

Übersicht

- 3 Graphalgorithmen
 - Darstellung von Graphen
 - Tiefensuche
 - Starke Komponenten
 - **Topologisches Sortieren**
 - Kürzeste Pfade
 - Netzwerkalgorithmen
 - Minimale Spannbäume

Topologisches Sortieren

Ein reflexiv-transitive Hüllen eines DAG entspricht einer Halbordnung.

Definition

Es sei M eine Menge. Wir sagen $\leq \subseteq M \times M$ ist eine **Halbordnung** auf M , wenn

- 1 Für alle $x \in M$ gilt $x \leq x$
- 2 Für alle $x, y \in M$ gilt $x \leq y \wedge y \leq x \Rightarrow x = y$
- 3 Für alle $x, y, z \in M$ gilt $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Topologisches Sortieren:

Bette eine Halbordnung in eine Ordnung ein.

Ordne die Knoten eines DAG so, daß keine Kante von einem größeren zu einem kleineren Knoten führt.

Topologisches Sortieren – Anwendungen

Topologisches Sortieren kann viele Probleme lösen.

- In welcher Reihenfolge sollen Vorlesungen gehört werden?
- Wie baut man ein kompliziertes Gerät?
- Entwurf von Klassenhierarchien.
- Task scheduling.
- Tabellenkalkulation.
- Optimierung beim Compilieren.
- ...

Frage: Kann jede Halbordnung topologisch sortiert werden?

Topologisches Sortieren

Theorem

Jede endliche Halbordnung kann in eine (totale) Ordnung eingebettet werden.

Beweis.

Sei $\leq \subseteq M \times M$ eine endliche Halbordnung auf M .

Da M endlich ist, dann gibt es ein $x \in M$, so daß es kein $y \in M$ mit $y \neq x$ und $y \leq x$ gibt.

Wir können die Ordnung mit x als kleinstem Element beginnen und dann eine Ordnung von $M \setminus \{x\}$ anfügen.

Durch Induktion sieht man, daß dies eine (totale) Ordnung auf M ist. □

Frage: Was ist mit unendlichen Mengen?

Topologisches Sortieren

Theorem

Jede endliche Halbordnung kann in eine (totale) Ordnung eingebettet werden.

Beweis.

Sei $\leq \subseteq M \times M$ eine endliche Halbordnung auf M .

Da M endlich ist, dann gibt es ein $x \in M$, so daß es kein $y \in M$ mit $y \neq x$ und $y \leq x$ gibt.

Wir können die Ordnung mit x als kleinstem Element beginnen und dann eine Ordnung von $M \setminus \{x\}$ anfügen.

Durch Induktion sieht man, daß dies eine (totale) Ordnung auf M ist. □

Frage: Was ist mit unendlichen Mengen?

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$

Daher kommt u nach v in der konstruierten Ordnung.

Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u .

Nur möglich, wenn keine Kante von u nach v . □

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$

Daher kommt u nach v in der konstruierten Ordnung.

Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u .

Nur möglich, wenn keine Kante von u nach v . □

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$
Daher kommt u nach v in der konstruierten Ordnung.

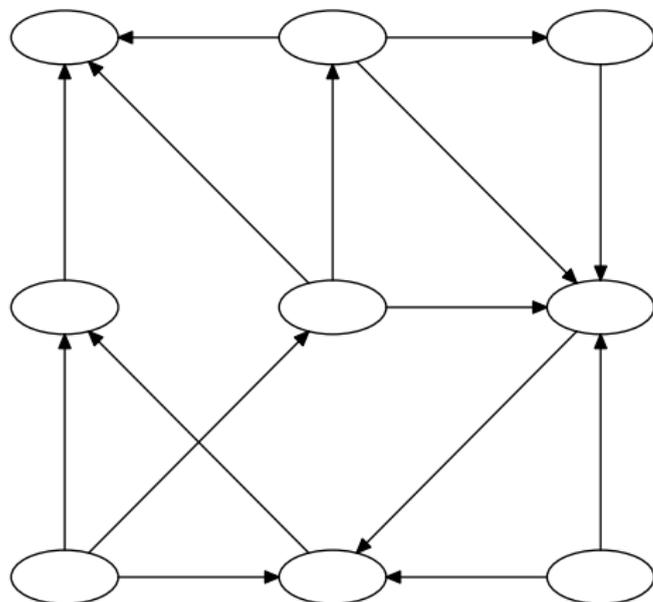
Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u .

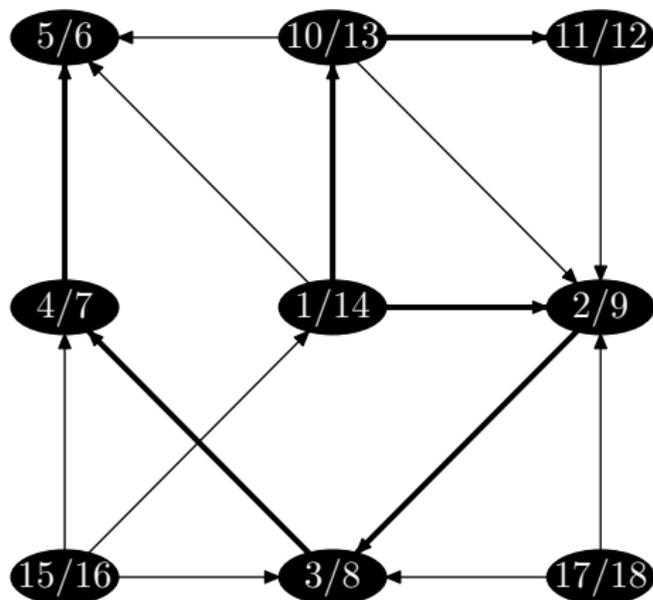
Nur möglich, wenn keine Kante von u nach v . □

Topologisches Sortieren – Beispiel



Topologisches Sortieren in $O(|V| + |E|)$ Schritten.

Topologisches Sortieren – Beispiel



Topologisches Sortieren in $O(|V| + |E|)$ Schritten.

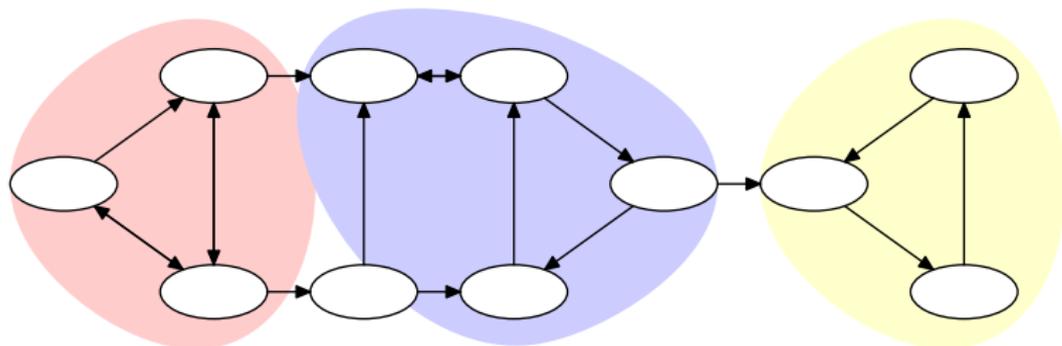
Übersicht

- 3 Graphalgorithmen
 - Darstellung von Graphen
 - Tiefensuche
 - Starke Komponenten
 - Topologisches Sortieren
 - **Kürzeste Pfade**
 - Netzwerkalgorithmen
 - Minimale Spannbäume

s - t Connectivity

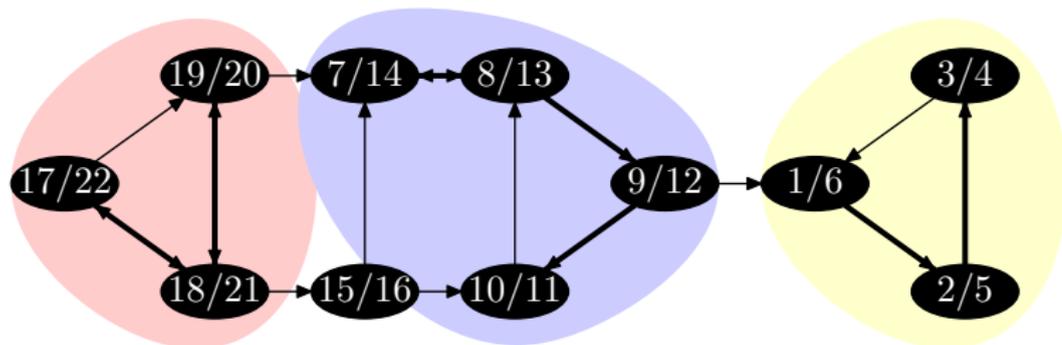
Gegeben: Knoten s und t in gerichtetem Graph

Frage: Ist t von s erreichbar (durch einen gerichteten Pfad)?



s - t Connectivity

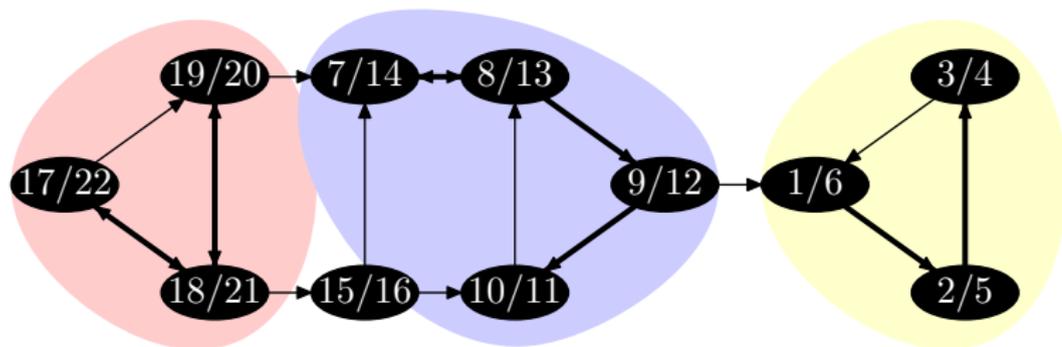
Führe eine DFS aus, starte bei s .



s - t Connectivity

Führe eine DFS aus, starte bei s .

