

Wie kommt man am schnellsten von A nach B?

9 Kürzeste Wege

Gegeben gewichteter, gerichteter graph $G = (V, E)$, $w : E \rightarrow \mathbb{R}$.

- ▶ SSSP (single source shortest paths):
finde kürzesten Weg von Quelle s zu allen anderen Knoten
- ▶ APSP (all pairs shortest paths):
finde kürzesten Weg zwischen allen Knotenpaaren

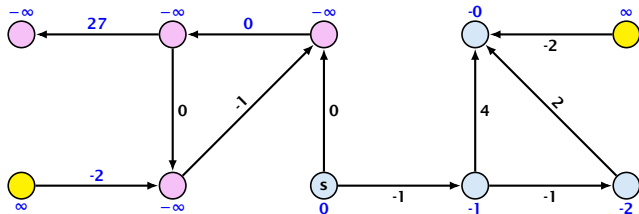
Manchmal wird auch von SSSP/APSP geredet wenn man nur die **Länge** der entsprechenden Wege berechnen möchte.

9 Kürzeste Wege

Die **Distanz** $d(x, y)$ zwischen Knoten x und y ist die Länge eines kürzesten Weges.

Formal:

$$d(s, v) = \begin{cases} +\infty & \text{kein Pfad von } s \text{ nach } v \\ -\infty & \text{kein kürzester Pfad von } s \text{ nach } v \\ \min_p \text{ path from } s \text{ to } v w(p) & \text{sonst} \end{cases}$$



9 Kürzeste Wege

Varianten des Problems:

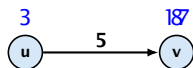
- ▶ uniform Gewichte (alle 1)
BFS, Laufzeit $\mathcal{O}(m + n)$
- ▶ beliebige Gewichte in einem **DAG** (**D**irected **A**cyclic **G**raph)
Topologische Sortierung + Kantenrelaxierung, Laufzeit
 $\mathcal{O}(m + n)$
- ▶ beliebiger Graph mit nichtnegativen Gewichten
Dijkstras Algorithmus,
Laufzeit $\mathcal{O}((m + n) \log n)$ oder $\mathcal{O}(m + n \log n)$
- ▶ beliebiger Graph mit beliebigen Gewichten
 - a) ohne negative Zyklen
 - b) mit negativen Zyklen

Kürzeste Wege

Idee SSSP:

- ▶ Jeder Knoten v hat Distanzlabel $\text{dist}(v)$.
- ▶ Anfangs ist $\text{dist}(s) = 0$ und $\text{dist}(v) = \infty$, d.h. $\text{dist}(x) \geq d(s, x)$ für alle Knoten x .
- ▶ Führe Kantenrelaxierungen durch...

Kantenrelaxierung:



$$\text{dist}(v) := \min\{\text{dist}(v), \text{dist}(u) + w(u, v)\}$$

es gilt immer noch $\text{dist}(v) \geq d(s, v)$...

Beobachtung:

- ▶ Es gilt immer $\text{dist}(v) \geq d(s, v)$ für alle Knoten.

Hoffnung:

Wenn wir genug Relaxierungen durchführen haben wir die richtigen Distanzen, da Distanzlabel nur kleiner werden.

Bei negativen Kreisen funktioniert das nicht.

Annahme: keine negativen Kreise.

⇒ die kürzesten Pfade haben endlich viele „hops“.

Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., e_5 und e_{10} könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \dots$ eine Folge von Kantenrelaxierungen, wobei die i -te Relaxierung R_i auf Kante e_i operiert.

