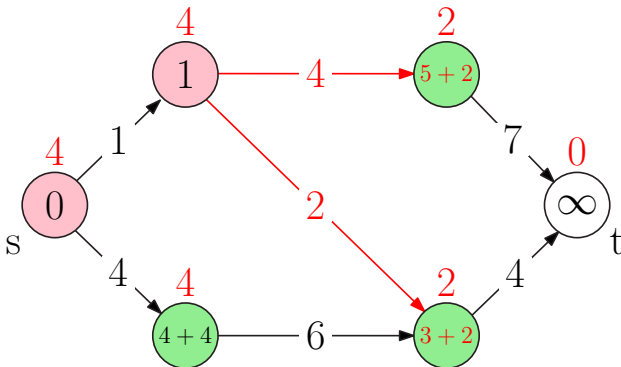


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

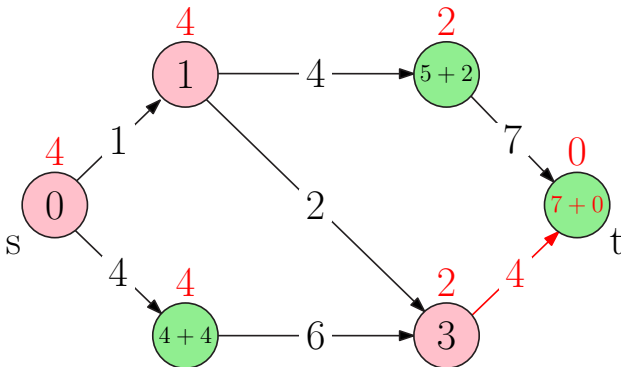


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

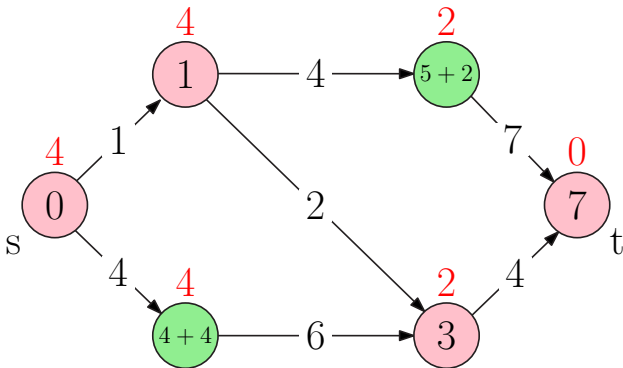


Suche in Graphen

A*-Suche

Beispiel

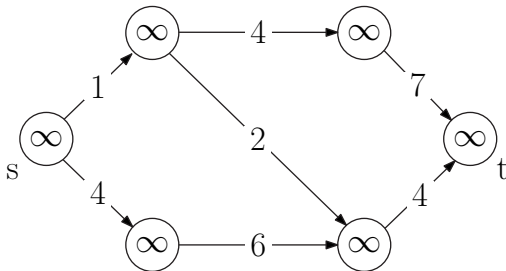
- Suche mit geänderten Schlüsseln in *Priority Queue*



Erster Ansatz

- $\text{pot}(v) := \mu(v, t)$ sei **optimales Potential**
 - Algorithmus besucht nur Knoten auf kürzesten Pfaden
 - **zu hoher Speicherverbrauch** → benötigt alle kürzesten Distanzen!

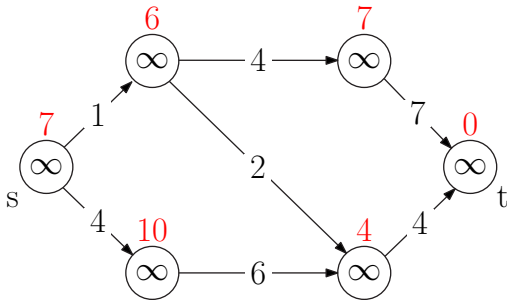
Beispiel



Erster Ansatz

- $\text{pot}(v) := \mu(v, t)$ sei **optimales Potential**
 - Algorithmus besucht nur Knoten auf kürzesten Pfaden
 - **zu hoher Speicherverbrauch** → benötigt alle kürzesten Distanzen!

