

Übung 3 – Algorithmen II

Daniel Seemaier, Tobias Heuer – daniel.seemaier@kit.edu, tobias.heuer@kit.edu
http://algo2.iti.kit.edu/AlgorithmenII_WS20.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

- SCC - Ein Ausblick
 - SCC Algorithmus
 - Wiederholung
 - Invarianten
 - vertiefendes Beispiel – Floyd Warshall
 - nicht relevant für Klausur

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

Dijkstras Algorithmus



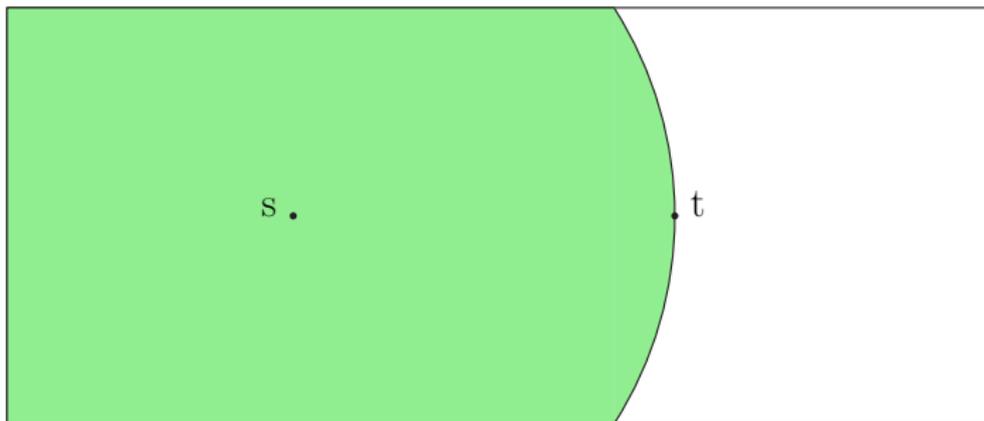
Bidirektionale Suche



A*-Suche



Bidirektionale A*-Suche



Suche in Graphen

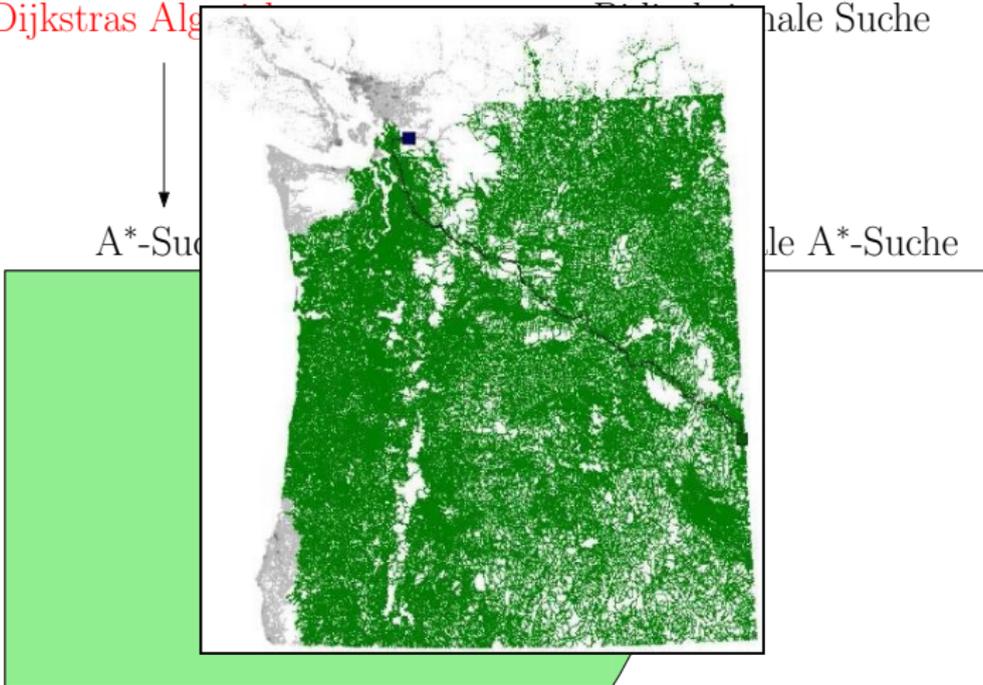
Übersicht über verschiedene Varianten

Dijkstras Algorithmus

A*-Suche

Bestmögliche Suche

Bestmögliche A*-Suche



Suche in Graphen

Übersicht über verschiedene Varianten

Dijkstras Algorithmus



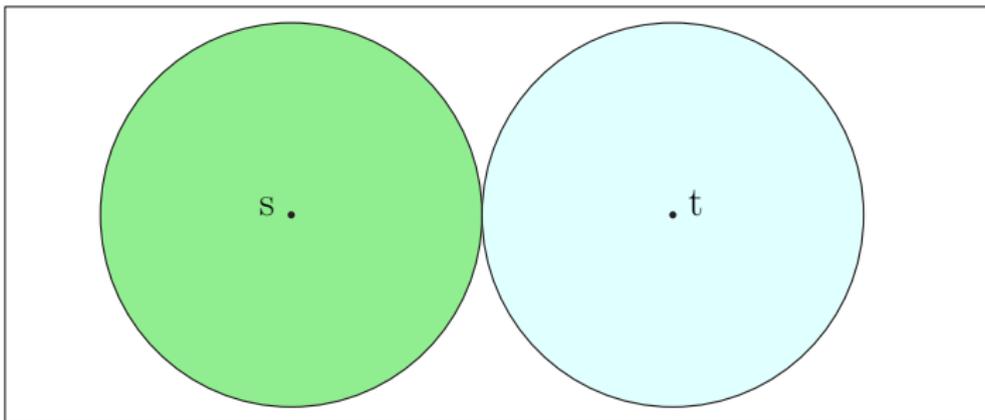
Bidirektionale Suche



A*-Suche

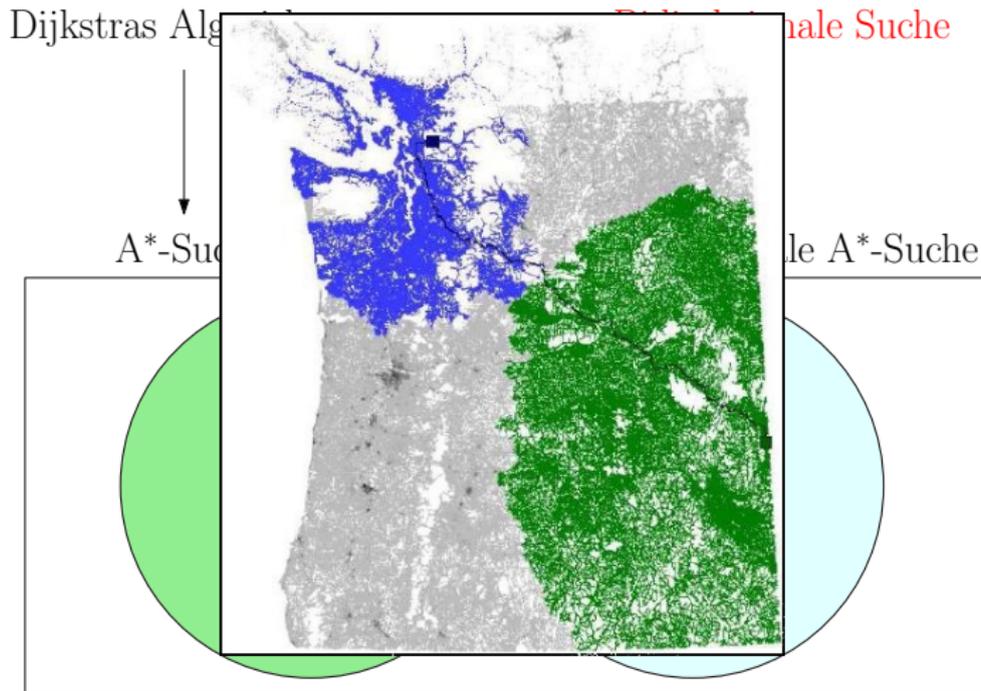


Bidirektionale A*-Suche



Suche in Graphen

Übersicht über verschiedene Varianten



Suche in Graphen

Übersicht über verschiedene Varianten

Dijkstras Algorithmus



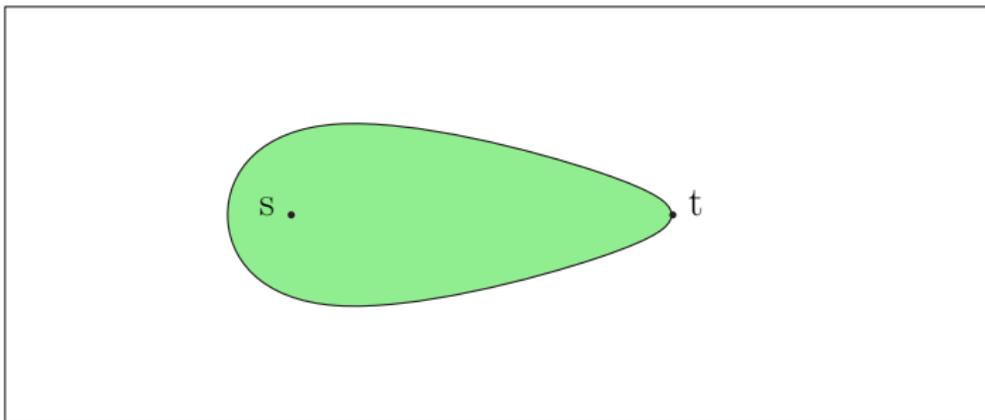
Bidirektionale Suche



A*-Suche



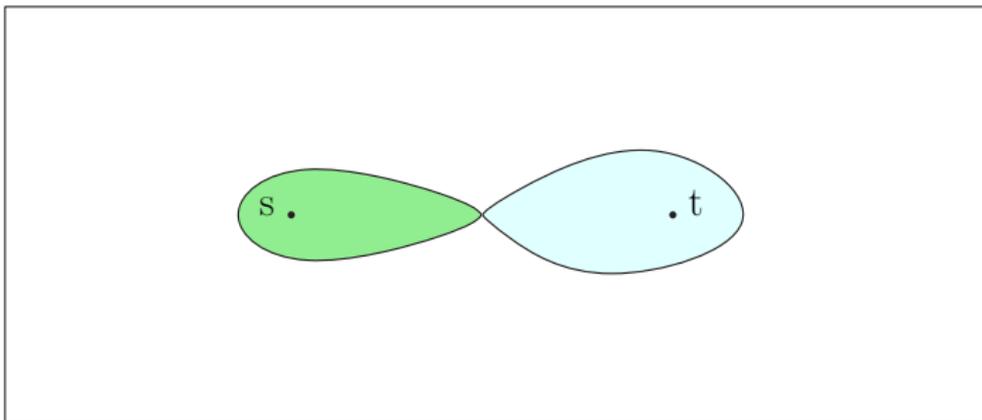
Bidirektionale A*-Suche



Suche in Graphen

Übersicht über verschiedene Varianten

Dijkstras Algorithmus \longrightarrow Bidirektionale Suche



Suche in Graphen

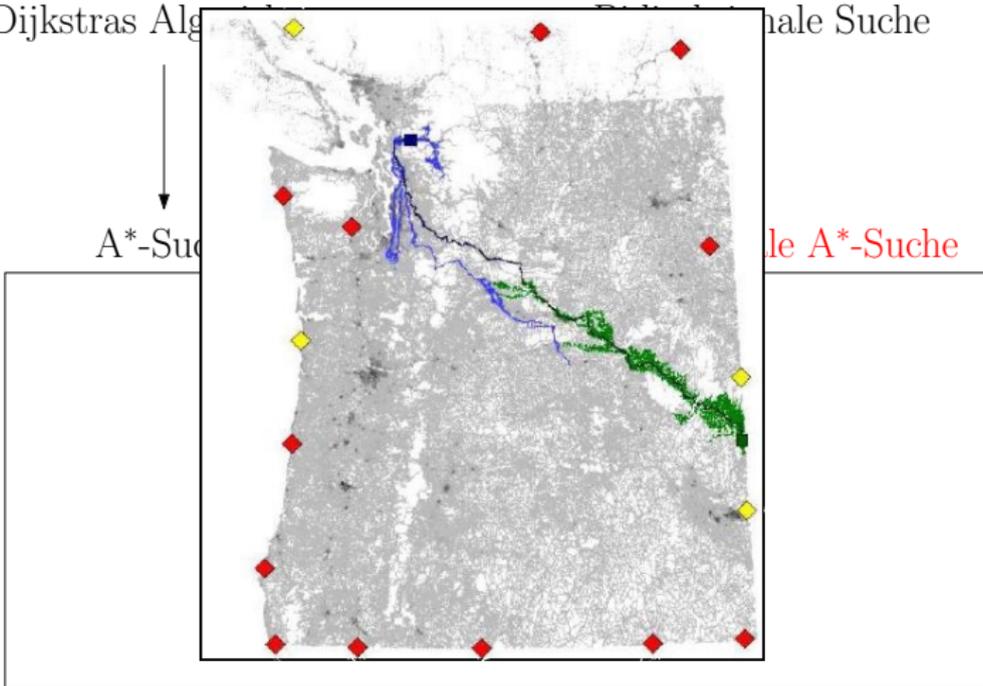
Übersicht über verschiedene Varianten

Dijkstras Alg

A*-Suche

nahe Suche

le A*-Suche



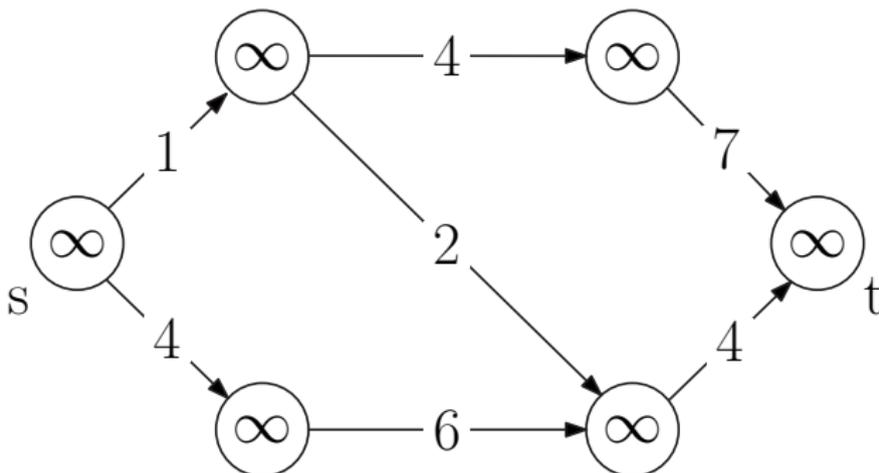
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



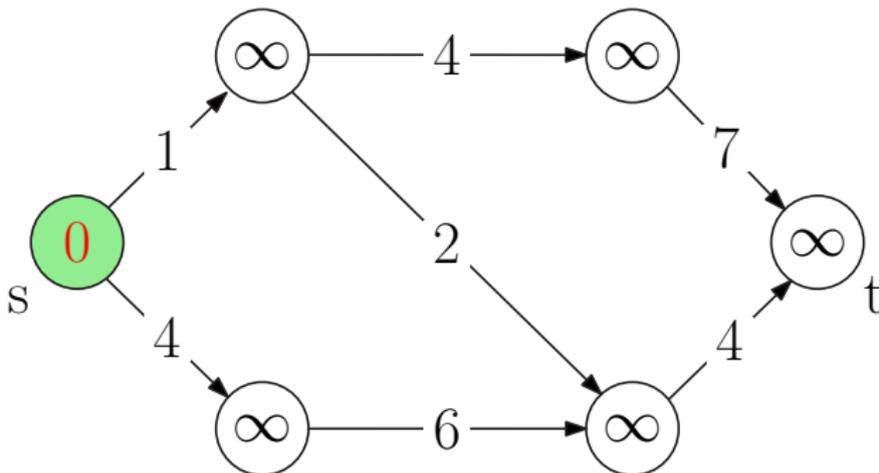
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



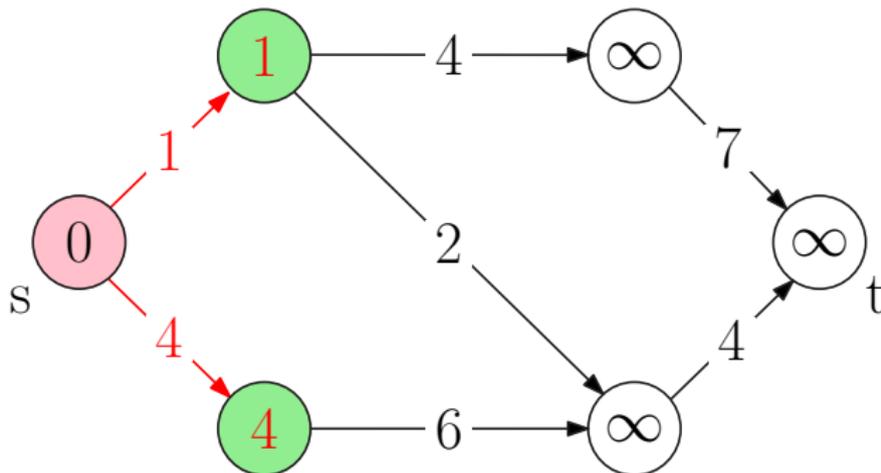
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



