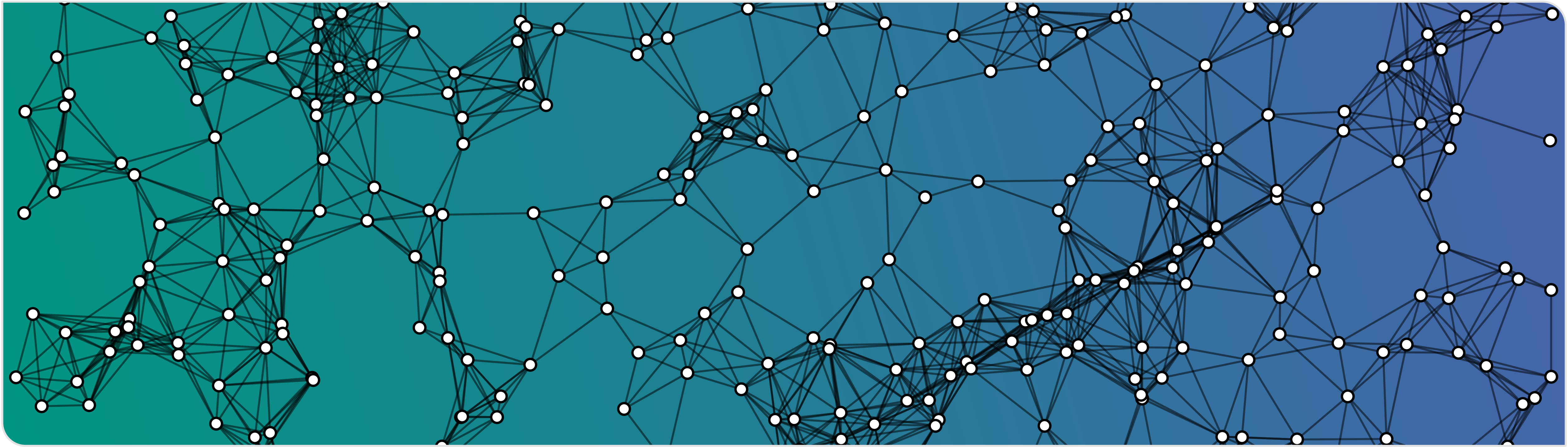
Karlsruher Institut für Technologie

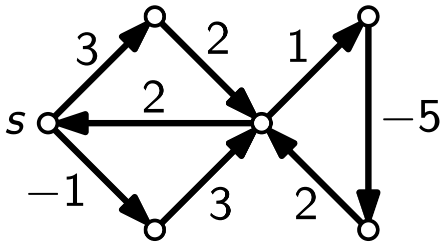
Algorithmen 1

Kürzeste Wege: negative Kanten und APSP



Kürzeste Pfade mit negativen Kantenlängen

Problem: Single-Source Shortest Path (SSSP)
Gegen einen gerichteten Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$ und $s \in V$, berechne $\mathrm{dist}(s, t)$ für alle $t \in V$.

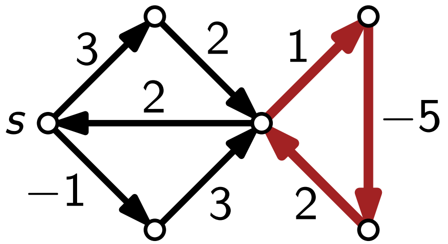


Kürzeste Pfade mit negativen Kantenlängen

Problem: Single-Source Shortest Path (SSSP)
Gegen einen gerichteten Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$ und $s \in V$, berechne $\mathrm{dist}(s, t)$ für alle $t \in V$.

Negative Kreise

- Pfad kann mehrfach im Kreis läuft
- beliebig kleine Länge möglich (Distanzen $-\infty$?)

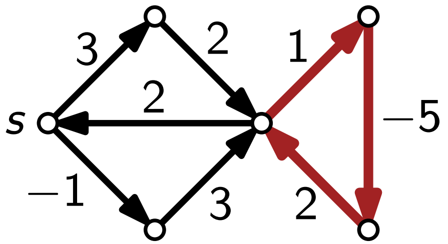


Kürzeste Pfade mit negativen Kantenlängen

Problem: Single-Source Shortest Path (SSSP)
Gegen einen gerichteten Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$ und $s \in V$, berechne $\mathrm{dist}(s, t)$ für alle $t \in V$.

Negative Kreise

- Pfad kann mehrfach im Kreis läuft
- beliebig kleine Länge möglich (Distanzen $-\infty$?)



Möglichkeit 1

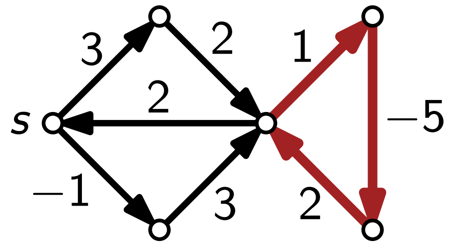
- erlaube nur **einfache Pfade**, die keine Knoten mehrfach besuchen
- Problem Hamilton-Pfad als Spezialfall

(vermutlich kein polynomieller Algorithmus möglich, siehe TGI nächstes Semester)

Kürzeste Pfade mit negativen Kantenlängen

Problem: Single-Source Shortest Path (SSSP)
Gegen einen gerichteten Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$ und $s \in V$, berechne $\mathrm{dist}(s, t)$ für alle $t \in V$.

Negative Kreise



- Pfad kann mehrfach im Kreis läuft
- beliebig kleine Länge möglich (Distanzen $-\infty$?)

Möglichkeit 1

- erlaube nur **einfache Pfade**, die keine Knoten mehrfach besuchen
- Problem Hamilton-Pfad als Spezialfall

(vermutlich kein polynomieller Algorithmus möglich, siehe TGI nächstes Semester)

Möglichkeit 2

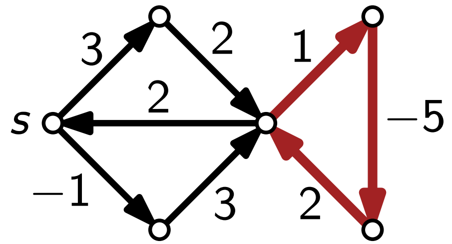
- erkenne negative Kreise
- Abbruch, wenn negativer Kreis gefunden

Kürzeste Pfade mit negativen Kantenlängen

Problem: Single-Source Shortest Path (SSSP)
Gegen einen gerichteten Graphen $G = (V, E)$, Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$ und $s \in V$, berechne $\mathrm{dist}(s, t)$ für alle $t \in V$.

Negative Kreise

- Pfad kann mehrfach im Kreis läuft
- beliebig kleine Länge möglich (Distanzen $-\infty$?)



Möglichkeit 1

- erlaube nur **einfache Pfade**, die keine Knoten mehrfach besuchen
- Problem Hamilton-Pfad als Spezialfall

(vermutlich kein polynomieller Algorithmus möglich, siehe TGI nächstes Semester)

Möglichkeit 2

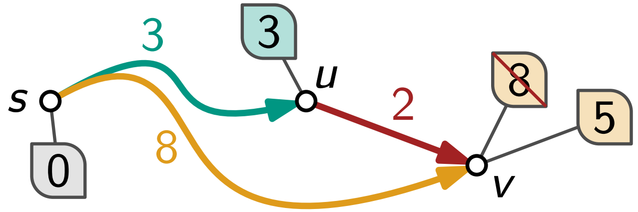
- erkenne negative Kreise
- Abbruch, wenn negativer Kreis gefunden

diese Variante betrachten wir im Folgenden
(daher auch gerichtete Graphen)

Erinnerung: Dijkstras Algorithmus

Grundsätzliches Vorgehen

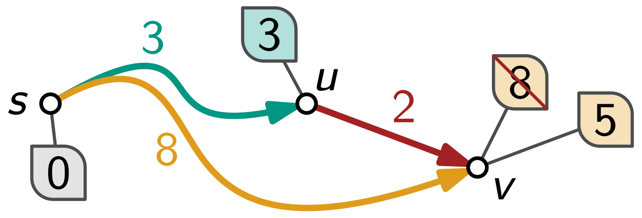
- speichere $d[v]$ für jeden Knoten v: Länge des kürzesten bekannten sv-Pfades
- **Relaxierung** der Kante (u, v): wenn $d[v] > d[u] + \text{len}(u, v)$, setze $d[v] = d[u] + \text{len}(u, v)$



Erinnerung: Dijkstras Algorithmus

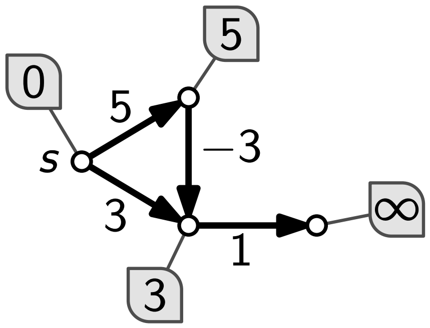
Grundsätzliches Vorgehen

- speichere $d[v]$ für jeden Knoten v: Länge des kürzesten bekannten sv-Pfades
- **Relaxierung** der Kante (u, v): wenn $d[v] > d[u] + \text{len}(u, v)$, setze $d[v] = d[u] + \text{len}(u, v)$



Reihenfolge der Relaxierungen bei Dijkstra

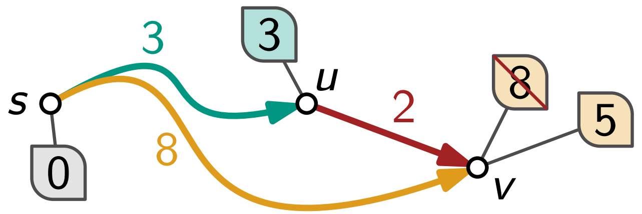
- Exploriere Knoten u: relaxiere alle ausgehenden Kanten von u
- exploriere Knoten aufsteigend nach $d[u]$
- Korrektheit: beim Explorieren gilt $d[u] = \text{dist}(s, u)$



Erinnerung: Dijkstras Algorithmus

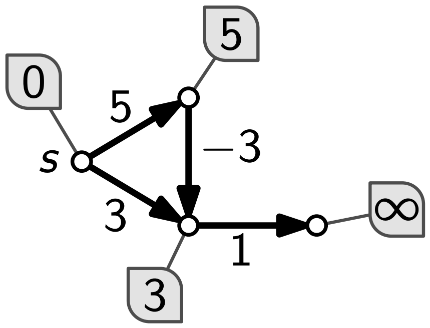
Grundsätzliches Vorgehen

- speichere $d[v]$ für jeden Knoten v: Länge des kürzesten bekannten sv-Pfades
- **Relaxierung** der Kante (u, v): wenn $d[v] > d[u] + \text{len}(u, v)$, setze $d[v] = d[u] + \text{len}(u, v)$



Reihenfolge der Relaxierungen bei Dijkstra

- Exploriere Knoten u: relaxiere alle ausgehenden Kanten von u
- exploriere Knoten aufsteigend nach $d[u]$
- Korrektheit: beim Explorieren gilt $d[u] = \text{dist}(s, u)$



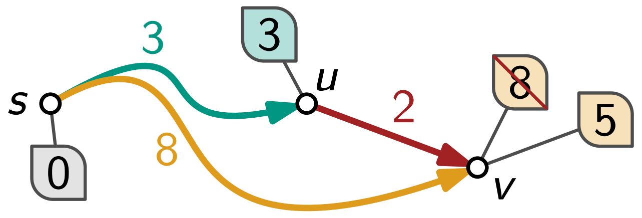
Mit negativen Kantenlängen

- Problem: $d[u]$ könnte später wegen negativem Pfad noch kleiner werden

Erinnerung: Dijkstras Algorithmus

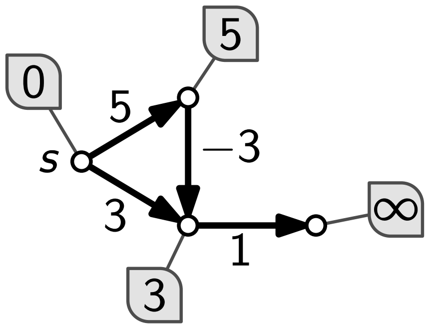
Grundsätzliches Vorgehen

- speichere $d[v]$ für jeden Knoten v: Länge des kürzesten bekannten sv-Pfades
- **Relaxierung** der Kante (u, v): wenn $d[v] > d[u] + \text{len}(u, v)$, setze $d[v] = d[u] + \text{len}(u, v)$



Reihenfolge der Relaxierungen bei Dijkstra

- Exploriere Knoten u: relaxiere alle ausgehenden Kanten von u
- exploriere Knoten aufsteigend nach $d[u]$
- Korrektheit: beim Explorieren gilt $d[u] = \text{dist}(s, u)$



Mit negativen Kantenlängen

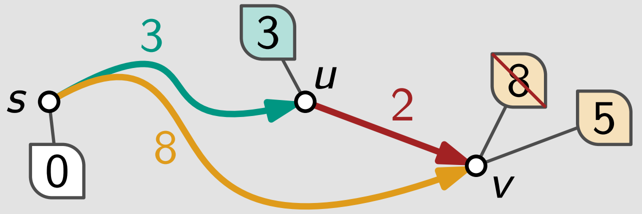
- Problem: $d[u]$ könnte später wegen negativem Pfad noch kleiner werden
- Hoffnung: einfach weiter relaxieren bis sich nichts mehr ändert führt zum Ziel

Algorithmus von Bellman und Ford

Grober Plan

- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Relaxierung der Kante (u, v)



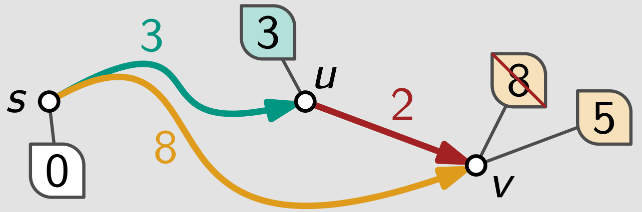
wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Algorithmus von Bellman und Ford

Grober Plan

- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Welche dieser Aussagen sind wahr?

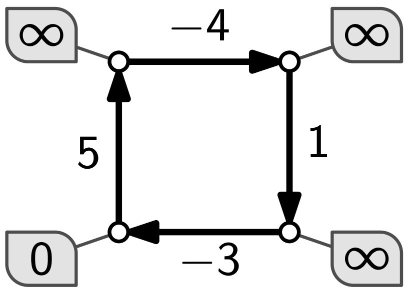
Algorithmus von Bellman und Ford

Grober Plan

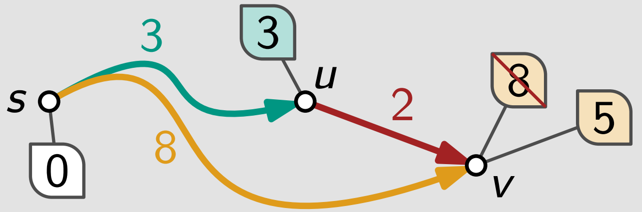
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

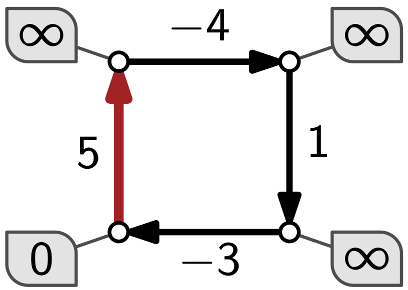


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

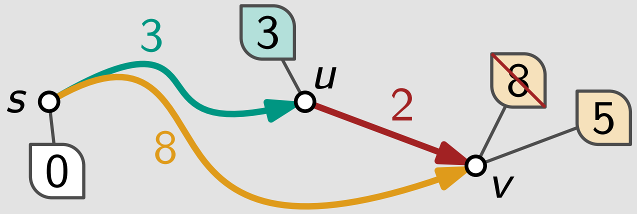
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

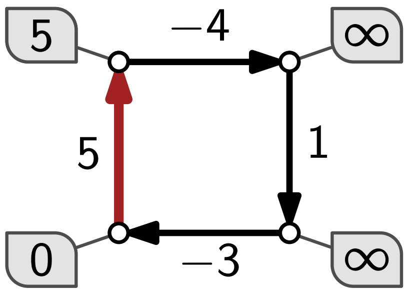
Algorithmus von Bellman und Ford

Grober Plan

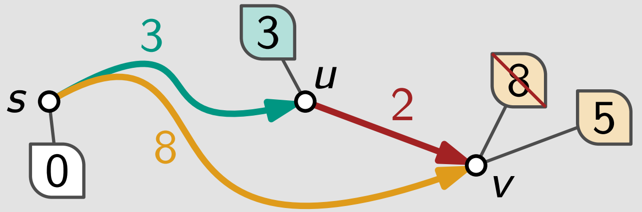
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

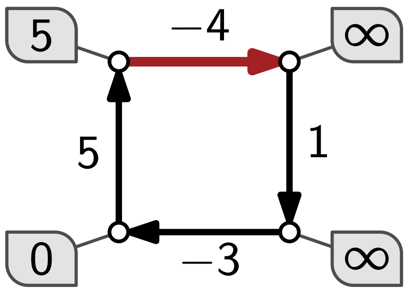


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

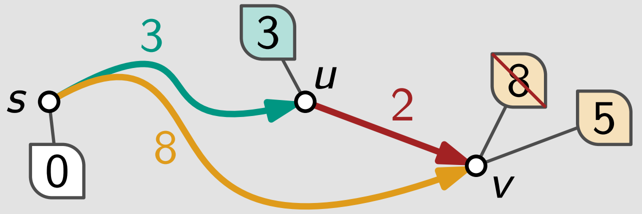
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