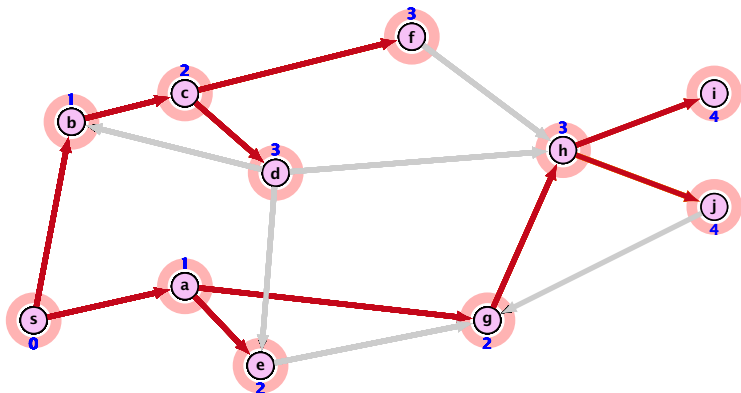
Sei $p = (s = v_0, v_1, \dots, v_k = x)$ ein kürzester Pfad von s nach x , und sei $a_i = (v_{i-1}, v_i)$ die i -te Kante dieses Pfades ($1 \leq i \leq k$).

Beobachtung

Falls ein k -elementige Teilfolge $S = S_1, S_2, \dots$ von \mathcal{R} existiert bei der S_i auf Kante a_i operiert hat der Knoten x nach Abarbeitung von \mathcal{R} das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem S_i durchgeführt ist hat v_i das korrekte Distanzlabel;

Uniforme Kantengewichte - BFS



visit-operation:

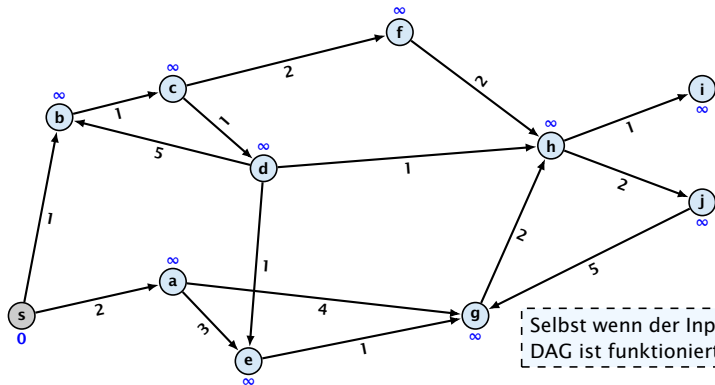
- ▶ relaxiere alle ausgehenden Kanten
- ▶ füge unbesuchte Nachbarn in Queue ein

Der Level ist die hop-Distanz von s zu v ; bei uniformen Kantengewichten entspricht dies der tatsächlichen Distanz.

- ▶ in der ersten Runde wird für jeden Level 1 Knoten eine eingehende Kante von Level 0 relaxiert
- ▶ in der zweiten Runde wird für jeden Level 2 Knoten eine eingehende Kante von Level 1 relaxiert
- ▶ ...

Bei einem kürzesten Pfad werden Relaxierungen in der Reihenfolge des Pfades durchgeführt.

Beliebige Gewichte – BFS funktioniert nicht



visit-operation:

- ▶ relaxiere alle ausgehenden Kanten
- ▶ füge unbesuchte Nachbarn in Queue ein

Kante (d, e) wird nach (e, g) relaxiert. Deshalb hat Knoten g am Ende falsche Distanz. Kante (d, h) wird erfolgreich relaxiert; der Vorgänger von h ist aber g (nach BFS-Regel).

Kürzeste Wege in DAGs

`finNum(v)` ist die bei DFS berechnete Reihenfolge, in der die rekursiven Aufrufe von `dfsVisit(v)` enden.

