Script generated by TTT

Title: Seidl: GAD (30.06.2015)

Date: Tue Jun 30 13:46:51 CEST 2015

Duration: 145:52 min

Pages: 70

Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
  Setze v auf nicht markiert;
init();
foreach (s ∈ V)
  if (s nicht markiert) {
    markiere s;
    root(s);
    DFS(s,s);
  }
```

```
DFS(Node u, Node v) {
  foreach ((v, w) ∈ E)
    if (w ist markiert)
      traverseNonTreeEdge(v,w);
    else {
      traverseTreeEdge(v,w);
      markiere w;
      DFS(v,w);
    }
  backtrack(u,v);
}
```


Tiefensuche

Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

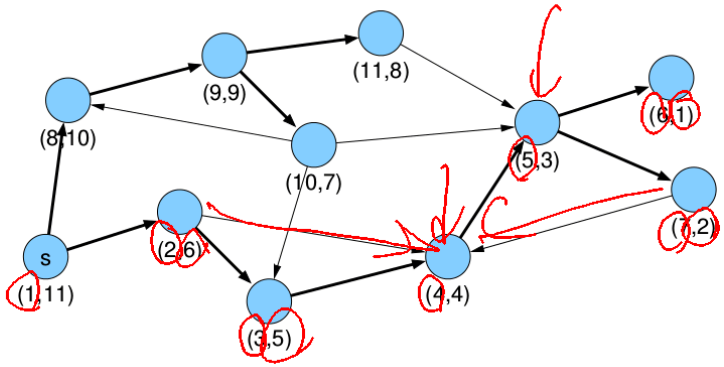
Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
  Setze v auf nicht markiert;
init();
foreach (s ∈ V)
  if (s nicht markiert) {
    markiere s;
    root(s);
    DFS(s,s);
  }
```

```
DFS(Node u, Node v) {
  foreach ((v, w) ∈ E)
    if (w ist markiert)
      traverseNonTreeEdge(v,w);
    else {
      traverseTreeEdge(v,w);
      markiere w;
      DFS(v,w);
    }
  backtrack(u,v);
}
```

Tiefensuche



DFS-Nummerierung

Beobachtung:

- Knoten im DFS-Rekursionsstack (aktiven Knoten) sind bezüglich dfsNum aufsteigend sortiert

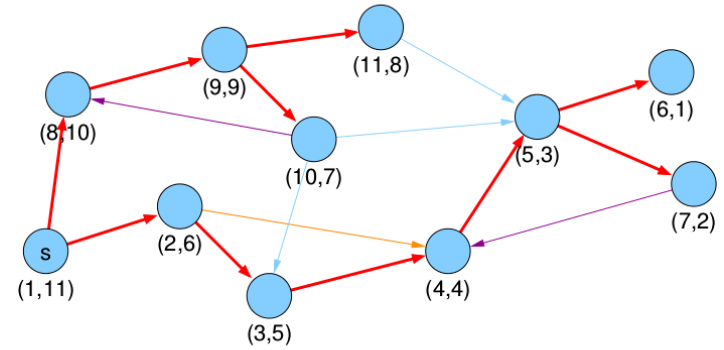
Begründung:

- dfsCount wird nach jeder Zuweisung von dfsNum inkrementiert
- neue aktive Knoten haben also immer die höchste dfsNum

DFS-Nummerierung

Kantentypen:

- Baumkanten: zum Kind
- Vorwärtskanten: zu einem Nachfahren
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



DFS-Nummerierung

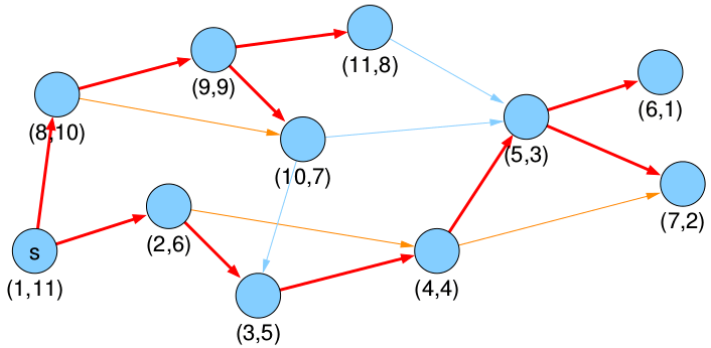
Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Anwendung:

- Erkennung von azyklischen gerichteten Graphen (engl. directed acyclic graph / DAG)



- keine gerichteten Kreise

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

DFS-Nummerierung

Beobachtung für Kante (v,w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- (2)⇒(3): Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- (3)⇒(2): Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

Graphen Graphtraversierung

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- (2)⇒(3): Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- (3)⇒(2): Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

H. Seidl (TUM) GAD SS'15 438

Graphen Graphtraversierung

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- ¬(2)⇒¬(1): Wenn Rückwärtskante (v,w) existiert, gibt es einen gerichteten Kreis ab Knoten w (und G ist kein DAG).
- ¬(1)⇒¬(2): Wenn es einen gerichteten Kreis gibt, ist mindestens eine von der DFS besuchte Kante dieses Kreises eine Rückwärtskante (Kante zu einem schon besuchten Knoten, dieser muss Vorfahr sein).

□

Graphen Graphtraversierung

Zusammenhang in Graphen

Definition

Ein ungerichteter Graph heißt zusammenhängend, wenn es von jedem Knoten einen Pfad zu jedem anderen Knoten gibt.