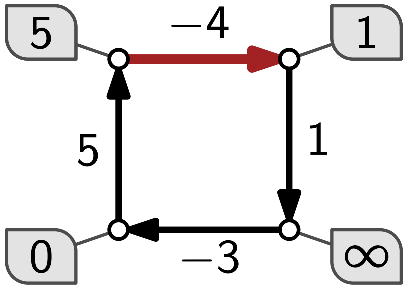


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

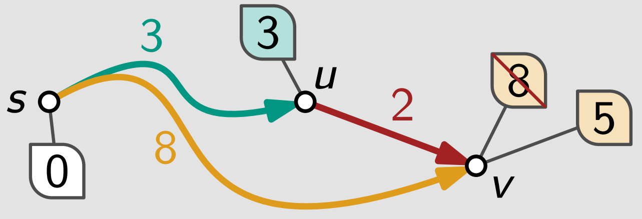
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

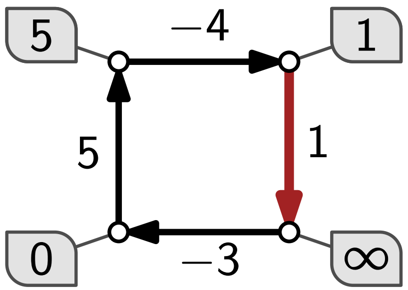
Algorithmus von Bellman und Ford

Grober Plan

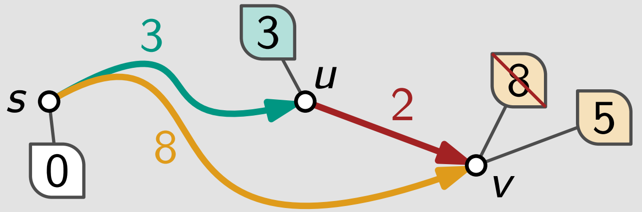
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

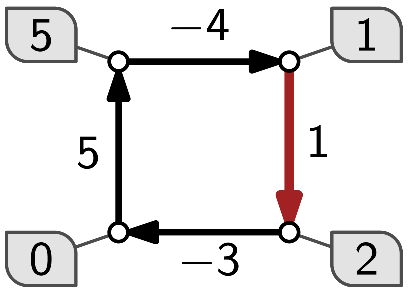
Algorithmus von Bellman und Ford

Grober Plan

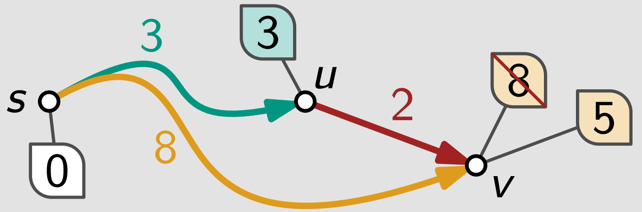
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

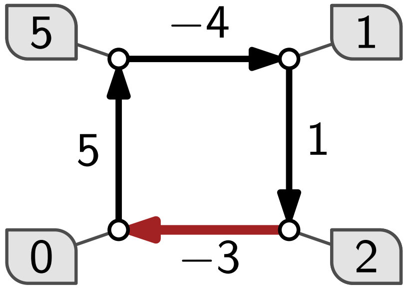
Algorithmus von Bellman und Ford

Grober Plan

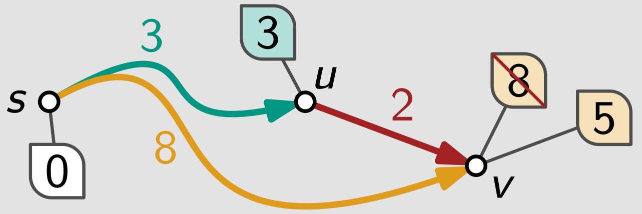
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

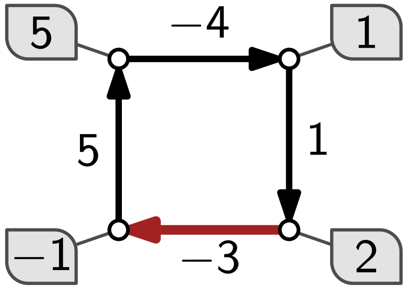
Algorithmus von Bellman und Ford

Grober Plan

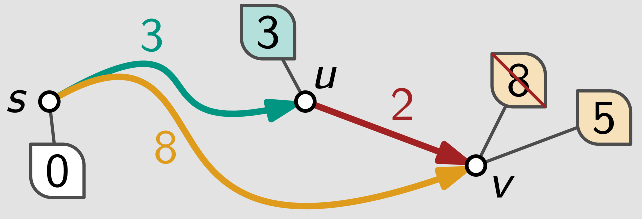
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

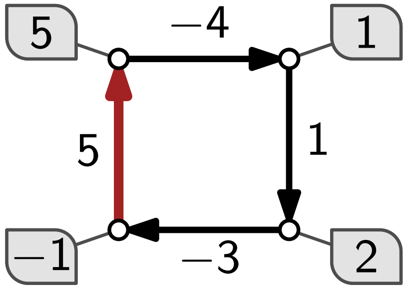


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

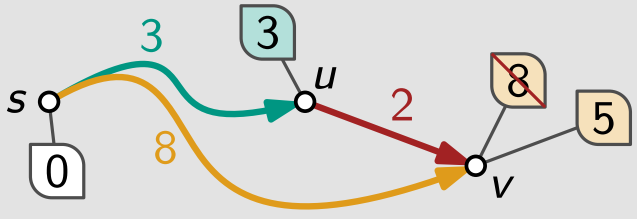
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

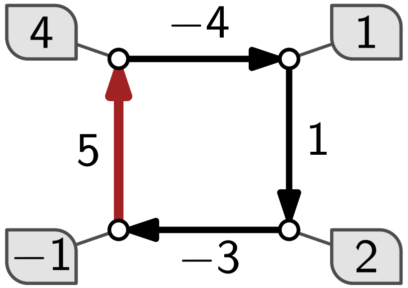
Algorithmus von Bellman und Ford

Grober Plan

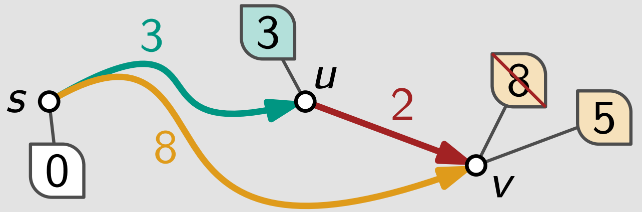
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

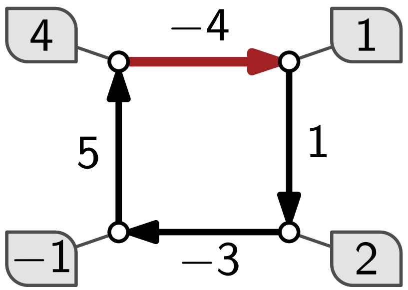
Algorithmus von Bellman und Ford

Grober Plan

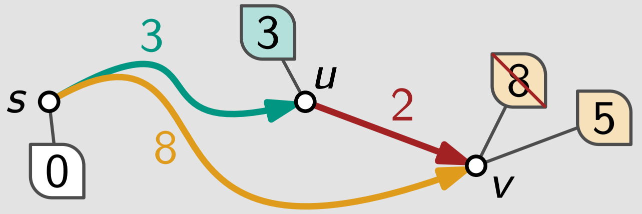
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer,
  je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

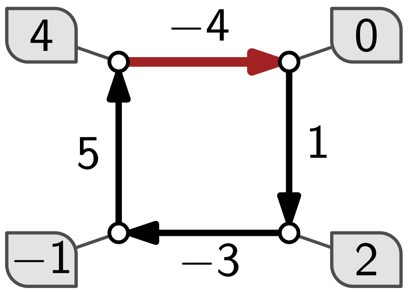
Algorithmus von Bellman und Ford

Grober Plan

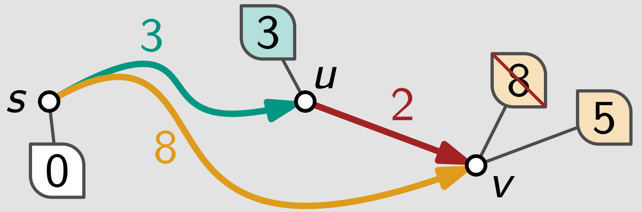
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

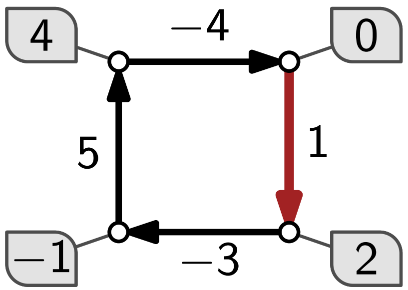


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

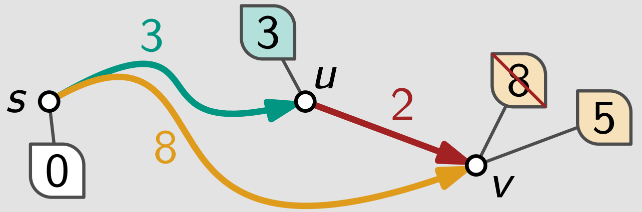
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

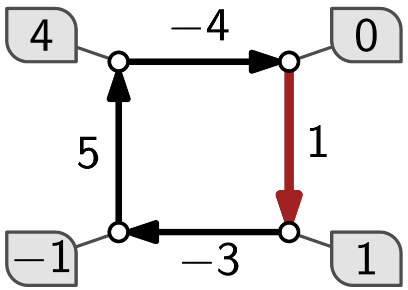
Algorithmus von Bellman und Ford

Grober Plan

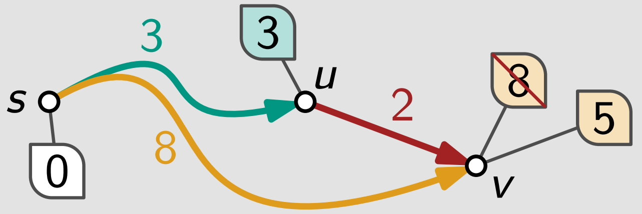
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer,
  je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)

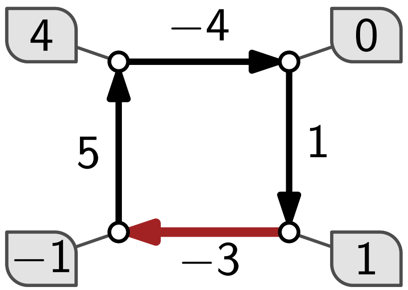


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

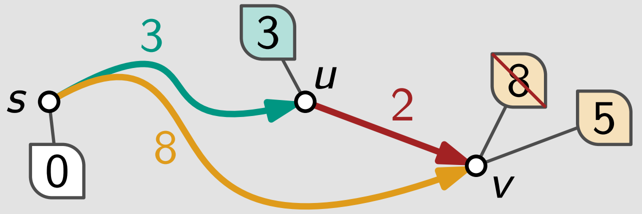
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer,
  je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Algorithmus von Bellman und Ford

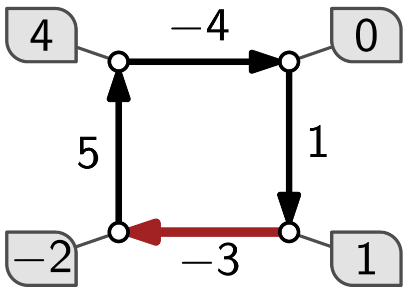
Karlsruher Institut für Technologie

Grober Plan

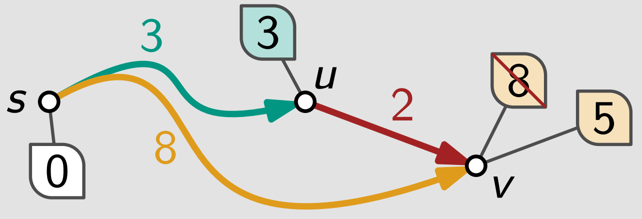
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

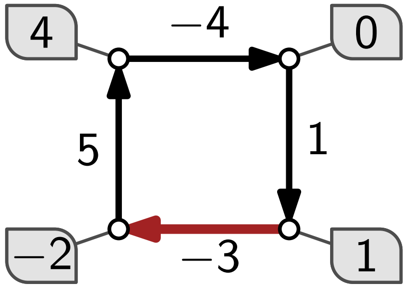
Algorithmus von Bellman und Ford

Grober Plan

- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise

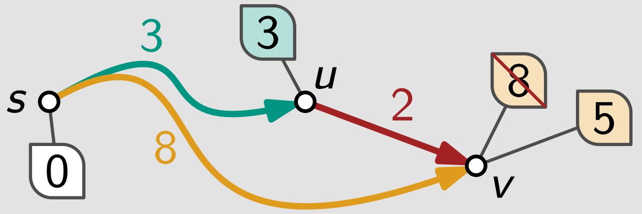
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Ohne negative Kreise

- in jeder Iteration wird die Gesamtsumme der $d[v]$ um mindestens 1 kleiner
- die Gesamtsumme der finalen Lösung ist von unten beschränkt ($\neq -\infty$)
- also: Terminierung nach endlich vielen Iterationen

Relaxierung der Kante (u, v)

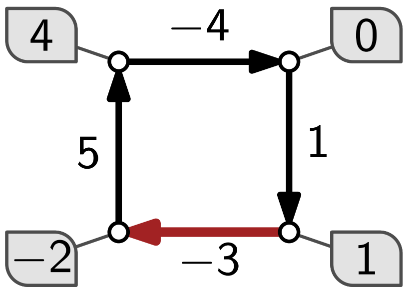


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

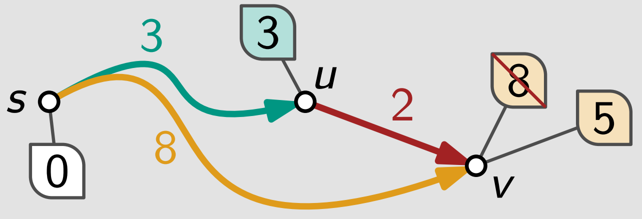
Algorithmus von Bellman und Ford

Grober Plan
- relaxiere jede Kante einmal
- iteriere, bis sich keines der $d[v]$ mehr ändert

Problemfall negative Kreise
- Pfade werden immer kürzer, je länger man im Kreis läuft
- terminiert nicht



Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

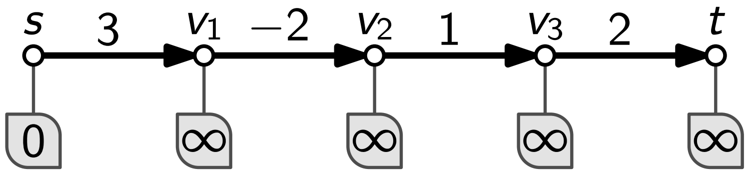
Ohne negative Kreise
- in jeder Iteration wird die Gesamtsumme der $d[v]$ um mindestens 1 kleiner
- die Gesamtsumme der finalen Lösung ist von unten beschränkt ($\neq -\infty$)
- also: Terminierung nach endlich vielen Iterationen

Ziel im Folgenden
- beweise obere Schranke für die Anzahl Iterationen
- Erkennung negativer Kreise

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

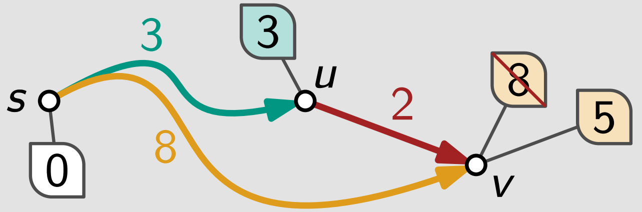
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

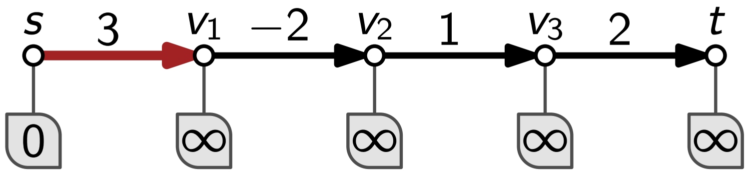


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

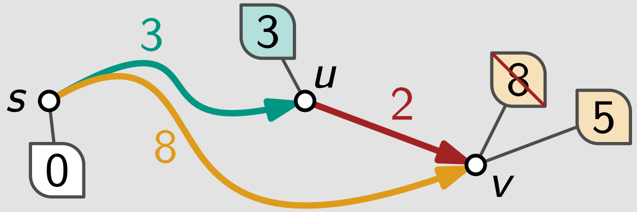
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

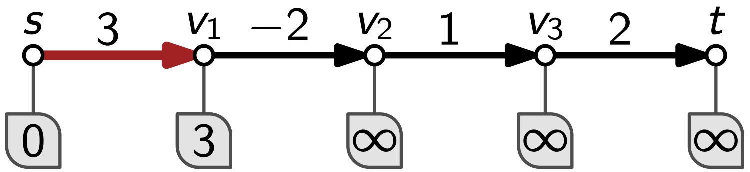


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

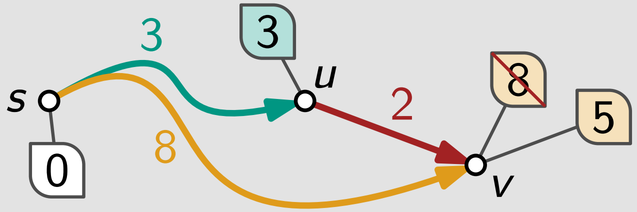
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