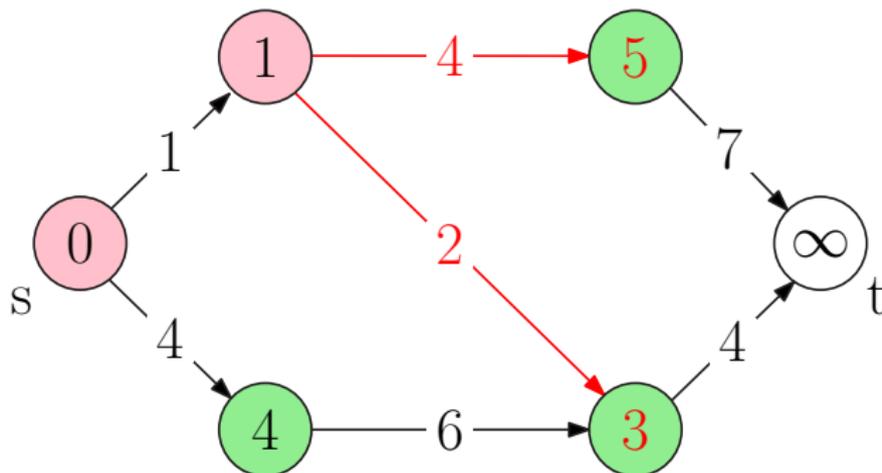
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



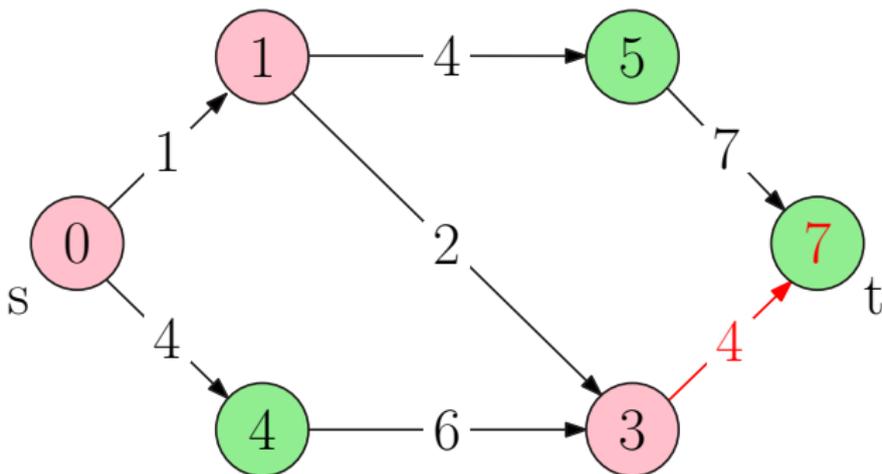
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



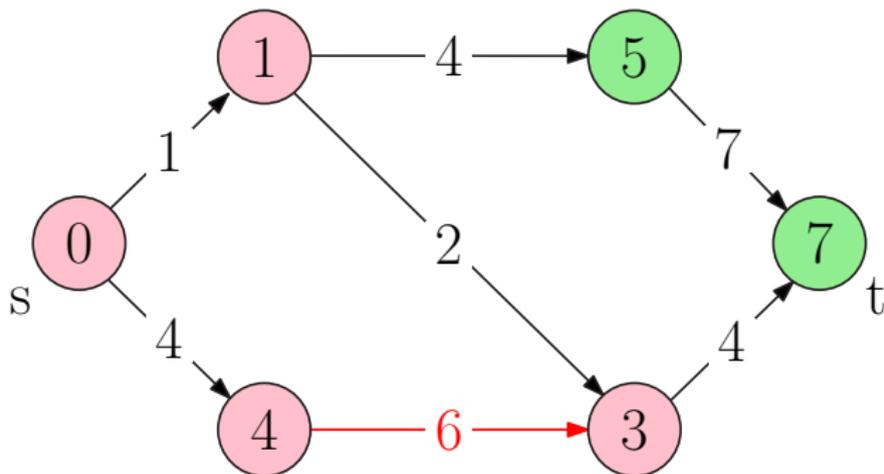
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



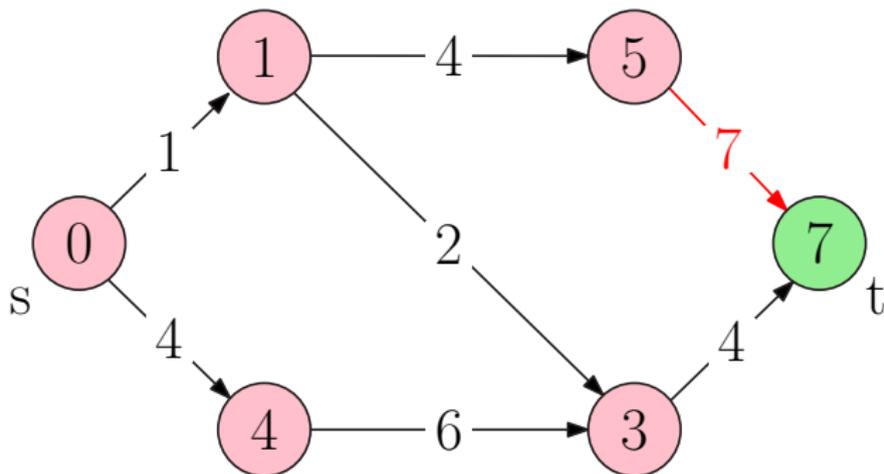
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



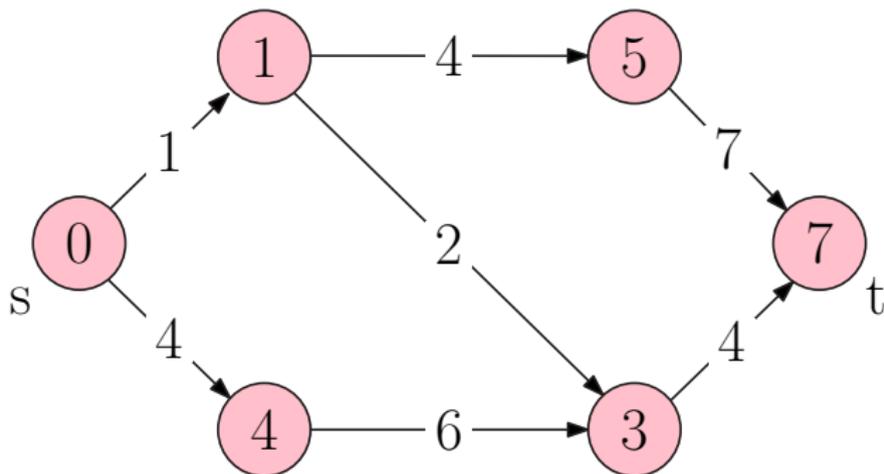
Suche in Graphen

Dijkstras Algorithmus

Einige Eigenschaften

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

Beispiel



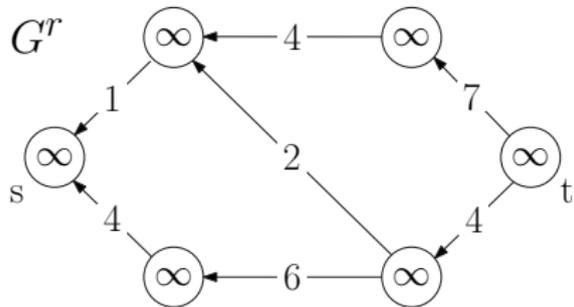
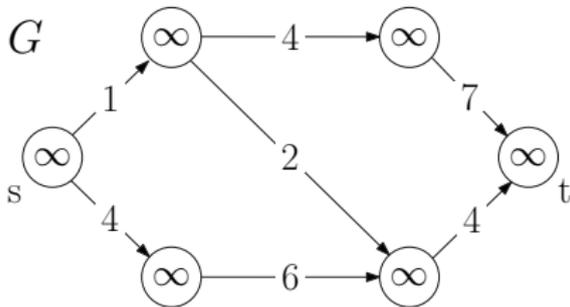
Suche in Graphen