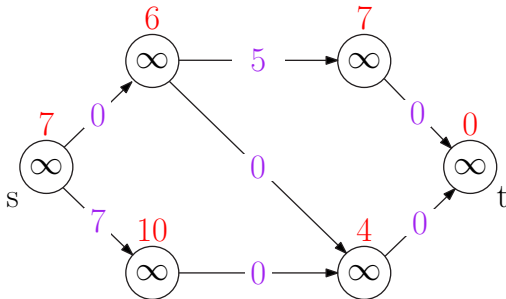
Beispiel



Erster Ansatz

- $\text{pot}(v) := \mu(v, t)$ sei **optimales Potential**
 - Algorithmus besucht nur Knoten auf kürzesten Pfaden
 - **zu hoher Speicherverbrauch** → benötigt alle kürzesten Distanzen!

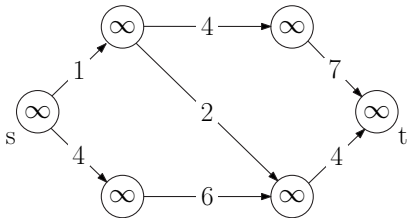
Beispiel



Kompromiss

- berechne Potential für einige Knoten l (Landmarken)
- bei konkreter Anfrage:
 - wähle Landmarke l hinter dem Ziel t (heuristisch)
 - verwende Potential $\text{pot}(v) := \mu(v, l) - \mu(t, l)$

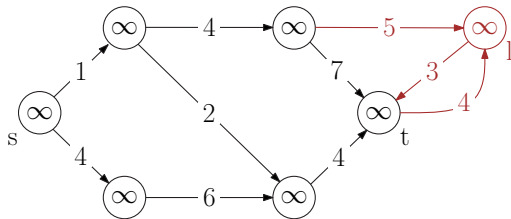
Beispiel



Kompromiss

- berechne Potential für einige Knoten l (Landmarken)
- bei konkreter Anfrage:
 - wähle Landmarke l hinter dem Ziel t (heuristisch)
 - verwende Potential $\text{pot}(v) := \mu(v, l) - \mu(t, l)$

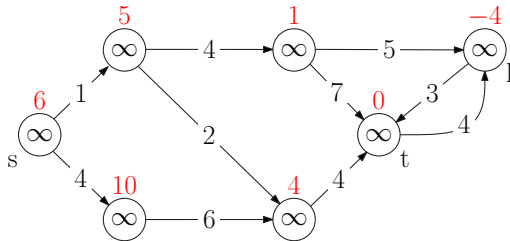
Beispiel



Kompromiss

- berechne Potential für einige Knoten l (Landmarken)
- bei konkreter Anfrage:
 - wähle Landmarke l hinter dem Ziel t (heuristisch)
 - verwende Potential $\text{pot}(v) := \mu(v, l) - \mu(t, l)$

Beispiel



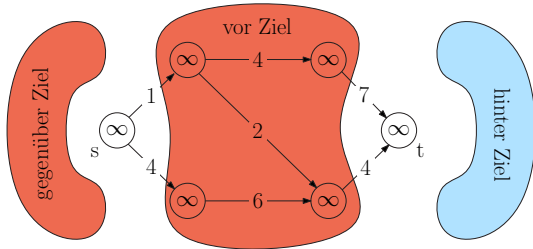
Suche in Graphen

A*-Suche – Landmarken

Qualität der Landmarken

- Was passiert bei **schlechter Landmarke** (vor/gegenüber Ziel)?
 - Korrektheit?
 - Laufzeit?