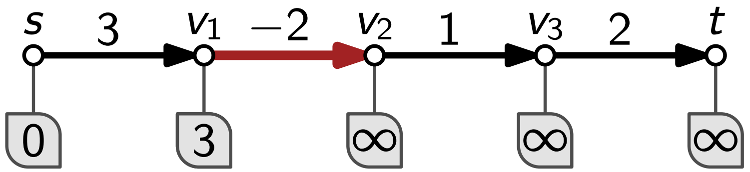


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

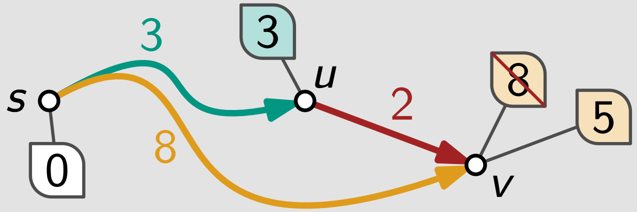
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

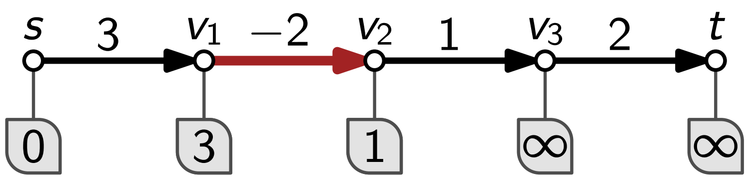


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten *st*-Pfad

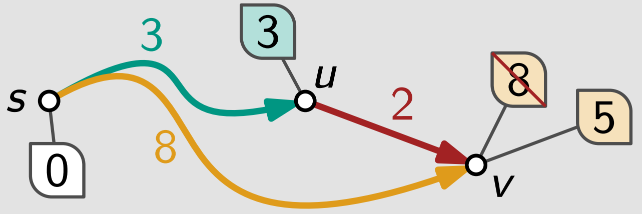
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

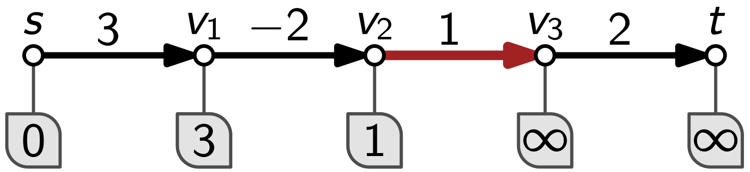


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

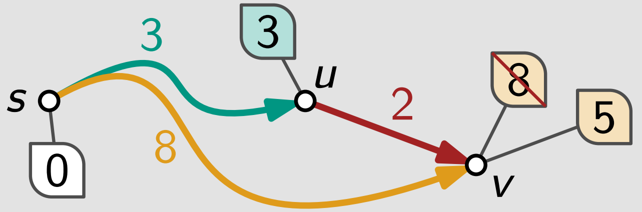
Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

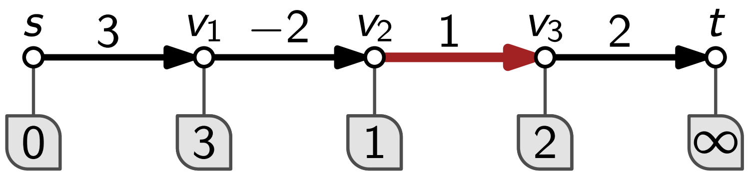


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

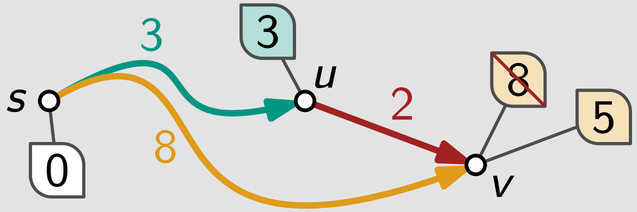
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

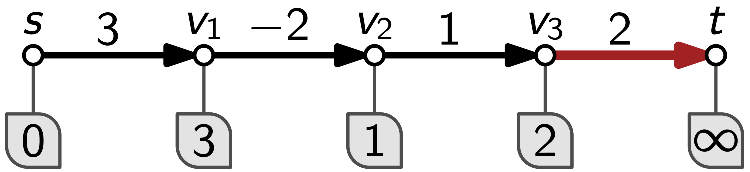


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

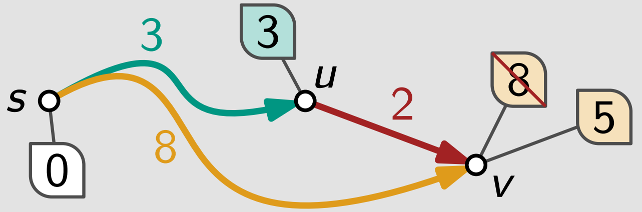
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

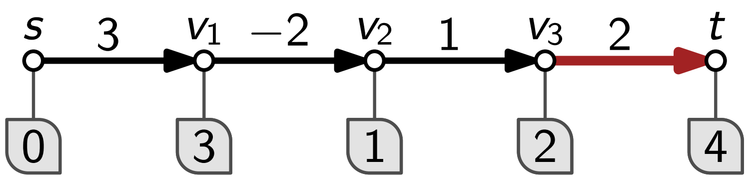


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

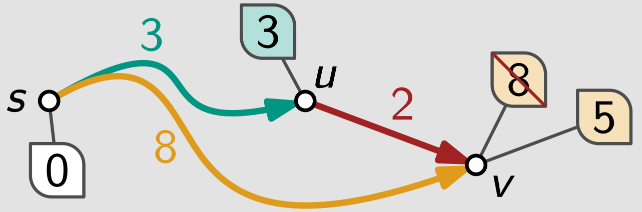
- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

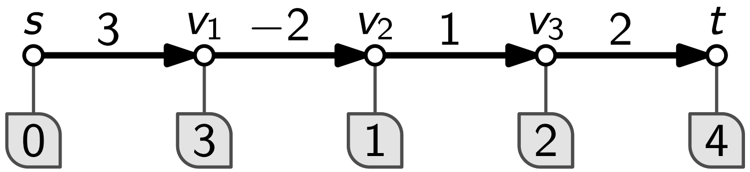


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig

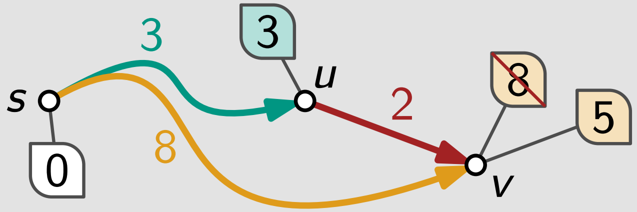


- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist

Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

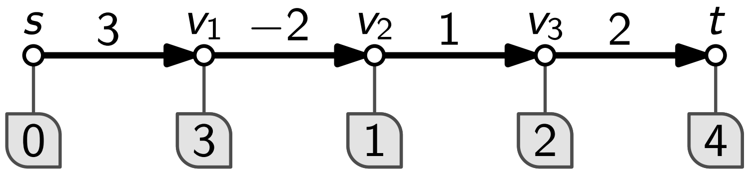


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

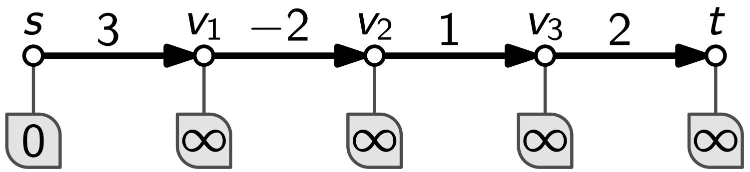
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



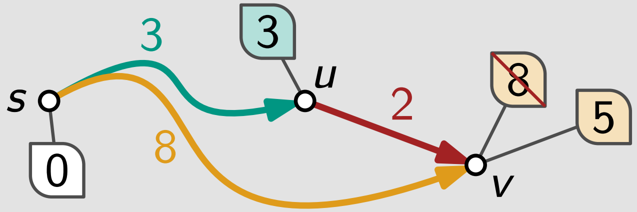
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)



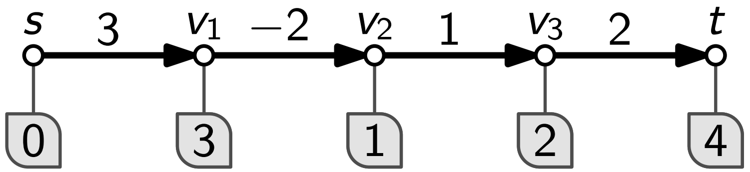
wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

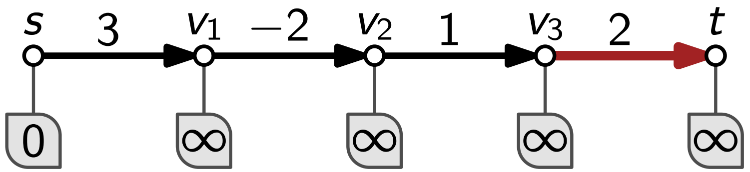
Karlsruher Institut für Technologie

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



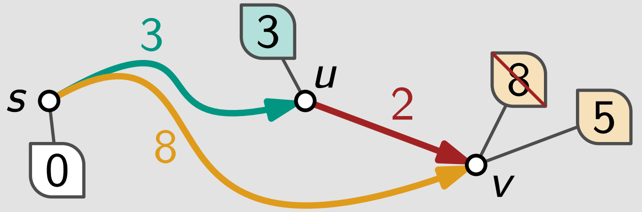
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

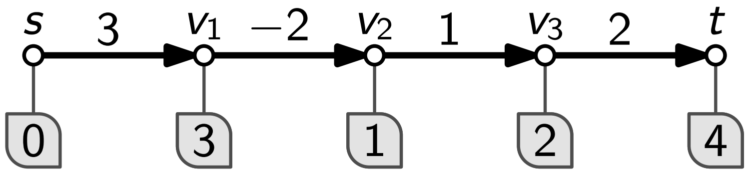


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

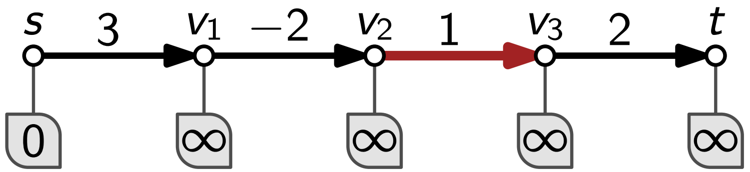
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



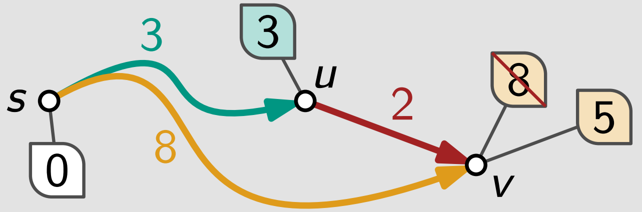
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

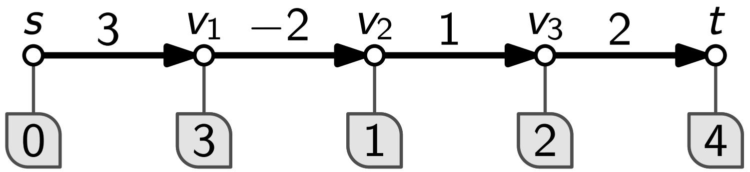


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

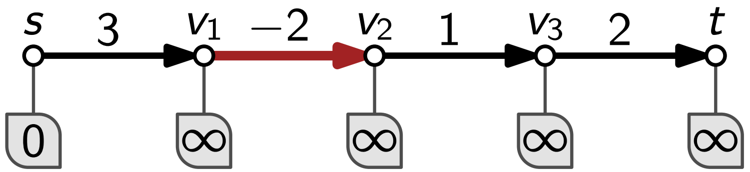
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten *st*-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



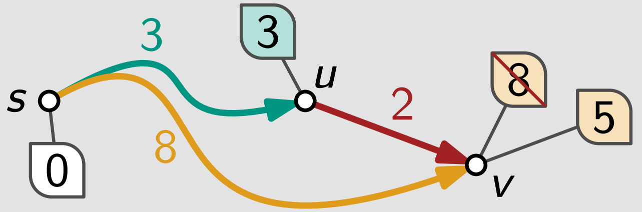
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

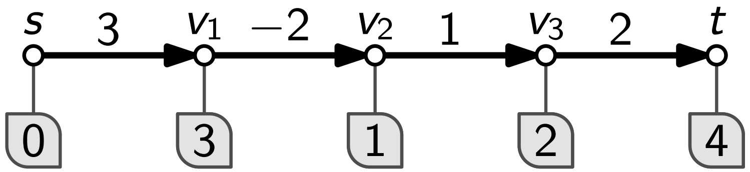


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

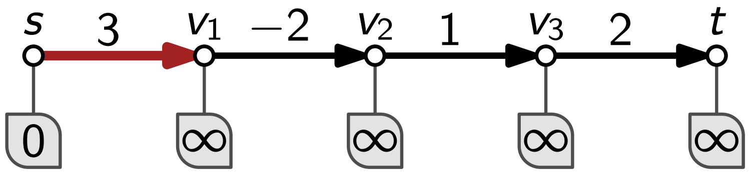
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig

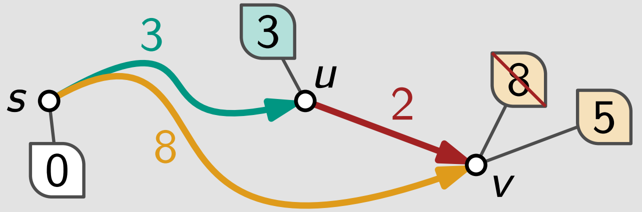


- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

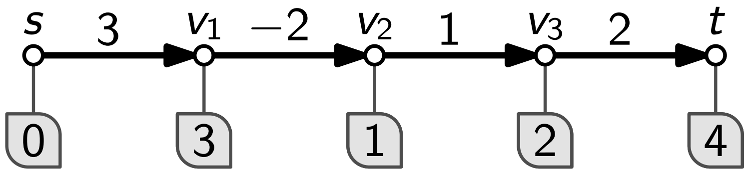


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

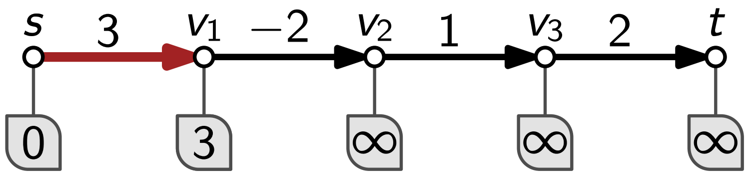
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



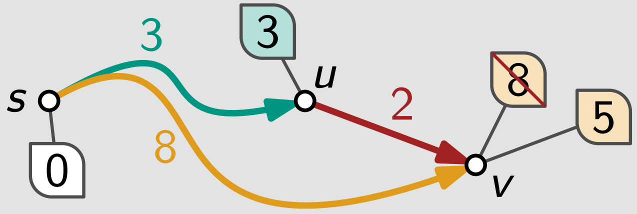
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

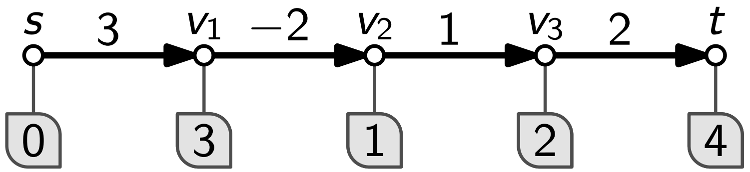


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

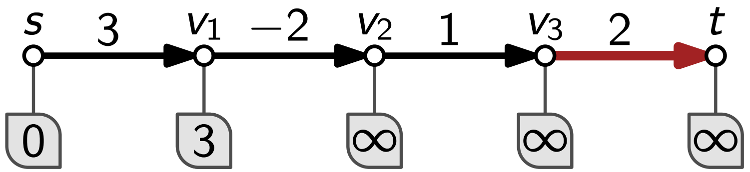
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



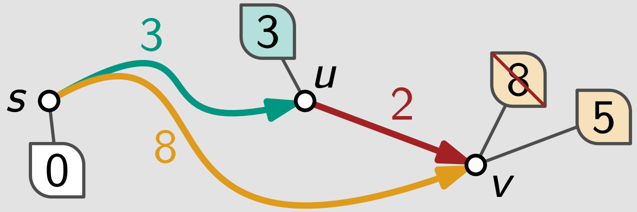
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

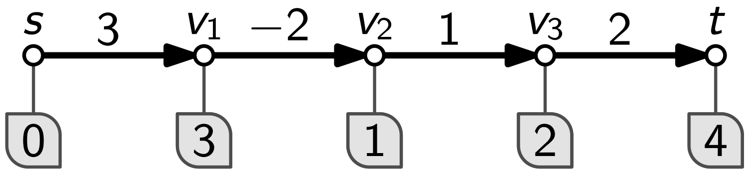


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

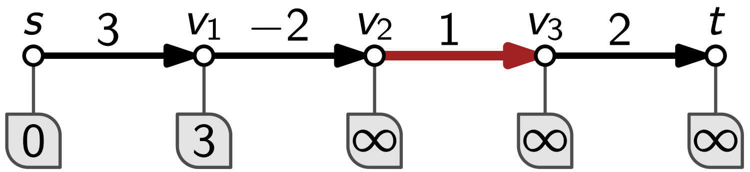
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



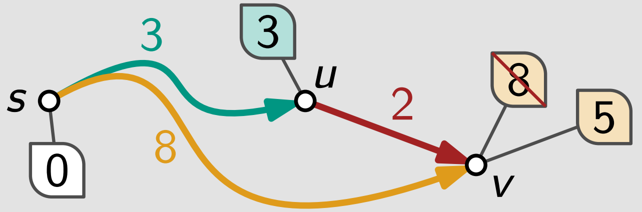
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