In DAGs existiert **topologische Sortierung** der Knoten:

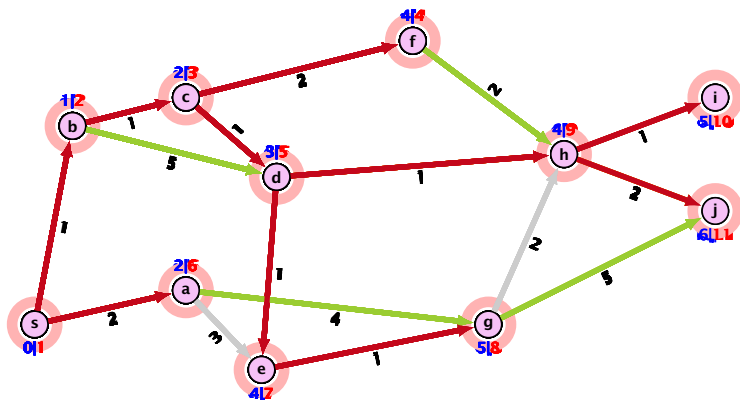
- ▶ $\text{top} : V \rightarrow \{1, \dots, n\}$, bijektiv
- ▶ für alle $(x, y) \in E$: $\text{top}(x) < \text{top}(y)$

Zum Beispiel: $\text{top}(v) = n - \text{finNum}(v) + 1$.

```
1 Input: weighted DAG  $G=(V,E,w)$ ; start vertex  $s$ ;  
2     array  $\text{top}$  mit  $\text{top}[i]$   $i$ -ter Knoten in TopSort  
3 Output:  $\text{key}$ -field of node contains distance from  $s$ ;  
4  
5 ShortestPathDag( $s, \text{top}$ )  
6     foreach  $v \in V$   
7          $v \rightarrow \text{key} = \infty$ ;  
8      $s \rightarrow \text{key} = 0$ ;  
9     for  $i = 1$  to  $n$  do  
10         $v = \text{top}[i]$ ;  
11        foreach  $x \in N[v]$   
12             $x \rightarrow \text{key} = \min(v \rightarrow \text{key} + w(v,x), x \rightarrow \text{key})$ ;
```

Kürzeste Wege in DAGs

Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten x besucht führe **Relaxierung** von ausgehenden Kanten durch (Kante (x, y)): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.



Funktioniert für beliebige Kantengewichte.

Kürzeste Wege in DAGs

Korrektheit:

Für **jeden** Pfad werden die Relaxierungen in der Reihenfolge des Pfades durchgeführt.

Kürzeste Wege in DAGs

Laufzeit:

DFS für topologische Sortierung

- ▶ Laufzeit $\mathcal{O}(n + m)$

Danach besucht der Algorithmus jede Kante genau einmal.
Zusätzlich wird jeder Knoten besucht:

- ▶ Laufzeit $\mathcal{O}(n + m)$

Also, Gesamtlaufzeit $\mathcal{O}(n + m)$.

Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- ▶ beliebiger Graph (gerichtet oder ungerichtet)
- ▶ nichtnegative Gewichte

Idee:

- ▶ Distanzrelaxierungen entlang **eines** kürzesten s - x für alle $x \in V, x \neq s$.

Problem:

- ▶ führe Relaxierungen in der richtigen Reihenfolge durch

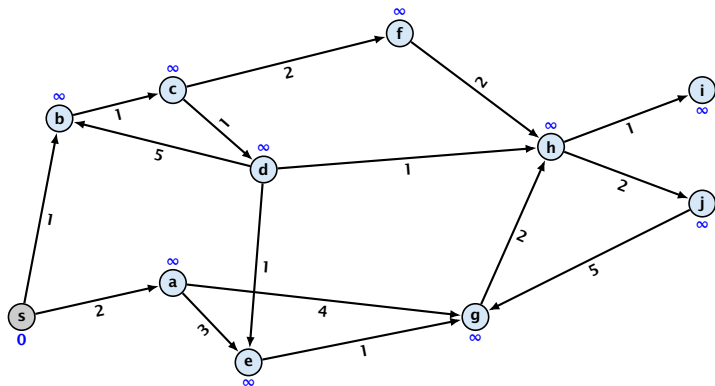
Lösung(?):