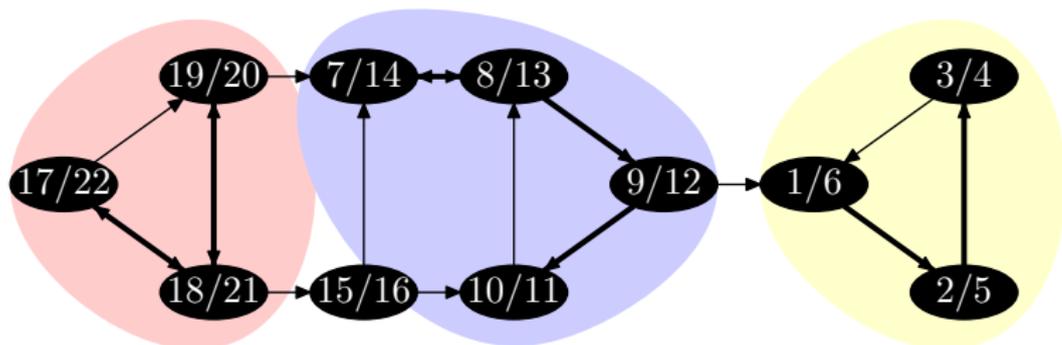
Pfad von s nach $t \iff f(t) < f(s)$.



s - t Connectivity

Führe eine DFS aus, starte bei s .

Pfad von s nach $t \iff f(t) < f(s)$.

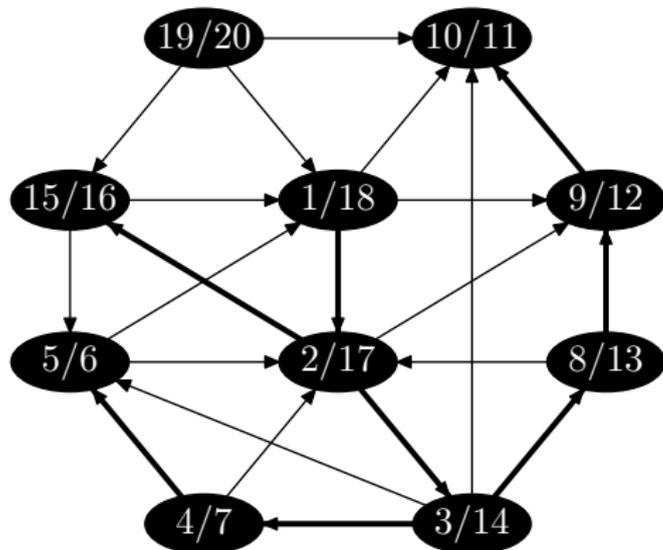


Beweis: Wenn s schwarz wird, sind alle von s erreichbaren Knoten schwarz.

s-t Connectivity

Theorem

Gegeben sei ein gerichteter Graph $G = (V, E)$ und $s \in V$.
Wir können in linearer Zeit alle von s erreichbaren Knoten finden.



Single Source Shortest Paths

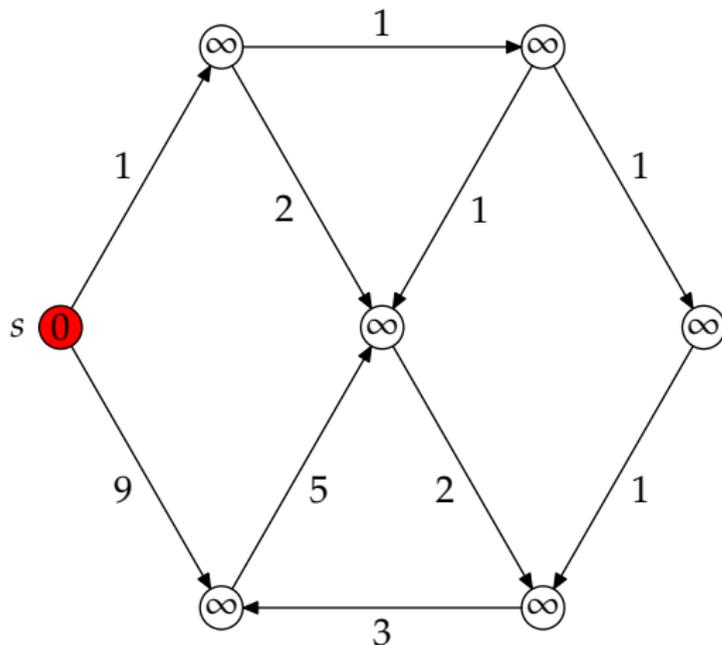
Gegeben ein gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $length : E \rightarrow \mathbf{Q}$ und ein Knoten $s \in V$, finde die kürzesten Wege von s zu allen Knoten.

- Wir lösen das Problem mit dynamischer Programmierung.
- Menge F von Knoten, deren Abstand bekannt ist.
- Anfangs ist $F = \{s\}$.
- F wird in jeder Iteration größer.
- Invariante: Kein Knoten $v \notin F$ hat kleineren Abstand zu s als jeder Knoten in F .

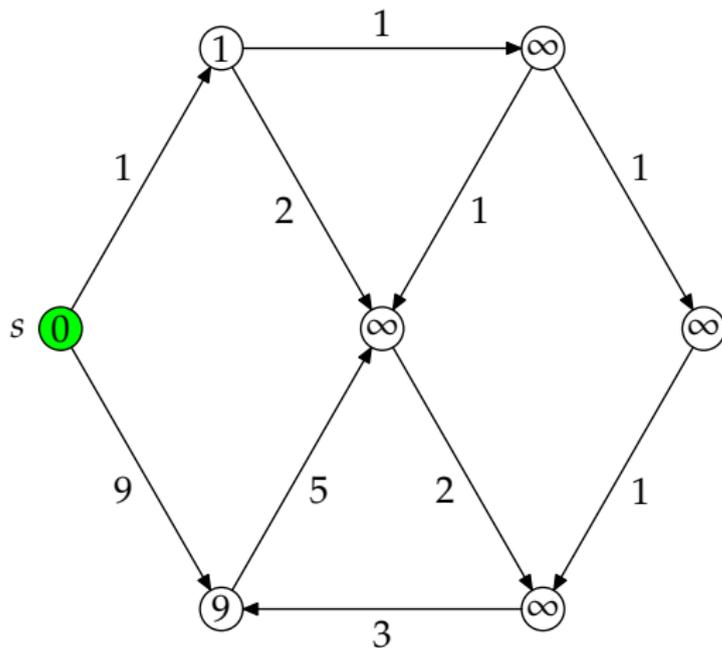
Single Source Shortest Paths

Gegeben ein gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $length : E \rightarrow \mathbf{Q}$ und ein Knoten $s \in V$, finde die kürzesten Wege von s zu allen Knoten.

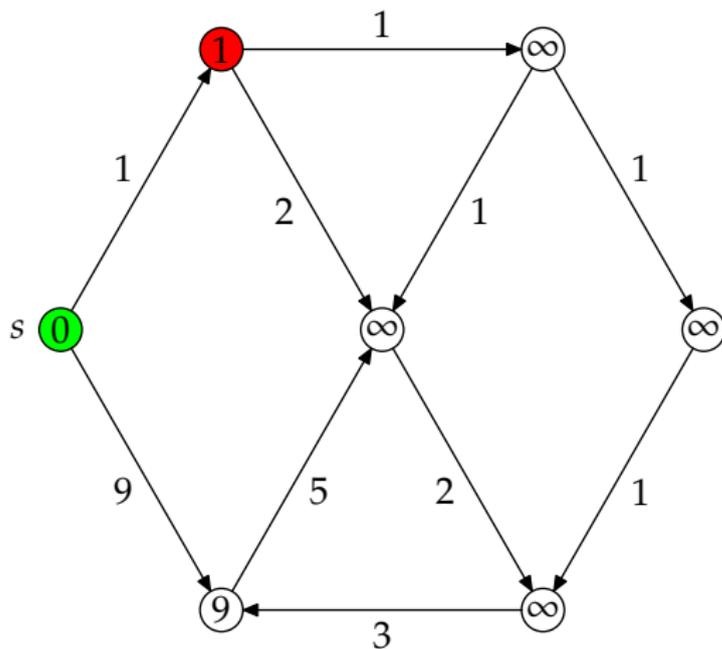
- Wir lösen das Problem mit dynamischer Programmierung.
- Menge F von Knoten, deren Abstand bekannt ist.
- Anfangs ist $F = \{s\}$.
- F wird in jeder Iteration größer.
- Invariante: Kein Knoten $v \notin F$ hat kleineren Abstand zu s als jeder Knoten in F .