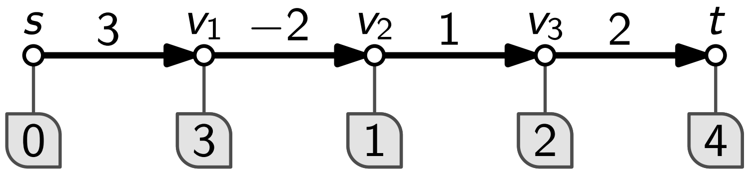


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

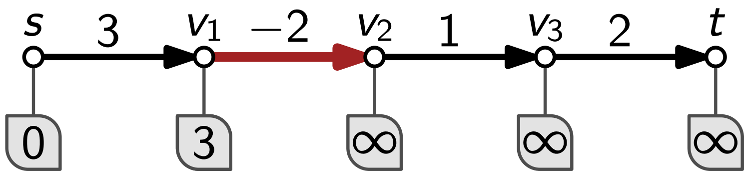
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten *st*-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig

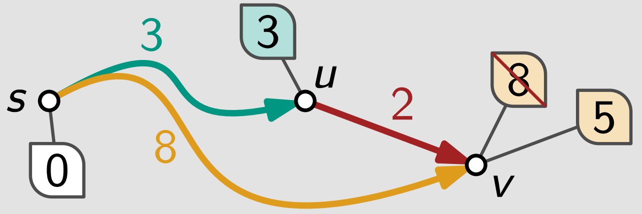


- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

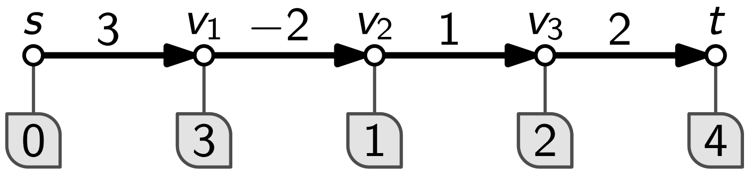


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

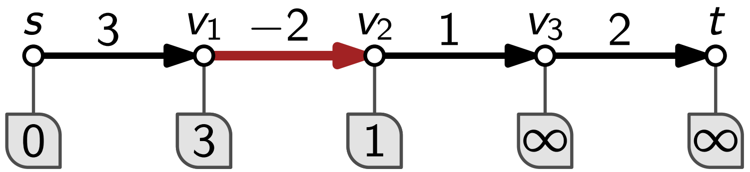
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten *st*-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



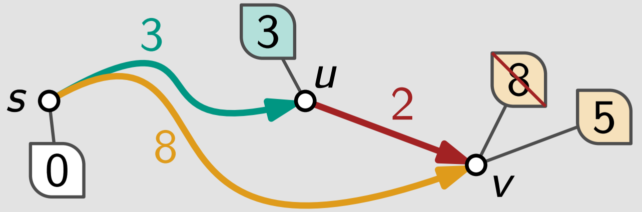
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

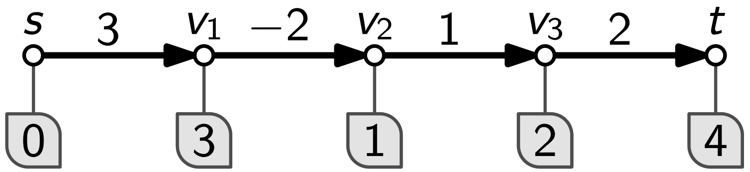


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

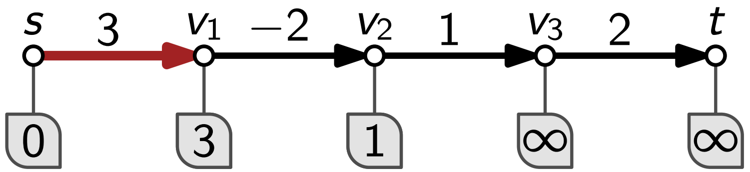
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



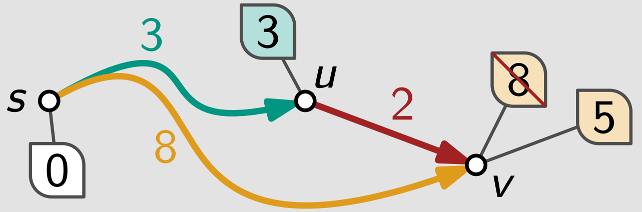
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

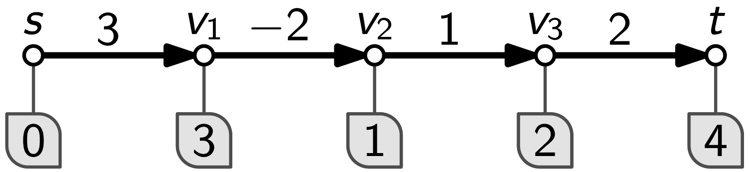


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

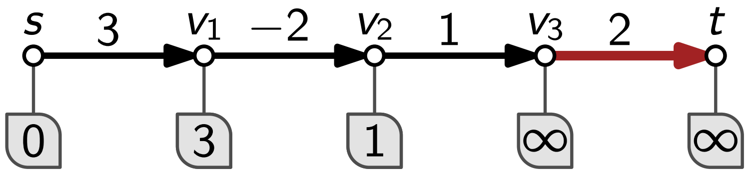
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



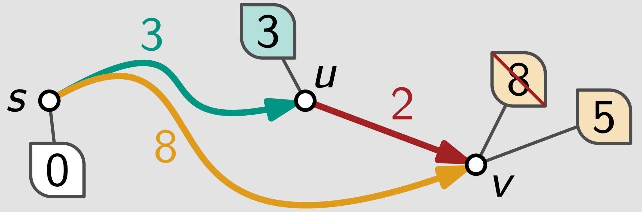
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

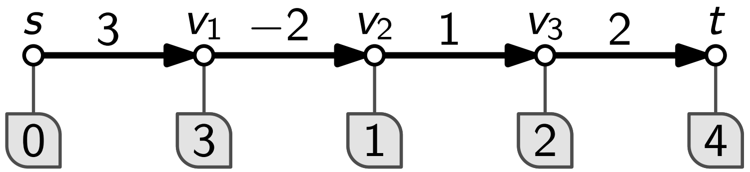


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

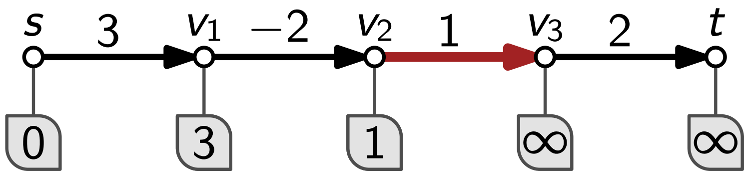
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



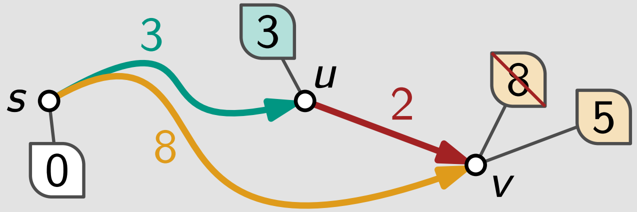
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

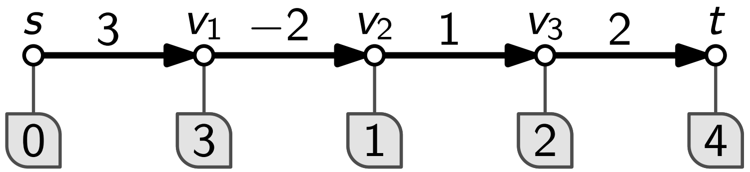


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

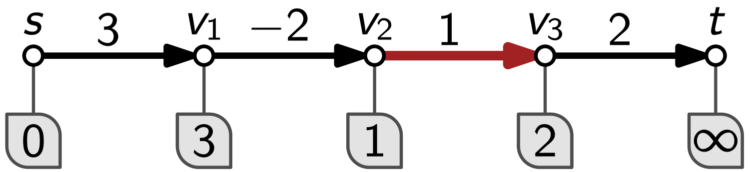
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



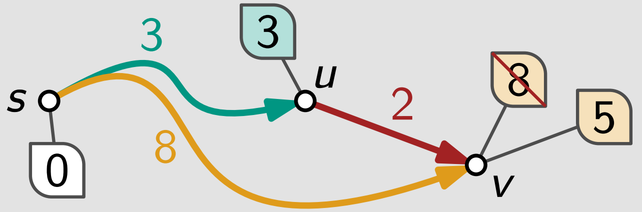
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

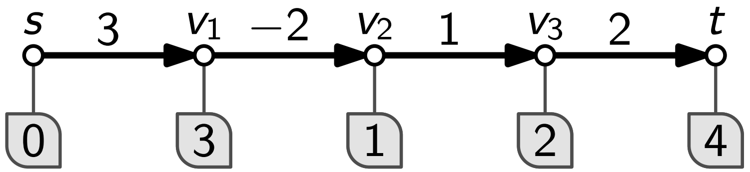


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

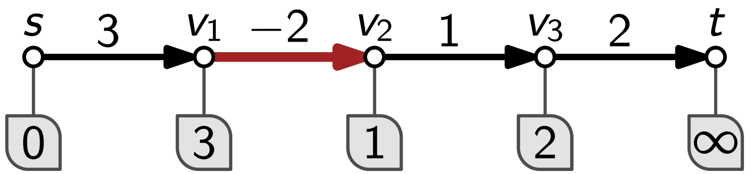
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



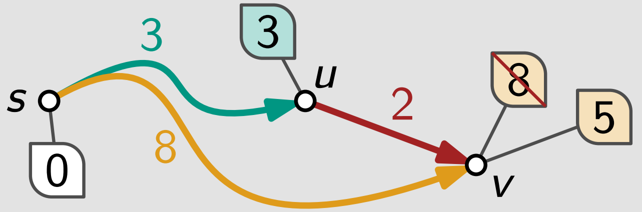
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)



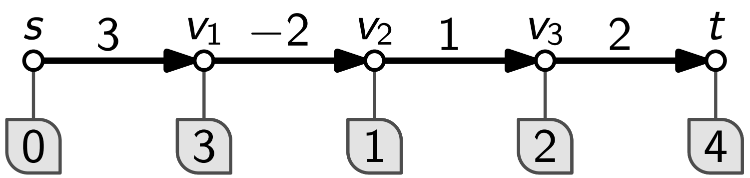
wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Wie viele Iterationen sind nötig?

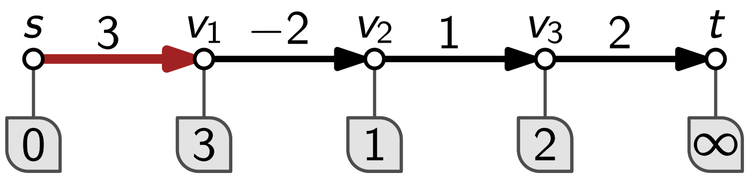
Karlsruher Institut für Technologie

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



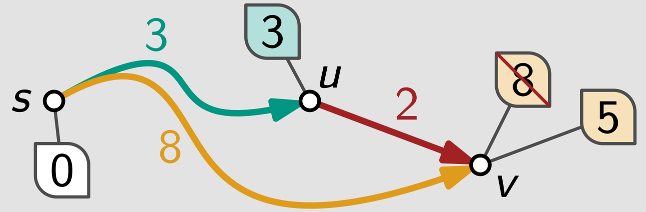
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

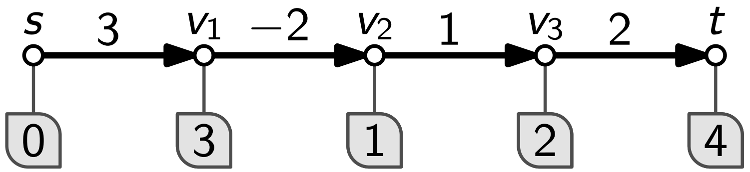


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

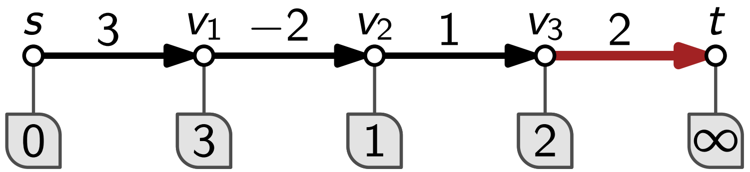
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



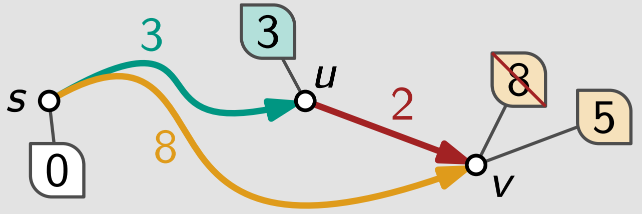
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

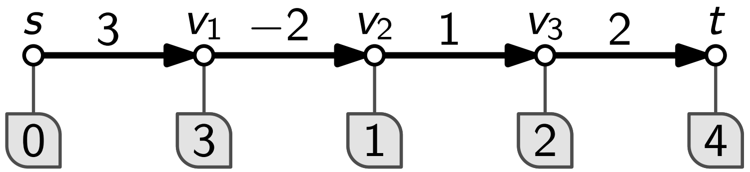


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

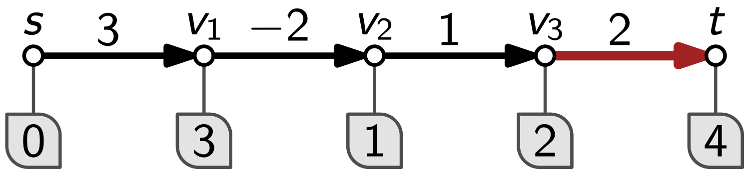
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



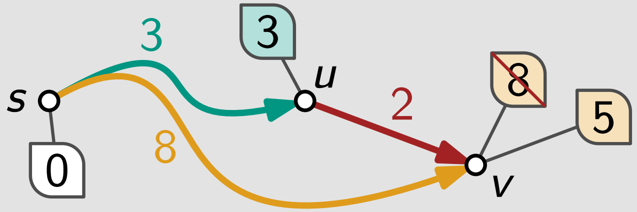
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

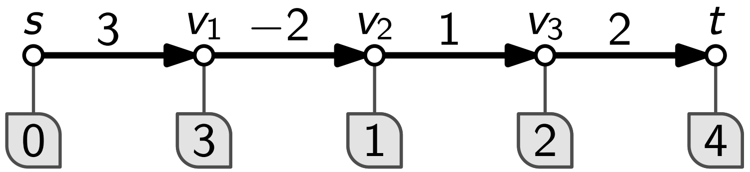


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

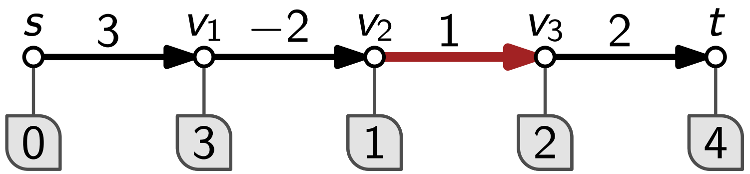
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



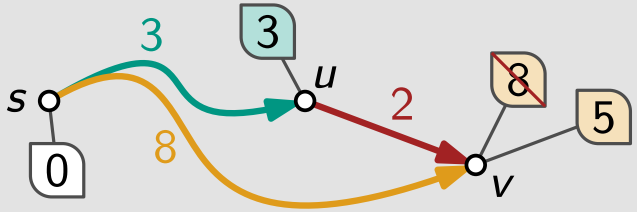
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

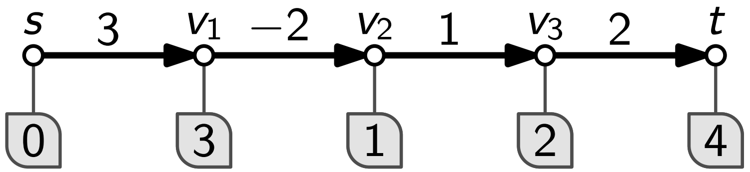


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

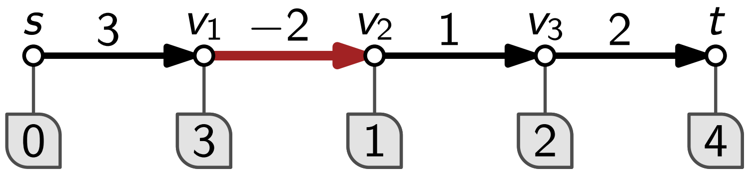
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



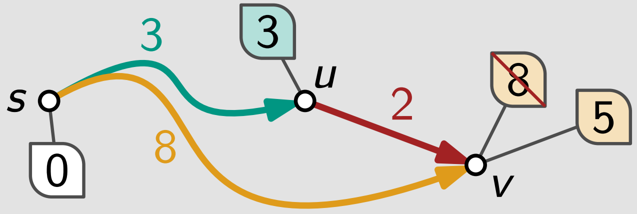
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

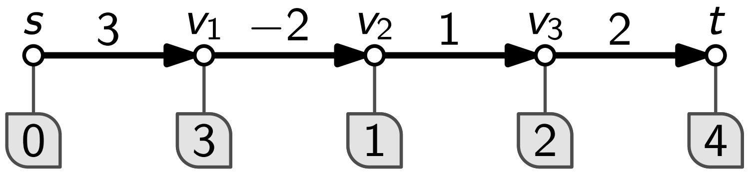


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

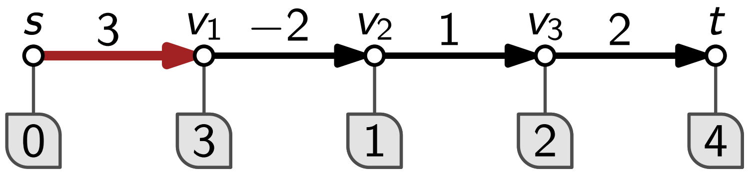
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