Ein maximaler zusammenhängender induzierter Teilgraph wird als Zusammenhangskomponente bezeichnet.

Die Zusammenhangskomponenten eines ungerichteten Graphen können mit DFS oder BFS in $O(n+m)$ bestimmt werden.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Bemerkung:

- "zusammenhängend" ist im wesentlichen gleichbedeutend mit "1-knotenzusammenhängend"

  Ausnahme: Graph mit nur einem Knoten ist zusammenhängend, aber nicht 1-zusammenhängend

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Definition

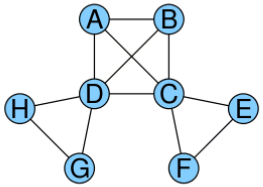
Die Zweifachzusammenhangskomponenten eines Graphen sind die maximalen Teilgraphen, die 2-fach zusammenhängend sind.

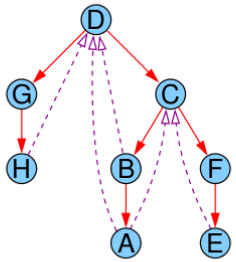
Ein Block ist ein maximaler zusammenhängender Teilgraph, der keinen Artikulationsknoten enthält.

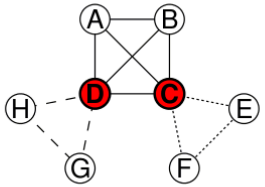
Die Menge der Blöcke besteht aus den Zweifachzusammenhangskomponenten, den Brücken (engl. *cut edges*), sowie den isolierten Knoten.

Artikulationsknoten und Blöcke per DFS

- bei Aufruf der DFS für Knoten v wird `num[v]` bestimmt und `low[v]` mit `num[v]` initialisiert
- nach Besuch eines Nachbarknotens w:
  Update von `low[v]` durch Vergleich mit
  - `low[w]` nach Rückkehr vom rekursiven Aufruf, falls (v,w) eine Baumkante war
  - `num[w]`, falls (v,w) eine Rückwärtskante war







Blöcke und DFS

Modifizierte DFS nach R. E. Tarjan:

- `num[v]`: DFS-Nummer von v
- `low[v]`: minimale Nummer `num[w]` eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten (abwärts), evt. gefolgt von einer einzigen Rückwärtskante (aufwärts) erreicht werden kann

- `low[v]`: Minimum von
  - `num[v]`
  - `low[w]`, wobei w ein Kind von v im DFS-Baum ist (Baumkante)
  - `num[w]`, wobei $\{v,w\}$ eine Rückwärtskante ist

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

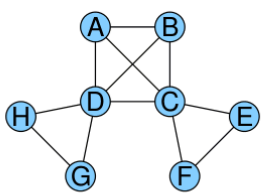
- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit `low[b]`$\geq$`num[a]` gibt.*

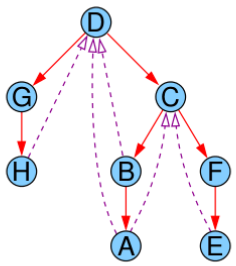
Beweisidee

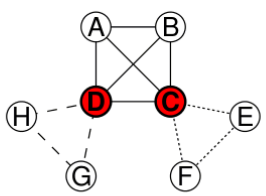
Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

Artikulationsknoten und Blöcke per DFS

- bei Aufruf der DFS für Knoten v wird `num[v]` bestimmt und `low[v]` mit `num[v]` initialisiert
- nach Besuch eines Nachbarknotens w:
  Update von `low[v]` durch Vergleich mit
  - `low[w]` nach Rückkehr vom rekursiven Aufruf, falls (v,w) eine Baumkante war
  - `num[w]`, falls (v,w) eine Rückwärtskante war

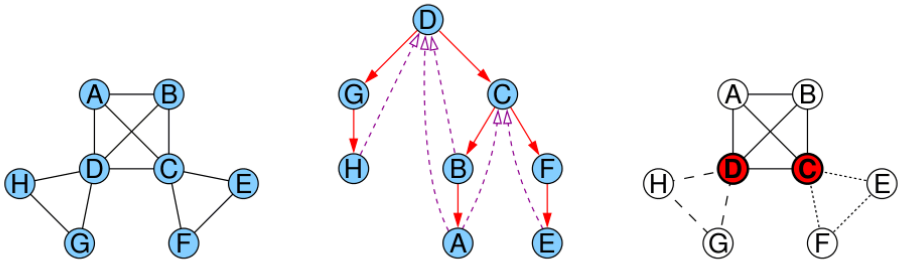






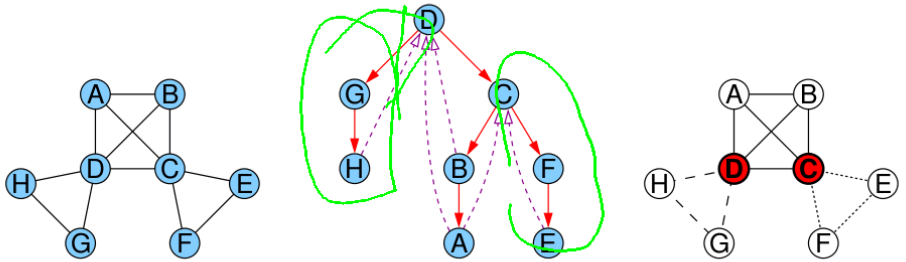
Artikulationsknoten und Blöcke per DFS

- bei Aufruf der DFS für Knoten v wird `num[v]` bestimmt und `low[v]` mit `num[v]` initialisiert
- nach Besuch eines Nachbarknotens w: Update von `low[v]` durch Vergleich mit
  - `low[w]` nach Rückkehr vom rekursiven Aufruf, falls (v,w) eine Baumkante war
  - `num[w]`, falls (v,w) eine Rückwärtskante war