Bidirektionale Suche

Eigenschaften

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

Beispiel



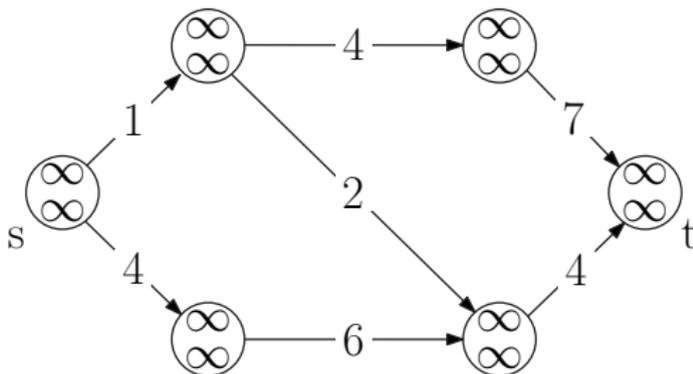
Suche in Graphen

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



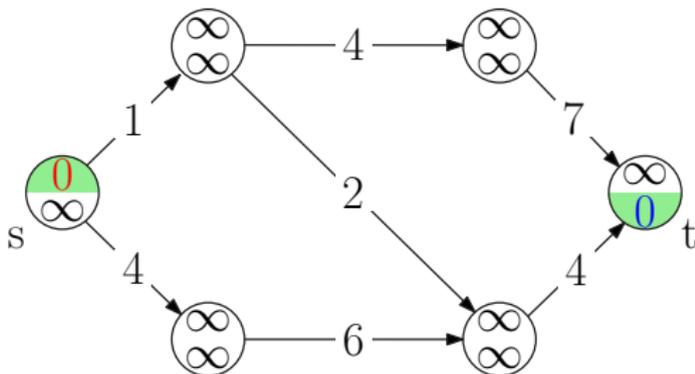
Suche in Graphen

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



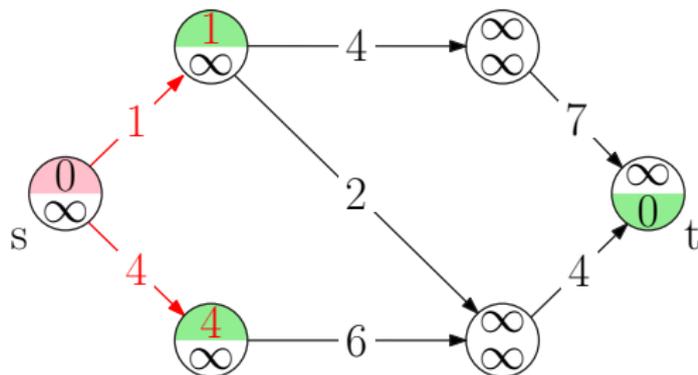
Suche in Graphen

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



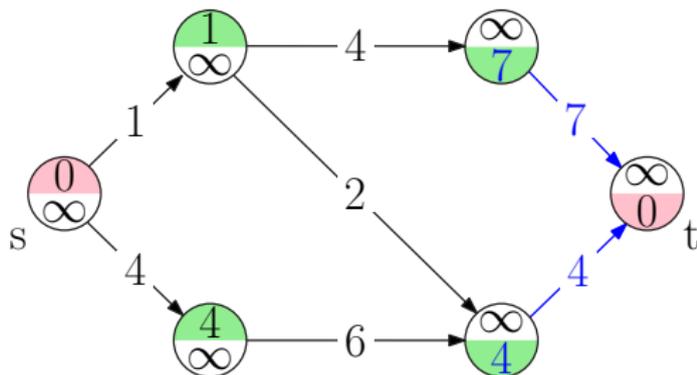
Suche in Graphen

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

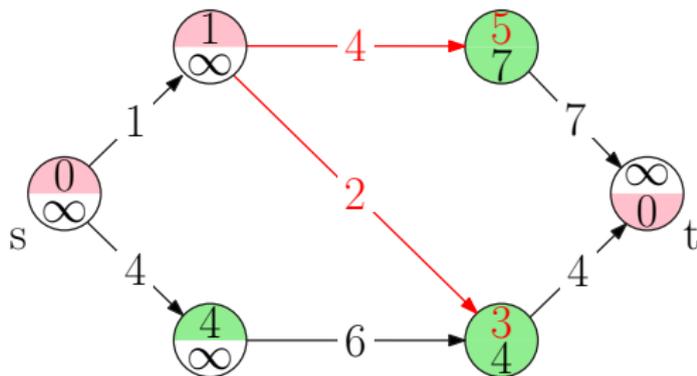
Suche in Graphen

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

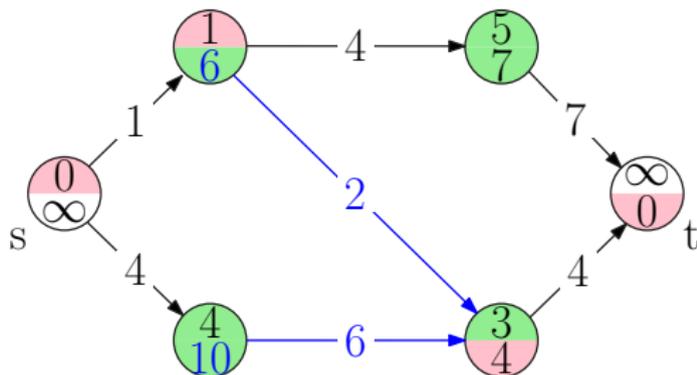
Suche in Graphen

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



backward

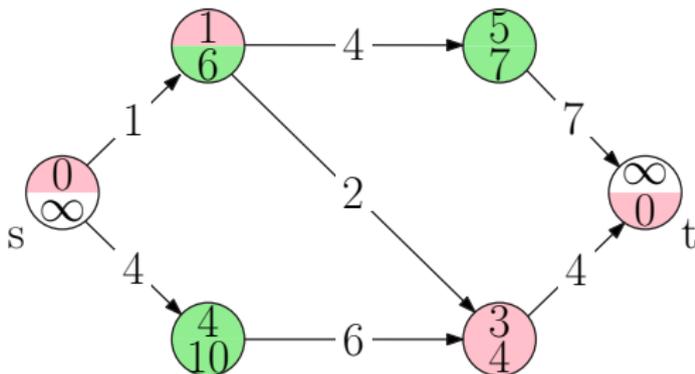
Suche in Graphen

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

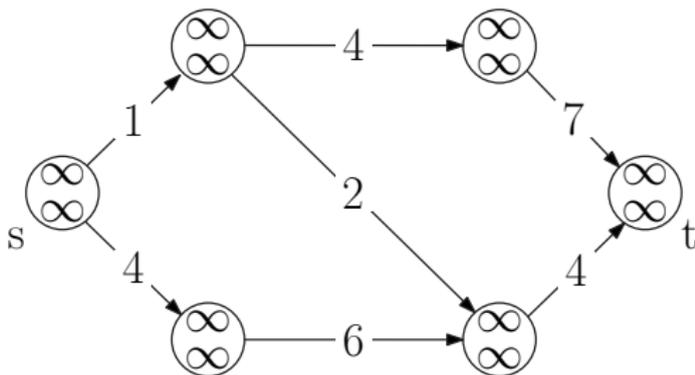
Suche in Graphen

Bidirektionale Suche

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



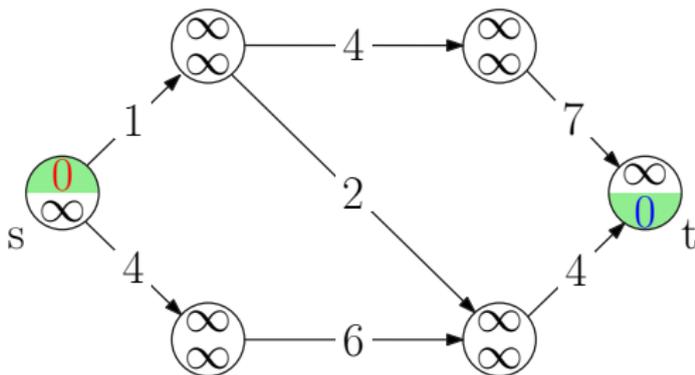
Suche in Graphen

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



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

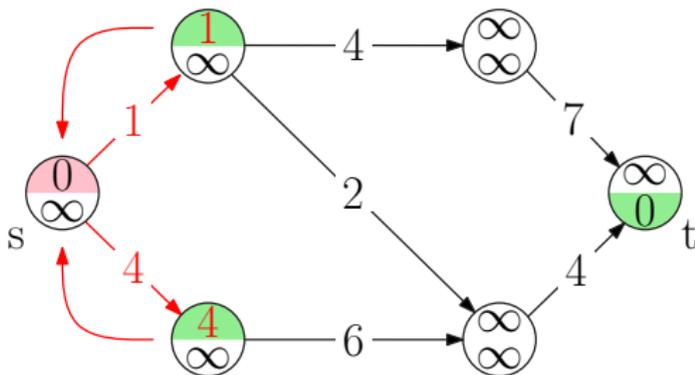
Suche in Graphen

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

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

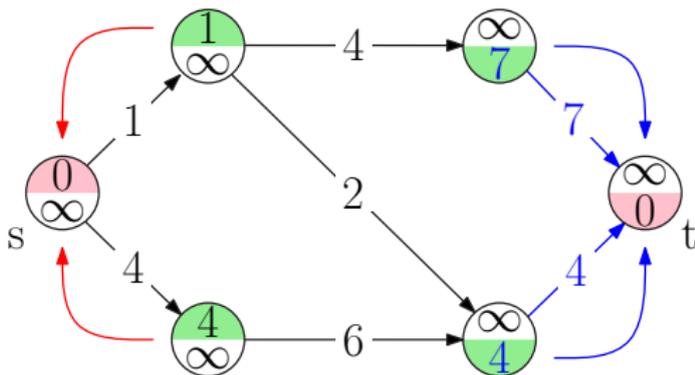
Suche in Graphen

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

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

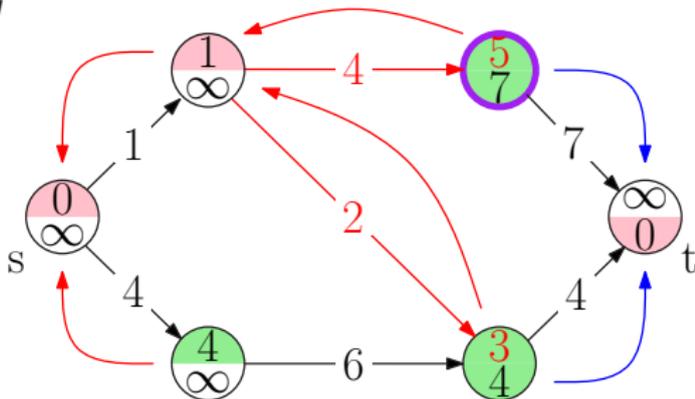
Suche in Graphen

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

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

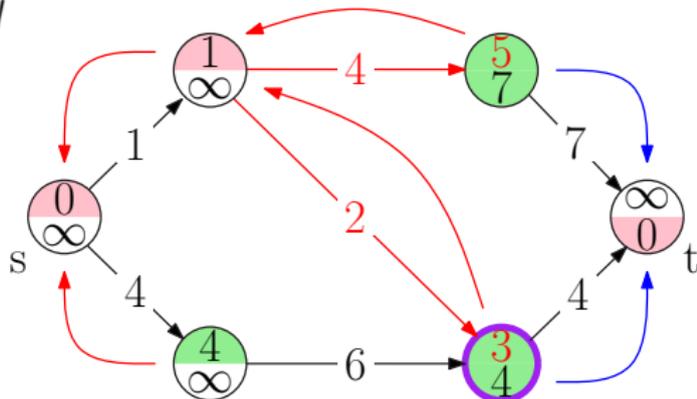
Suche in Graphen

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

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

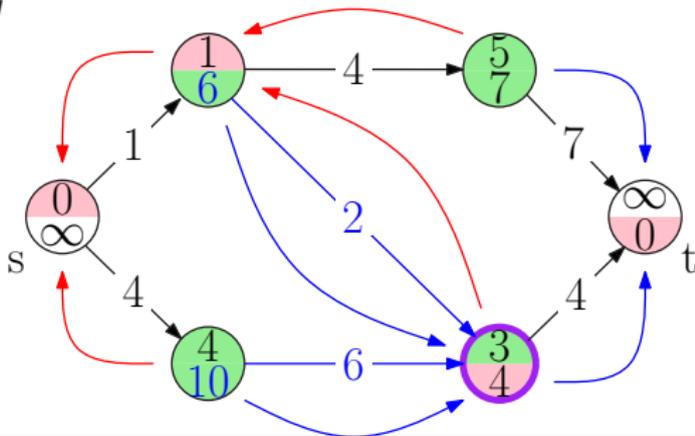
Suche in Graphen

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



backward

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

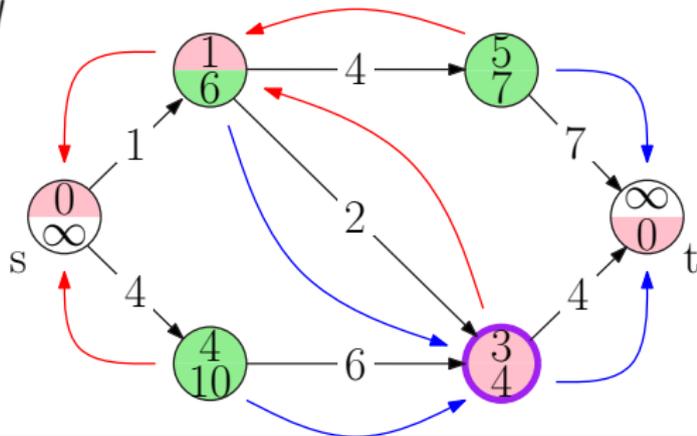
Suche in Graphen

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

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

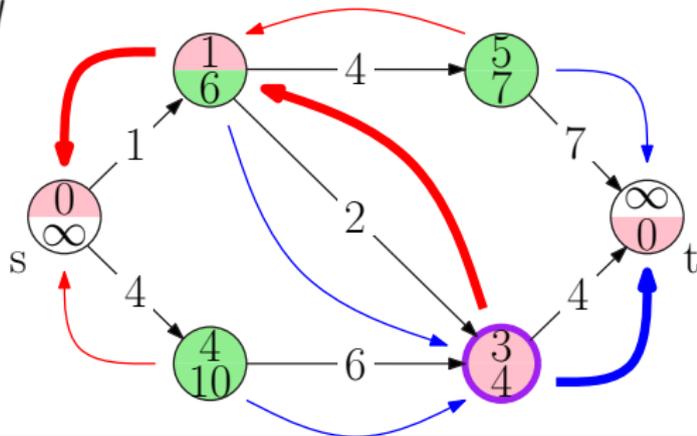
Suche in Graphen

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

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

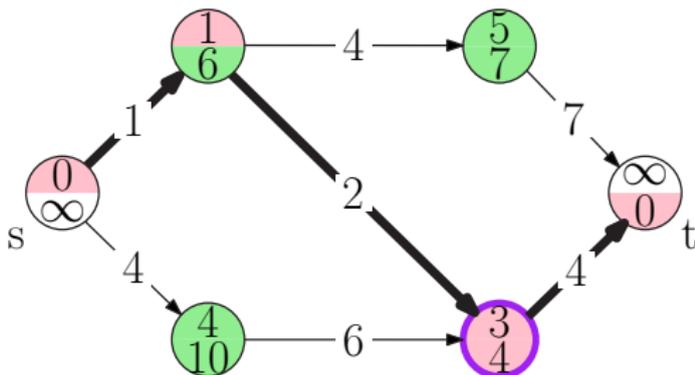
Suche in Graphen

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

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

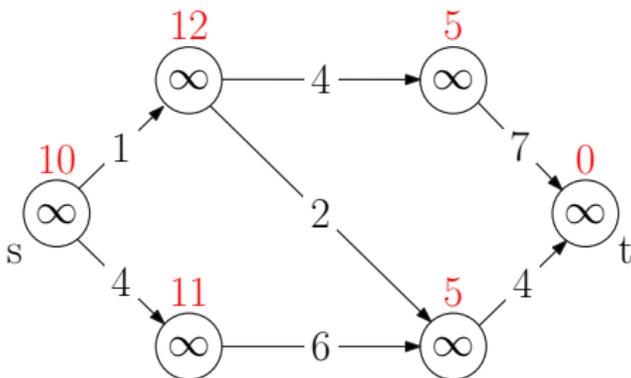
Suche in Graphen

A*-Suche

Eigenschaften

- zielgerichtete Suche
 - benötigt **Potentialfunktion** $\text{pot}(\cdot) : V \rightarrow \mathbb{R}$
 - **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



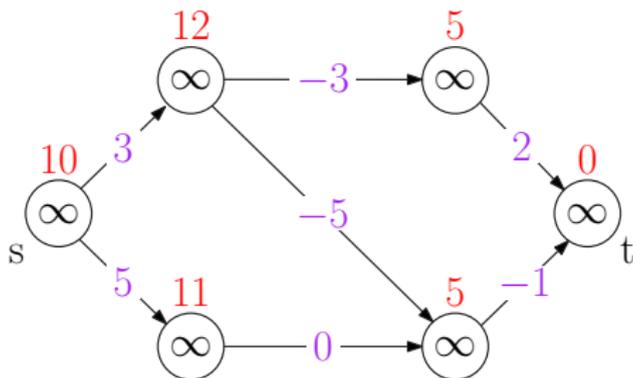
Suche in Graphen

A*-Suche

Eigenschaften

- zielgerichtete Suche
 - benötigt Potentialfunktion $\text{pot}(\cdot) : V \rightarrow \mathbb{R}$
 - 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



Suche in Graphen

A*-Suche – Potentialfunktionen

Eigenschaften

- Potentialfunktion $\text{pot}(\cdot) : V \rightarrow \mathbb{R}$
- 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)