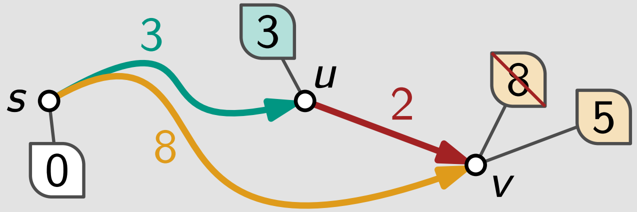
- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

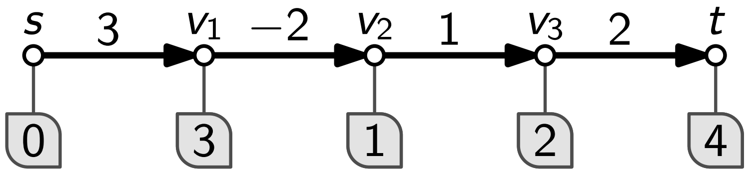


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

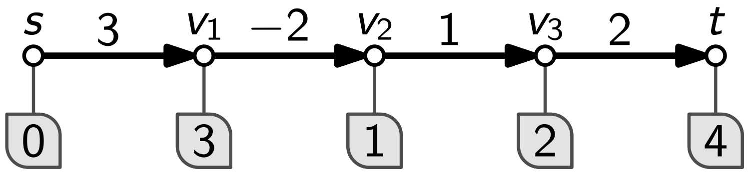
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne

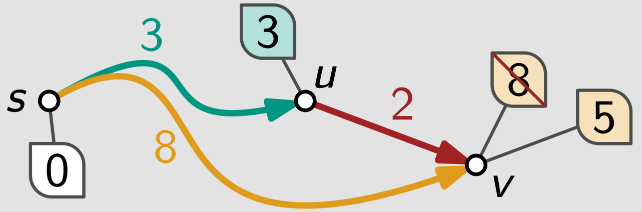


- nach i Iterationen: korrekte Distanz bis v_i
- kürzester st-Pfad besteht aus k Kanten $\Rightarrow$ höchstens k Iterationen

Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

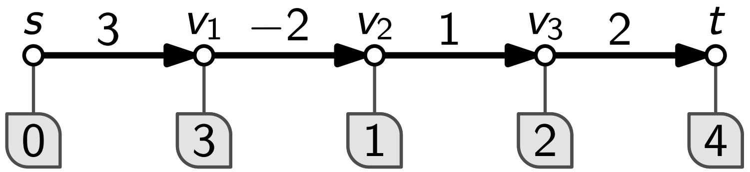


wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

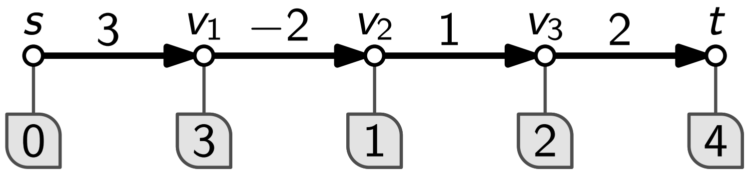
Wie viele Iterationen sind nötig?

Betrachte einen kürzesten st-Pfad

- **Best-Case:** wir relaxieren von vorne nach hinten
- sind nach einer Iteration schon fertig



- Problem: wir wissen vorher nicht, welche Reihenfolge gut ist
- **Worst-Case:** wir relaxieren von hinten nach vorne



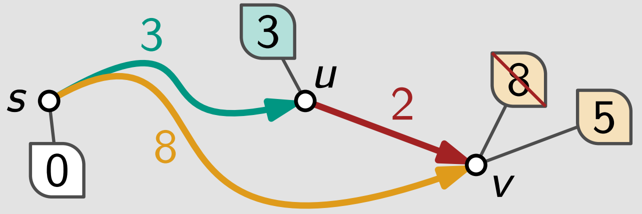
- nach i Iterationen: korrekte Distanz bis v_i
- kürzester st-Pfad besteht aus k Kanten $\Rightarrow$ höchstens k Iterationen

$\Rightarrow n-1$ Iterationen sollten ausreichen
(werden wir im Folgenden noch formal beweisen)

Algorithmus

- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

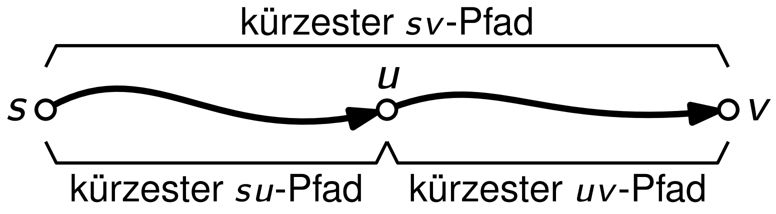
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

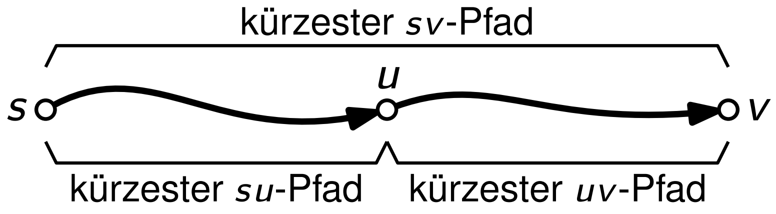
Beweis: $n - 1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.



Beweis: $n - 1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

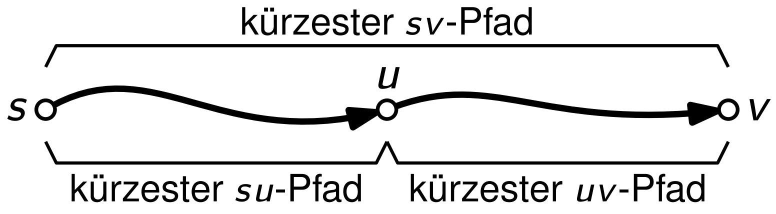


Begründung

- wenn es einen kürzeren su-Pfad gäbe
- dann wäre dieser zusammen mit dem uv-Pfad auch ein kürzerer sv-Pfad

Beweis: $n-1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

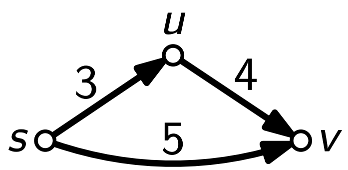


Begründung

- wenn es einen kürzeren su-Pfad gäbe
- dann wäre dieser zusammen mit dem uv-Pfad auch ein kürzerer sv-Pfad

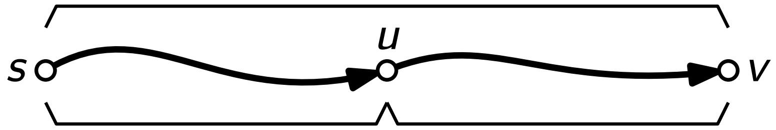
Achtung: Umkehrung gilt nicht

- kürzeste su- und uv-Pfade bilden zusammen **nicht** zwangsweise einen kürzesten sv-Pfad
- das gilt nur, wenn u auch auf dem kürzesten sv-Pfad liegt



Beweis: $n-1$ Iterationen genügen

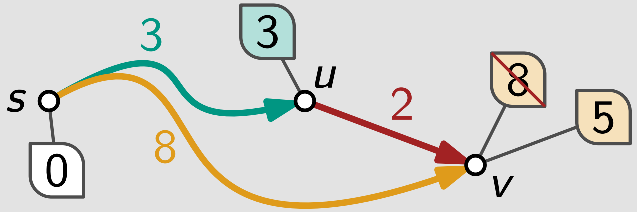
Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.



Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \mathrm{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

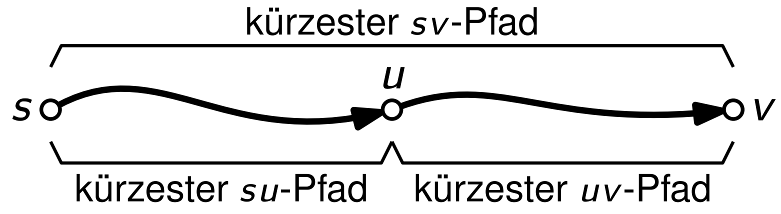
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Beweis: $n - 1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

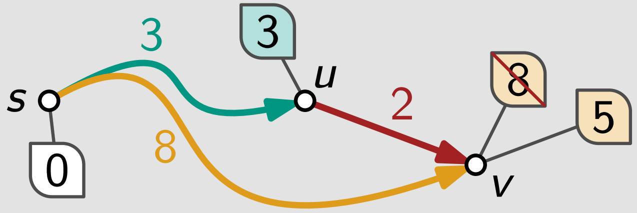


Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \mathrm{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Beweis: Induktion über k
- $k = 1$: Kante (s, v) ist kürzester Pfad $\Rightarrow$ eine Iteration genügt

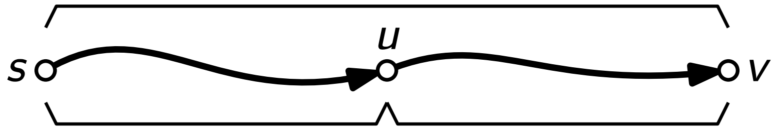
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Beweis: $n-1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

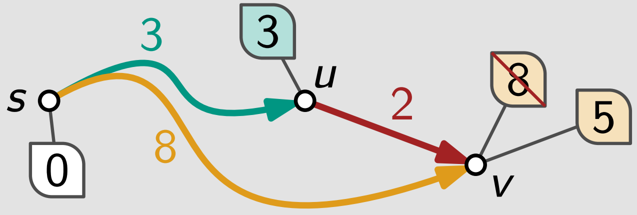


Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \mathrm{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Beweis: Induktion über k
- $k = 1$: Kante (s, v) ist kürzester Pfad $\Rightarrow$ eine Iteration genügt
- $k > 1$: betrachte Vorgänger u von v auf kürzestem sv-Pfad
- es gibt kürzesten su-Pfad mit $k-1$ Kanten

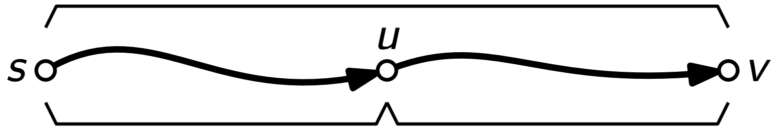
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Beweis: $n-1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

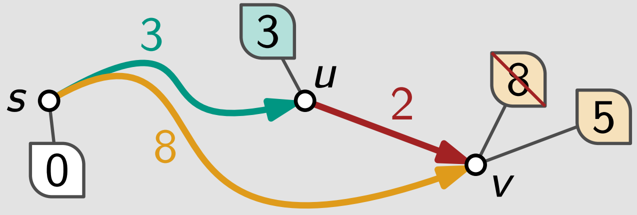


Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \text{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Beweis: Induktion über k
- $k = 1$: Kante (s, v) ist kürzester Pfad $\Rightarrow$ eine Iteration genügt
- $k > 1$: betrachte Vorgänger u von v auf kürzestem sv-Pfad
- es gibt kürzesten su-Pfad mit $k-1$ Kanten
- I.V. $\Rightarrow d[u] = \text{dist}(s, u)$ nach $k-1$ Iterationen

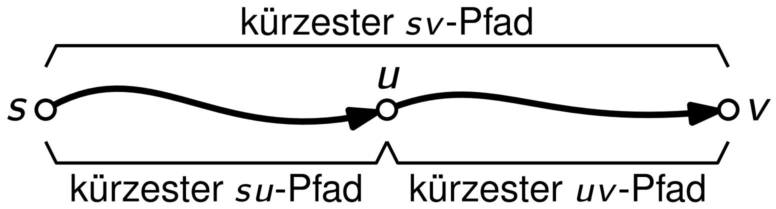
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Beweis: $n - 1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

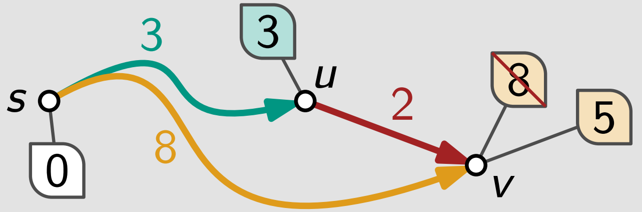


Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \text{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Beweis: Induktion über k
- $k = 1$: Kante (s, v) ist kürzester Pfad $\Rightarrow$ eine Iteration genügt
- $k > 1$: betrachte Vorgänger u von v auf kürzestem sv-Pfad
- es gibt kürzesten su-Pfad mit $k - 1$ Kanten
- I.V. $\Rightarrow d[u] = \text{dist}(s, u)$ nach $k - 1$ Iterationen
- Relaxierung von (u, v) in $k$ter Iteration setzt $d[v] = \text{dist}(s, v)$

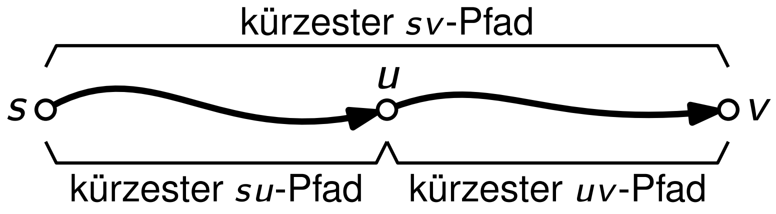
Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Beweis: $n - 1$ Iterationen genügen

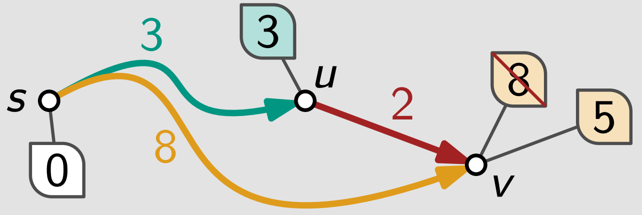
Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.



Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \mathrm{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)

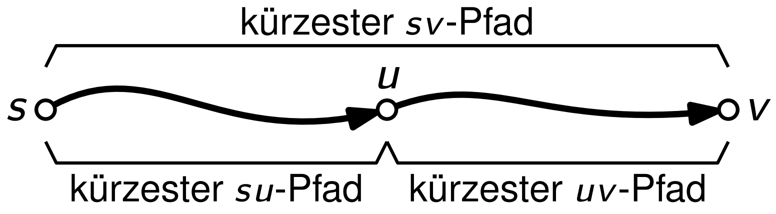


wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Beachte
- kürzester Pfad hat höchstens n Knoten und $n - 1$ Kanten
- (vorausgesetzt, wir haben keine negativen Kreise)

Beweis: $n - 1$ Iterationen genügen

Beobachtung
Teilpfade kürzester Pfade sind kürzeste Pfade.

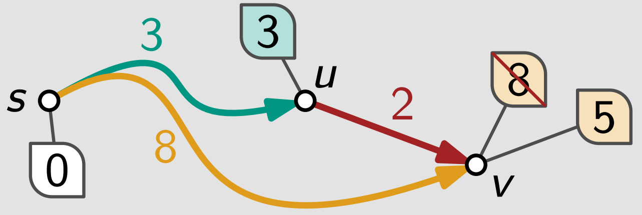


Lemma
Wenn es einen kürzesten sv-Pfad gibt, der aus k Kanten besteht, dann gilt $d[v] = \text{dist}(s, v)$ nach k Iterationen.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Beachte
- kürzester Pfad hat höchstens n Knoten und $n - 1$ Kanten
- (vorausgesetzt, wir haben keine negativen Kreise)

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

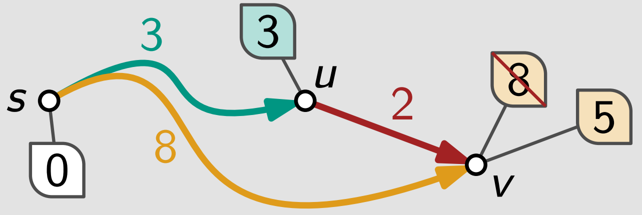
Theorem
Wenn der Graph keine negativen Kreise hat, so gilt nach $n - 1$ Iterationen für alle Knoten v, dass $d[v] = \text{dist}(s, v)$.

Erkennung von negativen Kreisen

Theorem
Wenn der Graph keine negativen Kreise hat, so gilt nach $n - 1$ Iterationen für alle Knoten v, dass $d[v] = \mathrm{dist}(s, v)$.

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Der Graph enthält einen negativen Kreis genau dann wenn…

Erkennung von negativen Kreisen

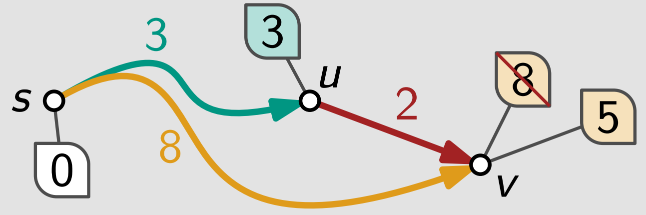
Theorem
Wenn der Graph keine negativen Kreise hat, so gilt nach $n-1$ Iterationen für alle Knoten v, dass $d[v] = \text{dist}(s, v)$.