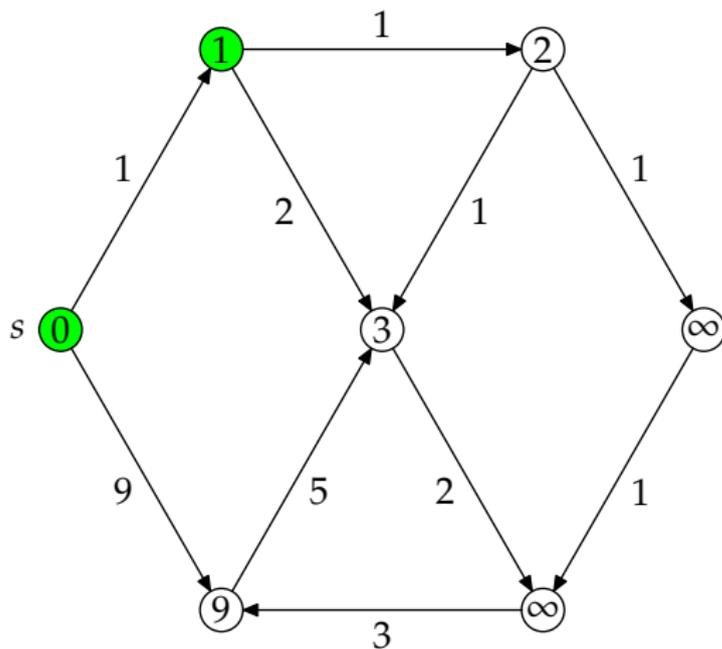
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



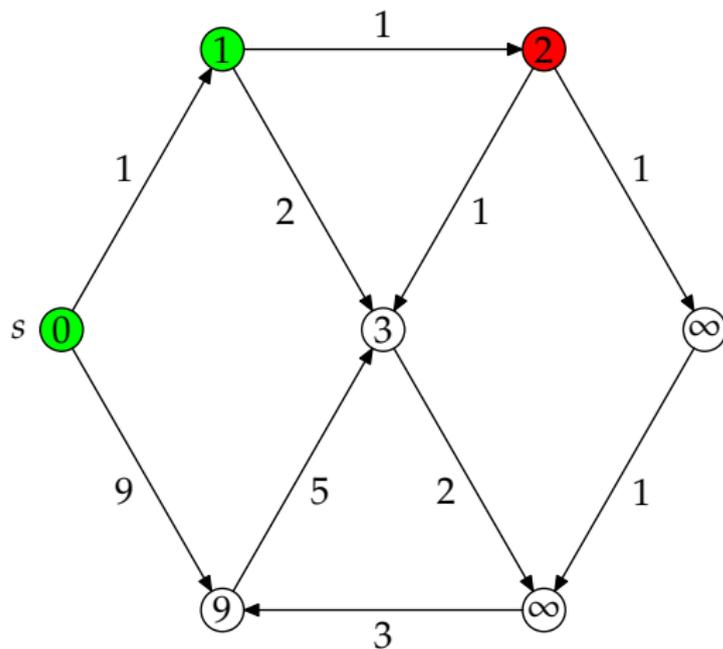
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



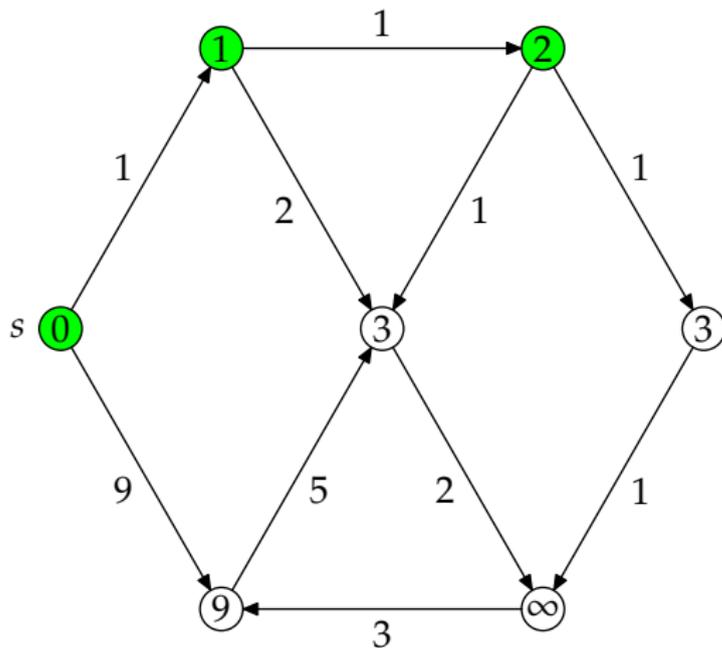
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



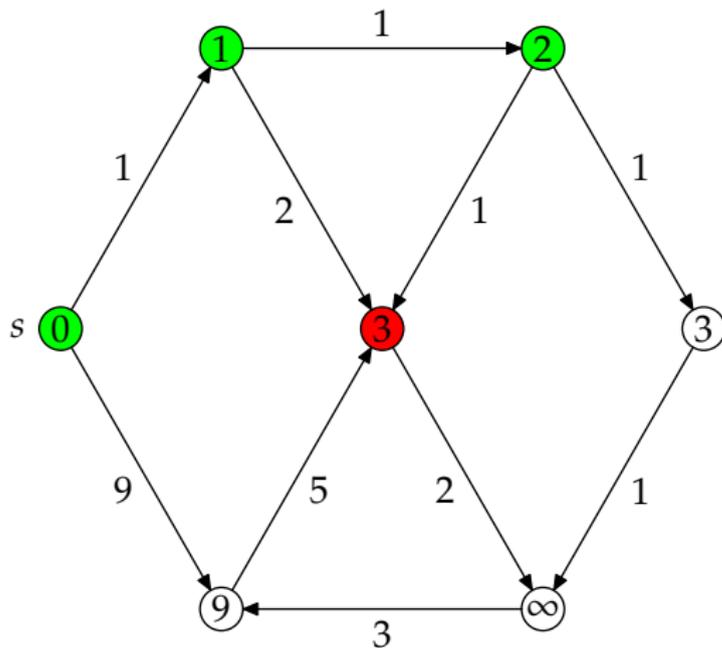
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



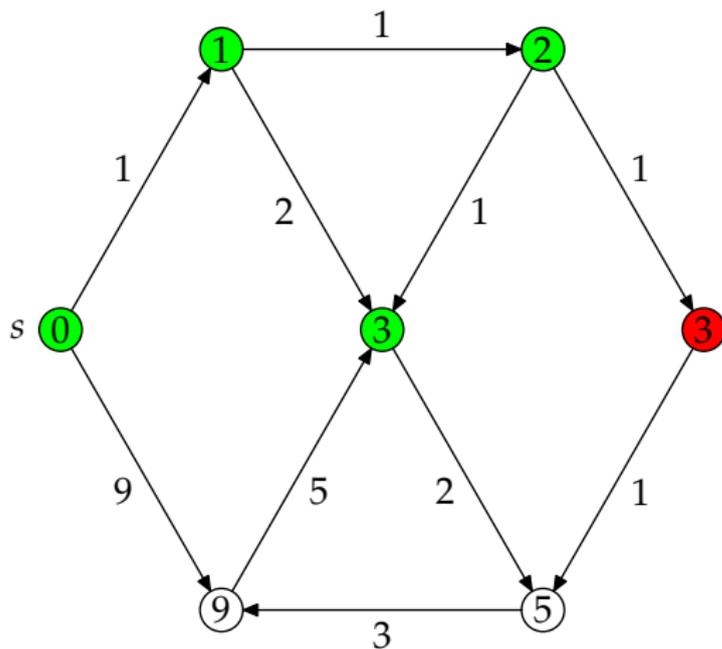
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



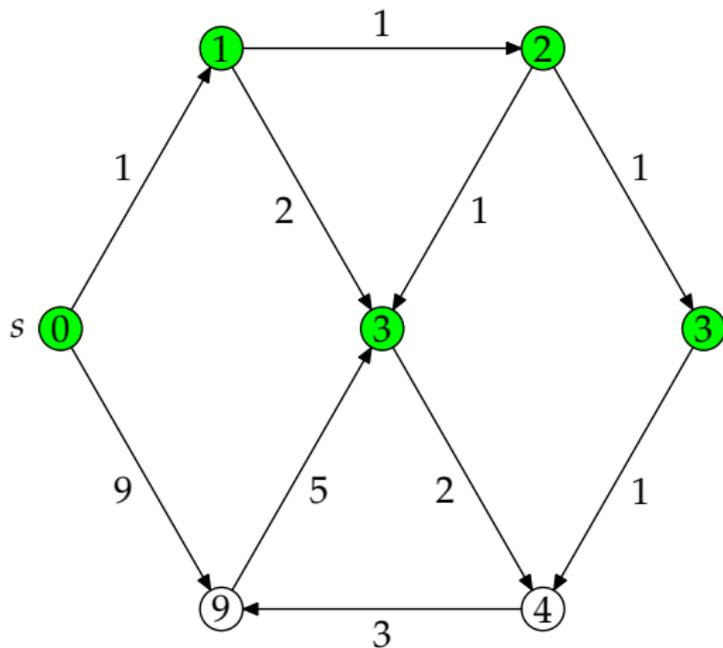
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



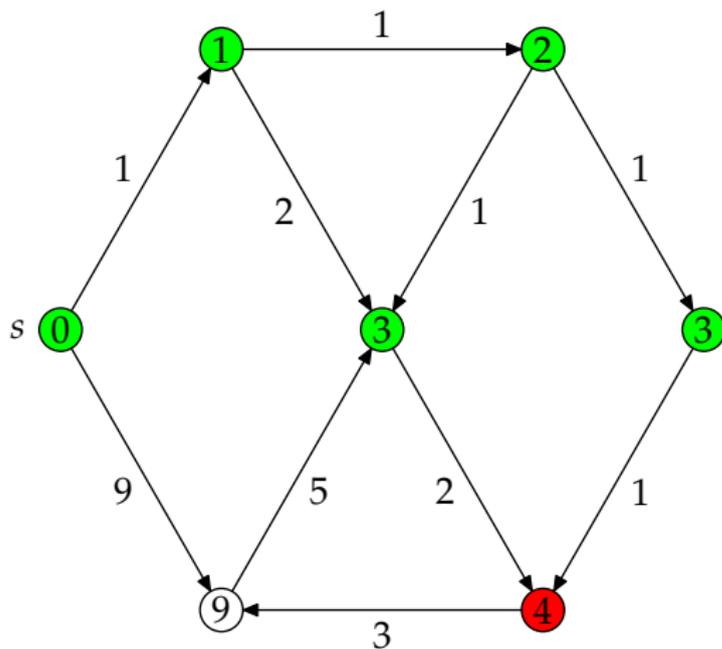
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