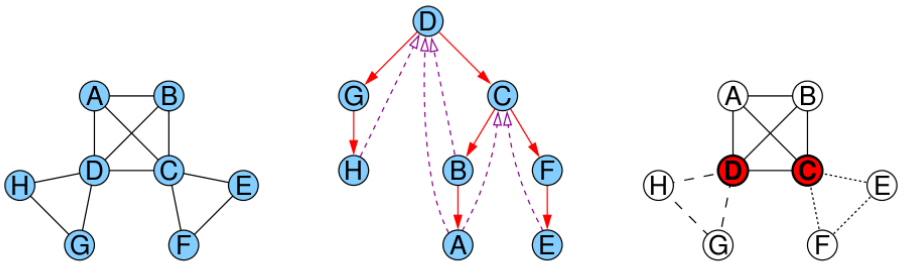
Artikulationsknoten und Blöcke per DFS

- Kanten werden auf einem anfangs leeren Stack gesammelt
- Rückwärtskanten kommen direkt auf den Stack (ohne rek. Aufruf)
- Baumkanten kommen vor dem rekursiven Aufruf auf den Stack
- nach Rückkehr von einem rekursiven Aufruf werden im Fall `low[w]`$\geq$`num[v]` die obersten Kanten vom Stack bis einschließlich der Baumkante $\{v,w\}$ entfernt und bilden den nächsten Block



Artikulationsknoten und Blöcke per DFS

- Kanten werden auf einem anfangs leeren Stack gesammelt
- Rückwärtskanten kommen direkt auf den Stack (ohne rek. Aufruf)
- Baumkanten kommen vor dem rekursiven Aufruf auf den Stack
- nach Rückkehr von einem rekursiven Aufruf werden im Fall `low[w]`$\geq$`num[v]` die obersten Kanten vom Stack bis einschließlich der Baumkante $\{v,w\}$ entfernt und bilden den nächsten Block



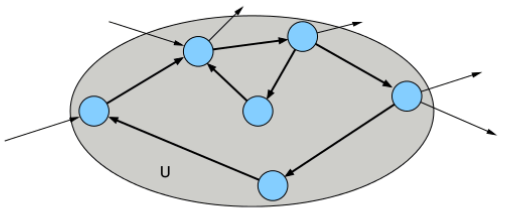
Starke Zusammenhangskomponenten

Definition

Sei $G = (V, E)$ ein gerichteter Graph.

Knotenteilmenge $U \subseteq V$ heißt stark zusammenhängend genau dann, wenn für alle $u, v \in U$ ein gerichteter Pfad von u nach v in G existiert.

Für Knotenteilmenge $U \subseteq V$ heißt der induzierte Teilgraph $G[U]$ starke Zusammenhangskomponente von G, wenn U stark zusammenhängend und (inklusions-)maximal ist.



Starke Zusammenhangskomponenten

Beobachtungen:

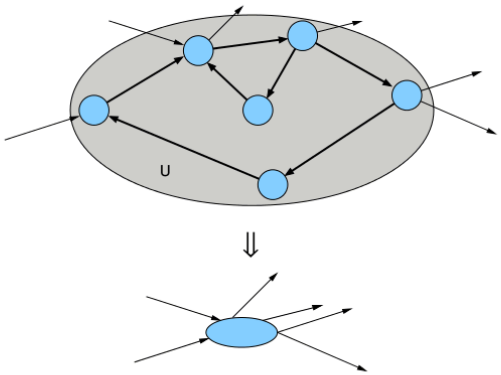
- Knoten $x, y \in V$ sind stark zusammenhängend, falls beide Knoten auf einem gemeinsamen gerichteten Kreis liegen (oder $x = y$).
- Die starken Zusammenhangskomponenten bilden eine Partition der Knotenmenge.

  (im Gegensatz zu 2-Zhk. bei ungerichteten Graphen, wo nur die Kantenmenge partitioniert wird, sich aber zwei verschiedene 2-Zhk. in einem Knoten überlappen können)

Starke Zusammenhangskomponenten

Beobachtungen:

- Schrumpft man alle starken Zusammenhangskomponenten zu einzelnen (Super-)Knoten, ergibt sich ein DAG.



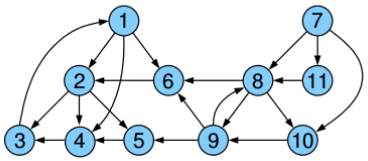
Starke Zhk. und DFS

Idee:

- beginne mit Graph ohne Kanten, jeder Knoten ist eigene SCC
- füge nach und nach einzelne Kanten ein

$\Rightarrow$ aktueller (current) Graph $G_c = (V, E_c)$

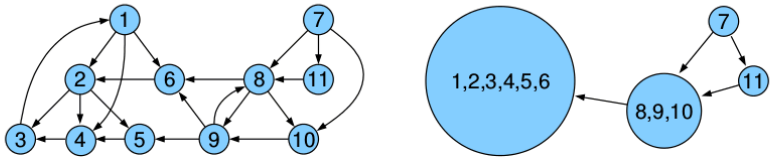
- Update der starken Zusammenhangskomponenten (SCCs)