Situation: nach $n-1$ Iterationen

Wenn es keinen negativen Kreis gibt
- weitere Relaxierungen ändern nichts mehr
- für jede Kante $(u, v) \in E$ gilt: $d[v] \leq d[u] + \text{len}(u, v)$

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Erkennung von negativen Kreisen

Theorem
Wenn der Graph keine negativen Kreise hat, so gilt nach $n-1$ Iterationen für alle Knoten v, dass $d[v] = \text{dist}(s, v)$.

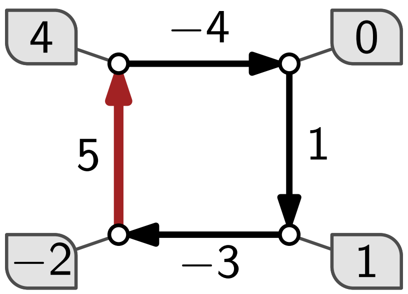
Situation: nach $n-1$ Iterationen

Wenn es keinen negativen Kreis gibt

- weitere Relaxierungen ändern nichts mehr
- für jede Kante $(u, v) \in E$ gilt: $d[v] \leq d[u] + \text{len}(u, v)$

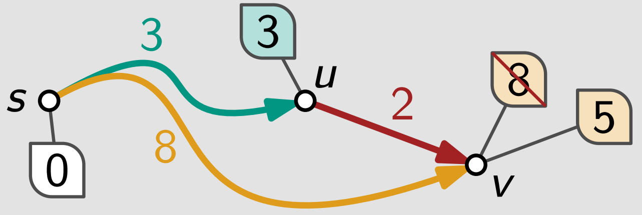
Wenn es negativen Kreis gibt

- man kann immer kürzere Pfade finden
- Relaxierungen haben weiterhin Effekt
- es gibt $(u, v) \in E$ mit: $d[v] > d[u] + \text{len}(u, v)$



Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Erkennung von negativen Kreisen

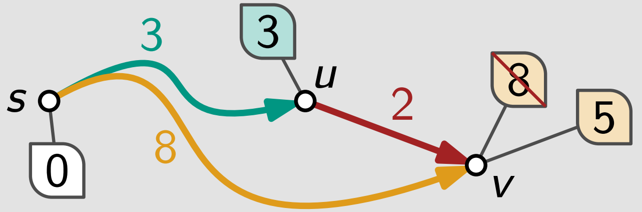
Theorem
Wenn der Graph keine negativen Kreise hat, so gilt nach $n-1$ Iterationen für alle Knoten v, dass $d[v] = \mathrm{dist}(s, v)$.

Situation: nach $n-1$ Iterationen

Algorithmus
- relaxiere jede Kante einmal
- iteriere, bis sich kein $d[v]$ ändert

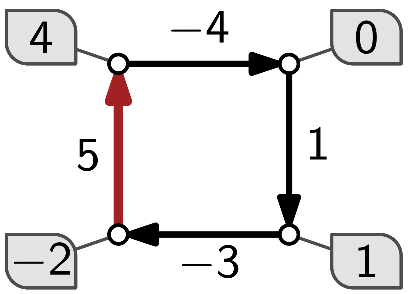
Wenn es keinen negativen Kreis gibt
- weitere Relaxierungen ändern nichts mehr
- für jede Kante $(u, v) \in E$ gilt: $d[v] \leq d[u] + \mathrm{len}(u, v)$

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \mathrm{len}(u, v)$
setze $d[v] = d[u] + \mathrm{len}(u, v)$

Wenn es negativen Kreis gibt
- man kann immer kürzere Pfade finden
- Relaxierungen haben weiterhin Effekt
- es gibt $(u, v) \in E$ mit: $d[v] > d[u] + \mathrm{len}(u, v)$

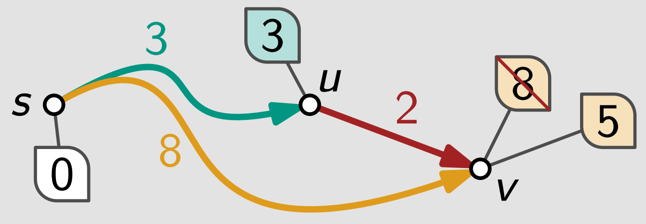


Test auf negativen Kreis: überprüfe ob $d[v] \leq d[u] + \mathrm{len}(u, v)$ für alle $(u, v) \in E$

Pseudocode und Laufzeit

```
BellmanFord(Graph G, Node s)
    d := Array of size n initialized with ∞
    d[s] := 0
    for n − 1 iterations do
        for Edge (u, v) ∈ E do
            if d[v] > d[u] + len(u, v) then
                d[v] := d[u] + len(u, v)
    // test for negative cycle
    for Edge (u, v) ∈ E do
        if d[v] > d[u] + len(u, v) then
            return negative cycle
    return d
```

Relaxierung der Kante (u, v)



wenn $d[v] > d[u] + \text{len}(u, v)$
setze $d[v] = d[u] + \text{len}(u, v)$

Pseudocode und Laufzeit

```
BellmanFord(Graph G, Node s)
    d := Array of size n initialized with ∞
    d[s] := 0
    for n − 1 iterations do
        for Edge (u, v) ∈ E do
            if d[v] > d[u] + len(u, v) then
                d[v] := d[u] + len(u, v)
    // test for negative cycle
    for Edge (u, v) ∈ E do
        if d[v] > d[u] + len(u, v) then
            return negative cycle
    return d
```

Laufzeit

- n mal über alle Kanten iterieren
- damit: $\Theta(n \cdot m)$

Pseudocode und Laufzeit

```
BellmanFord(Graph G, Node s)
    d := Array of size n initialized with ∞
    d[s] := 0
    for n − 1 iterations do
        for Edge (u, v) ∈ E do
            if d[v] > d[u] + len(u, v) then
                d[v] := d[u] + len(u, v)
    // test for negative cycle
    for Edge (u, v) ∈ E do
        if d[v] > d[u] + len(u, v) then
            return negative cycle
    return d
```

Laufzeit

- n mal über alle Kanten iterieren
- damit: $\Theta(n \cdot m)$

Abbrechen, wenn alle $d[v]$ gleich bleiben

- in der Praxis sinnvoll
- hilft im Worst-Case nicht asymptotisch

Pseudocode und Laufzeit

```
BellmanFord(Graph G, Node s)
  d := Array of size n initialized with ∞
  d[s] := 0
  for n − 1 iterations do
    for Edge (u, v) ∈ E do
      if d[v] > d[u] + len(u, v) then
        d[v] := d[u] + len(u, v)
  // test for negative cycle
  for Edge (u, v) ∈ E do
    if d[v] > d[u] + len(u, v) then
      return negative cycle
  return d
```

Laufzeit

- n mal über alle Kanten iterieren
- damit: $\Theta(n \cdot m)$

Abbrechen, wenn alle $d[v]$ gleich bleiben

- in der Praxis sinnvoll
- hilft im Worst-Case nicht asymptotisch

Geschickte Wahl der Kantenreihenfolge

- kann konstante Faktoren sparen
- hilft im Worst-Case nicht asymptotisch

Pseudocode und Laufzeit

BellmanFord(*Graph* G, *Node* s)
- $d :=$ *Array* of size n initialized with ∞
- $d[s] := 0$
- **for** $n-1$ iterations **do**
  - **for** *Edge* $(u, v) \in E$ **do**
    - **if** $d[v] > d[u] + \text{len}(u, v)$ **then**
      - $d[v] := d[u] + \text{len}(u, v)$
- // test for negative cycle
- **for** *Edge* $(u, v) \in E$ **do**
  - **if** $d[v] > d[u] + \text{len}(u, v)$ **then**
    - **return** negative cycle
- **return** d

Laufzeit
- n mal über alle Kanten iterieren
- damit: $\Theta(n \cdot m)$

Abbrechen, wenn alle $d[v]$ gleich bleiben
- in der Praxis sinnvoll
- hilft im Worst-Case nicht asymptotisch

Geschickte Wahl der Kantenreihenfolge
- kann konstante Faktoren sparen
- hilft im Worst-Case nicht asymptotisch

Weitere Anmerkungen zu Laufzeit
- kein asymptotisch besserer Algo bekannt
- sehr langsam verglichen mit $\Theta(n \log n + m)$
  (Dijkstras Algo für nicht-negative Kantenlängen)

All Pair Shortest Path

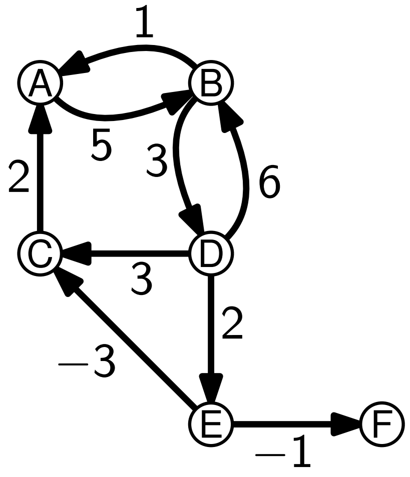
Problem: All-Pair Shortest Path (APSP)
Gegen einen gerichteten Graphen $G = (V, E)$ mit Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$. Berechne $\mathrm{dist}(s, t)$ für alle Knotenpaare $s, t \in V$.

All Pair Shortest Path

Problem: All-Pair Shortest Path (APSP)
Gegen einen gerichteten Graphen $G = (V, E)$ mit Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$. Berechne $\mathrm{dist}(s, t)$ für alle Knotenpaare $s, t \in V$.

Anmerkungen

- Distanzmatrix als Ausgabe
- Ausgabe ist bereits quadratisch groß
- wir nehmen erstmal an, dass es keine negativen Kreise gibt



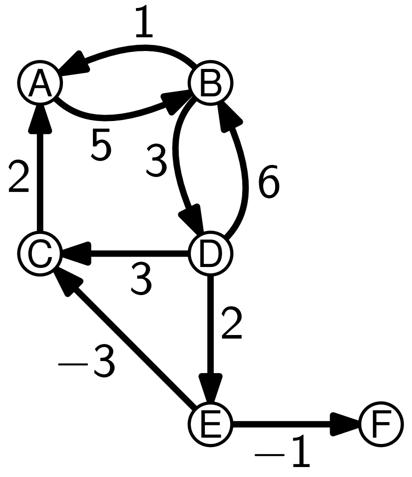
	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

All Pair Shortest Path

Problem: All-Pair Shortest Path (APSP)
Gegen einen gerichteten Graphen $G = (V, E)$ mit Kantenlängen $\mathrm{len}\colon E \to \mathbb{Z}$. Berechne $\mathrm{dist}(s, t)$ für alle Knotenpaare $s, t \in V$.

Anmerkungen

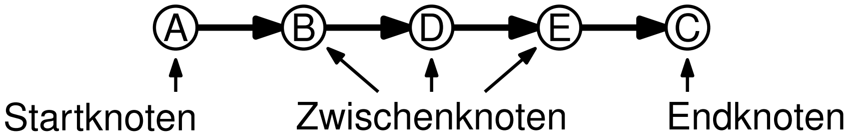
- Distanzmatrix als Ausgabe
- Ausgabe ist bereits quadratisch groß
- wir nehmen erstmal an, dass es keine negativen Kreise gibt



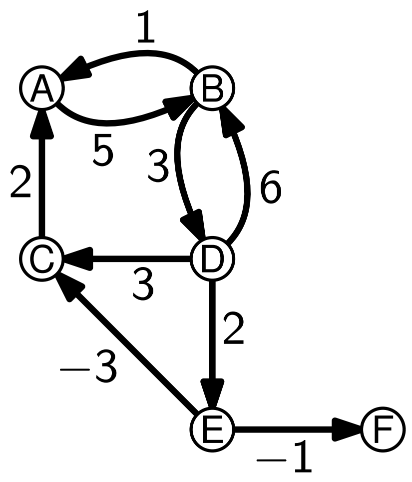
	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten

- Pfad $\langle v_1, v_2, \ldots, v_k \rangle$
- v_1 ist **Startknoten**, v_k **Endknoten**
- alle anderen sind **Zwischenknoten**



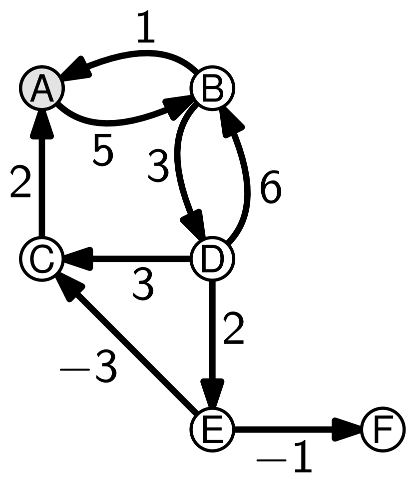
Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



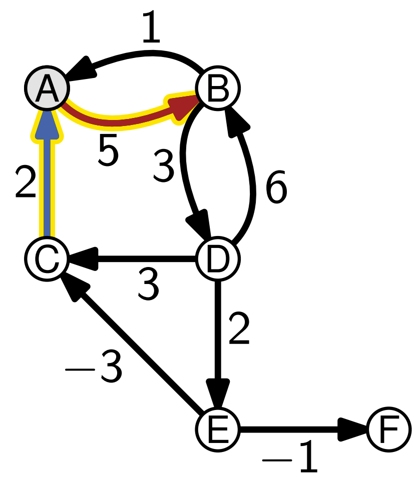
ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



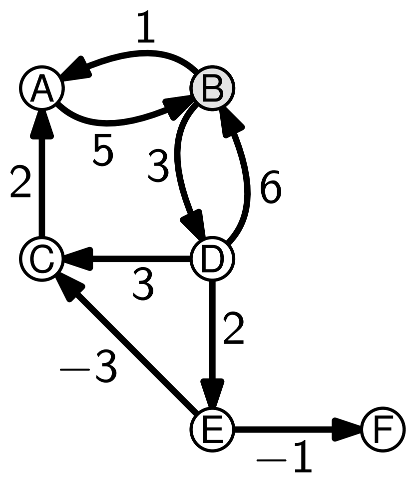
ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

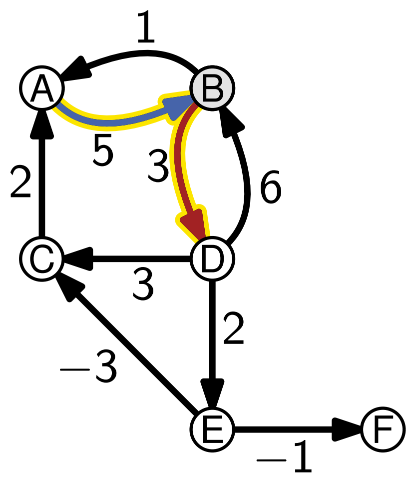
Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

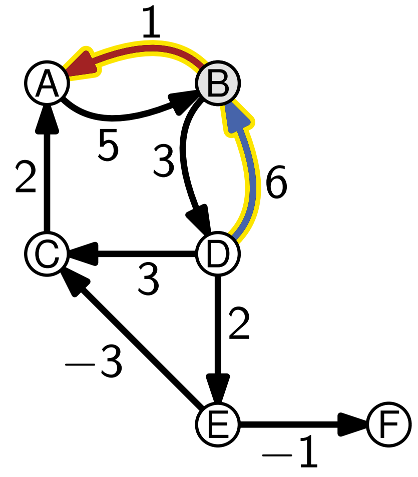
Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

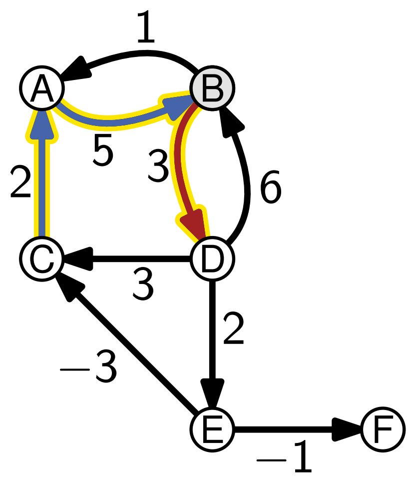
Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

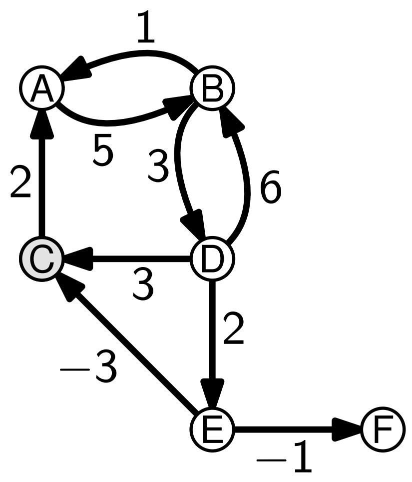
Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

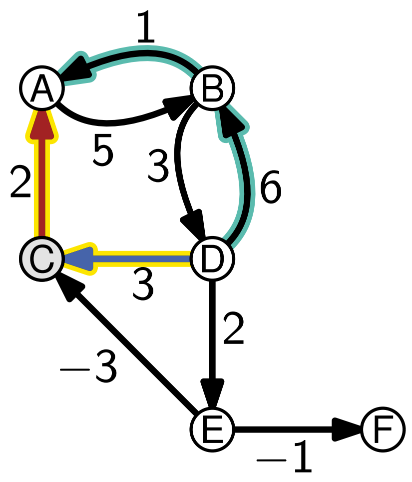
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

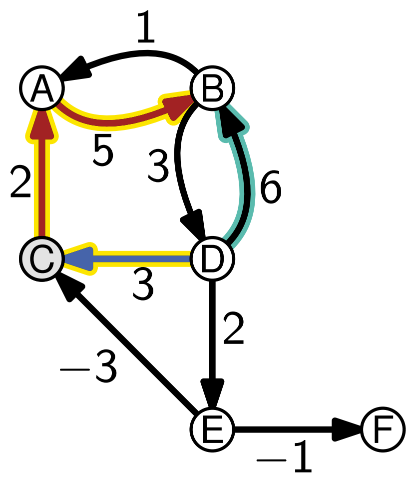
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