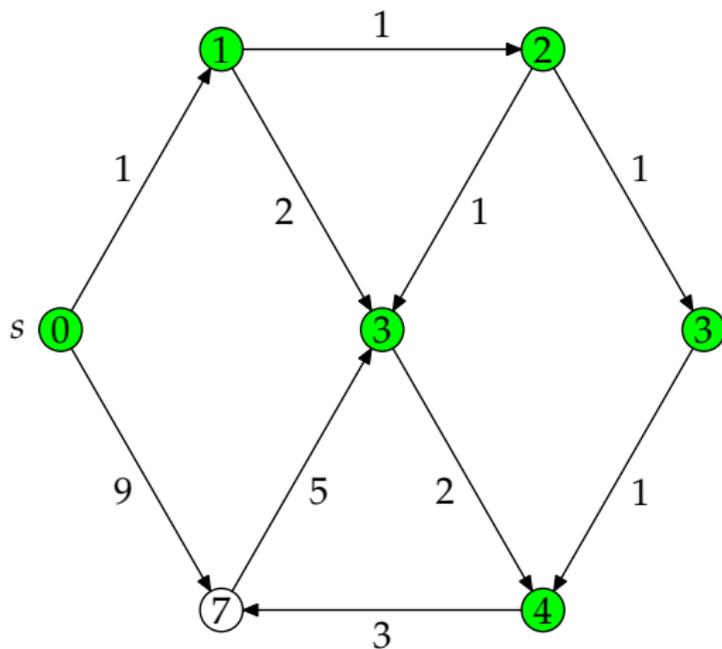
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



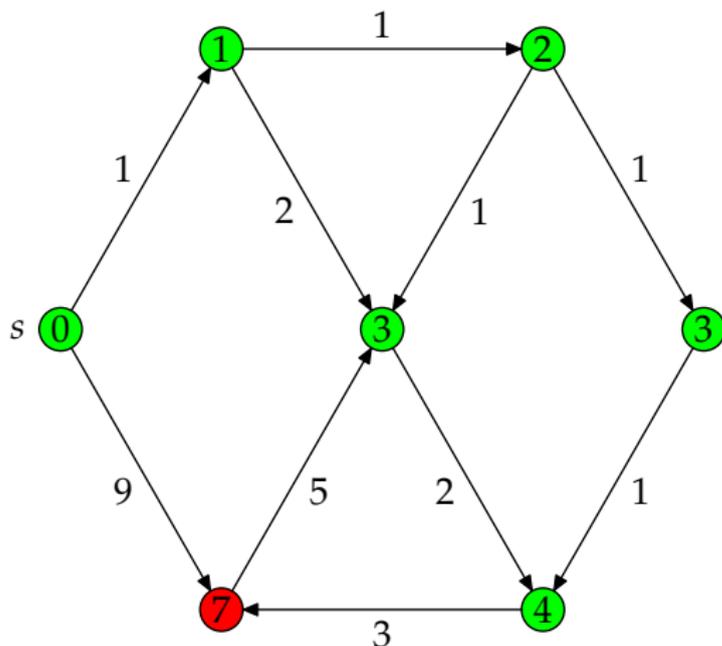
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



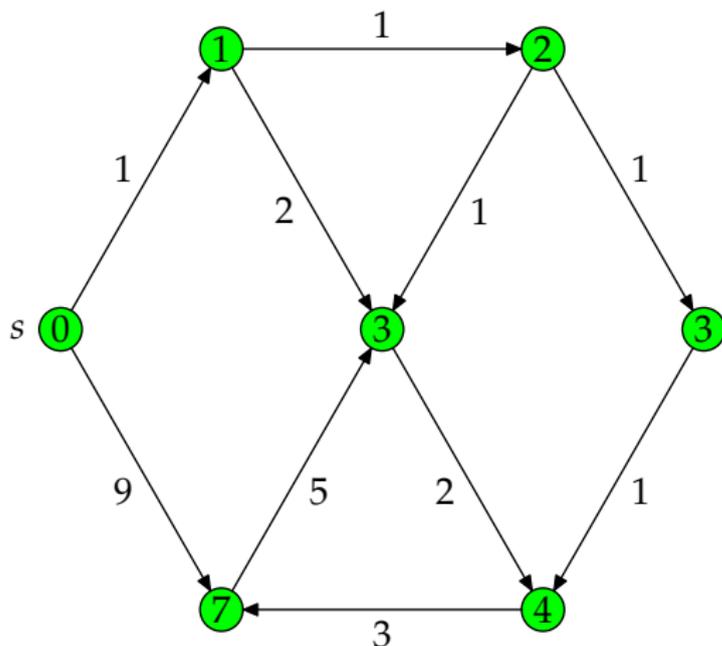
- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, **relaxiert** seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.

Korrektheit

Lemma

- *Jeder grüne Knoten enthält den Abstand von s .*
- *Jeder weiße Knoten enthält Abstand von s über grüne Knoten.*

Beweis.

Sei v einer weißer Knoten, dessen Beschriftung minimal ist. Betrachte einen kürzesten Pfad von s nach v und den ersten weißen Knoten u auf diesem Pfad.

Es gilt $u = v$, da der Abstand von s zu u mindestens so groß ist wie der Abstand von s zu v .

Wenn ein Knoten grün wird, garantiert die Relaxation, daß die zweite Bedingung weiter gilt. □

Korrektheit

Lemma

- *Jeder grüne Knoten enthält den Abstand von s .*
- *Jeder weiße Knoten enthält Abstand von s über grüne Knoten.*

Beweis.

Sei v einer weißer Knoten, dessen Beschriftung minimal ist. Betrachte einen kürzesten Pfad von s nach v und den ersten weißen Knoten u auf diesem Pfad.

Es gilt $u = v$, da der Abstand von s zu u mindestens so groß ist wie der Abstand von s zu v .

Wenn ein Knoten grün wird, garantiert die Relaxation, daß die zweite Bedingung weiter gilt. □

Der Algorithmus von Dijkstra

Algorithmus

procedure Dijkstra(s) :

$Q := V - \{s\};$

for v in Q **do** $d[v] := \text{infty}$ **od**;

$d[s] := 0;$

while $Q \neq \text{emptyset}$ **do**

 choose v in Q with minimal $d[v]$;

$Q := Q - \{v\};$

forall u adjacent **to** v **do**

$d[u] := \min \{ d[u], d[v] + \text{length}(v, u) \}$

od

od

Wie implementieren wir Q?

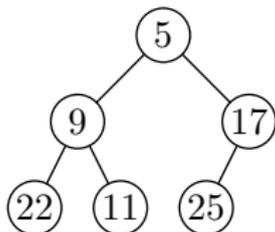
Der Algorithmus von Dijkstra – Beispiel



Priority Queues (Prioritätswarteschlangen)

Operationen einer **Prioritätswarteschlange** Q :

- 1 Einfügen von x mit Gewicht w (insert)
- 2 Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
- 3 Das Gewicht eines Elements x auf w verringern (decrease-weight)

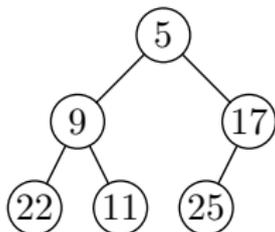


Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Priority Queues (Prioritätswarteschlangen)

Operationen einer **Prioritätswarteschlange** Q :

- 1 Einfügen von x mit Gewicht w (insert)
- 2 Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
- 3 Das Gewicht eines Elements x auf w verringern (decrease-weight)



Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ Schritten.

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt.

Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein **Fibonacci-Heap** benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ Schritten.

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt.

Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein **Fibonacci-Heap** benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ Schritten.

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt.

Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein **Fibonacci-Heap** benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

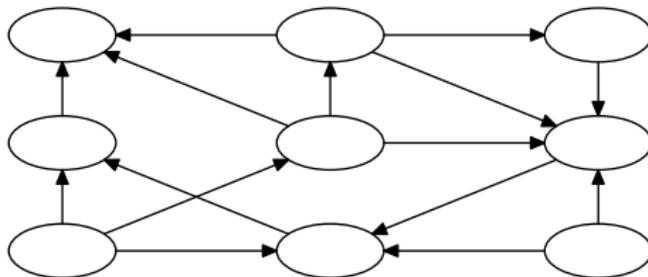
```
public Map<V, Double> dijkstra(V s, Map<Edge<V>, Double> length,
                               Map<V, V> pred) {
    if(pred == null) pred = new HashMap<V, V>();
    Map<V, Double> dist = new HashMap<V, Double>();
    PriorityQueue<V, Double> queue = new SplayPriorityQueue<>();
    for(V u : allNodes()) dist.put(u, Double.MAX_VALUE);
    dist.put(s, 0.0); queue.insert(s, 0.0);
    while(!queue.isEmpty()) {
        V u = queue.extractMin();
        for(V v : neighbors(u)) {
            Double l = length.get(new Edge<V>(u, v));
            if(l == null) continue;
            double d = dist.get(u) + l;
            if(d < dist.get(v)) {
                queue.decreaseKey(v, d); dist.put(v, d); pred.put(v, u); }
        }
    }
    return dist;
}
```

Spezialfall DAG

Können wir kürzeste Pfade in DAGs schneller finden?