- ▶ führe Relaxierungen gemäß der Distanz von s durch

Dijkstra's Algorithmus

```
1 Input: weighted graph  $G=(V,E,w)$ ; start vertex  $s$ ;  
2 Output: key-field of node contains distance from  $s$ ;  
3  
4 Dijkstra( $s$ )  
5      $S = \text{new PriorityQueue}()$   
6     foreach  $v \in V$   
7          $v \rightarrow \text{key} = \infty$ ;  
8          $v \rightarrow \text{par} = \text{NULL}$ ;  
9          $v \rightarrow \text{han} = S \rightarrow \text{insert}(v)$ ;  
10     $s \rightarrow \text{key} = 0$ ;  $S \rightarrow \text{decreaseKey}(s \rightarrow \text{han}, 0)$ ;  
11    while ( $!S \rightarrow \text{empty}()$ ) {  
12         $v = S \rightarrow \text{extractMin}()$ ;  
13        foreach  $x \in N[v]$   
14            if ( $x \rightarrow \text{key} > v \rightarrow \text{key} + w(v,x)$ )  
15                 $S \rightarrow \text{decreaseKey}(x \rightarrow \text{han}, v \rightarrow \text{key} + w(v,x))$ ;  
16                 $x \rightarrow \text{key} = v \rightarrow \text{key} + w(v,x)$ ;  
17                 $x \rightarrow \text{par} = v$ ;  
18    }
```


Beliebige Graphen — nichtnegative Gewichte



Warum funktioniert das?

Invariante:

Sei A die Menge der Knoten, die schon aus der PQ entfernt wurden.

- A** Das Distanzlabel $x.key$ erfüllt $x.key \leq w(P)$ für alle $s-x$ Pfade P die **entweder** mit Kante (a, x) , $a \in A$ enden **oder** leer sind (nur aus Knoten s bestehen).
- B** Knoten in A haben ihre korrekte Distanz.

Außerdem nutzen wir, dass $x.key$ eine obere Schranke an die Länge eines kürzesten $s-x$ Pfades ist (folgt da wir nur Relaxierungen durchführen).

Initialisierung:

- A** Initial ist A leer; d.h. s ist der einzig erlaubte Pfad. Deshalb ist Invariante erfüllt wenn man s Distanz 0 gibt and allen anderen Knoten Distanz ∞ .
- B** Gilt, da A leer ist.

Beibehaltung der Invariante:

- B** Wir nehmen den Knoten v mit kleinstem key-Wert. Ein Pfad P von s nach v muss Länge mindestens $v.key$ haben.

Dies gilt, da der Pfad wenn er A verläßt (zum Knoten x) schon Länge mindestens $x.key \geq v.key$ hat (durch Invariante A für $x.key$). Er kann nicht kürzer werden da Kantengewichte **nichtnegativ** sind.

Deshalb, ist das Distanzlabel für v korrekt (es ist eine obere Schranke und kein Pfad ist kürzer)

D.h. Invariante B gilt für $A' = A \cup \{v\}$.

A Da v korrekte Distanz hat gilt $x.\text{key} \leq w(P)$ für alle Pfade die in Kante (v, x) enden.

Da durch die Invariante gilt, dass $x.\text{key} \leq w(P)$ für Pfade, die in (a, x) , $a \in A$ enden (oder leer sind), gilt Invariante A für $A' = A \cup \{v\}$.

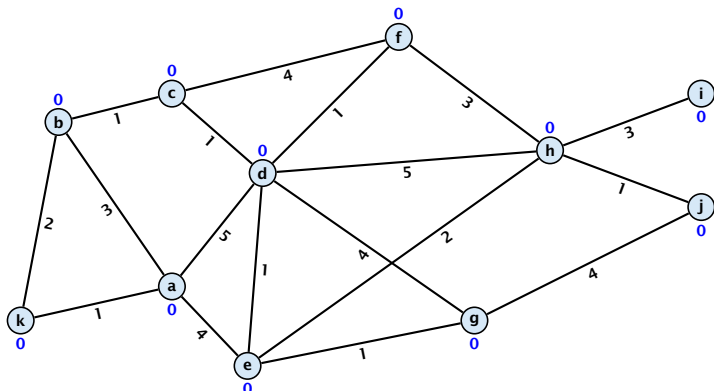
Laufzeit: m decreaseKey Operationen, n Einfügeoperationen, n extractMin() Operationen

Laufzeit: $\mathcal{O}((m + n) \log n)$.