Beispiel



Qualität der Landmarken

■ Korrektheit:

immer korrekt dank **Dreiecksungleichung**

■ untere Schranke für Distanz zum Ziel t

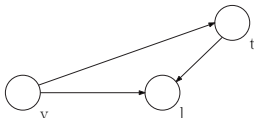
$$\text{pot}(v) = \mu(v, l) - \mu(t, l) \leq \mu(v, t) \Leftrightarrow \mu(v, l) \leq \mu(v, t) + \mu(t, l)$$

■ nicht-negative reduzierte Kantengewichte

$$\bar{c}(u, v) = c(u, v) + \text{pot}(v) - \text{pot}(u)$$

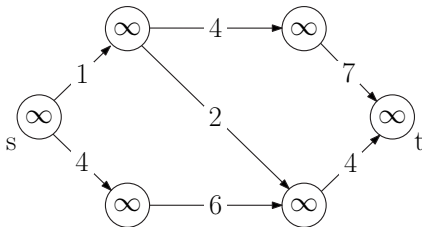
$$= c(u, v) + \mu(v, l) - \mu(t, l) - \mu(u, l) + \mu(t, l)$$

$$= c(u, v) + \mu(v, l) - \mu(u, l) \geq 0 \quad \forall (u, v) \in E$$



Qualität der Landmarken

- Laufzeit:
Algorithmus sucht in **falscher Richtung** → **langsam!**
- Beispiel

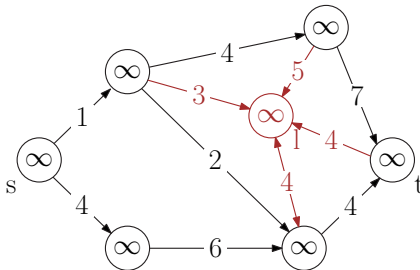


Suche in Graphen

A*-Suche – Landmarken

Qualität der Landmarken

- Laufzeit:
Algorithmus sucht in **falscher Richtung** → **langsam!**
- Beispiel

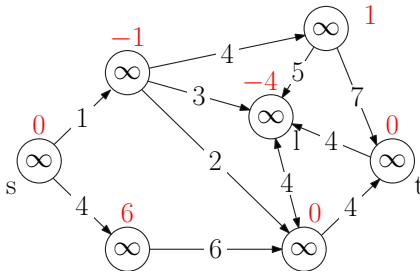


Qualität der Landmarken

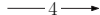
- Laufzeit:

Algorithmus sucht in **falscher Richtung** → **langsam!**

- Beispiel



Legende

	Kante mit Gewicht 4		Zeiger auf Vorgänger (in Vorwärtsrichtung)
	Kante mit Gewicht 4, wird gerade in Vorwärtsrichtung betrachtet		
	Kante mit Gewicht 4, wird gerade in Rückwärtsrichtung betrachtet		Zeiger auf Vorgänger (in Rückwärtsrichtung)
	Kante mit reduziertem Gewicht 1		
	neu eingefügte Kante mit Gewicht 4		
	Knoten v mit oberer Schranke für Distanz vom Start $d[v] = \infty$ und Potential $pot[v] = 2$		neu eingefügter Knoten
	<u>erreichter</u> Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 2$ (in <i>Priority Queue</i>)		
	<u>erreichter</u> Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 1$ (in <i>Priority Queue</i>) (obere Schranke gerade verkleinert)		
	<u>gescannt</u> Knoten v mit exakter Distanz vom Start $d[v] = 1$ (aus <i>Priority Queue</i> entfernt)		
	Knoten in Vorwärtsrichtung <u>gescannt</u> mit exakter Distanz vom Start $d_{fwd}[v] = 1$, in Rückwärtsrichtung <u>erreicht</u> mit oberer Schranke für Distanz zum Ziel $d_{bwd}[v] = 3$		
	Knoten in Vorwärtsrichtung <u>gescannt</u> mit $d_{fwd}[v] = 1$, in Rückwärtsrichtung <u>erreicht</u> mit $d_{bwd}[v] = 2$ (obere Schranke in Rückwärtsrichtung gerade verkleinert)		
	aktueller Treffpunkt von Vorwärts- und Rückwärtssuche mit minimalem $d_{fwd}[v] + d_{bwd}[v] = 3$		
	<u>erreichter</u> Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 2$ (in <i>Priority Queue</i>) (Schlüssel in <i>Priority Queue</i> ist $d[v] + pot[v] = 3 + 2$)		

Ende!



Feierabend!