Theorem

Kürzeste Pfade von einem Knoten s in einem DAG können in linearer Zeit gefunden werden.

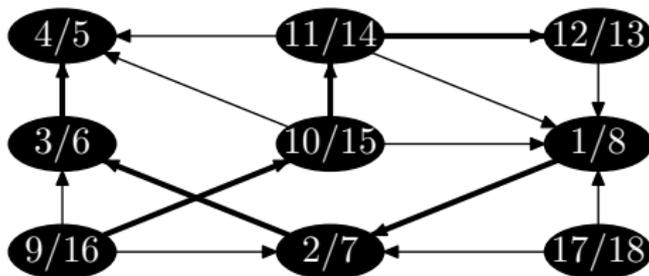


Spezialfall DAG

Können wir kürzeste Pfade in DAGs schneller finden?

Theorem

Kürzeste Pfade von einem Knoten s in einem DAG können in linearer Zeit gefunden werden.



Spezialfall DAG

Können wir kürzeste Pfade in DAGs schneller finden?

Theorem

Kürzeste Pfade von einem Knoten s in einem DAG können in linearer Zeit gefunden werden.

Beweis.

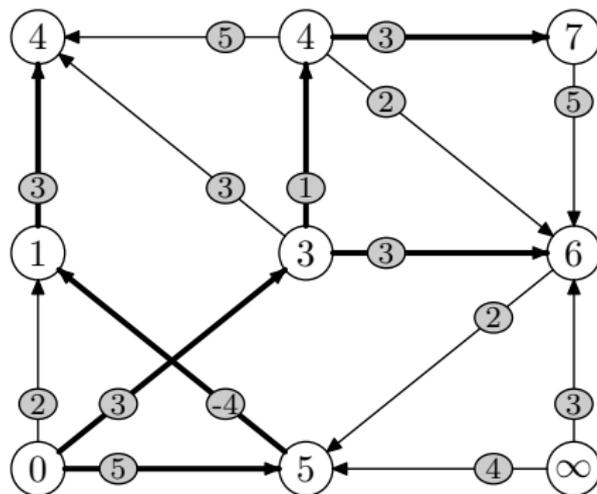
Relaxiere Knoten in topologischer Reihenfolge.

Laufzeit: $O(|V| + |E|)$.

Korrektheit: Jeder Knoten, der relaxiert, kennt zu diesem Zeitpunkt seinen echten Abstand. □

Frage: Sind negative Gewichte hier erlaubt?

Kürzeste Wege mit negativen Kantengewichten



Dijkstra funktioniert nicht mit negativen Kantengewichten.

Der Algorithmus von Bellman und Ford

Idee:

- Relaxiere alle Kanten
- Wiederhole dies, bis keine Änderung

Warum korrekt?

Induktion über die Länge eines kürzesten Pfads.
(Also genügen n Wiederholungen)

Was passiert bei Kreisen mit negativem Gewicht?

→ Keine Terminierung.

Der Algorithmus von Bellman und Ford

Idee:

- Relaxiere alle Kanten
- Wiederhole dies, bis keine Änderung

Warum korrekt?

Induktion über die Länge eines kürzesten Pfads.
(Also genügen n Wiederholungen)

Was passiert bei Kreisen mit negativem Gewicht?

→ Keine Terminierung.

Der Algorithmus von Bellman und Ford

Idee:

- Relaxiere alle Kanten
- Wiederhole dies, bis keine Änderung

Warum korrekt?

Induktion über die Länge eines kürzesten Pfads.
(Also genügen n Wiederholungen)

Was passiert bei Kreisen mit negativem Gewicht?

→ Keine Terminierung.

Der Algorithmus von Bellman und Ford

Algorithmus

```
function Bellman – Ford(s) boolean :  
for v in V do d[v] := infty od;  
d[s] := 0;  
for i = 1 to |V| – 1 do  
  forall(u, v) in E do  
    d[u] := min { d[u], d[v] + length(v, u) }  
  od  
od;  
forall(u, v) in E do  
  if d[u] > d[v] + length(v, u) then return false fi  
od;  
return true
```

Der Algorithmus von Bellman und Ford

Theorem

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $E \rightarrow \mathbf{R}$ und ein Knoten $s \in V$.

Wir können in $O(|V| \cdot |E|)$ feststellen, ob ein Kreis mit negativem Gewicht existiert, und falls nicht, die kürzesten Wege von s zu allen Knoten berechnen.

Beweis.

Wir haben die Korrektheit bereits nachgewiesen.

Zur Laufzeit: Jeder Knoten wird $|V|$ mal relaxiert, also wird auch jede Kante $|V|$ mal relaxiert (in konstanter Zeit). □

Der Algorithmus von Bellman und Ford

Theorem

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $E \rightarrow \mathbf{R}$ und ein Knoten $s \in V$.

Wir können in $O(|V| \cdot |E|)$ feststellen, ob ein Kreis mit negativem Gewicht existiert, und falls nicht, die kürzesten Wege von s zu allen Knoten berechnen.

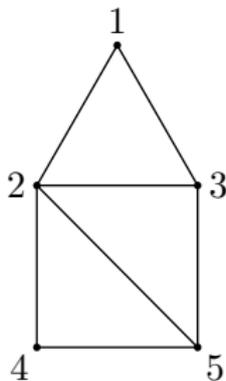
Beweis.

Wir haben die Korrektheit bereits nachgewiesen.

Zur Laufzeit: Jeder Knoten wird $|V|$ mal relaxiert, also wird auch jede Kante $|V|$ mal relaxiert (in konstanter Zeit). □

Rekapitulation: Darstellung von Graphen

Adjazenzliste



1		2, 3
2		1, 3, 4, 5
3		1, 2, 5
4		2, 5
5		2, 3, 4

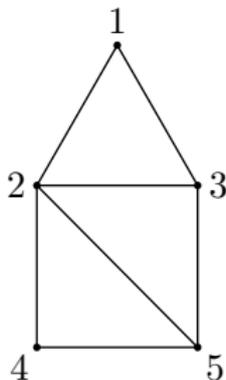
Alle bisherigen Algorithmen:

Adjazenzliste gute Darstellung

Kritische Operation: Alle ausgehenden Kanten besuchen.

Rekapitulation: Darstellung von Graphen

Adjazenzliste



1		2, 3
2		1, 3, 4, 5
3		1, 2, 5
4		2, 5
5		2, 3, 4

Alle bisherigen Algorithmen:

Adjazenzliste gute Darstellung

Kritische Operation: Alle ausgehenden Kanten besuchen.

All Pairs Shortest Paths

Eingabe: Gerichteter Graph mit Kantengewichten

Ausgabe: Abstände und kürzeste Wege zwischen allen Knotenpaaren

Laufzeit $O((|V|^2 + |V| \cdot |E|) \log |V|)$ mit Dijkstra.

→ Wende Dijkstra auf jeden Knoten an.

All Pairs Shortest Paths

Eingabe: Gerichteter Graph mit Kantengewichten

Ausgabe: Abstände und kürzeste Wege zwischen allen Knotenpaaren

Laufzeit $O((|V|^2 + |V| \cdot |E|) \log |V|)$ mit Dijkstra.

→ Wende Dijkstra auf jeden Knoten an.

Algorithmus von Floyd

Algorithmus

```
procedure Floyd1 :  
for  $i = 1, \dots, n$  do  
  for  $j = 1, \dots, n$  do  $d[i, j, 0] := \text{length}[i, j]$  od  
od;  
for  $k = 1, \dots, n$  do  
  for  $i = 1, \dots, n$  do  
    for  $j = 1, \dots, n$  do  
       $d[i, j, k] := \min \{ d[i, j, k - 1], d[i, k, k - 1] + d[k, j, k - 1] \}$   
    od  
  od  
od
```

Der Abstand von i nach j ist in $d[i, j, n]$ zu finden.

Theorem

Die kürzesten Wege zwischen allen Knotenpaaren eines gerichteten Graphen $G = (V, E)$ können in $O(|V|^3)$ Schritten gefunden werden.

Beweis.

Per Induktion: $d[i, j, k]$ enthält die Länge des kürzesten Pfades von i nach j , wenn nur $1, \dots, k$ als Zwischenstationen erlaubt sind. Dann enthält $d[i, j, n]$ den wirklichen Abstand. \square

Theorem

Die kürzesten Wege zwischen allen Knotenpaaren eines gerichteten Graphen $G = (V, E)$ können in $O(|V|^3)$ Schritten gefunden werden.

Beweis.

Per Induktion: $d[i, j, k]$ enthält die Länge des kürzesten Pfades von i nach j , wenn nur $1, \dots, k$ als Zwischenstationen erlaubt sind. Dann enthält $d[i, j, n]$ den wirklichen Abstand. □

Algorithmus von Floyd

Einfachere Version:

Algorithmus

```
procedure Floyd :  
for i = 1, ..., n do  
  for j = 1, ..., n do d[i, j] := length[i, j] od  
od;  
for k = 1, ..., n do  
  for i = 1, ..., n do  
    for j = 1, ..., n do  
      d[i, j] := min { d[i, j], d[i, k] + d[k, j] }  
    od  
  od  
od
```

Spezialfall: Transitive Hülle – Algorithmus von Warshall

Frage: Zwischen welchen Knotenpaaren gibt es einen Weg?

Algorithmus

```
procedure Warshall :  
for i = 1, ..., n do  
  for j = 1, ..., n do D[i,j] := A[i,j] od  
od;  
for k = 1, ..., n do  
  for i = 1, ..., n do  
    for j = 1, ..., n do  
      D[i,j] := D[i,j] || D[i,k] && D[k,j]  
    od  
  od  
od
```

Transitive Hülle

Theorem

Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

- 1 Berechne die starken Zusammenhangskomponenten
- 2 Schrumpfe jede SCC X zu einem Knoten
- 3 Berechne die transitive Hülle H dieses Graphen
- 4 Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.