Starke Zhk. und DFS

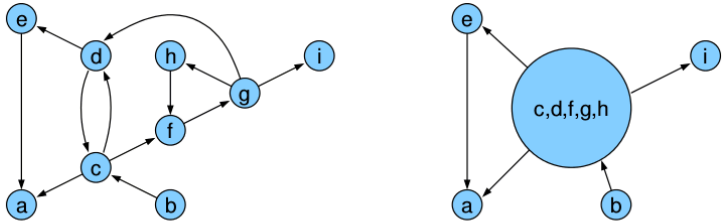
Idee:

- betrachte geschrumpften (shrunken) Graph G_c^s:
  Knoten entsprechen SCCs von G_c, Kante (C, D) genau dann, wenn es Knoten $u \in C$ und $v \in D$ mit $(u, v) \in E_c$ gibt
- geschrumpfter Graph G_c^s ist ein DAG
- Ziel: Aktualisierung des geschrumpften Graphen beim Einfügen



Starke Zhk. und DFS

Geschrumpfter Graph
(Beispiel aus Mehlhorn / Sanders)



Starke Zhk. und DFS

Update des geschrumpften Graphen nach Einfügen einer Kante:

3 Möglichkeiten:

- beide Endpunkte gehören zu derselben SCC

⇒ geschrumpfter Graph unverändert

- Kante verbindet Knoten aus zwei verschiedenen SCCs, aber schließt keinen Kreis

⇒ SCCs im geschrumpften Graph unverändert, aber eine Kante wird im geschrumpften Graph eingefügt (falls nicht schon vorhanden)

- Kante verbindet Knoten aus zwei verschiedenen SCCs und schließt einen oder mehrere Kreise

⇒ alle SCCs, die auf einem der Kreise liegen, werden zu einer einzigen SCC verschmolzen

Starke Zhk. und DFS

Prinzip:

- Tiefensuche
  - V_c schon markierte (entdeckte) Knoten
  - E_c schon gefundene Kanten
- 3 Arten von SCC: unentdeckt, offen, geschlossen
- unentdeckte Knoten haben Ein- / Ausgangsgrad Null in G_c

⇒ zunächst bildet jeder Knoten eine eigene unentdeckte SCC, andere SCCs enthalten nur markierte Knoten

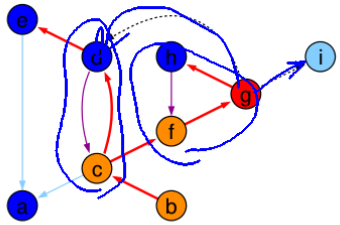
- SCCs mit mindestens einem aktiven Knoten (ohne finishNum) heißen offen
- SCC heißt geschlossen, falls sie nur fertige Knoten (mit finishNum) enthält
- Knoten in offenen / geschlossenen SCCs heißen offen / geschlossen

Starke Zhk. und DFS

- Knoten in geschlossenen SCCs sind immer fertig (mit finishNum)
- Knoten in offenen SCCs können fertig oder noch aktiv (ohne finishNum) sein
- Repräsentant einer SCC: Knoten mit kleinster dfsNum

Starke Zhk. und DFS

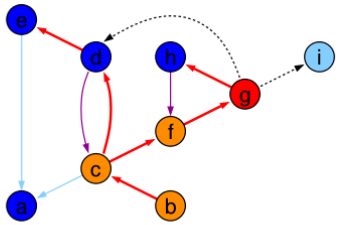
DFS-Snapshot:

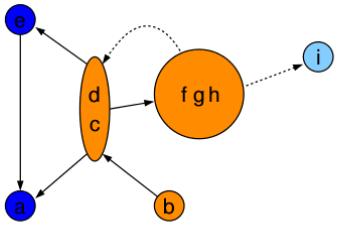


- erste DFS startete bei Knoten a, zweite bei b
- aktueller Knoten ist g, auf dem Rekursionsstack liegen b, c, f, g
- (g, d) und (g, i) wurden noch nicht exploriert
- (d, c) und (h, f) sind Rückwärtskanten
- (c, a) und (e, a) sind Querkanten
- (b, c), (c, d), (d, e), (c, f), (f, g) und (g, h) sind Baumkanten

Starke Zhk. und DFS

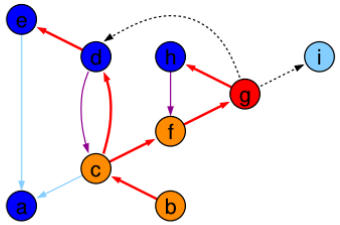
DFS-Snapshot mit geschrumpftem Graph:

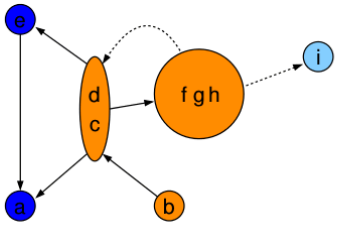




- unentdeckt: $\{i\}$ offen: $\{b\}$, $\{c, d\}$, $\{f, g, h\}$ geschlossen: $\{a\}$, $\{e\}$
- offene SCCs bilden Pfad im geschrumpften Graph
- aktueller Knoten gehört zur letzten SCC
- offene Knoten wurden in Reihenfolge b, c, d, f, g, h erreicht und werden von den Repräsentanten b, c und f genau in die offenen SCCs partitioniert

Starke Zhk. und DFS





Beobachtungen (Invarianten für G_c):