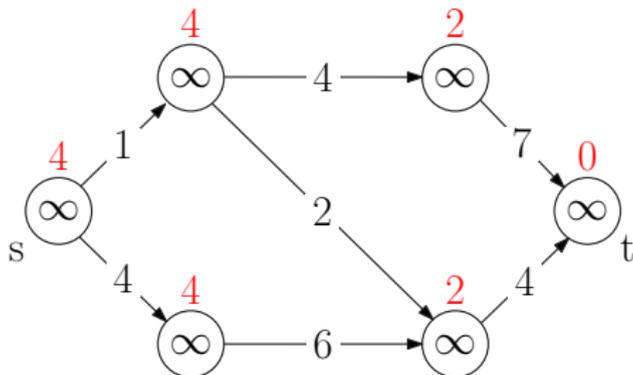
Suche in Graphen

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



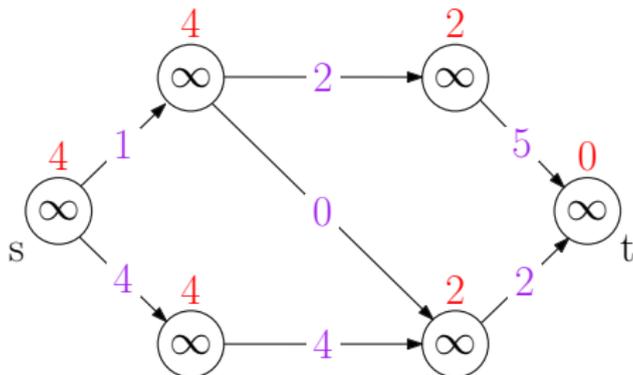
Suche in Graphen

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

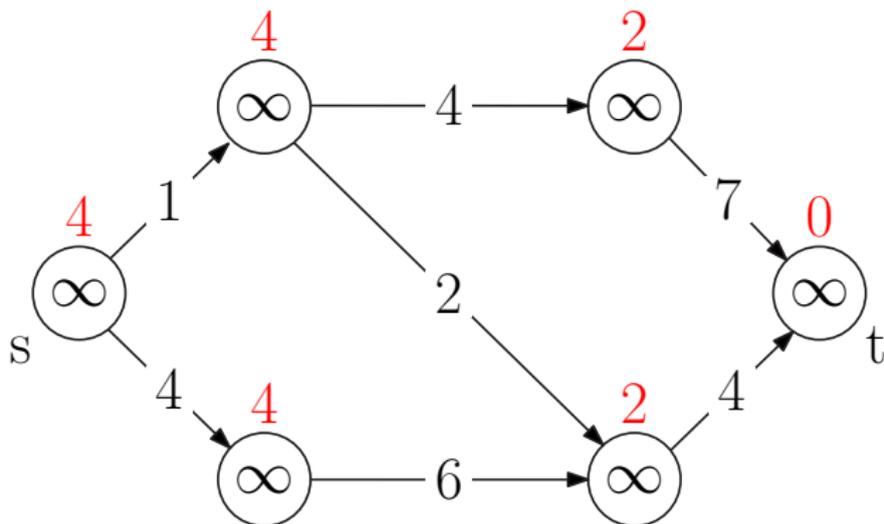


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

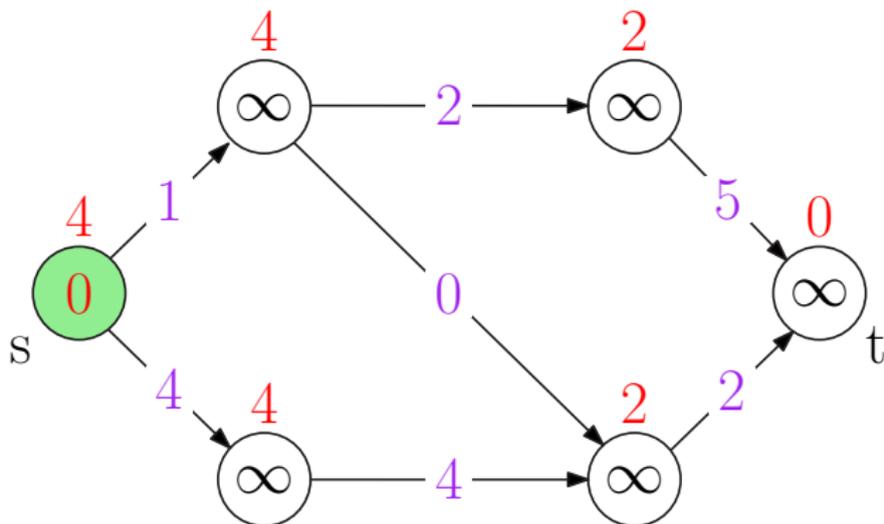


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

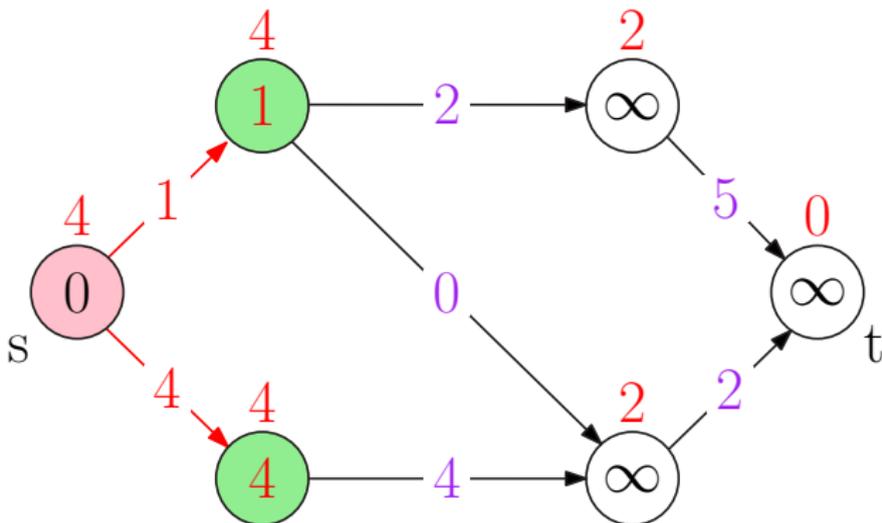


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

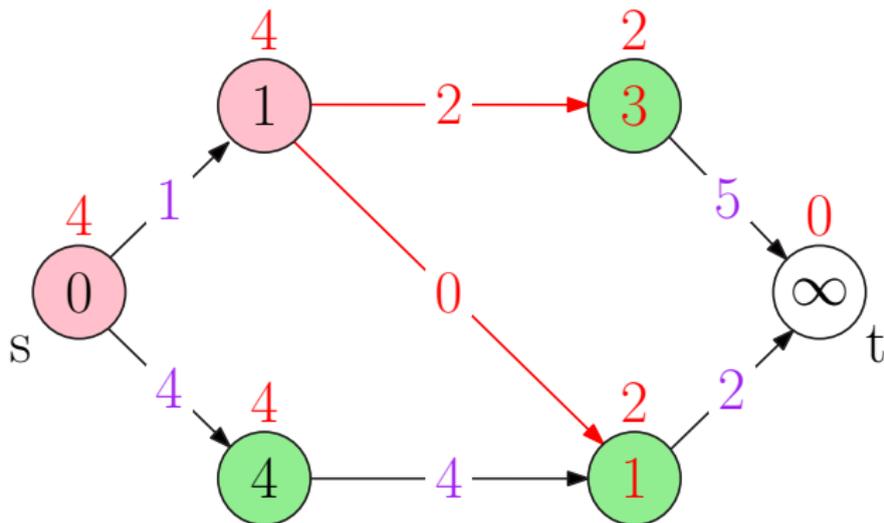


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

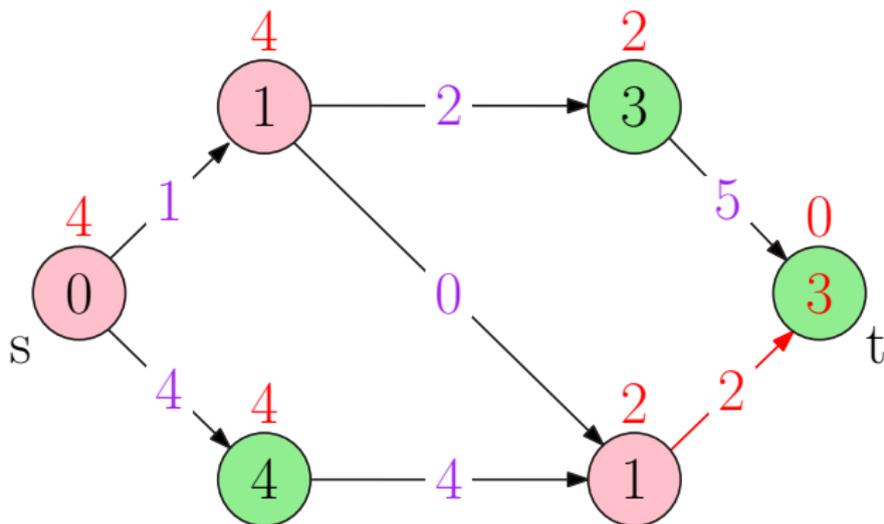


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

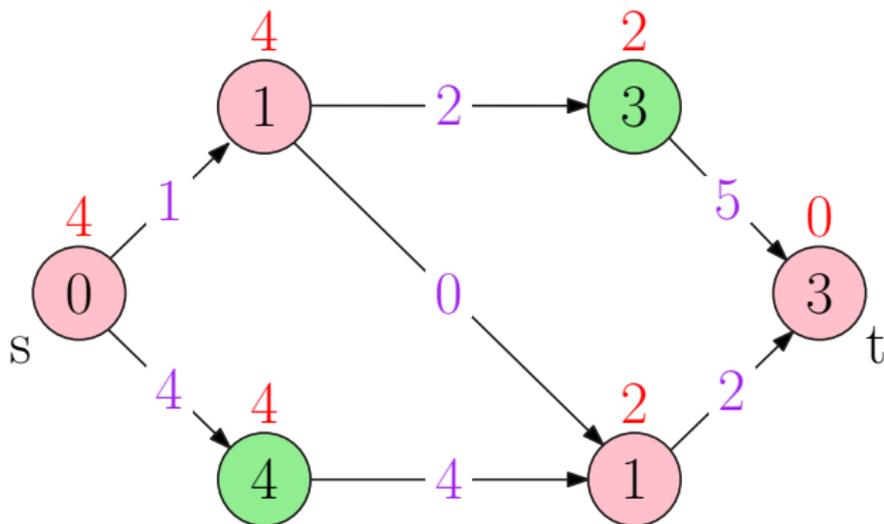


Suche in Graphen

A*-Suche

Beispiel

- Suche mit reduzierten Kantengewichten

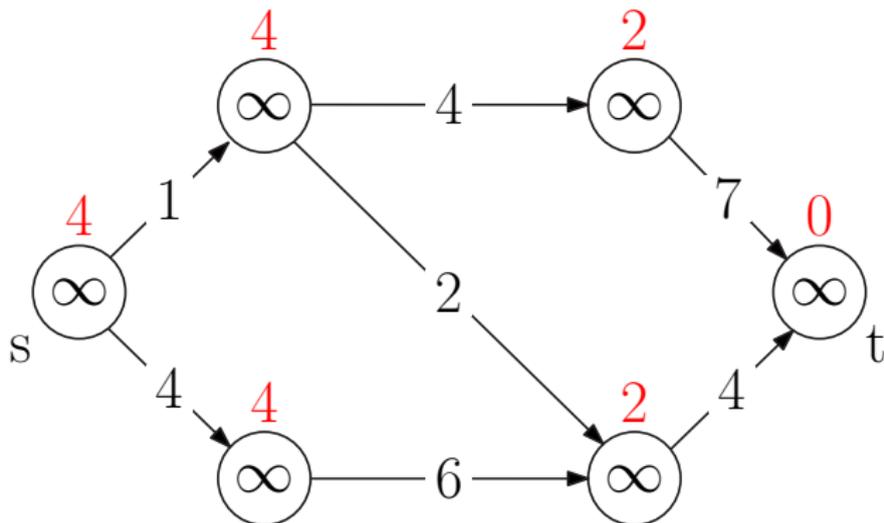


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

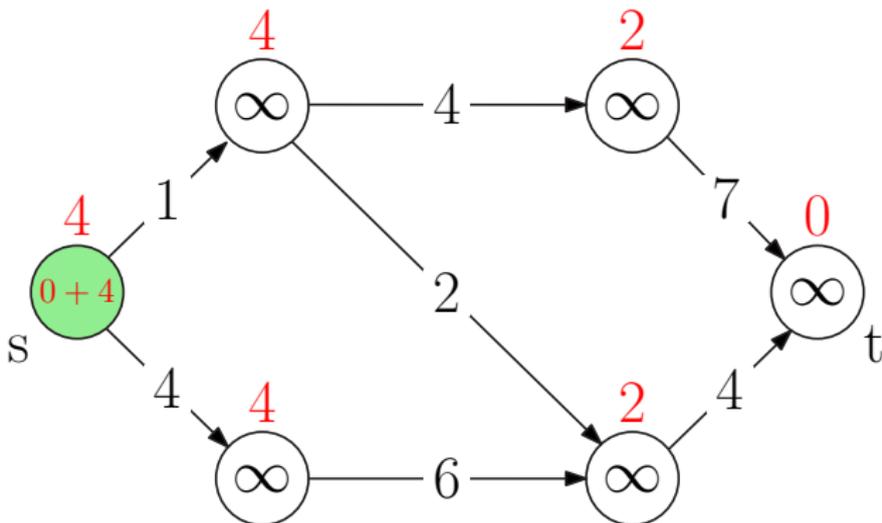


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

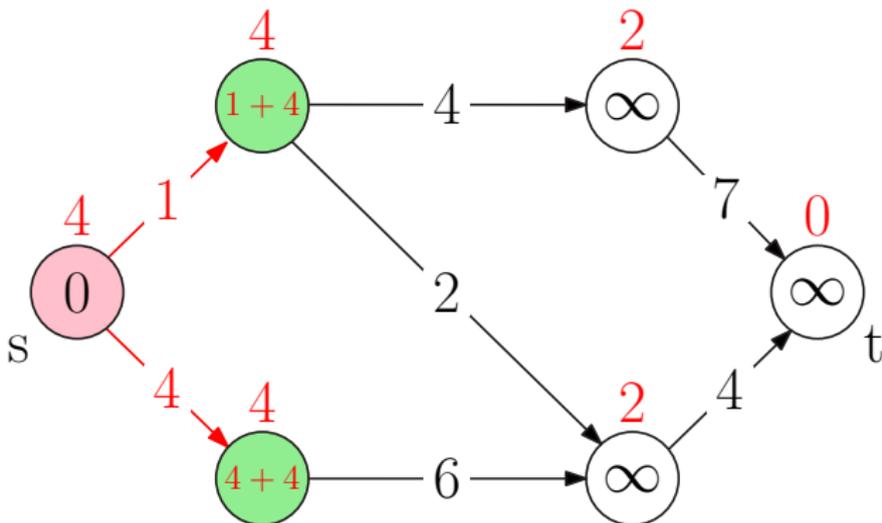


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

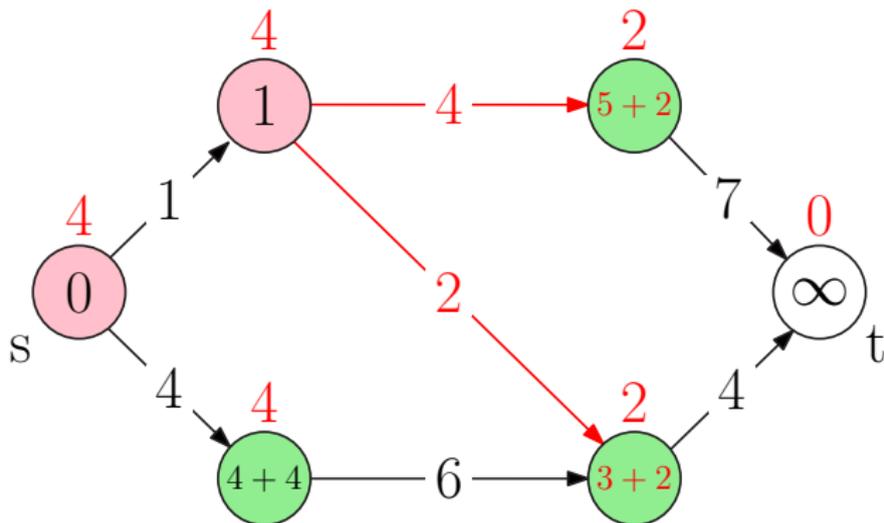


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

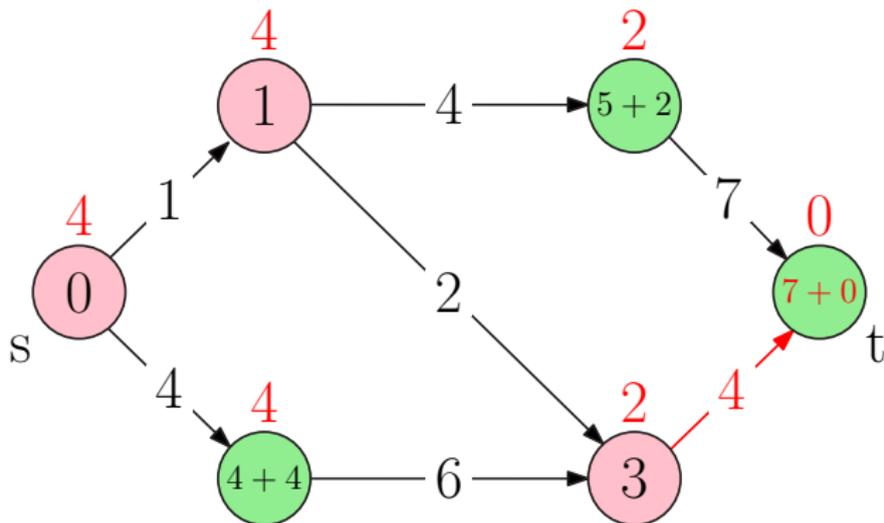


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*

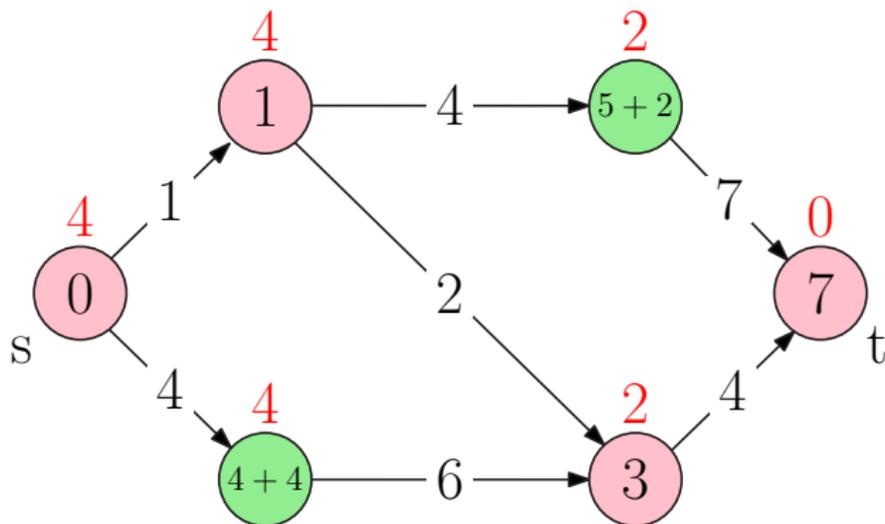


Suche in Graphen

A*-Suche

Beispiel

- Suche mit geänderten Schlüsseln in *Priority Queue*



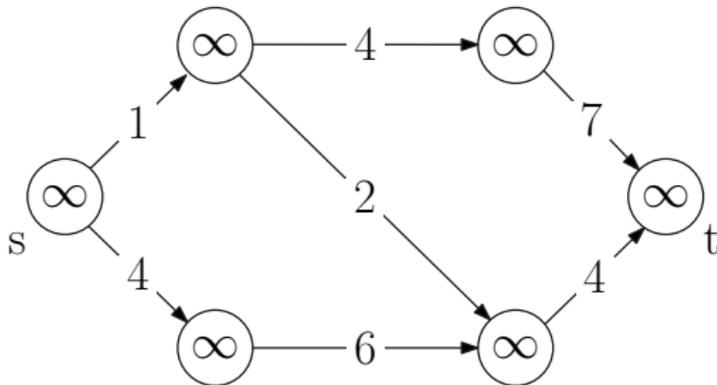
Suche in Graphen

A*-Suche – Landmarken

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



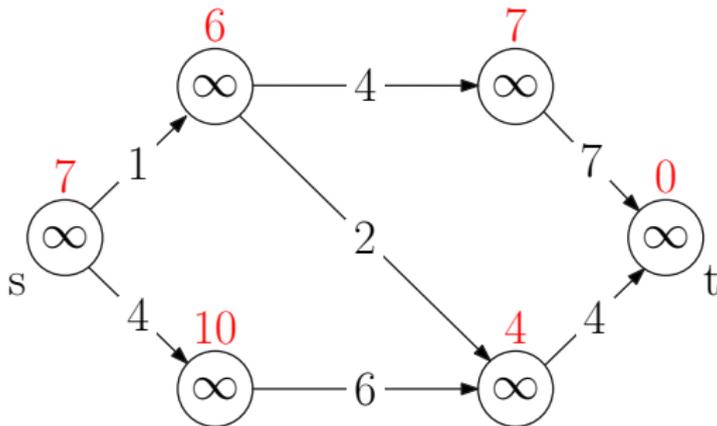
Suche in Graphen

A*-Suche – Landmarken

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



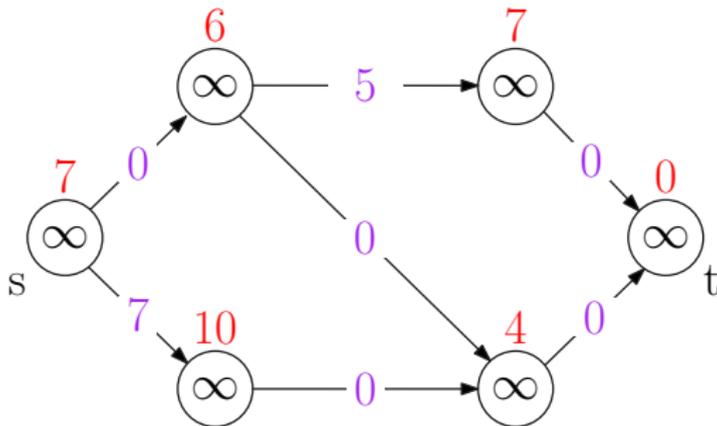
Suche in Graphen

A*-Suche – Landmarken

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



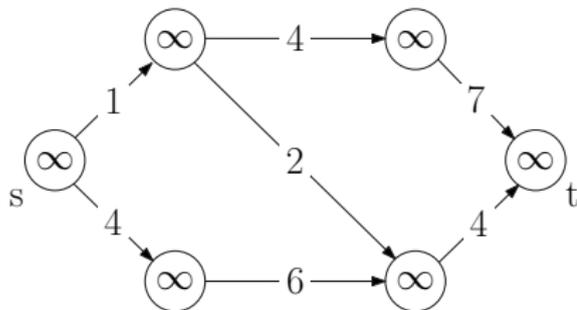
Suche in Graphen

A*-Suche – Landmarken

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



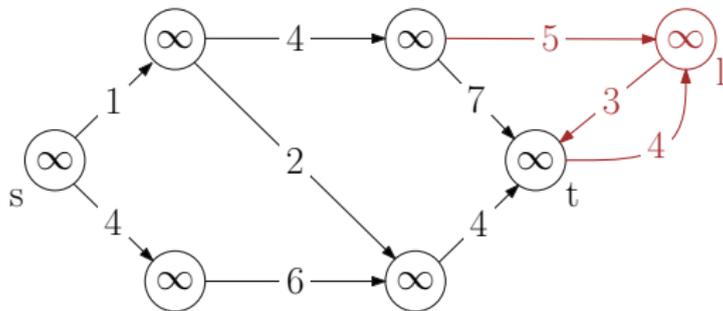
Suche in Graphen

A*-Suche – Landmarken

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



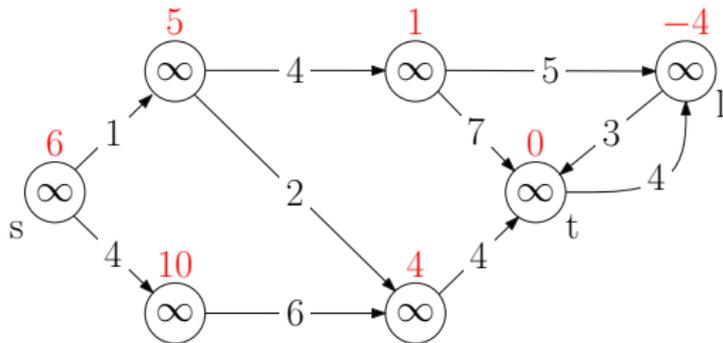
Suche in Graphen

A*-Suche – Landmarken

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



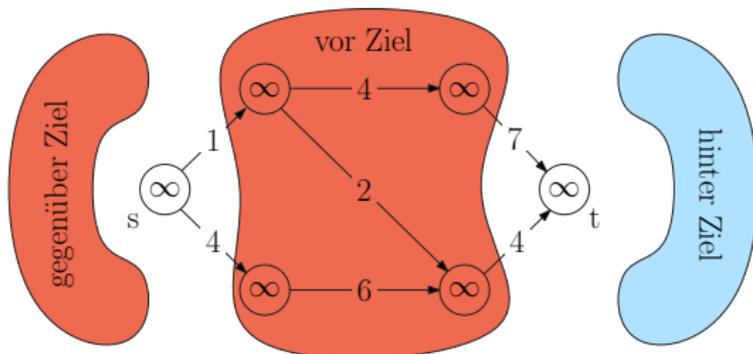
Suche in Graphen

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

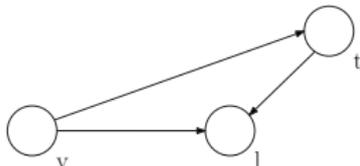
$$\text{pot}(v) = \mu(v, l) - \mu(t, l) \leq \mu(v, t) \Leftrightarrow \mu(v, l) \leq \mu(v, t) + \mu(t, l)$$