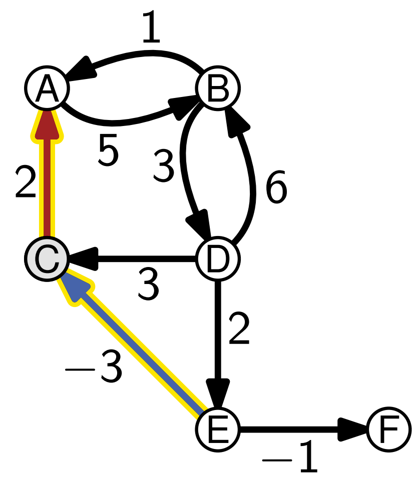
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

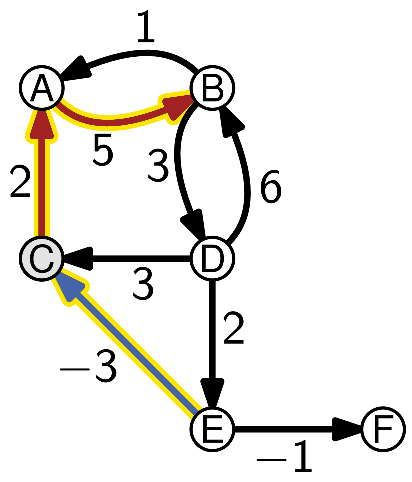
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

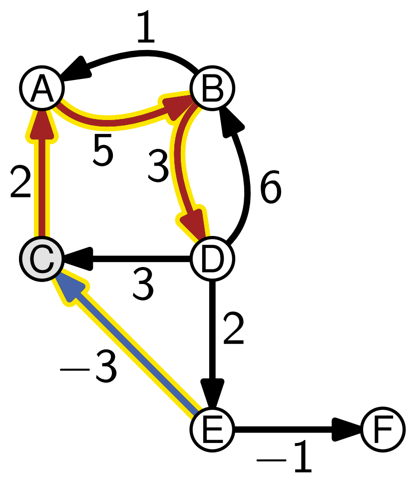
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

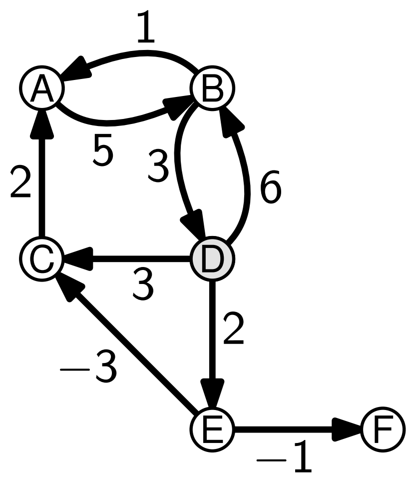
Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

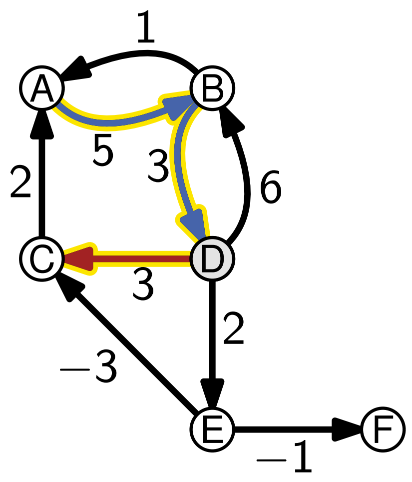
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

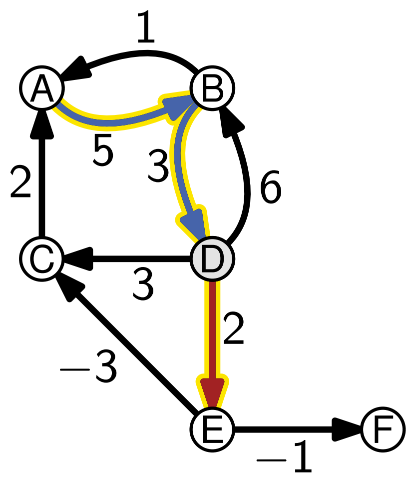
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

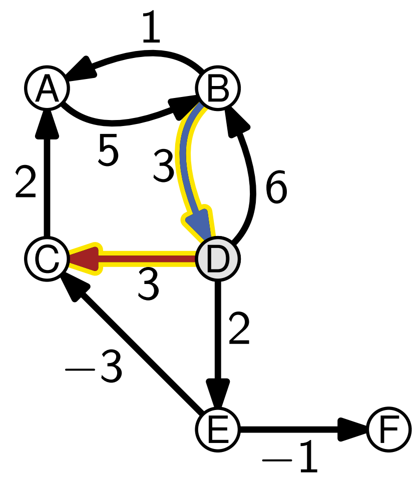
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

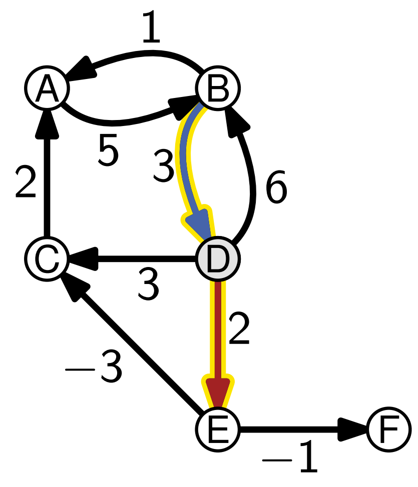
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

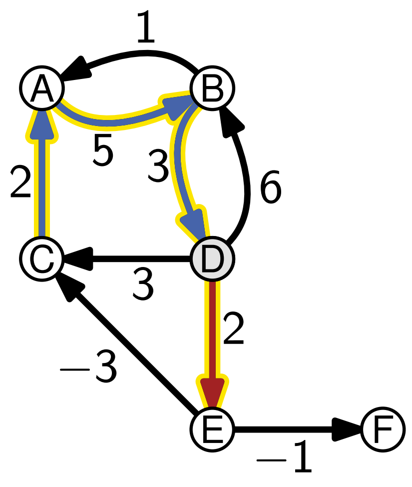
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

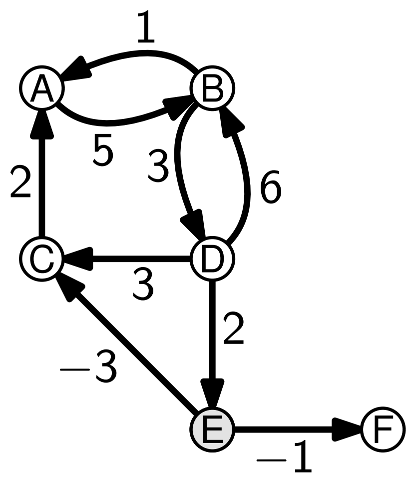
Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

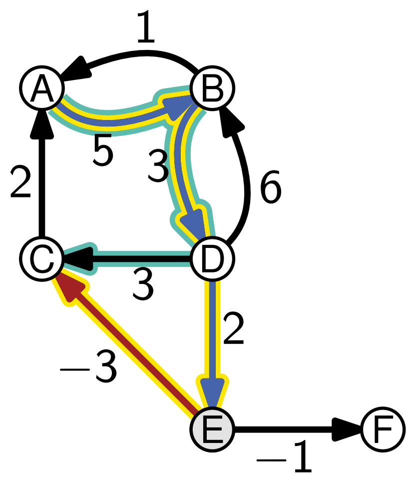
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

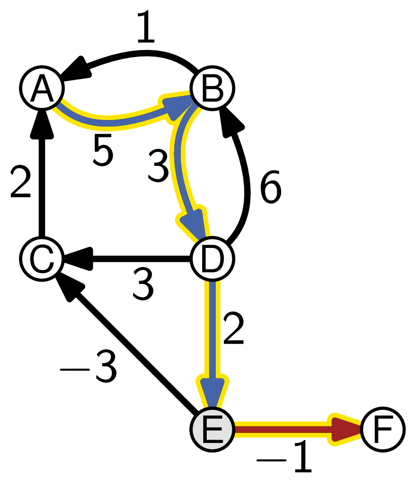
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

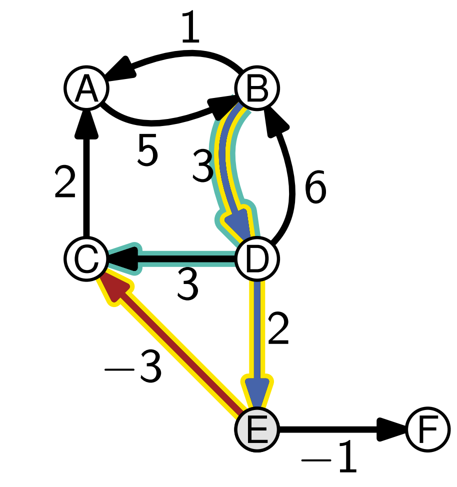
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

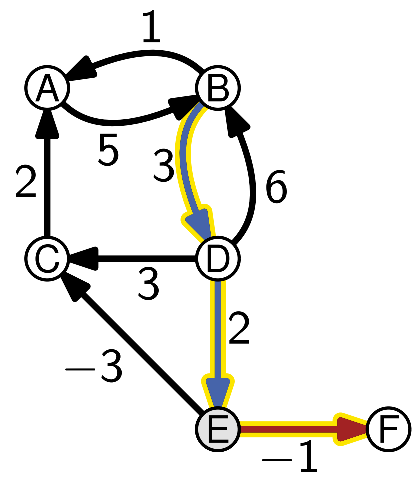
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

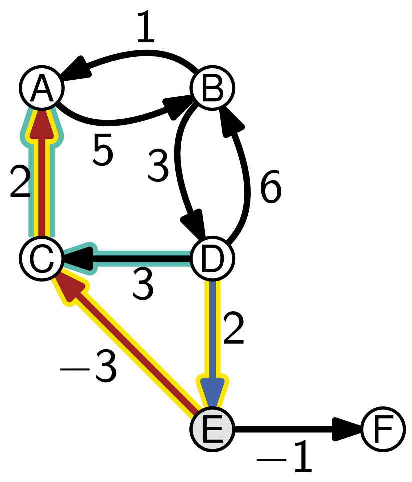
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

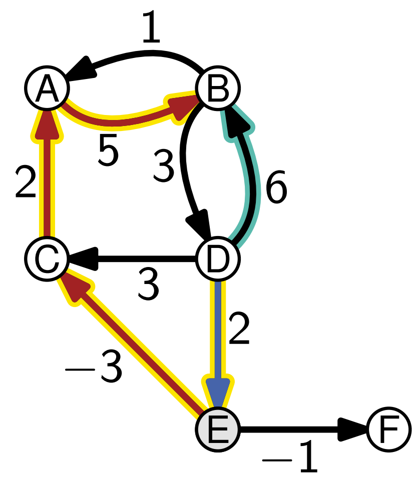
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

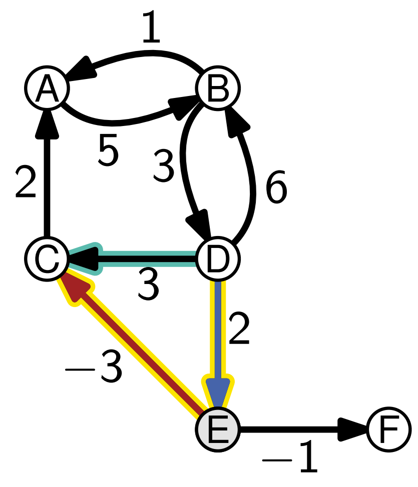
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

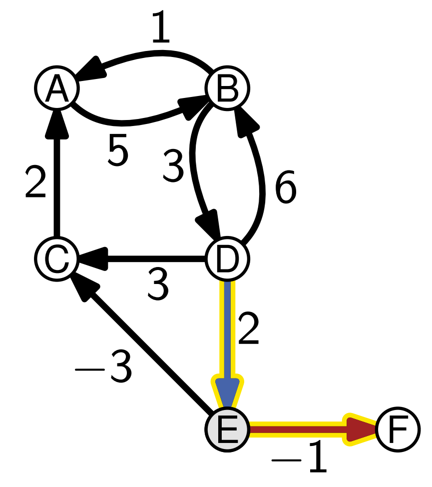
	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten

Karlsruher Institut für Technologie



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	−3	∞	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

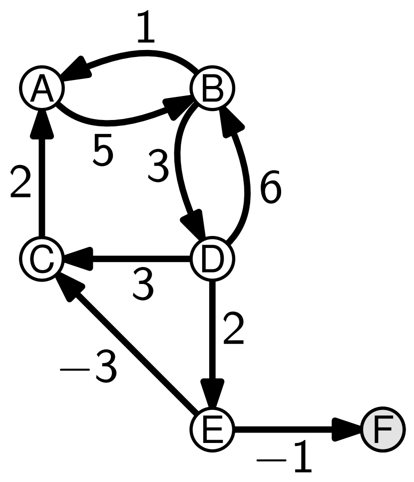
Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	−1	0	2	1
E	−1	4	−3	7	0	−1
F	∞	∞	∞	∞	∞	0

Eingeschränkte Menge an Zwischenknoten



ohne Zwischenknoten

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	∞	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	-3	∞	0	-1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A}

	A	B	C	D	E	F
A	0	5	∞	∞	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	∞	∞	∞
D	∞	6	3	0	2	∞
E	∞	∞	-3	∞	0	-1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	7	6	3	0	2	∞
E	∞	∞	-3	∞	0	-1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C}

	A	B	C	D	E	F
A	0	5	∞	8	∞	∞
B	1	0	∞	3	∞	∞
C	2	7	0	10	∞	∞
D	5	6	3	0	2	∞
E	-1	4	-3	7	0	-1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D}

	A	B	C	D	E	F
A	0	5	11	8	10	∞
B	1	0	6	3	5	∞
C	2	7	0	10	12	∞
D	5	6	3	0	2	∞
E	-1	4	-3	7	0	-1
F	∞	∞	∞	∞	∞	0

Zwischenknoten: {A, B, C, D, E}

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	-1	0	2	1
E	-1	4	-3	7	0	-1
F	∞	∞	∞	∞	∞	0

alle Zwischenknoten

	A	B	C	D	E	F
A	0	5	7	8	10	9
B	1	0	2	3	5	4
C	2	7	0	10	12	11
D	1	6	-1	0	2	1
E	-1	4	-3	7	0	-1
F	∞	∞	∞	∞	∞	0

Umsetzung als Algorithmus

Teillösungen für die Distanzen

- ordne Knotenmenge $V = \{v_1, \dots, v_n\}$
- D_i: Distanzmatrix bezüglich Zwischenknoten aus $V_i = \{v_1, \dots, v_i\} \to$ Endergebnis D_n
- $D_i[u][v]$: Länge eines kürzesten uv-Pfads der nur Zwischenknoten aus V_i verwendet

Umsetzung als Algorithmus

Teillösungen für die Distanzen

- ordne Knotenmenge $V = \{v_1, \dots, v_n\}$
- D_i: Distanzmatrix bezüglich Zwischenknoten aus $V_i = \{v_1, \dots, v_i\} \to$ Endergebnis D_n
- $D_i[u][v]$: Länge eines kürzesten uv-Pfads der nur Zwischenknoten aus V_i verwendet

Algorithmus von Floyd und Warshall

- D_0 ist leicht zu berechnen: quasi die Adjazenzmatrix
- berechne iterativ D_i aus D_{i-1}
  - v_i ist neuer erlaubter Zwischenknoten

Wie können wir D_i aus D_{i-1} berechnen?