1. Pfade aus geschlossenen SCCs führen immer zu geschlossenen SCCs
2. Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen SCCs

   offene Komponenten bilden Pfad im geschrumpften Graph
3. Knoten der offenen SCCs in Reihenfolge der DFS-Nummern werden durch Repräsentanten in die offenen SCCs partitioniert

Starke Zhk. und DFS

Geschlossene SCCs von G_c sind auch SCCs in G:

- Sei v geschlossener Knoten und S / S_c seine SCC in G / G_c.
- zu zeigen: $S = S_c$
- G_c ist Subgraph von G, also $S_c \subseteq S$
- somit zu zeigen: $S \subseteq S_c$

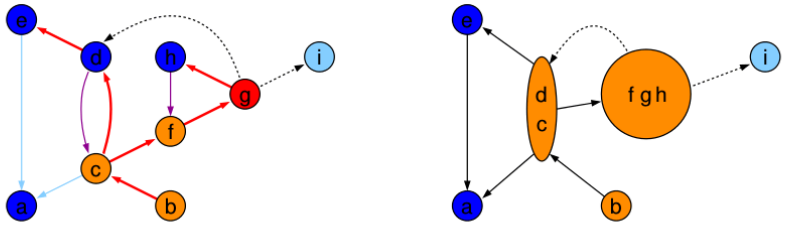
- Sei w ein Knoten in S.

$\Rightarrow$ $\exists$ Kreis C durch v und w.

- Invariante 1: alle Knoten von C sind geschlossen und somit erledigt (alle ausgehenden Kanten exploriert)
- C ist in G_c enthalten, also $w \in S_c$
- damit gilt $S \subseteq S_c$, also $S = S_c$

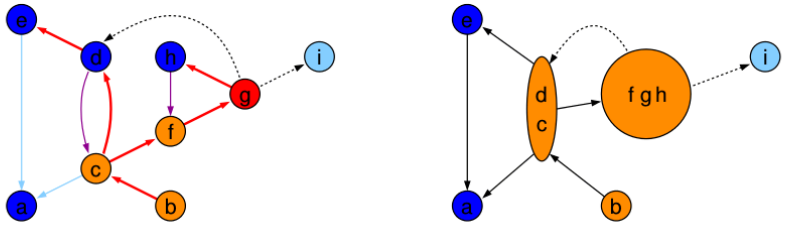
Starke Zhk. und DFS

DFS-Snapshot mit geschrumpftem Graph:



- unentdeckt: $\{i\}$ offen: $\{b\}, \{c, d\}, \{f, g, h\}$ geschlossen: $\{a\}, \{e\}$
- offene SCCs bilden Pfad im geschrumpften Graph
- aktueller Knoten gehört zur letzten SCC
- offene Knoten wurden in Reihenfolge b, c, d, f, g, h erreicht und werden von den Repräsentanten b, c und f genau in die offenen SCCs partitioniert

Starke Zhk. und DFS



Beobachtungen (Invarianten für G_c):

1. Pfade aus geschlossenen SCCs führen immer zu geschlossenen SCCs
2. Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen SCCs

   offene Komponenten bilden Pfad im geschrumpften Graph
3. Knoten der offenen SCCs in Reihenfolge der DFS-Nummern werden durch Repräsentanten in die offenen SCCs partitioniert

Starke Zhk. und DFS

Geschlossene SCCs von G_c sind auch SCCs in G:

- Sei v geschlossener Knoten und S / S_c seine SCC in G / G_c.
- zu zeigen: $S = S_c$
- G_c ist Subgraph von G, also $S_c \subseteq S$
- somit zu zeigen: $S \subseteq S_c$
- Sei w ein Knoten in S.

$\Rightarrow$ $\exists$ Kreis C durch v und w.

- Invariante 1: alle Knoten von C sind geschlossen und somit erledigt (alle ausgehenden Kanten exploriert)
- C ist in G_c enthalten, also $w \in S_c$
- damit gilt $S \subseteq S_c$, also $S = S_c$

Starke Zhk. und DFS

Vorgehen:

- Invarianten 2 und 3 helfen bei Verwaltung der offenen SCCs
- Knoten in offenen SCCs auf Stack oNodes
  (in Reihenfolge steigender dfsNum)
- Repräsentanten der offenen SCCs auf Stack oReps
- zu Beginn Invarianten gültig (alles leer)
- vor Markierung einer neuen Wurzel sind alle markierten Knoten erledigt, also keine offenen SCCs, beide Stacks leer
  dann: neue offene SCC für neue Wurzel s,
  s kommt auf beide Stacks

Starke Zhk. und DFS

Prinzip: betrachte Kante $e = (v, w)$

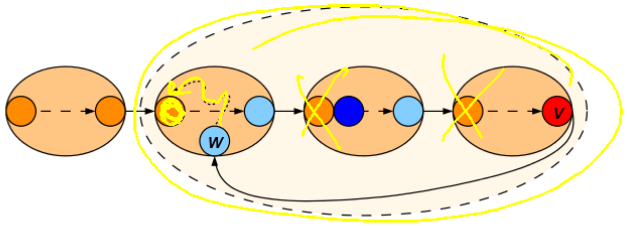
- Kante zu unbekanntem Knoten w (Baumkante):
  neue eigene offene SCC für w (w kommt auf oNodes und oReps)
- Kante zu Knoten w in geschlossener SCC (Nicht-Baumkante):
  von w gibt es keinen Weg zu v, sonst wäre die SCC von w noch nicht geschlossen (geschlossene SCCs sind bereits komplett), also SCCs unverändert
- Kante zu Knoten w in offener SCC (Nicht-Baumkante):
  falls v und w in unterschiedlichen SCCs liegen, müssen diese mit allen SCCs dazwischen zu einer einzigen SCC verschmolzen werden (durch Löschen der Repräsentanten)