- nicht-negative reduzierte Kantengewichte

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

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

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

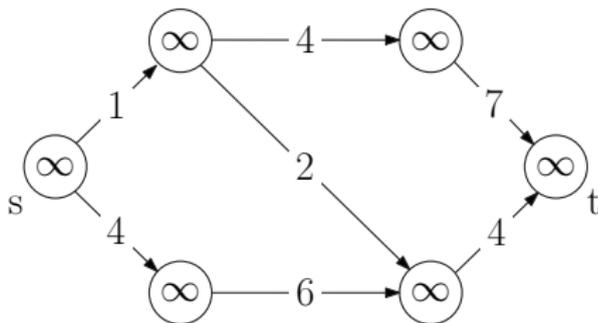


Suche in Graphen

A*-Suche – Landmarken

Qualität der Landmarken

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

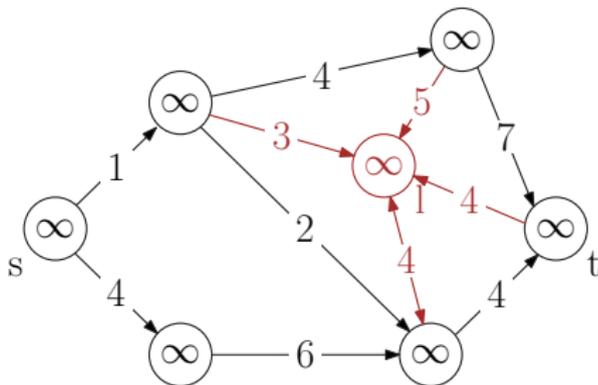


Suche in Graphen

A*-Suche – Landmarken

Qualität der Landmarken

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

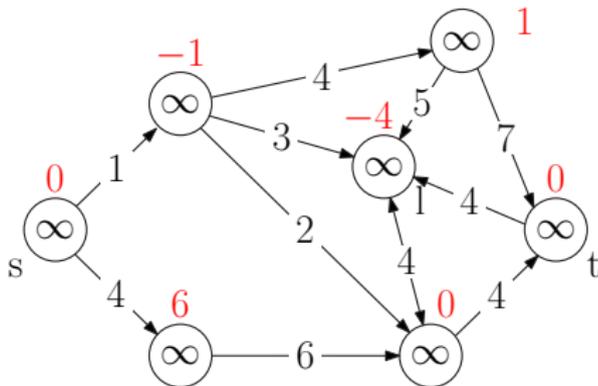


Suche in Graphen

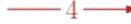
A*-Suche – Landmarken

Qualität der Landmarken

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



Legende

-  Kante mit Gewicht 4
-  Kante mit Gewicht 4, wird gerade in Vorwärtsrichtung betrachtet
-  Kante mit Gewicht 4, wird gerade in Rückwärtsrichtung betrachtet
-  Kante mit reduziertem Gewicht 1
-  neu eingefügte Kante mit Gewicht 4

-  Zeiger auf Vorgänger (in Vorwärtsrichtung)
-  Zeiger auf Vorgänger (in Rückwärtsrichtung)

 Knoten v mit oberer Schranke für Distanz vom Start $d[v] = \infty$ und Potential $pot[v] = 2$

 neu eingefügter Knoten

 erreichter Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 2$ (in *Priority Queue*)

 erreichter Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 1$ (in *Priority Queue*) (obere Schranke gerade verkleinert)

 gescannter Knoten v mit exakter Distanz vom Start $d[v] = 1$ (aus *Priority Queue* entfernt)

 Knoten in Vorwärtsrichtung gescannt mit exakter Distanz vom Start $d_{fwd}[v] = 1$, in Rückwärtsrichtung erreicht mit oberer Schranke für Distanz zum Ziel $d_{bwd}[v] = 3$

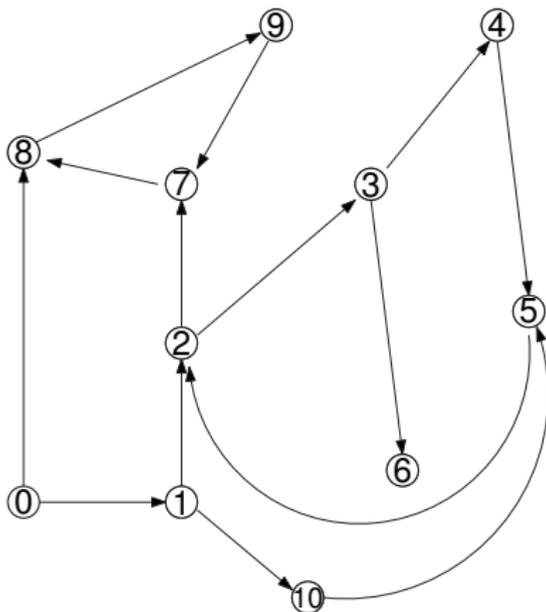
 Knoten in Vorwärtsrichtung gescannt mit $d_{fwd}[v] = 1$, in Rückwärtsrichtung erreicht mit $d_{bwd}[v] = 2$ (obere Schranke in Rückwärtsrichtung gerade verkleinert)

 aktueller Treffpunkt von Vorwärts- und Rückwärtssuche mit minimalem $d_{fwd}[v] + d_{bwd}[v] = 3$

 erreichter Knoten v mit oberer Schranke für Distanz vom Start $d[v] = 2$ (in *Priority Queue*) (Schlüssel in *Priority Queue* ist $d[v] + pot[v] = 3 + 2$)

Starke Zusammenhangskomponenten

SCC (Wiederholung)



oNodes | oReps

Invariante 1

Kein Kanten von geschlossenen in offene Komponenten.

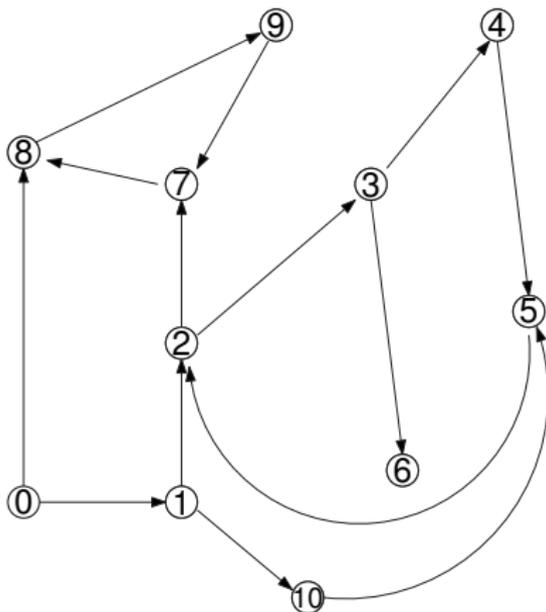
Invariante 2

Offene Komponenten liegen auf Pfad.

Invariante 3

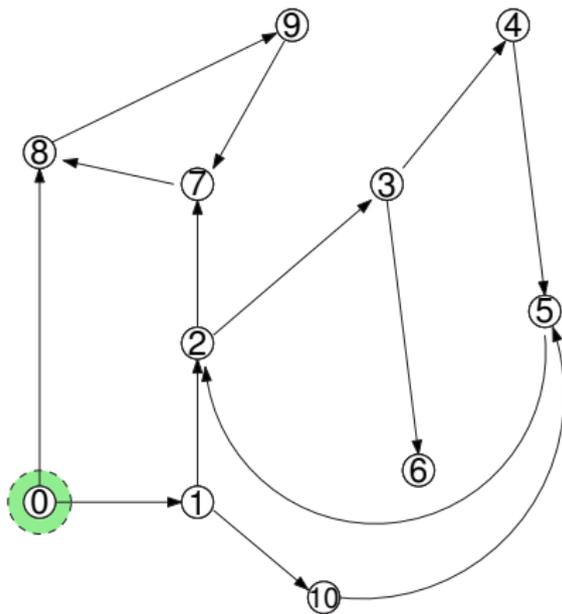
Repräsentanten partitionieren offene Komponenten bzgl. dfsNum.

SCC (Wiederholung)



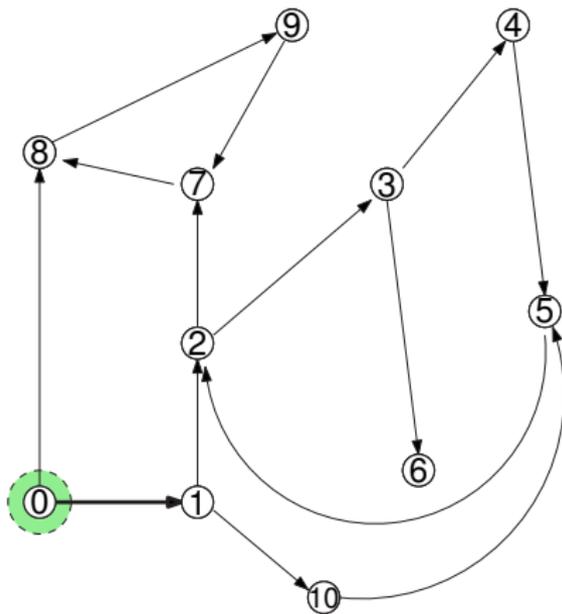
oNodes | oReps

SCC (Wiederholung)



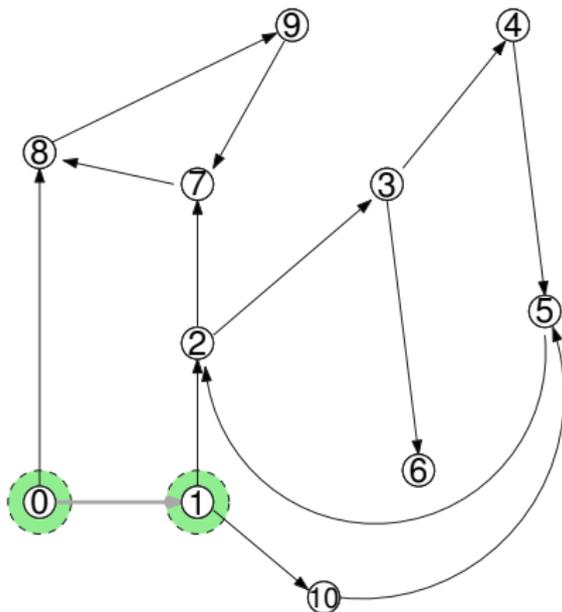
oNodes	oReps
0	0

SCC (Wiederholung)



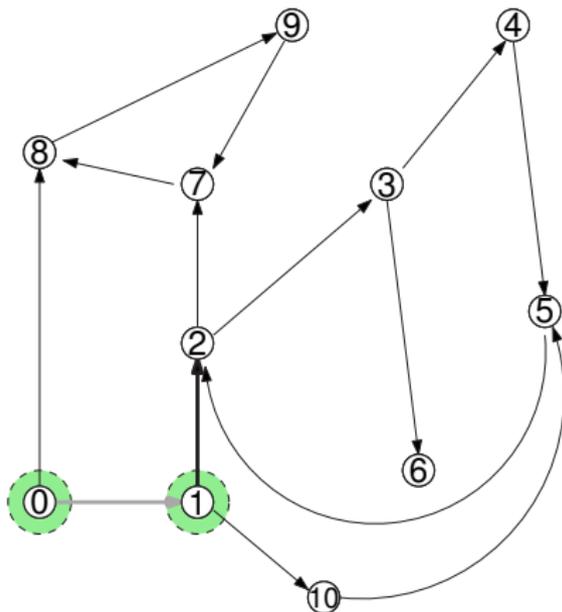
oNodes	oReps
0	0

SCC (Wiederholung)



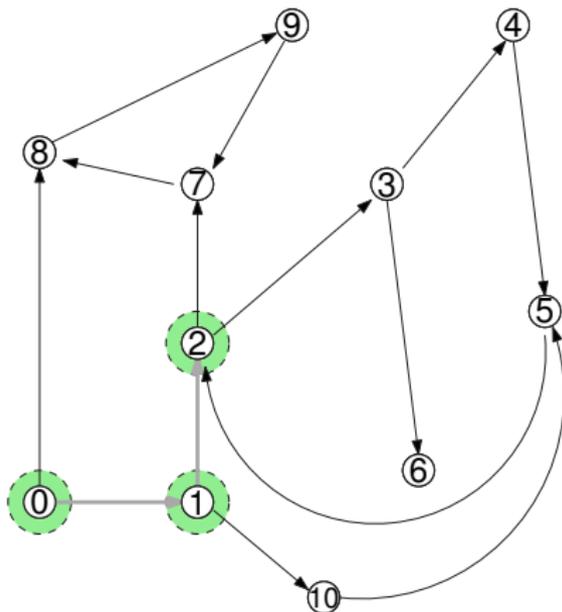
oNodes	oReps
0	0
1	1

SCC (Wiederholung)



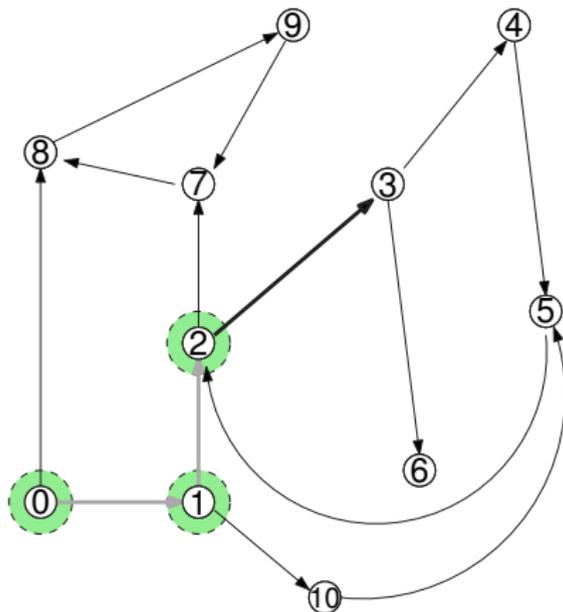
oNodes	oReps
0	0
1	1

SCC (Wiederholung)



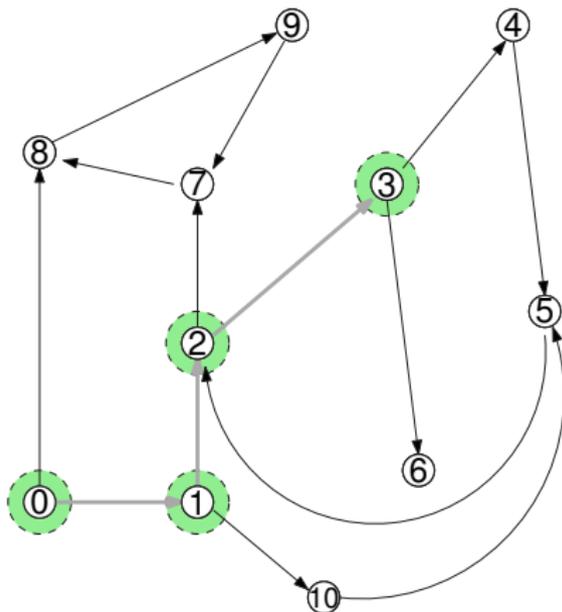
oNodes	oReps
0	0
1	1
2	2

SCC (Wiederholung)