Transitive Hülle

Theorem

Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

- 1 Berechne die starken Zusammenhangskomponenten
- 2 Schrumpfe jede SCC X zu einem Knoten
- 3 Berechne die transitive Hülle H dieses Graphen
- 4 Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.



Transitive Hülle

Theorem

Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

- 1 Berechne die starken Zusammenhangskomponenten
- 2 Schrumpfe jede SCC X zu einem Knoten
- 3 Berechne die transitive Hülle H dieses Graphen
- 4 Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.



Transitive Hülle

Theorem

Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

- 1 Berechne die starken Zusammenhangskomponenten
- 2 Schrumpfe jede SCC X zu einem Knoten
- 3 Berechne die transitive Hülle H dieses Graphen
- 4 Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.



Transitive Hülle

Theorem

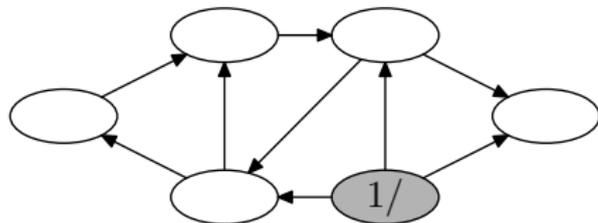
Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

- 1 Berechne die starken Zusammenhangskomponenten
- 2 Schrumpfe jede SCC X zu einem Knoten
- 3 Berechne die transitive Hülle H dieses Graphen
- 4 Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.



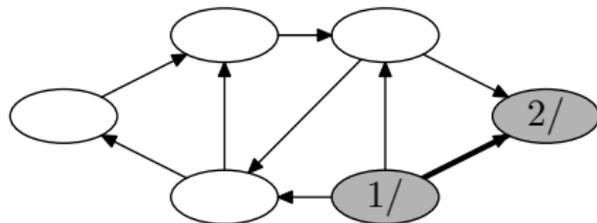
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

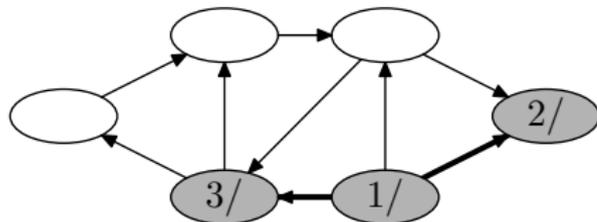
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

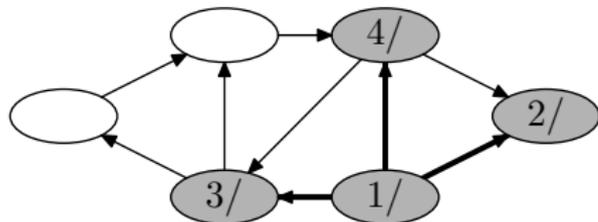
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

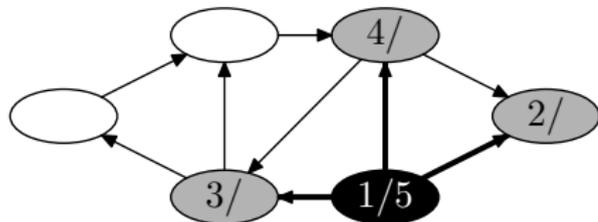
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

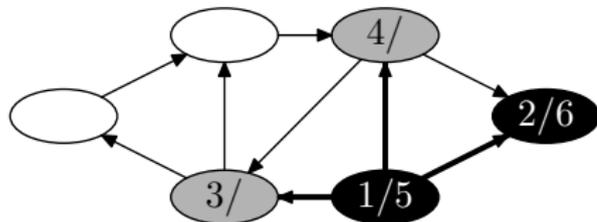
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

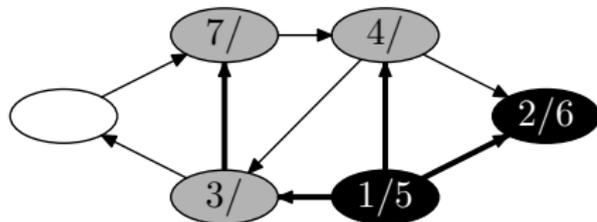
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

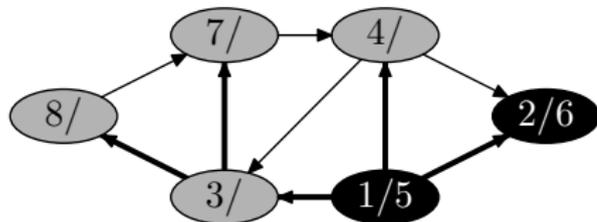
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

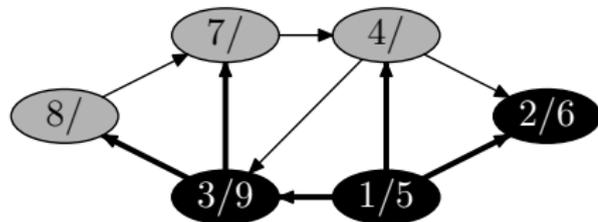
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

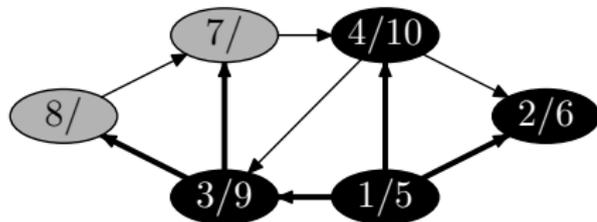
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

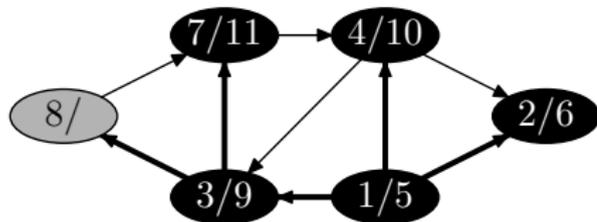
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

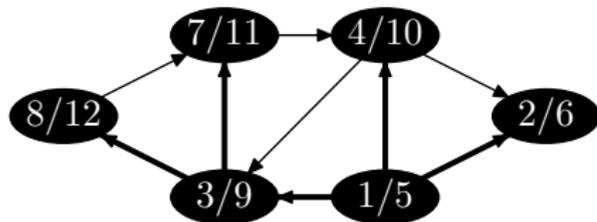
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

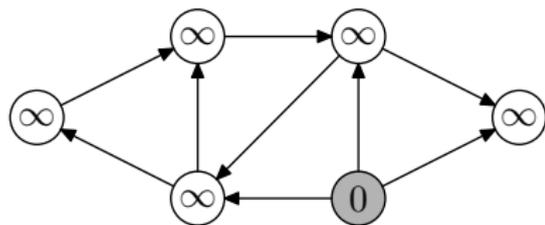
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

Breitensuche



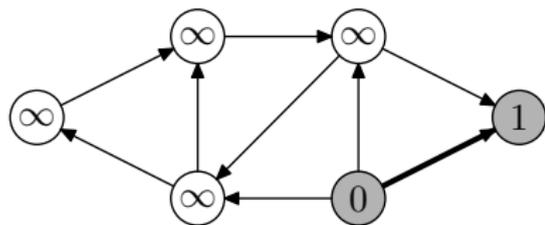
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



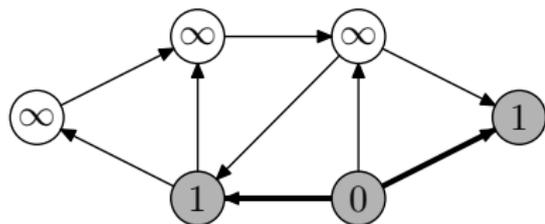
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



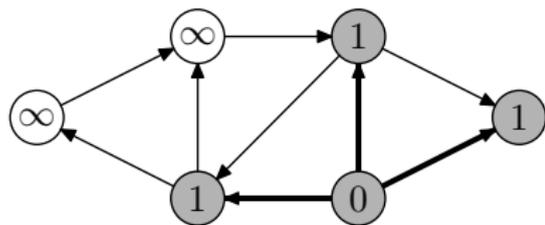
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



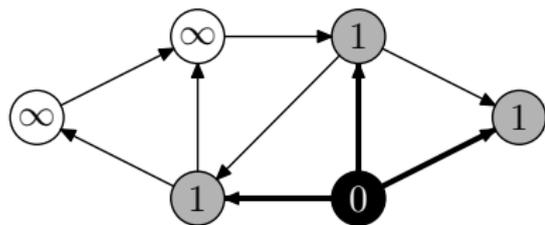
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



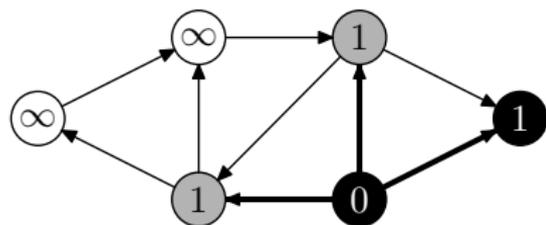
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



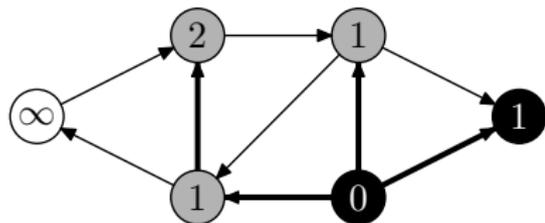
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