Wenn Knoten keine ausgehenden Kanten mehr hat:

- Knoten fertig
- wenn Knoten Repräsentant seiner SCC ist, dann SCC schließen

Starke Zhk. und DFS

Vereinigung offener SCCs im Kreisfall:



- offene SCC entsprechen Ovalen, Knoten sortiert nach dfsNum
- alle Repräsentanten offener SCCs liegen auf Baumpfad zum aktuellen Knoten v in SCC S_k
- Nicht-Baumkante (v, w) endet an Knoten w in offener SCC S_i mit Repräsentant r_i
- Pfad von w nach r_i muss existieren (innerhalb SCC S_i)

$\Rightarrow$ Kante (v, w) vereinigt $S_i, \ldots, S_k$

Starke Zhk. und DFS

- init() {
  component = new int[n];
  oReps = ⟨ ⟩;
  oNodes = ⟨ ⟩;
  dfsCount = 1;
  }
- root(Node w) / traverseTreeEdge(Node v, Node w) {
  oReps.push(w); // Repräsentant einer neuen SCC
  oNodes.push(w); // neuer offener Knoten
  dfsNum[w] = dfsCount;
  dfsCount++;
  }

Starke Zhk. und DFS

- traverseNonTreeEdge(Node v, Node w) {
  if (w ∈ oNodes) // verschmelze SCCs
  while (dfsNum[w] < dfsNum[oReps.top()])
  oReps.pop();
  }
- backtrack(Node u, Node v) {
  if (v == oReps.top()) { // v Repräsentant?
  oReps.pop(); // ja: entferne v
  do { // und offene Knoten bis v
  w = oNodes.pop();
  component[w] = v;
  } while (w!=v);
  }
  }

Starke Zhk. und DFS

Zeit: $O(n+m)$

Begründung:

- init, root: $O(1)$, $O(n)$
- traverseTreeEdge: $(n-1) \times O(1)$
- backtrack, traverseNonTreeEdge:
  da jeder Knoten höchstens einmal in oReps und oNodes landet,
  insgesamt $O(n+m)$
- DFS-Gerüst: $O(n+m)$
- gesamt: $O(n+m)$

Übersicht

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Fälle:

- Kantenkosten 1
- DAG, beliebige Kantenkosten
- beliebiger Graph, positive Kantenkosten
- beliebiger Graph, beliebige Kantenkosten

Kürzeste-Wege-Problem

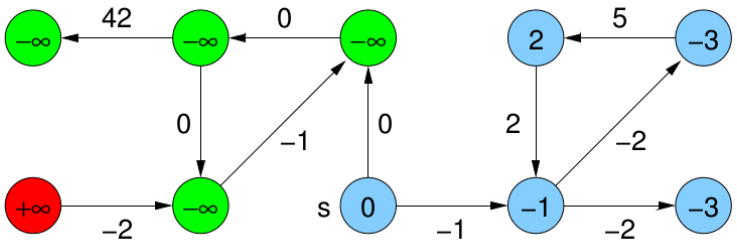
gegeben:

- gerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}$

2 Varianten:

- SSSP (single source shortest paths):
  kürzeste Wege von einer Quelle zu allen anderen Knoten
- APSP (all pairs shortest paths):
  kürzeste Wege zwischen allen Paaren

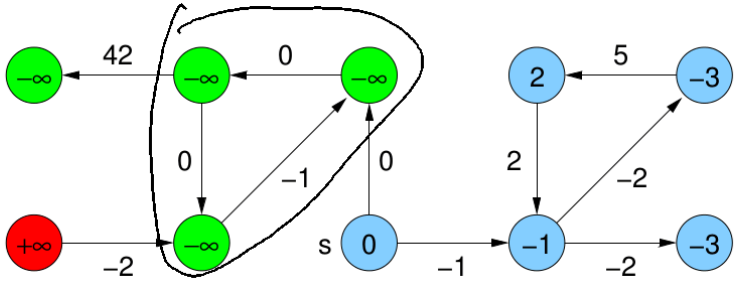
Distanzen



$\mu(s,v)$: Distanz von s nach v

$$\mu(s,v) = \begin{cases} +\infty & \text{kein Weg von } s \text{ nach } v \\ -\infty & \text{Weg beliebig kleiner Kosten von } s \text{ nach } v \\ \min\{c(p) : p \text{ ist Weg von } s \text{ nach } v\} \end{cases}$$

Distanzen



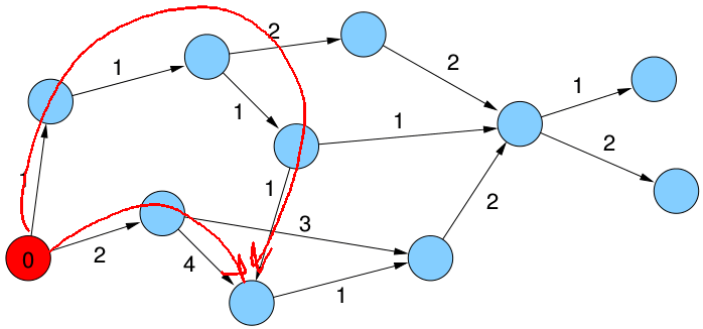
Wann sind die Kosten $-\infty$?