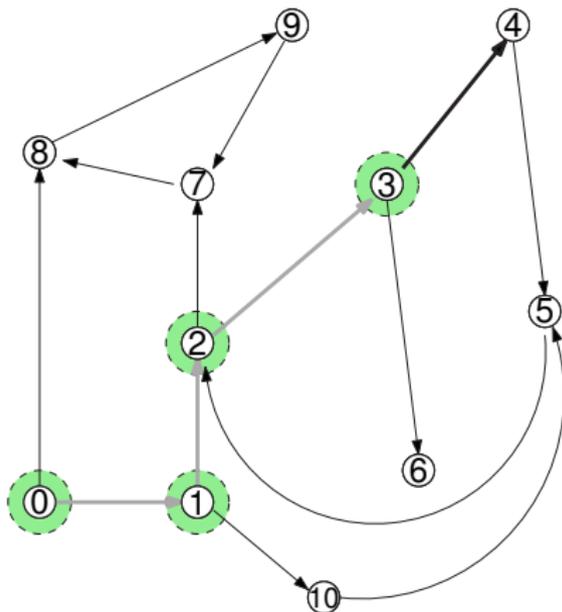
oNodes	oReps
0	0
1	1
2	2

SCC (Wiederholung)



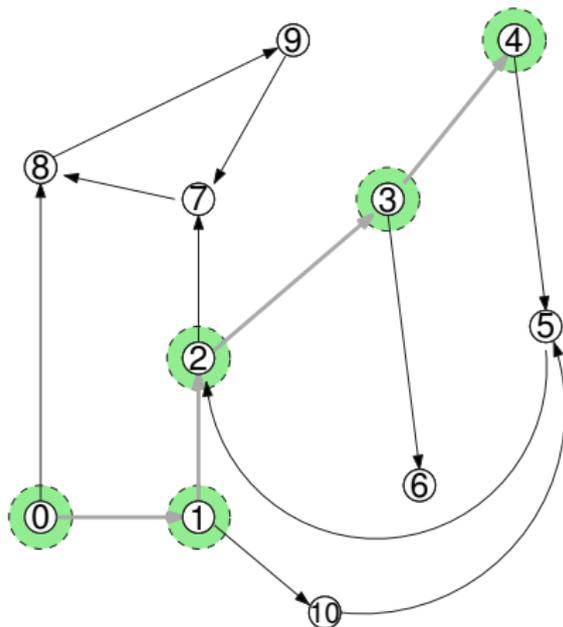
oNodes	oReps
0	0
1	1
2	2
3	3

SCC (Wiederholung)



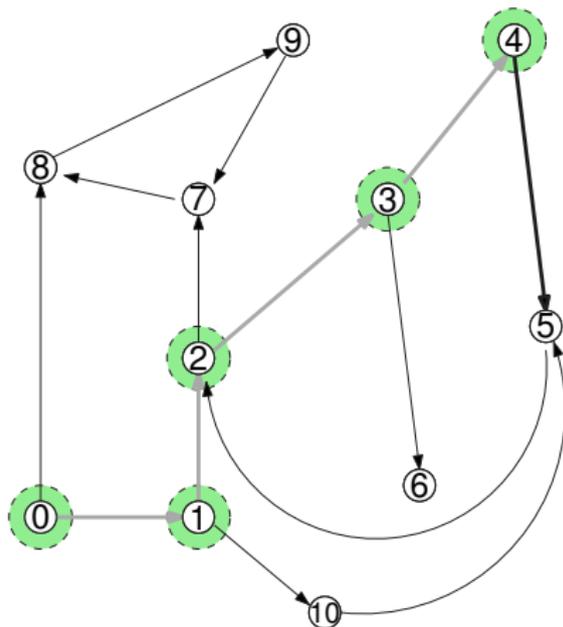
oNodes	oReps
0	0
1	1
2	2
3	3

SCC (Wiederholung)



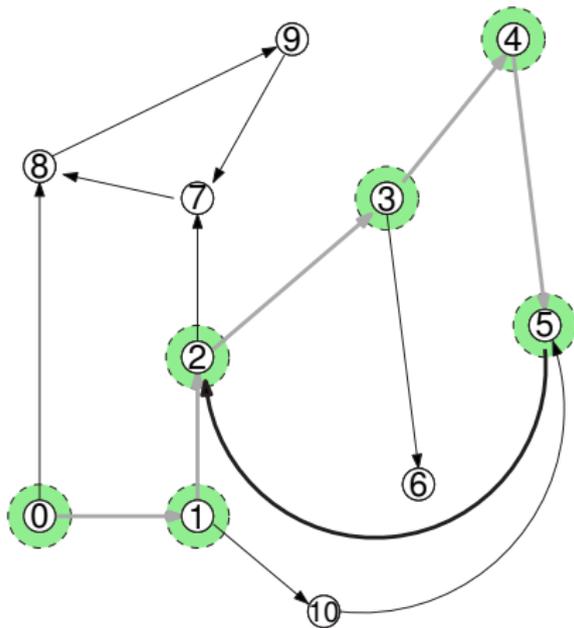
oNodes	oReps
0	0
1	1
2	2
3	3
4	4

SCC (Wiederholung)



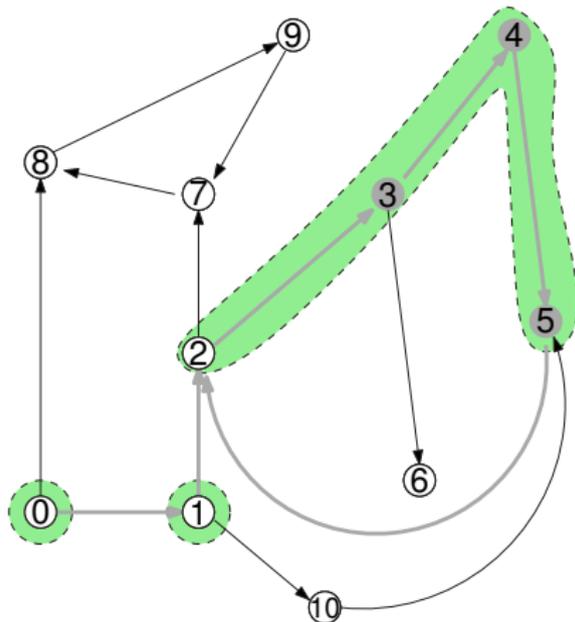
oNodes	oReps
0	0
1	1
2	2
3	3
4	4

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	3
4	4
5	5

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

Invariante 1

Kein Kanten von geschlossenen in offene Komponenten.

- Tiefensuche sucht Pfad durch Graphen
- Nur Rückwärtskanten ergeben Kreise
- Kreise vereinigen alle auf dem Kreis liegenden offenen Komponenten

Invariante 2

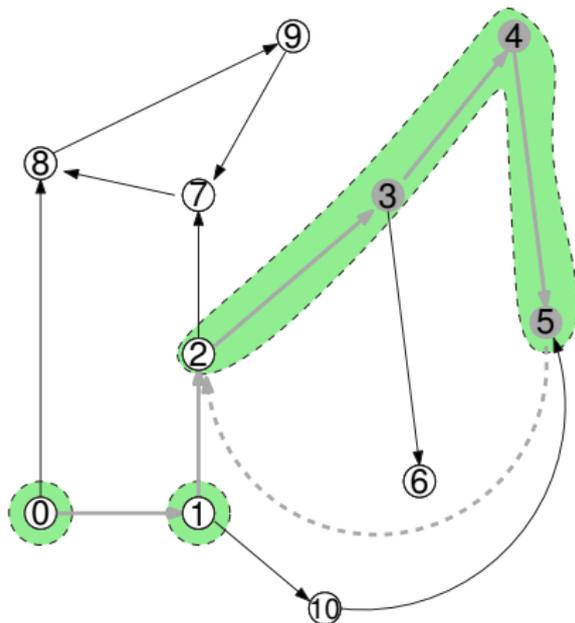
Offene Komponenten liegen auf Pfad.

- Repräsentant ist minimal auf Kreis

Invariante 3

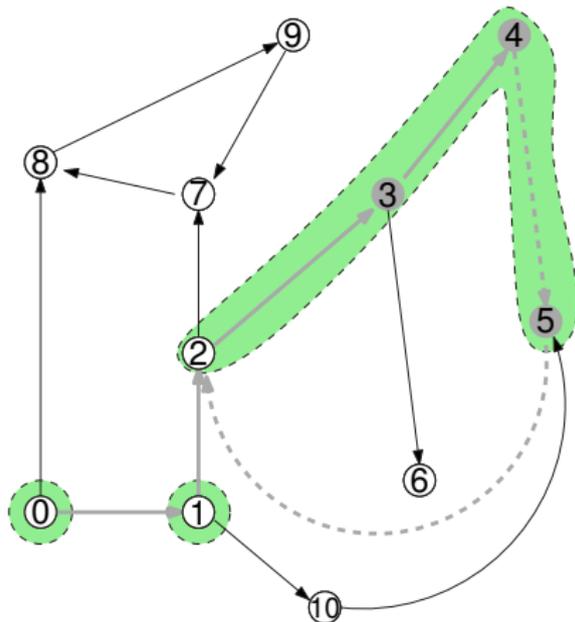
Repräsentanten partitionieren offene Komponenten bzgl. dfsNum.

SCC (Wiederholung)



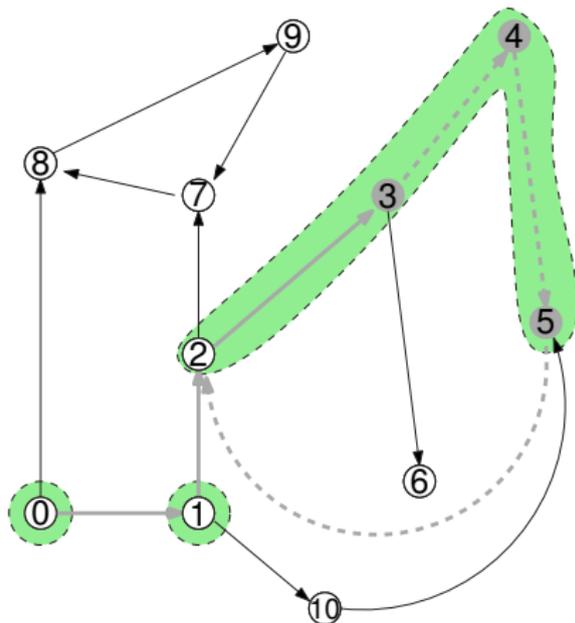
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



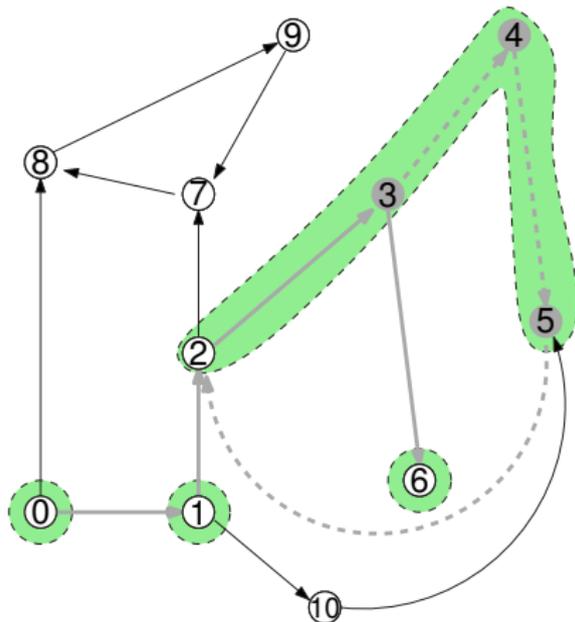
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



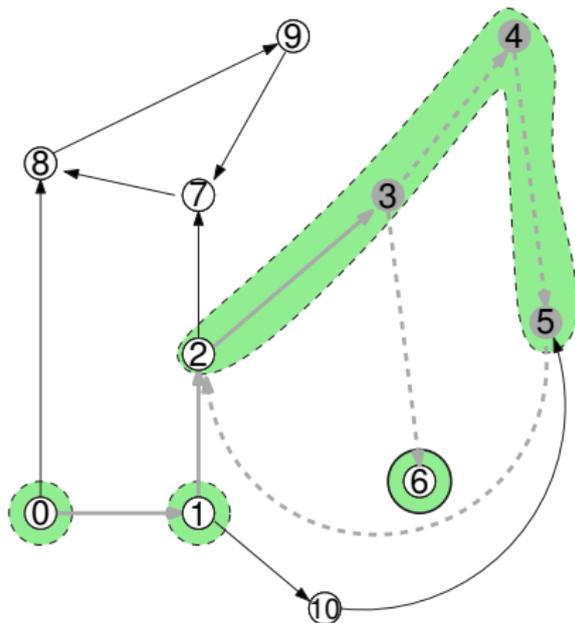
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
6	6

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
6	6

- Komponenten werden nach Bearbeitung aller ausgehenden Kanten geschlossen
- Alle offenen Komponenten liegen auf Stack
- Kante von geschlossener in offene Komponenten hätte bei Bearbeitung Kreis induziert

Invariante 1

Kein Kanten von geschlossenen in offene Komponenten.

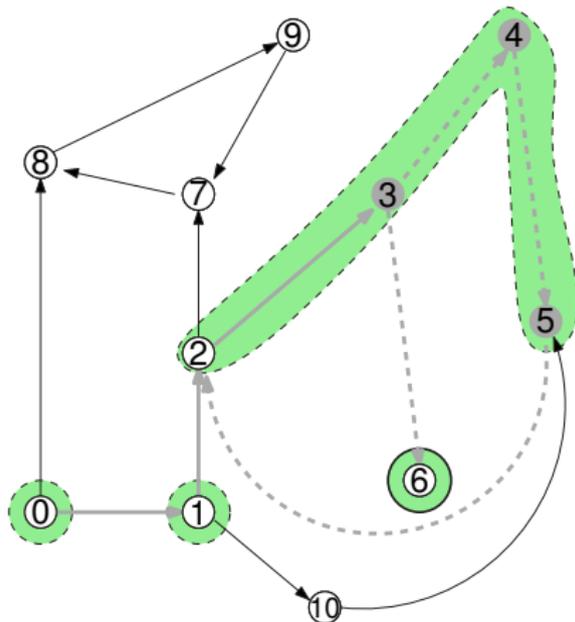
Invariante 2

Offene Komponenten liegen auf Pfad.

Invariante 3

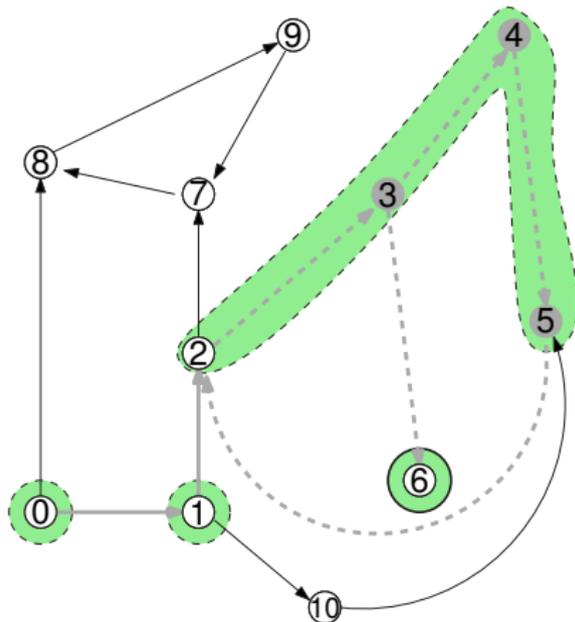
Repräsentanten partitionieren offene Komponenten bzgl. dfsNum.