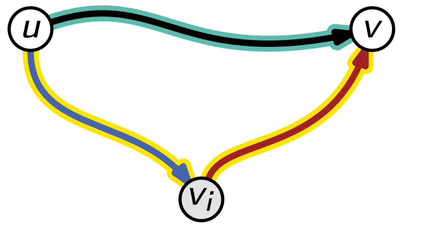
Umsetzung als Algorithmus

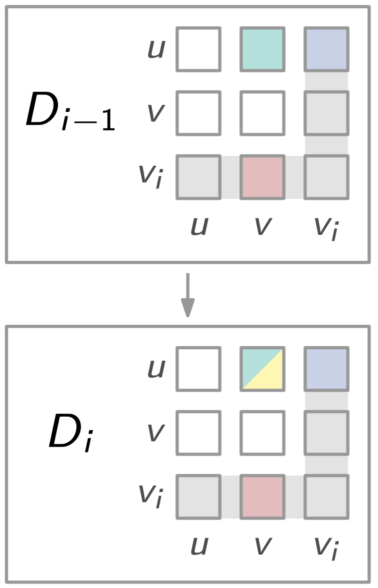
Teillösungen für die Distanzen

- ordne Knotenmenge $V = \{v_1, \dots, v_n\}$
- D_i: Distanzmatrix bezüglich Zwischenknoten aus $V_i = \{v_1, \dots, v_i\} \to$ Endergebnis D_n
- $D_i[u][v]$: Länge eines kürzesten uv-Pfads der nur Zwischenknoten aus V_i verwendet

Algorithmus von Floyd und Warshall

- D_0 ist leicht zu berechnen: quasi die Adjazenzmatrix
- berechne iterativ D_i aus D_{i-1}
  - v_i ist neuer erlaubter Zwischenknoten
  - Fall 1: kürzester uv-Pfad enthält v_i
    $\Rightarrow D_i[u][v] = D_{i-1}[u][v_i] + D_{i-1}[v_i][v]$
  - Fall 2: kürzester uv-Pfad enthält v_i nicht
    $\Rightarrow D_i[u][v] = D_{i-1}[u][v]$





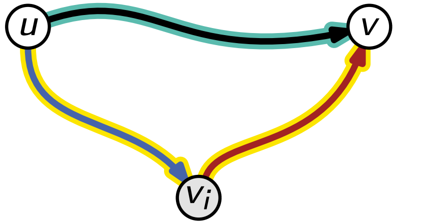
Umsetzung als Algorithmus

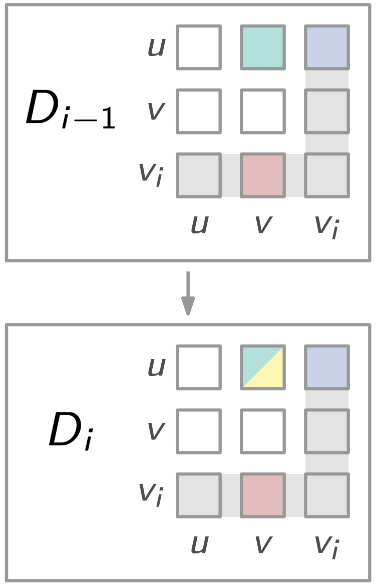
Teillösungen für die Distanzen

- ordne Knotenmenge $V = \{v_1, \ldots, v_n\}$
- D_i: Distanzmatrix bezüglich Zwischenknoten aus $V_i = \{v_1, \ldots, v_i\} \rightarrow$ Endergebnis D_n
- $D_i[u][v]$: Länge eines kürzesten uv-Pfads der nur Zwischenknoten aus V_i verwendet

Algorithmus von Floyd und Warshall

- D_0 ist leicht zu berechnen: quasi die Adjazenzmatrix
- berechne iterativ D_i aus D_{i-1}
  - v_i ist neuer erlaubter Zwischenknoten
  - Fall 1: kürzester uv-Pfad enthält v_i
    $\Rightarrow D_i[u][v] = D_{i-1}[u][v_i] + D_{i-1}[v_i][v]$
  - Fall 2: kürzester uv-Pfad enthält v_i nicht
    $\Rightarrow D_i[u][v] = D_{i-1}[u][v]$
  - Entscheidung für den richtigen Fall: teste ob $D_{i-1}[u][v_i] + D_{i-1}[v_i][v] < D_{i-1}[u][v]$





Anmerkungen und Pseudocode

Beachte

- Korrektheit folgt direkt induktiv
  (mit der Fallunterscheidung der vorherigen Folie)

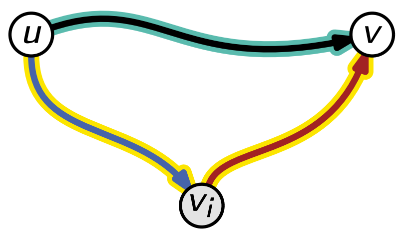
Anmerkungen und Pseudocode

Beachte

- Korrektheit folgt direkt induktiv
  (mit der Fallunterscheidung der vorherigen Folie)

FloydWarshall(*Graph* G)

$D := n \times n$ *Matrix* initialized with ∞
for $(u, v) \in E$ **do** $D[u][v] := \text{len}(u, v)$
for $v \in V$ **do** $D[v][v] := 0$
for $i := 1, \dots, n$ **do**
  // compute D_i from D_{i-1}
  for all pairs of nodes $(u, v) \in V \times V$ **do**
    $D[u][v] := \min(D[u][v], D[u][v_i] + D[v_i][v])$
return D



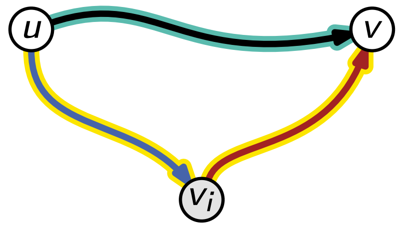
Anmerkungen und Pseudocode

Beachte

- Korrektheit folgt direkt induktiv
  (mit der Fallunterscheidung der vorherigen Folie)
- Änderungen in Iteration i machen für diese Iteration nichts kaputt
  $\rightarrow$ eine Matrix für alle D_i genügt

FloydWarshall(*Graph* G)

$D := n \times n$ *Matrix* initialized with ∞
for $(u, v) \in E$ **do** $D[u][v] := \mathrm{len}(u, v)$
for $v \in V$ **do** $D[v][v] := 0$
for $i := 1, \dots, n$ **do**
  // compute D_i from D_{i-1}
  for all pairs of nodes $(u, v) \in V \times V$ **do**
    $D[u][v] := \min(D[u][v], D[u][v_i] + D[v_i][v])$
return D



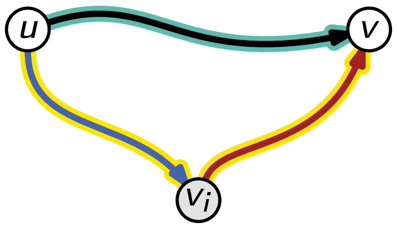
Anmerkungen und Pseudocode

Beachte

- Korrektheit folgt direkt induktiv
  (mit der Fallunterscheidung der vorherigen Folie)
- Änderungen in Iteration i machen für diese Iteration nichts kaputt
  $\rightarrow$ eine Matrix für alle D_i genügt
- Betrachtung aller Paare: für $u = v$, $u = v_i$ oder $v_i = v$ passiert nichts
  (Ausnahme: es gibt negativen Kreis $\rightarrow$ stellt man so fest)

FloydWarshall(*Graph* G)

```
D := n × n Matrix initialized with ∞
for (u, v) ∈ E do D[u][v] := len(u, v)
for v ∈ V do D[v][v] := 0
for i := 1, . . . , n do
    // compute D_i from D_{i−1}
    for all pairs of nodes (u, v) ∈ V × V do
        D[u][v] := min(D[u][v], D[u][v_i] + D[v_i][v])
return D
```



Anmerkungen und Pseudocode

Beachte

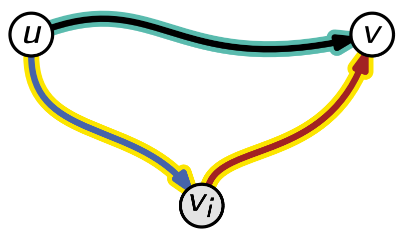
- Korrektheit folgt direkt induktiv
  (mit der Fallunterscheidung der vorherigen Folie)
- Änderungen in Iteration i machen für diese Iteration nichts kaputt
  $\rightarrow$ eine Matrix für alle D_i genügt
- Betrachtung aller Paare: für $u = v$, $u = v_i$ oder $v_i = v$ passiert nichts
  (Ausnahme: es gibt negativen Kreis $\rightarrow$ stellt man so fest)

Laufzeit

- n Iterationen
- n^2 Knotenpaare pro Iteration
- $\Theta(n^3)$

FloydWarshall(*Graph* G)

```
D := n × n Matrix initialized with ∞
for (u, v) ∈ E do D[u][v] := len(u, v)
for v ∈ V do D[v][v] := 0
for i := 1, ..., n do
    // compute D_i from D_{i−1}
    for all pairs of nodes (u, v) ∈ V × V do
        D[u][v] := min(D[u][v], D[u][v_i] + D[v_i][v])
return D
```



Dynamische Programmierung (DP)

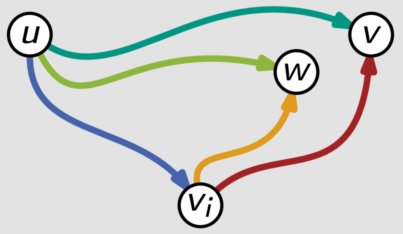
Karlsruher Institut für Technologie

Algorithmische Technik

- definiere Teillösungen
- Berechnung größerer aus kleinen Teillösungen
- Wiederverwendung berechneter Teillösungen

Beispiel: Floyd-Warshall

- $D_i[u][v]$: Distanz bzgl. Zwischenknoten aus $V_i = \{v_1, \dots, v_i\}$
- $D_i[u][v] = \min(D_{i-1}[u][v], D_{i-1}[u][v_i] + D_{i-1}[v_i][v])$
- $D_i[u][w] = \min(D_{i-1}[u][w], D_{i-1}[u][v_i] + D_{i-1}[v_i][w])$



Dynamische Programmierung (DP)

Algorithmische Technik

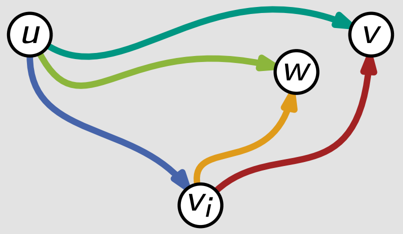
- definiere Teillösungen
- Berechnung größerer aus kleinen Teillösungen
- Wiederverwendung berechneter Teillösungen

Anmerkung

- Berechnung ist meist eine Rekurrenz
- Korrektheit folgt meist induktiv
- rekursive Umsetzung: sehr teuer (meist exponentiell), da keine Wiederverwendung von Teillösungen

Beispiel: Floyd-Warshall

- $D_i[u][v]$: Distanz bzgl. Zwischenknoten aus $V_i = \{v_1, \dots, v_i\}$
- $D_i[u][v] = \min(D_{i-1}[u][v], D_{i-1}[u][v_i] + D_{i-1}[v_i][v])$
- $D_i[u][w] = \min(D_{i-1}[u][w], D_{i-1}[u][v_i] + D_{i-1}[v_i][w])$



Dynamische Programmierung (DP)

Algorithmische Technik

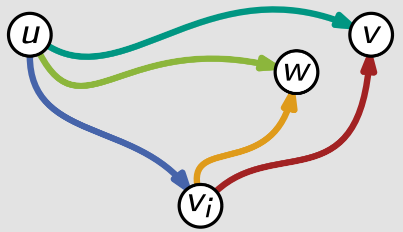
- definiere Teillösungen
- Berechnung größerer aus kleinen Teillösungen
- Wiederverwendung berechneter Teillösungen

Anmerkung

- Berechnung ist meist eine Rekurrenz
- Korrektheit folgt meist induktiv
- rekursive Umsetzung: sehr teuer (meist exponentiell), da keine Wiederverwendung von Teillösungen

Beispiel: Floyd-Warshall

- $D_i[u][v]$: Distanz bzgl. Zwischenknoten aus $V_i = \{v_1, \dots, v_i\}$
- $D_i[u][v] = \min(D_{i-1}[u][v], D_{i-1}[u][v_i] + D_{i-1}[v_i][v])$
- $D_i[u][w] = \min(D_{i-1}[u][w], D_{i-1}[u][v_i] + D_{i-1}[v_i][w])$



Hauptschwierigkeit: Definition geeigneter Teillösungen

- Welche Infos sind essentiell für eine Teillösung?
- X_i aus X_{i-1} berechnen ist schwierig, aber es geht, wenn ich zusätzlich Y_{i-1} kenne
- jetzt muss ich aber auch Y_i berechnen (aus X_{i-1} und Y_{i-1})

Dynamische Programmierung (DP) – Hinweise

Karlsruher Institut für Technologie

Vergleich mit Teile und Herrsche

- ähnlich: auch hier werden Teillösungen kombiniert (typischerweise disjunkte Teilprobleme)
- Besonderheit beim DP: Wiederverwendung von Teilergebnissen (Teilprobleme nicht disjunkt)

Dynamische Programmierung (DP) – Hinweise

Karlsruher Institut für Technologie

Vergleich mit Teile und Herrsche

- ähnlich: auch hier werden Teillösungen kombiniert (typischerweise disjunkte Teilprobleme)
- Besonderheit beim DP: Wiederverwendung von Teilergebnissen (Teilprobleme nicht disjunkt)

Häufiger Fehler: Fokus auf Rekurrenz statt Definition von Teillösungen

- Knackpunkt: geeignete Definition der Teillösungen
- Aufschrieb eines DP: sollte immer mit Definition der Teillösungen beginnen
- Rekurrenz ist dann oft mehr oder weniger offensichtlich

Dynamische Programmierung (DP) – Hinweise

Karlsruher Institut für Technologie

Vergleich mit Teile und Herrsche

- ähnlich: auch hier werden Teillösungen kombiniert (typischerweise disjunkte Teilprobleme)
- Besonderheit beim DP: Wiederverwendung von Teilergebnissen (Teilprobleme nicht disjunkt)

Häufiger Fehler: Fokus auf Rekurrenz statt Definition von Teillösungen

- Knackpunkt: geeignete Definition der Teillösungen
- Aufschrieb eines DP: sollte immer mit Definition der Teillösungen beginnen
- Rekurrenz ist dann oft mehr oder weniger offensichtlich

Teillösungen iterativ aufbauen

- meist wird irgendetwas Schritt für Schritt vergrößert
- Floyd-Warshall: Menge der Zwischenknoten V_i
- Sprechweise: **DP über Menge der Zwischenknoten** (so, wie man von einer **Induktion über *i*** spricht)

Dynamische Programmierung (DP) – Hinweise

Vergleich mit Teile und Herrsche

- ähnlich: auch hier werden Teillösungen kombiniert (typischerweise disjunkte Teilprobleme)
- Besonderheit beim DP: Wiederverwendung von Teilergebnissen (Teilprobleme nicht disjunkt)

Häufiger Fehler: Fokus auf Rekurrenz statt Definition von Teillösungen

- Knackpunkt: geeignete Definition der Teillösungen
- Aufschrieb eines DP: sollte immer mit Definition der Teillösungen beginnen
- Rekurrenz ist dann oft mehr oder weniger offensichtlich

Teillösungen iterativ aufbauen

- meist wird irgendetwas Schritt für Schritt vergrößert
- Floyd-Warshall: Menge der Zwischenknoten V_i
- Sprechweise: **DP über Menge der Zwischenknoten** (so, wie man von einer **Induktion über i** spricht)

  Mentaler Shortcut: Wenn man mit DPs vertraut ist, dann kommt das einer vollständigen Beschreibung des Floyd-Warshall Algos gleich.

Zusammenfassung

SSSP mit negativen Kanten

- Bellman–Ford: $\Theta(nm)$
- Erkennung negativer Kreise
- ist im Prinzip ein DP über die Anzahl Kanten auf dem kürzesten Weg

APSP

- Floyd–Warshall: $\Theta(n^3)$
- DP über die Menge der Zwischenknoten $V_i = \{v_1, \ldots, v_i\}$

Dynamsiche Programmierung (DP)

- wichtige algorithmische Technik
- wir werden später noch weitere DPs kennenlernen

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

$\Bigg\}$ nicht-negative Kantenlängen

Beachte

- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\rightarrow \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

} nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3 / 2^{\Omega(\sqrt{\log n})}$

} natürliche Kantenlängen

Beachte
- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

Beachte
- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

} nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3 / 2^{\Omega(\sqrt{\log n})}$

} natürliche Kantenlängen

- Johnson: Bellman–Ford + Längenanpassung + n Mal Dijkstra $\to \Theta(n^2 \log n + nm)$

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\rightarrow \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

} nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3 / 2^{\Omega(\sqrt{\log n})}$

} natürliche Kantenlängen

- Johnson: Bellman–Ford + Längenanpassung + n Mal Dijkstra $\rightarrow \Theta(n^2 \log n + nm)$
- substantiell besser als n^3 bzw. nm geht vermutlich nicht (Stichwort: Fine-Grained Complexity)

Beachte
- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

Karlsruher Institut für Technologie

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\rightarrow \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3 / 2^{\Omega(\sqrt{\log n})}$

natürliche Kantenlängen

- Johnson: Bellman–Ford + Längenanpassung + n Mal Dijkstra $\rightarrow \Theta(n^2 \log n + nm)$
- substantiell besser als n^3 bzw. nm geht vermutlich nicht (Stichwort: Fine-Grained Complexity)

Beachte
- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

Warum lernen wir hier nur den langsamen Floyd-Warshall kennen?

Bild: Grumpy cat line art / XXspiritwolf2000XX / Creative Commons

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

Beachte
- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

} nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3/2^{\Omega(\sqrt{\log n})}$

} natürliche Kantenlängen

- Johnson: Bellman–Ford + Längenanpassung + n Mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- substantiell besser als n^3 bzw. nm geht vermutlich nicht (Stichwort: Fine-Grained Complexity)

Warum lernen wir hier nur den langsamen Floyd-Warshall kennen?
- schön einfach
- Algorithmus der Wahl für dichte Graphen

Bild: Grumpy cat line art / XXspiritwolf2000XX / Creative Commons

Bonus: APSP – Geht es besser als $\Theta(n^3)$?

(Möglicherweise) bessere Algorithmen

- n mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- Pettie, Ramachandran: $\Theta(nm \log \alpha(m, n))$
  ($\alpha(m, n)$: sehr langsam wachsende inverse Ackermannfunktion)

} nicht-negative Kantenlängen

- Thorup: $\Theta(nm)$
- Williams: $n^3/2^{\Omega(\sqrt{\log n})}$

} natürliche Kantenlängen

- Johnson: Bellman–Ford + Längenanpassung + n Mal Dijkstra $\to \Theta(n^2 \log n + nm)$
- substantiell besser als n^3 bzw. nm geht vermutlich nicht (Stichwort: Fine-Grained Complexity)

Beachte

- $m \in O(n^2)$
- dünne Graphen: $m \ll n^2$

Warum lernen wir hier nur den langsamen Floyd-Warshall kennen?

- schön einfach
- Algorithmus der Wahl für dichte Graphen
- algorithmische Technik: dynamische Programmierung

Bild: Grumpy cat line art / XXspiritwolf2000XX / Creative Commons