wenn es einen Kreis mit negativer Gewichtssumme gibt
(hinreichende und notwendige Bedingung)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

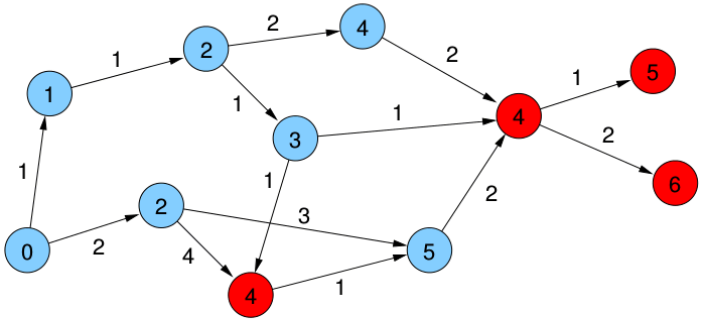
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Einfache Breitensuche funktioniert nicht.

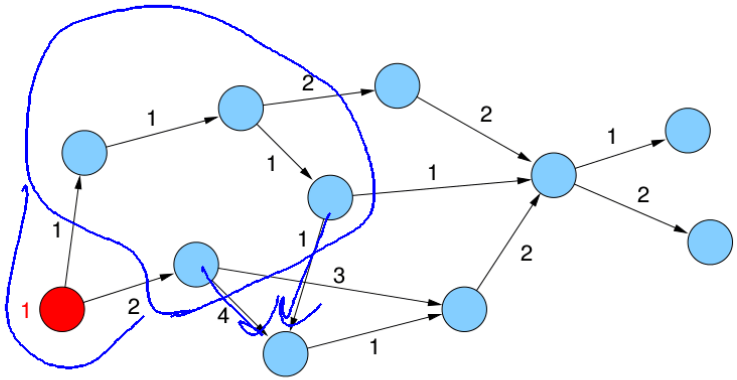


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

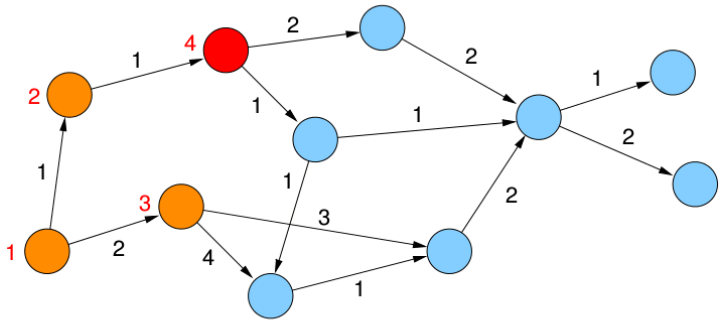


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

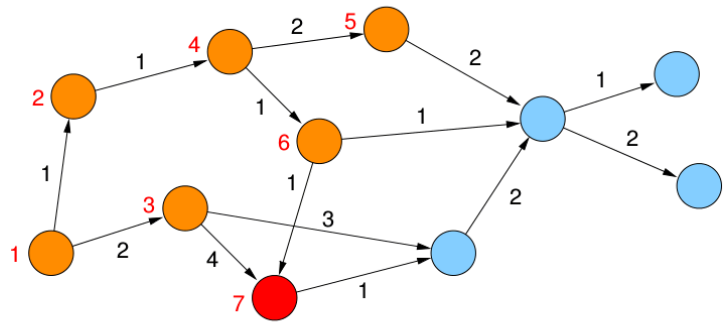


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

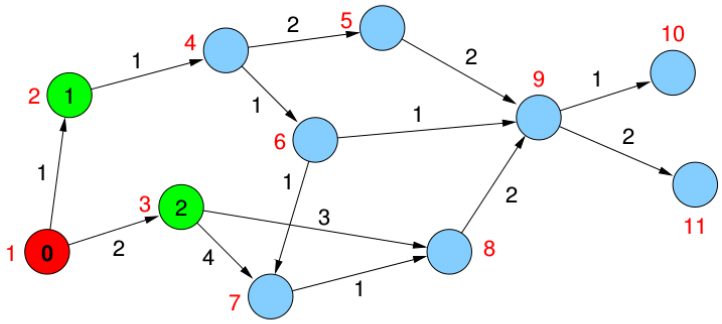


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

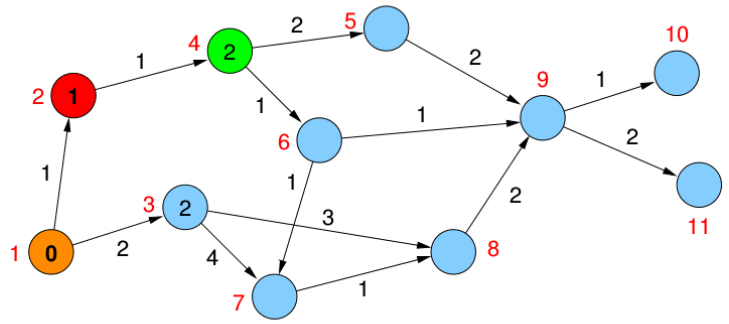


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

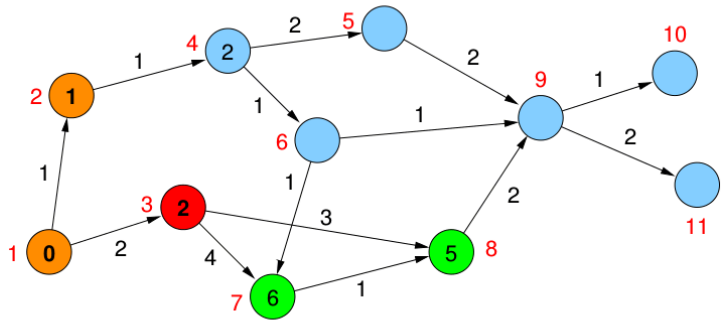


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

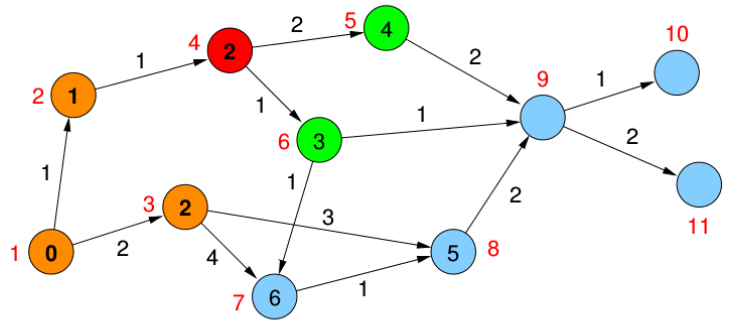


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

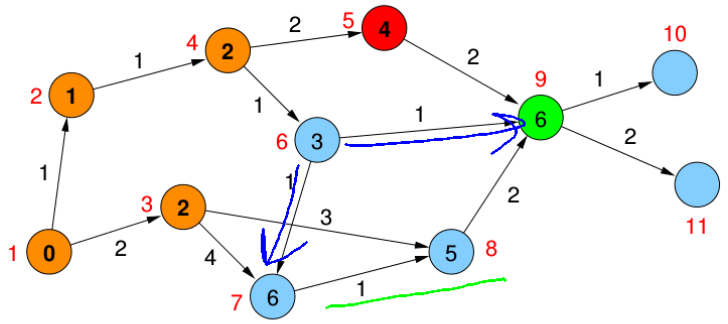


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

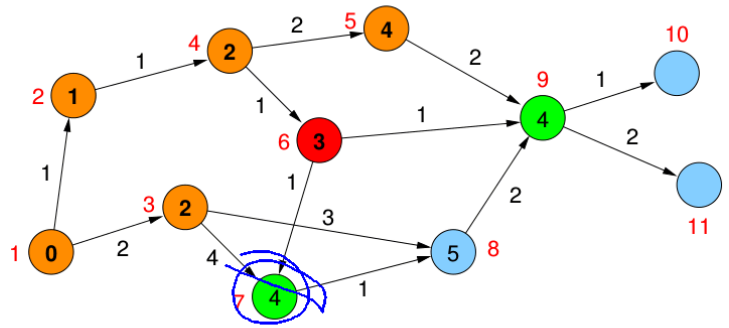


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Topologische Sortierung – warum funktioniert das?

- betrachte einen kürzesten Weg von s nach v
- der ganze Pfad beachtet die topologische Sortierung
- d.h., die Distanzen werden in der Reihenfolge der Knoten vom Anfang des Pfades zum Ende hin betrachtet
- damit ergibt sich für v der richtige Distanzwert

- ein Knoten x kann auch nie einen Wert erhalten, der echt kleiner als seine Distanz zu s ist