SCC (Wiederholung)



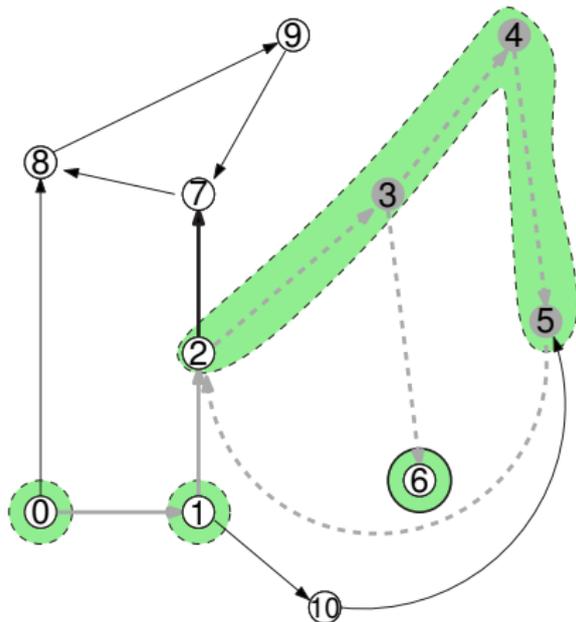
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



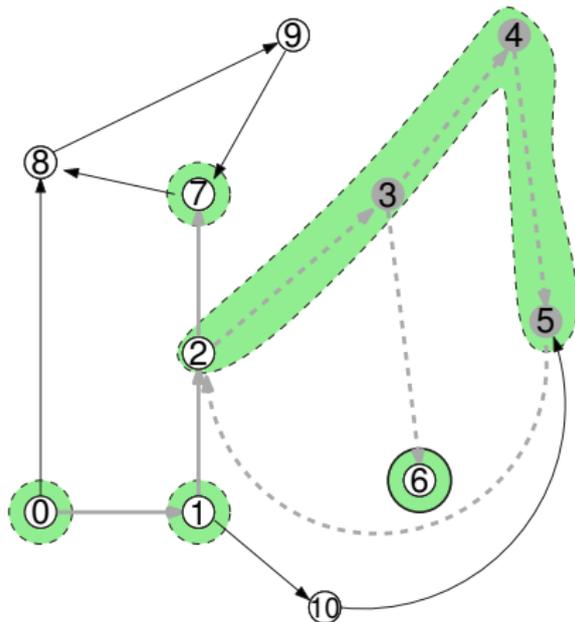
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



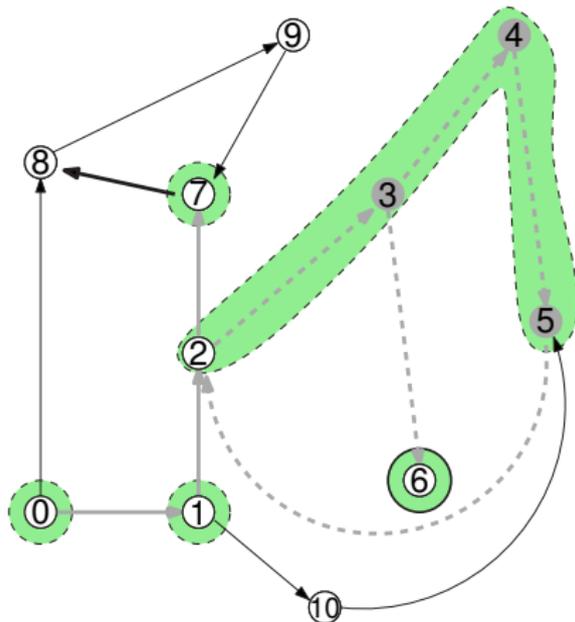
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

SCC (Wiederholung)



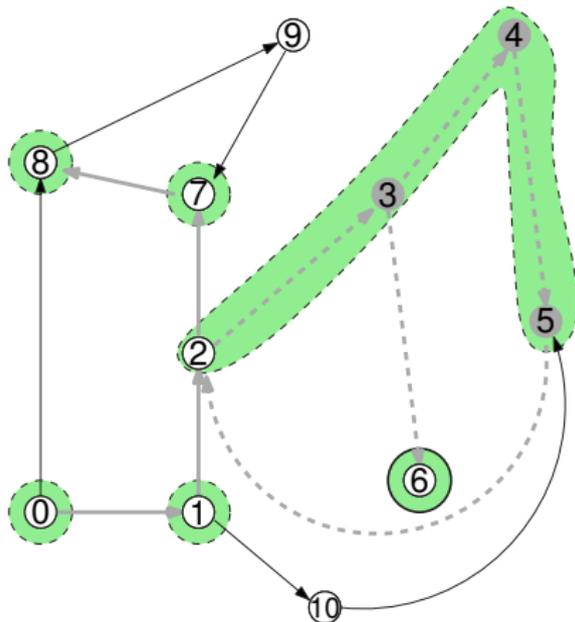
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7

SCC (Wiederholung)



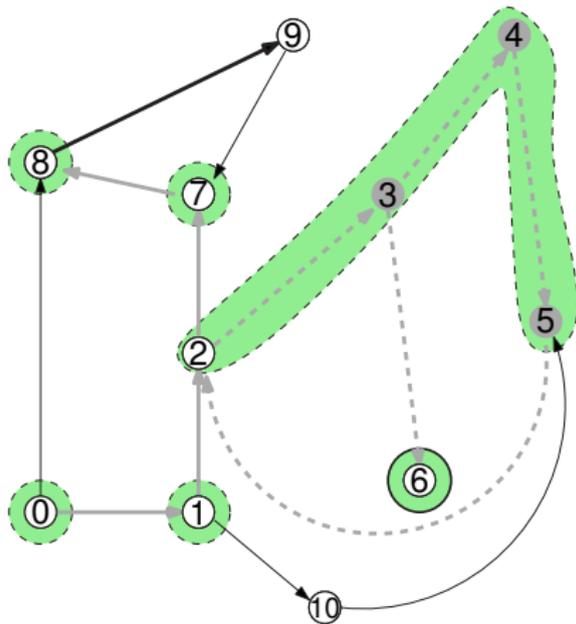
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7

SCC (Wiederholung)



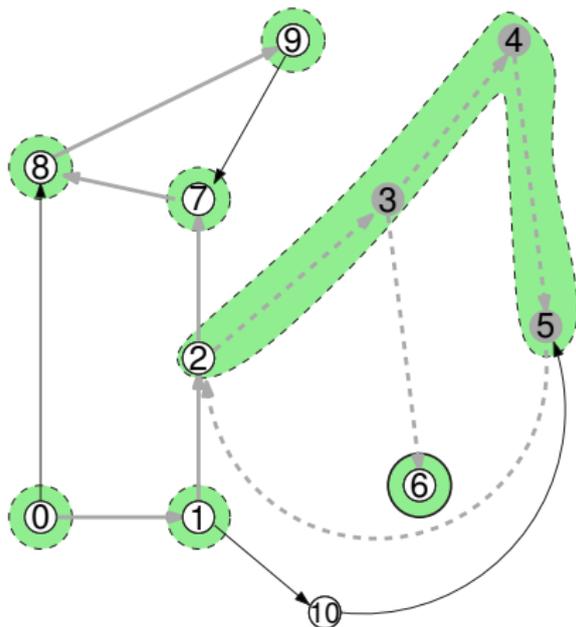
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	8

SCC (Wiederholung)



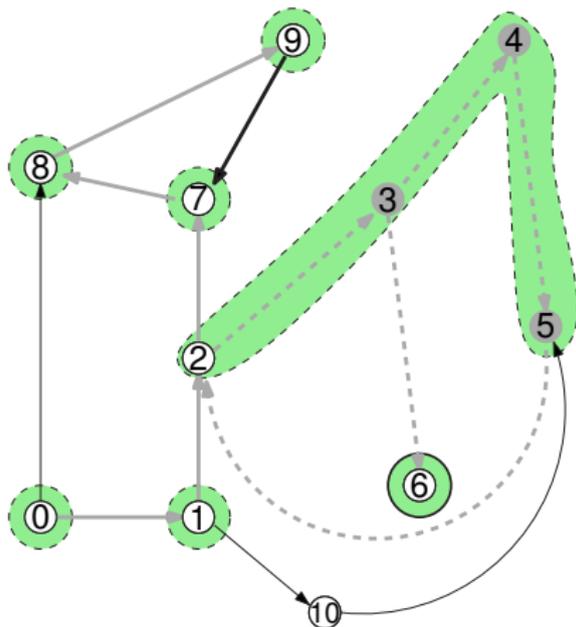
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	8

SCC (Wiederholung)



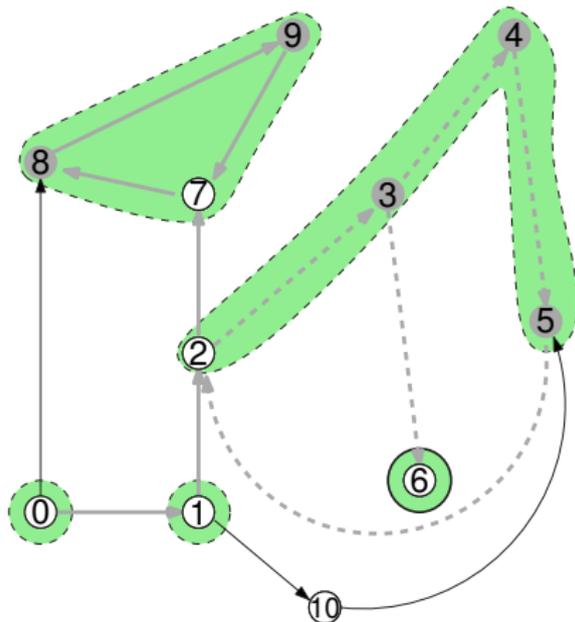
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	8
9	9

SCC (Wiederholung)



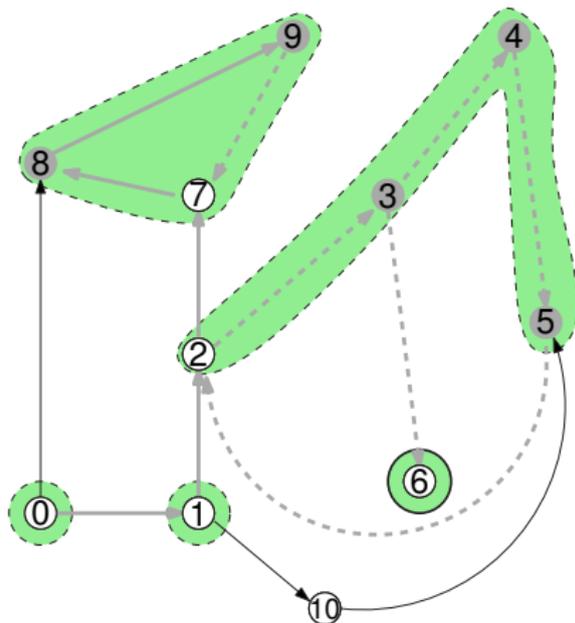
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	8
9	9

SCC (Wiederholung)



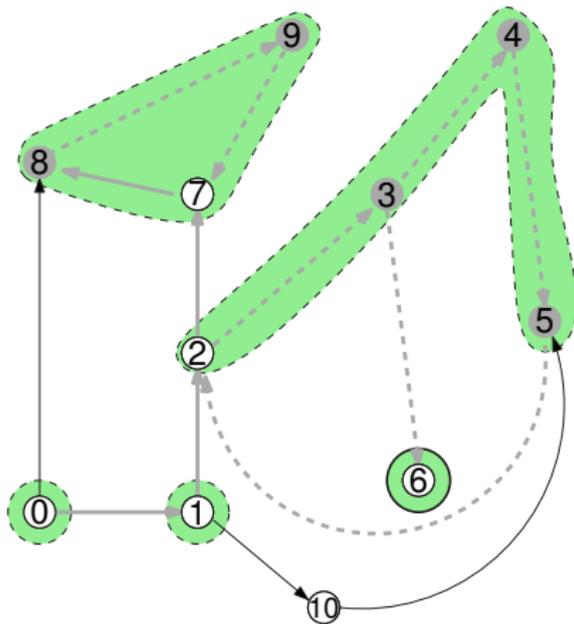
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	
9	

SCC (Wiederholung)



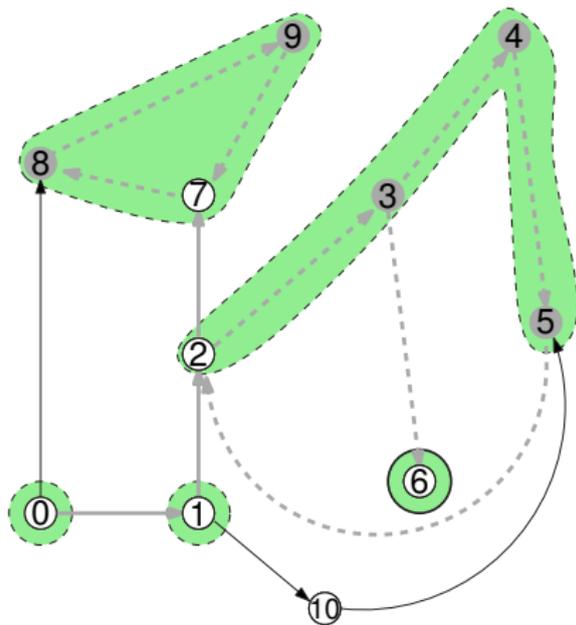
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	
9	

SCC (Wiederholung)



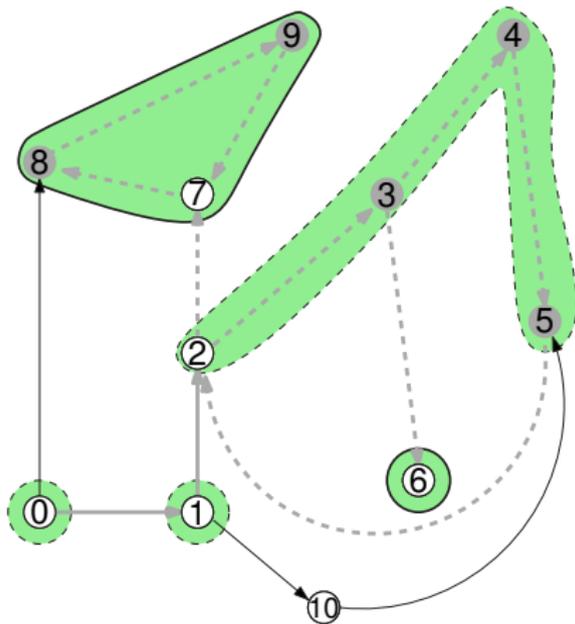
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	
9	

SCC (Wiederholung)



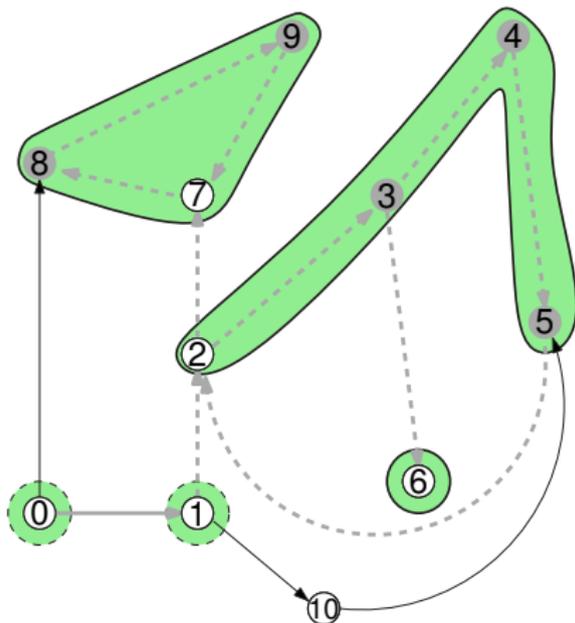
oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	
9	

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	
4	
5	
7	7
8	
9	

SCC (Wiederholung)



oNodes	oReps
0	0
1	1
2	2
3	
4	
5	

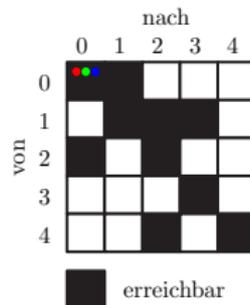
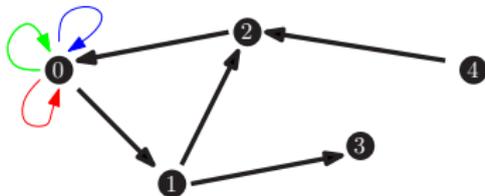
Floyd Warshall: SCC als Speedup Technik

Floyd Warshall

- kubischer Algorithmus
 - berechnet transitive Hülle
- auch anwendbar für *all-to-all* kürzeste Wege

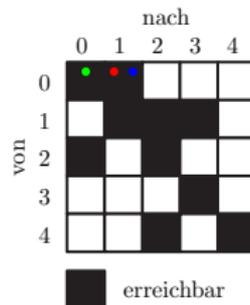
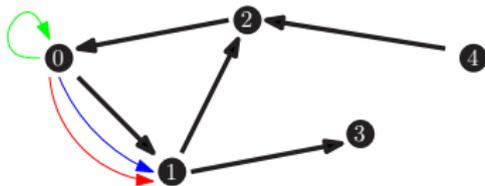
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



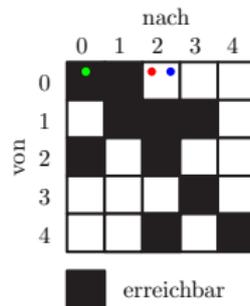
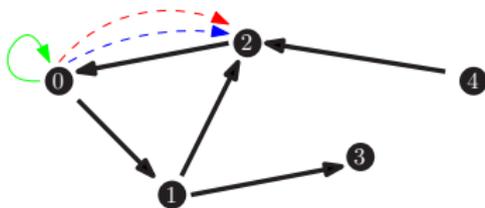
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