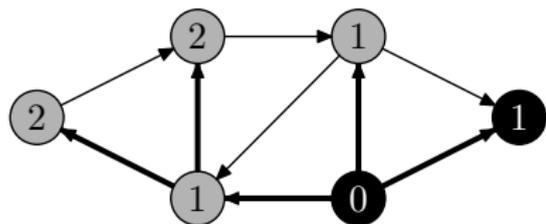
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



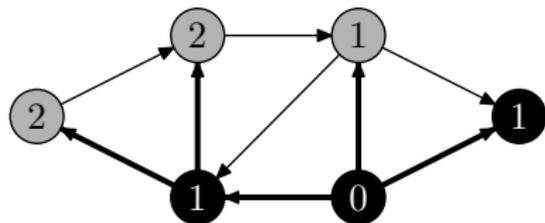
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



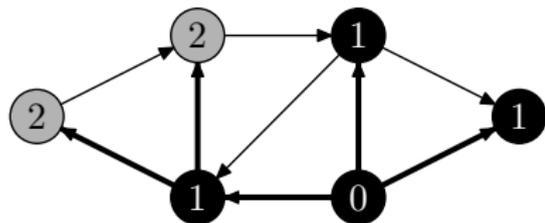
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



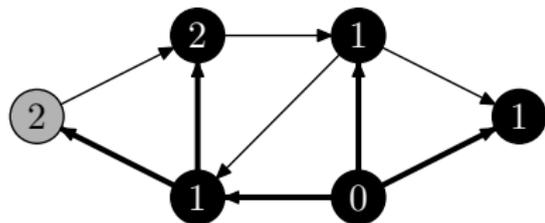
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



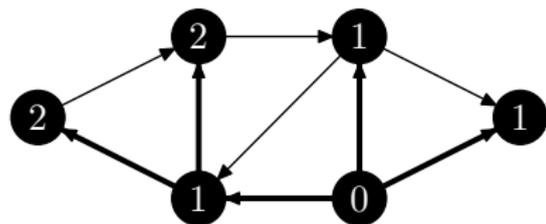
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche



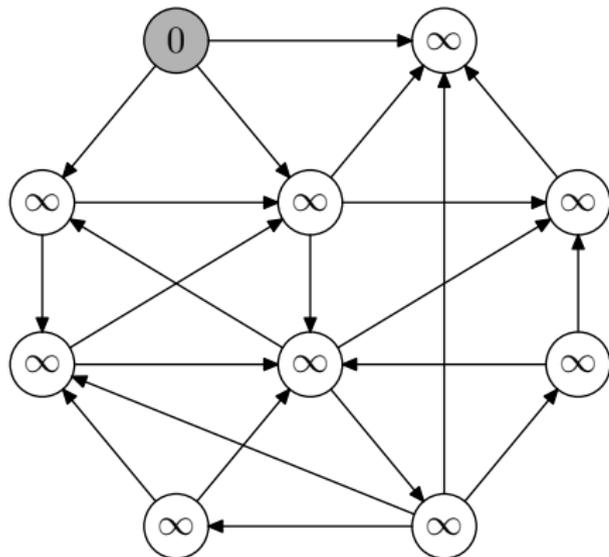
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

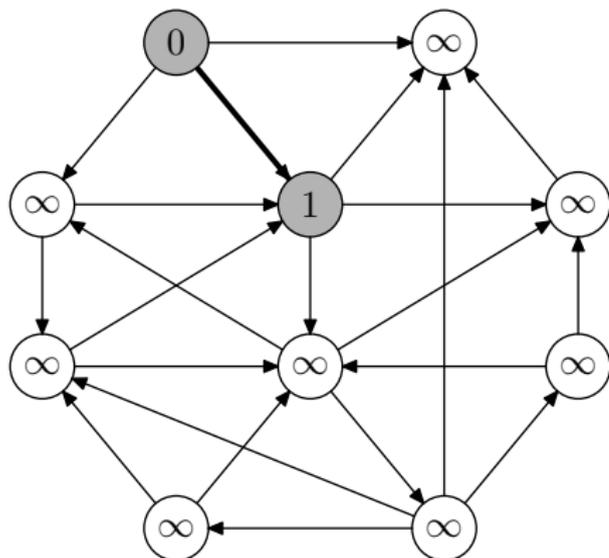
Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