- die Kantenfolge von s zu x, die jeweils zu den Distanzwerten an den Knoten geführt hat, wäre dann ein kürzerer Pfad (Widerspruch)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Allgemeine Strategie:

- Anfang: setze $d(s) = 0$ und
  für alle anderen Knoten v setze $d(v) = \infty$
- besuche Knoten in einer Reihenfolge, die sicherstellt, dass mindestens ein kürzester Weg von s zu jedem v in der Reihenfolge seiner Knoten besucht wird
- für jeden besuchten Knoten v aktualisiere die Distanzen der Knoten w mit $(v, w) \in E$, d.h. setze

$$d(w) = \min\{\, d(w),\ d(v) + c(v, w) \,\}$$

Kürzeste Wege in DAGs

Topologische Sortierung

- verwende FIFO-Queue q
- verwalte für jeden Knoten einen Zähler für die noch nicht markierten eingehenden Kanten
- initialisiere q mit allen Knoten, die keine eingehende Kante haben (Quellen)
- nimm nächsten Knoten v aus q und
  markiere alle $(v, w) \in E$, d.h. dekrementiere Zähler für w
- falls der Zähler von w dabei Null wird, füge w in q ein
- wiederhole das, bis q leer wird

Kürzeste Wege in DAGs

Topologische Sortierung

Korrektheit

- Knoten wird erst dann nummeriert, wenn alle Vorgänger nummeriert sind

Laufzeit

- für die Anfangswerte der Zähler muss der Graph einmal traversiert werden $O(n+m)$
- danach wird jede Kante genau einmal betrachtet

$\Rightarrow$ gesamt: $O(n+m)$

Test auf DAG-Eigenschaft

- topologische Sortierung erfasst genau dann alle Knoten, wenn der Graph ein DAG ist
- bei gerichteten Kreisen erhalten diese Knoten keine Nummer

Kürzeste Wege in DAGs

DAG-Strategie

1. Topologische Sortierung der Knoten

   Laufzeit $O(n+m)$

2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

   Laufzeit $O(n+m)$

Gesamtlaufzeit: $O(n+m)$

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s