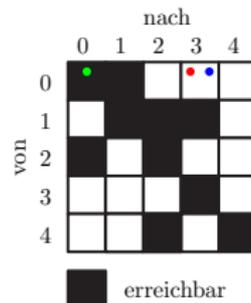
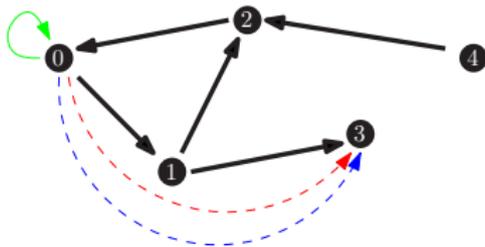
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



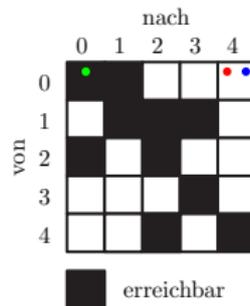
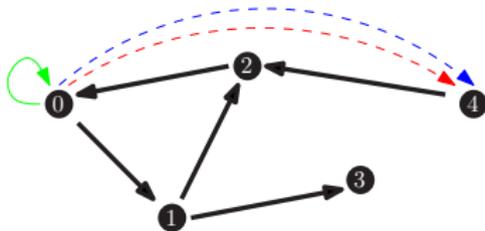
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



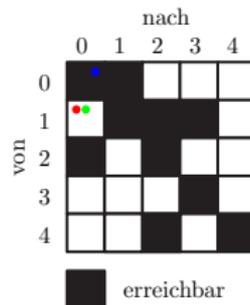
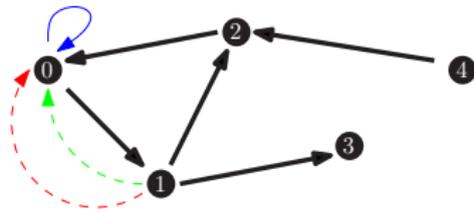
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



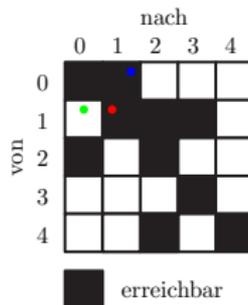
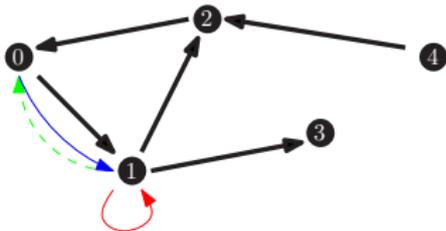
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



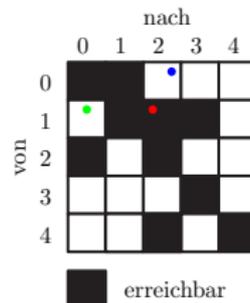
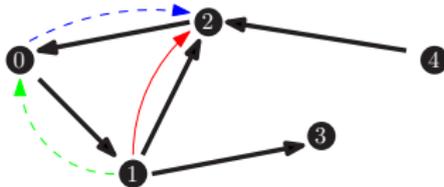
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



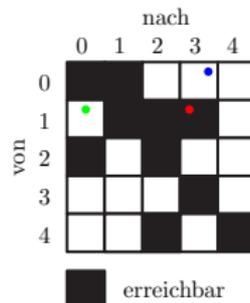
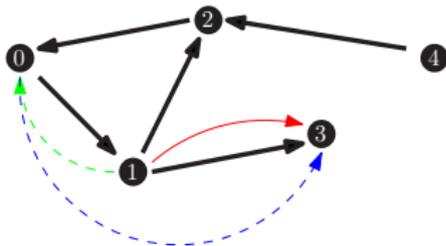
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



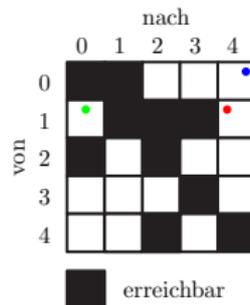
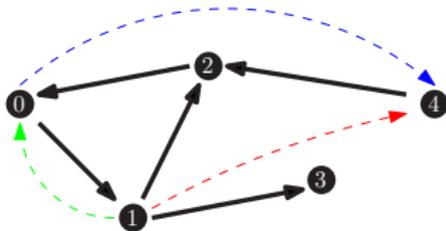
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



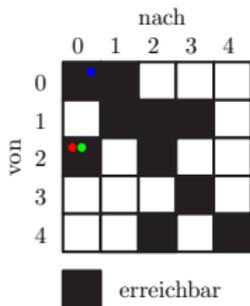
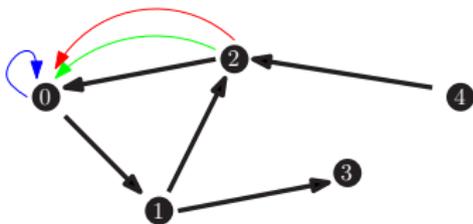
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



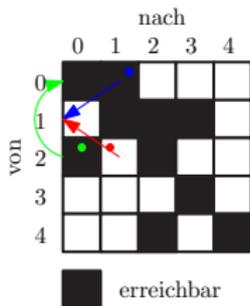
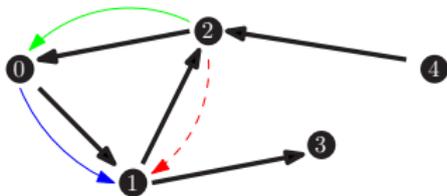
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



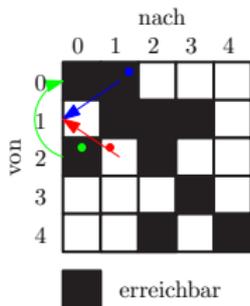
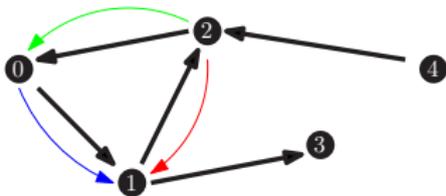
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



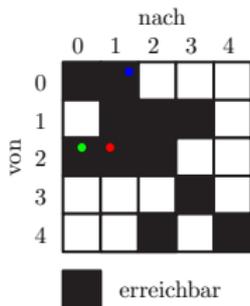
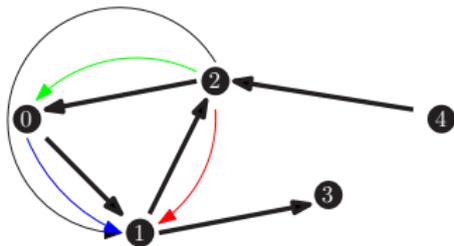
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



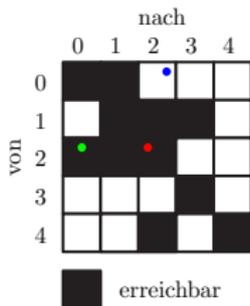
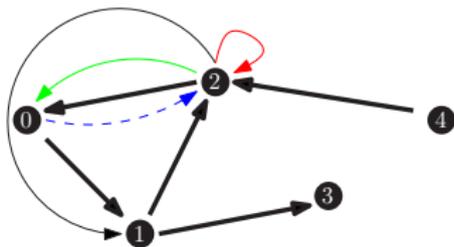
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



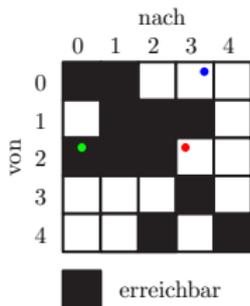
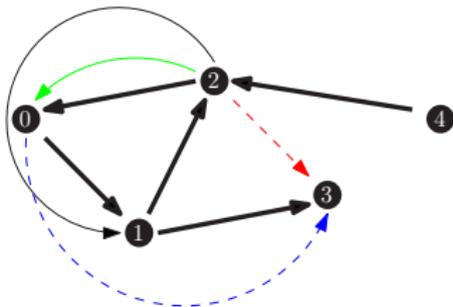
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



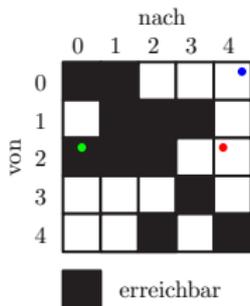
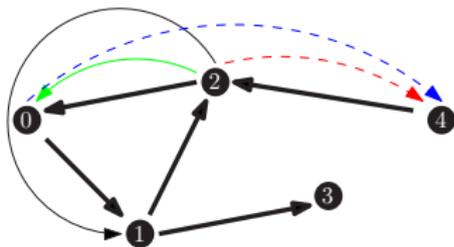
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



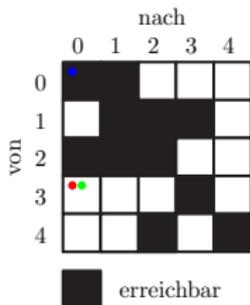
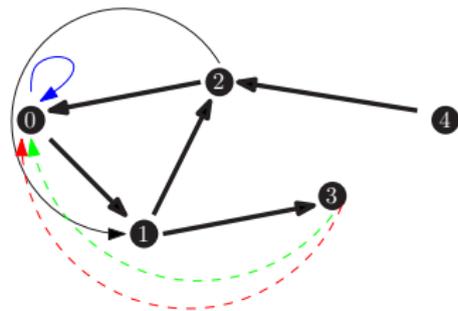
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall



```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

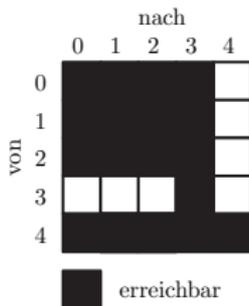
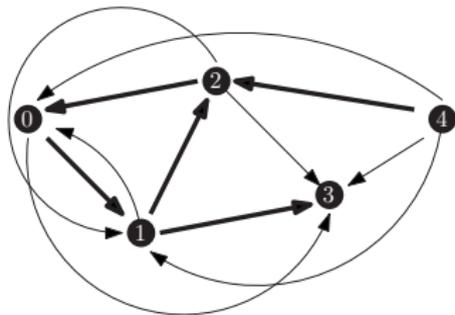
Floyd Warshall



```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

107 Iterationen später ...

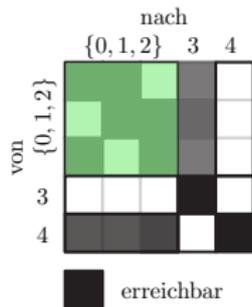
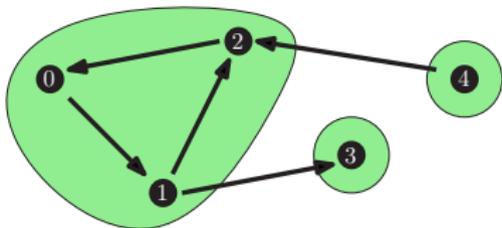
Floyd Warshall



```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

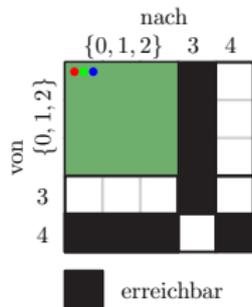
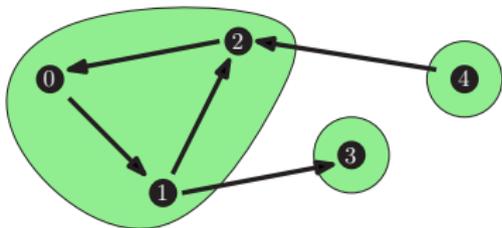
- Transitive Hülle einer SCC ist ein vollständiger Graph
- Betrachte Schrumpfggraphen: Floyd Warshall in $\#SCC^3$
- Deutlich Schneller falls $\#SCC < n$

Floyd Warshall und SCC



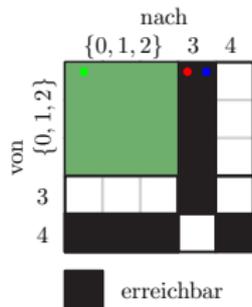
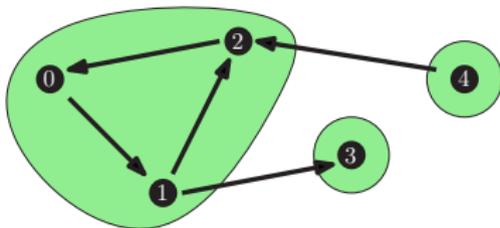
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



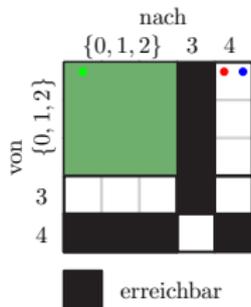
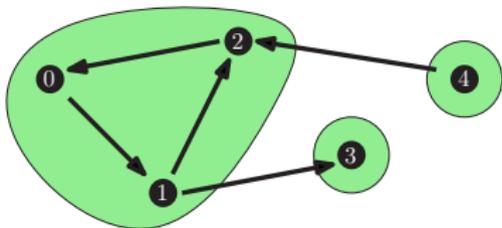
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



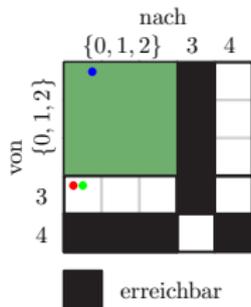
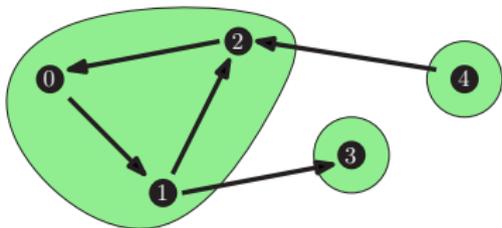
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



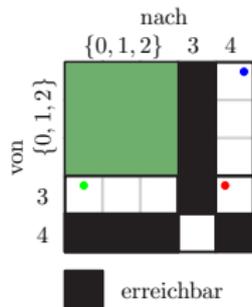
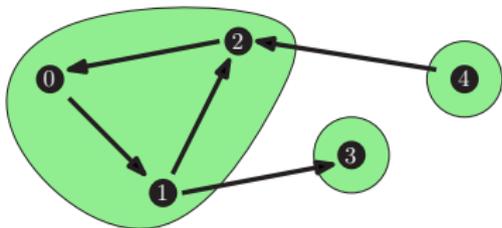
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



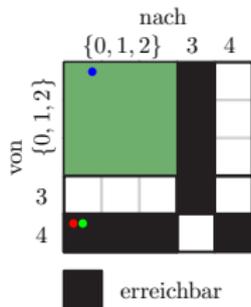
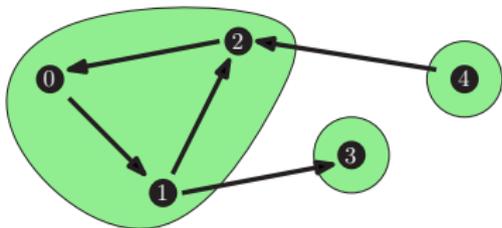
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```


Floyd Warshall und SCC



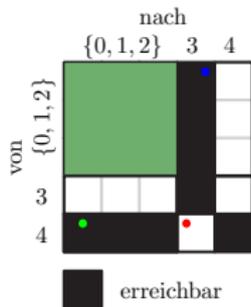
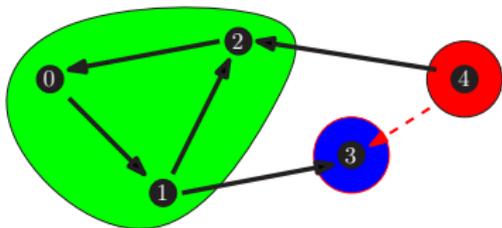
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



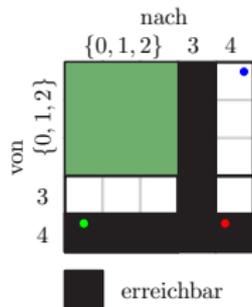
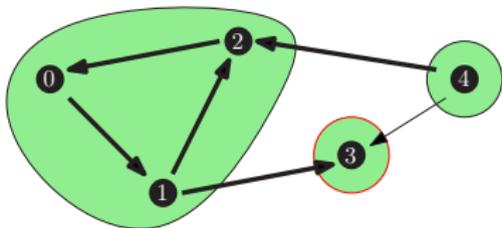
```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

Floyd Warshall und SCC



```
for int k = 0; k < n; ++k do
  for int i = 0; i < n; ++i do
    for int j = 0; j < n; ++j do
      array[i][j] = array[i][j] ||
        (array[i][k] && array[k][j]);
    end
  end
end
```

- SCC-Algorithmus nicht nur eleganter Algorithmus ...
- ... sondern auch Tool für bessere Algorithmen
- Linearzeit Algorithmus kann teurere Algorithmen deutlich beschleunigen

Ende!



Feierabend!