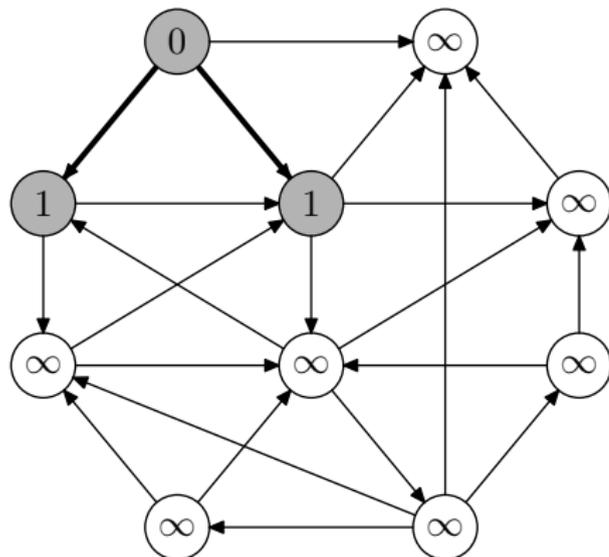
Breitensuche – Größeres Beispiel



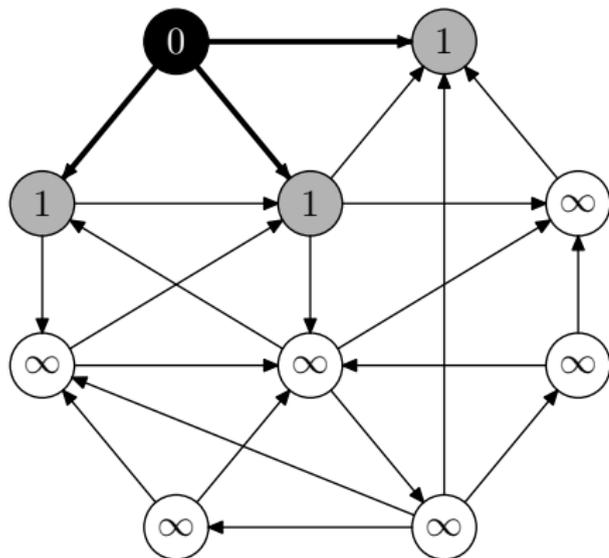
Breitensuche – Größeres Beispiel



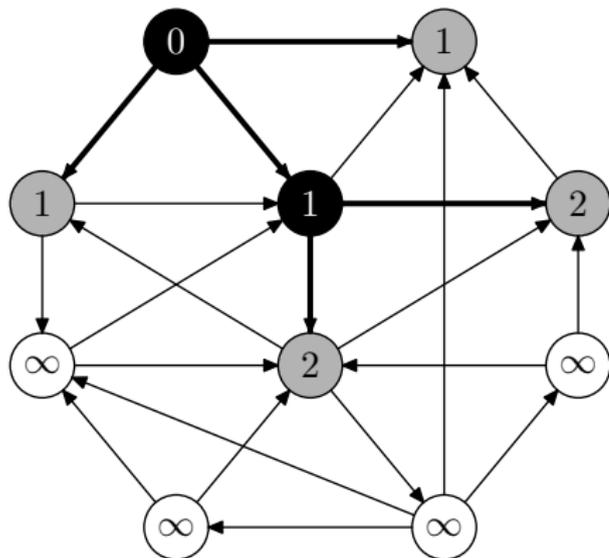
Breitensuche – Größeres Beispiel



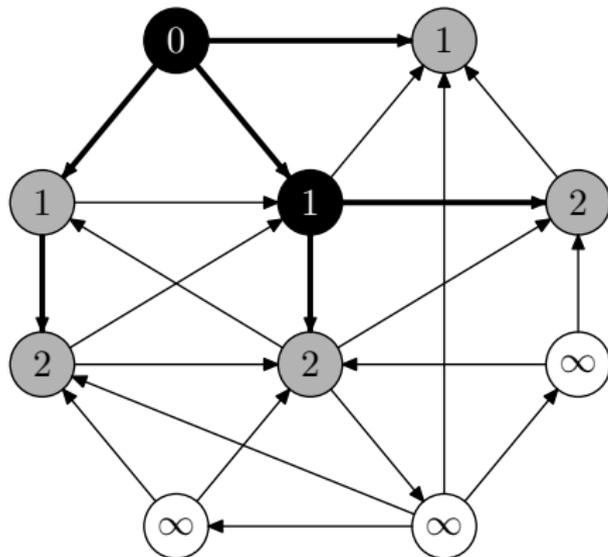
Breitensuche – Größeres Beispiel



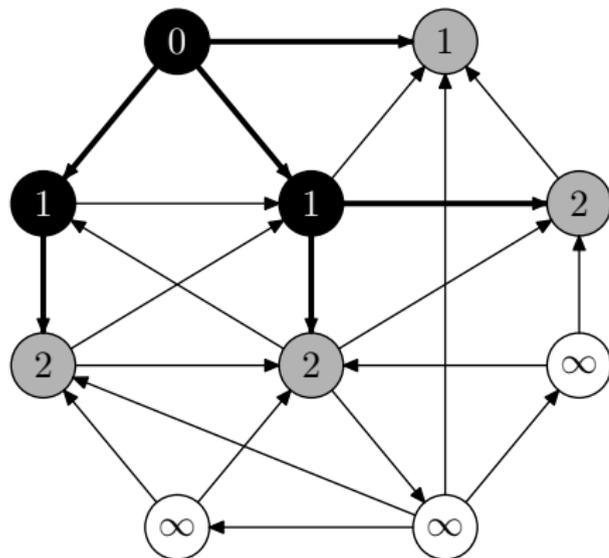
Breitensuche – Größeres Beispiel



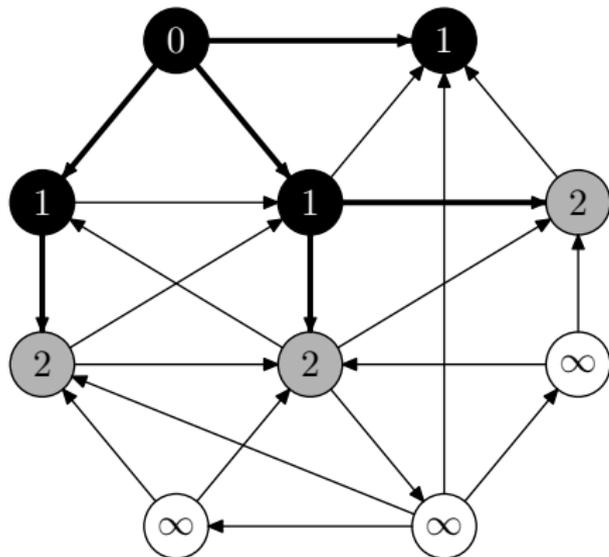
Breitensuche – Größeres Beispiel



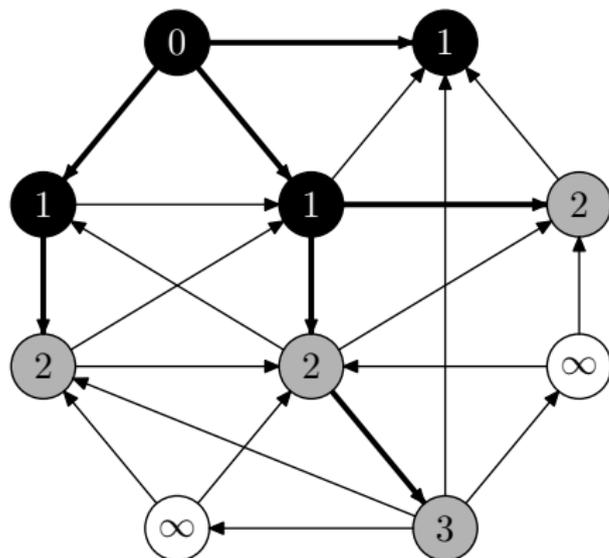
Breitensuche – Größeres Beispiel



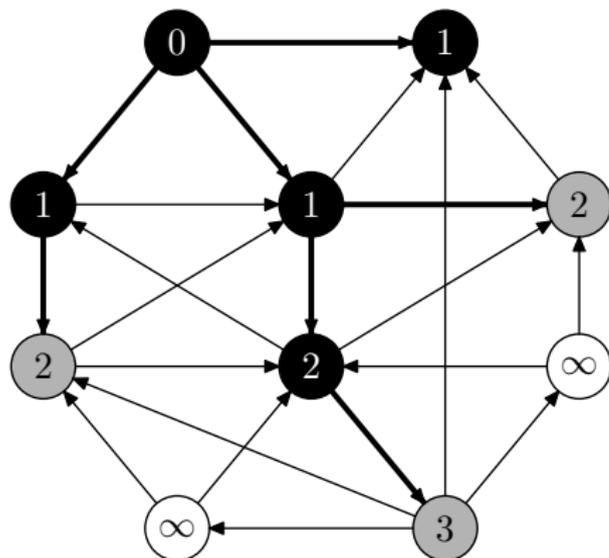
Breitensuche – Größeres Beispiel



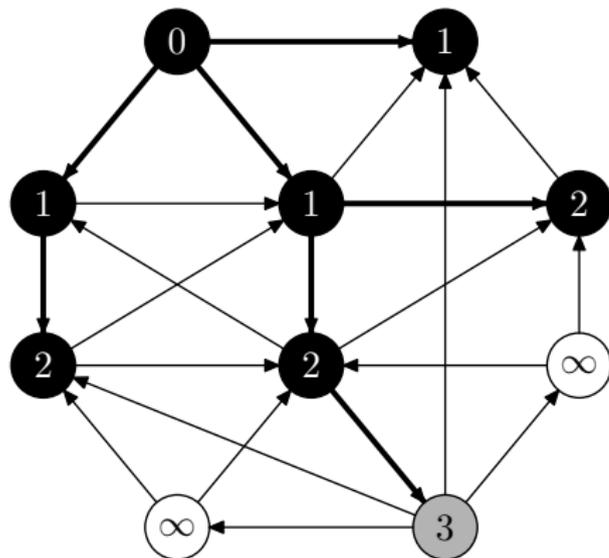
Breitensuche – Größeres Beispiel



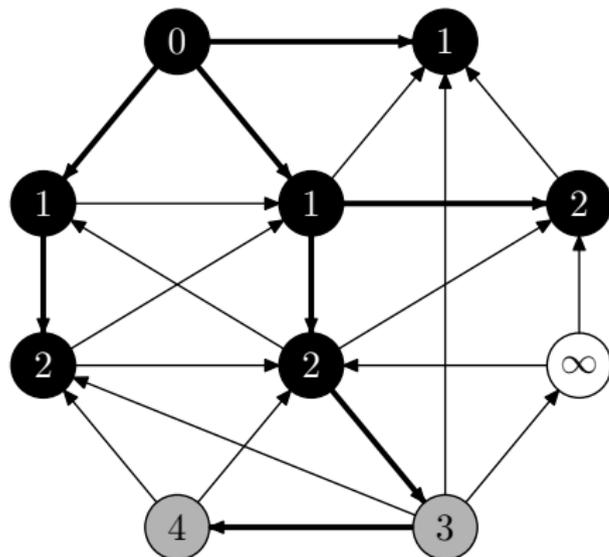
Breitensuche – Größeres Beispiel



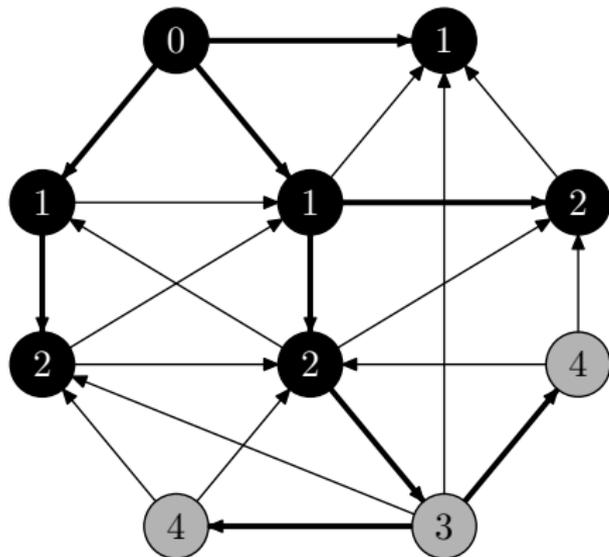
Breitensuche – Größeres Beispiel



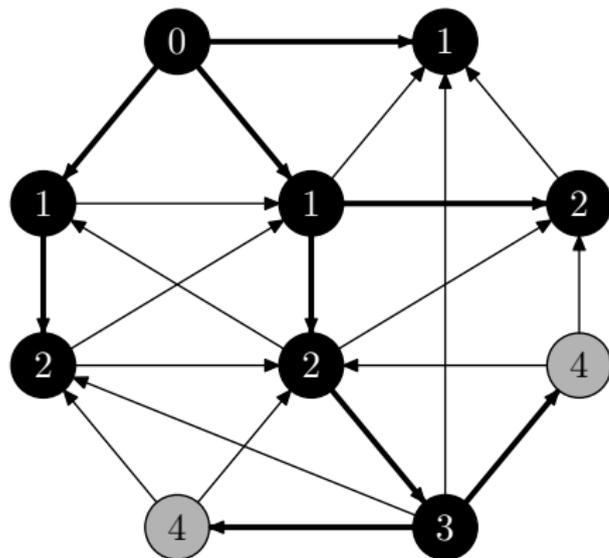
Breitensuche – Größeres Beispiel



Breitensuche – Größeres Beispiel



Breitensuche – Größeres Beispiel




```
public Map<V, V> BFS(V s, Map<Edge<V>, Double> length) {  
    Map<V, V> pred = new HashMap<V, V>();  
    Map<V, Integer> color = new HashMap<V, Integer>();  
    for(V u : allNodes()) color.put(u, WHITE);  
    Queue<V> q = new Queue<V>();  
    q.enqueue(s); color.put(s, GRAY);  
    while(!q.isEmpty()) {  
        V u = q.dequeue();  
        for(V v : neighbors(u)) {  
            if(color.get(v) == WHITE) {  
                color.put(v, GRAY);  
                q.enqueue(v);  
                pred.put(v, u);  
            }  
        }  
        color.put(u, WHITE);  
    }  
    return pred;  
}
```

Breitensuche

Theorem

Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $s \in V$.

Die kürzesten Wege von s zu allen anderen Knoten und die zugehörigen Abstände können in $O(|V| + |E|)$ berechnet werden.

Beweis.

Wir verwenden Breitensuche.

Die Laufzeit ist offensichtlich linear.

Der Korrektheitsbeweis sei hier weggelassen (etwas lang und technisch). □

Breitensuche

Theorem

Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $s \in V$.

Die kürzesten Wege von s zu allen anderen Knoten und die zugehörigen Abstände können in $O(|V| + |E|)$ berechnet werden.

Beweis.

Wir verwenden Breitensuche.

Die Laufzeit ist offensichtlich linear.

Der Korrektheitsbeweis sei hier weggelassen (etwas lang und technisch). □