Eine Laufzeit von $\mathcal{O}(m + n \log n)$ kann man durch Implementierung der Prioritätswarteschlange mit **Fibonacciheaps** erreichen.

Minimaler Spannbaum

Welche Kanten sollte man wählen um alle Knoten zu verbinden?



Minimiere Kosten!!!

Minimaler Spannbaum

Eingabe:

- ▶ ungerichteter **zusammenhängender** graph $G = (V, E)$
- ▶ positive Kantengewichte (Kosten) $w : E \rightarrow \mathbb{R}^+$

Ausgabe:

- ▶ Teilmenge $T \subseteq E$ s.t. $G = (V, T)$ verbunden und $w(T) = \sum_{e \in T} w(e)$ minimal

Beobachtung:

- ▶ da Kantengewichte positiv ist T ein Baum

Minimaler Spannbaum

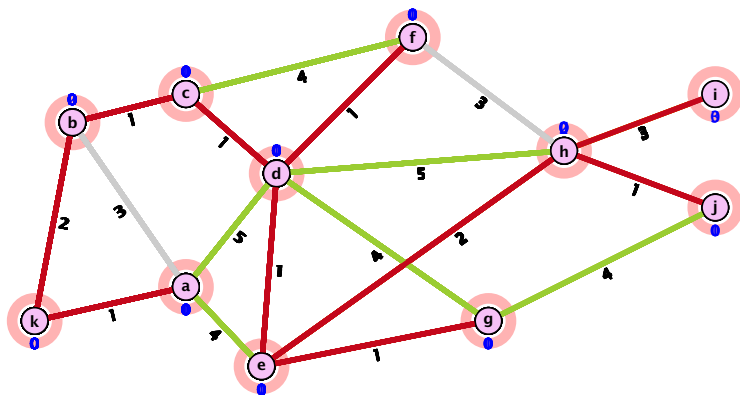
Lemma 1

Sei X, Y eine *Partitionierung* von V (d.h., $X \cup Y = V$ and $X \cap Y = \emptyset$), und sei $e = (x, y)$ eine billigste Kante zwischen X und Y . Dann existiert ein Minimaler Spannbaum der e enthält.

Prims MST-Algorithmus

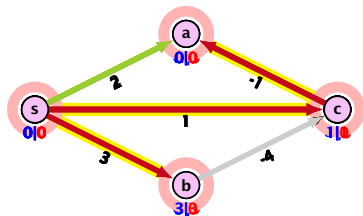
```
1 Input: weighted graph  $G=(V,E,w)$ ; start vertex  $s$ ;  
2 Output: parent-fields of nodes encode MST;  
3  
4 PrimMST( $s$ )  
5    $S = \text{new PriorityQueue}()$   
6   foreach  $v \in V$   
7      $v \rightarrow \text{key} = \infty$ ;  
8      $v \rightarrow \text{par} = \text{NULL}$ ;  
9      $v \rightarrow \text{han} = S \rightarrow \text{insert}(v)$ ;  
10   $s \rightarrow \text{key} = 0$ ;  $S \rightarrow \text{decreaseKey}(s \rightarrow \text{han}, 0)$ ;  
11  while ( $!S \rightarrow \text{empty}()$ ) {  
12     $v = S \rightarrow \text{extractMin}()$ ;  
13    foreach  $x \in N[v]$   
14      if ( $x \rightarrow \text{key} > w(v,x)$ )  
15         $S \rightarrow \text{decreaseKey}(x \rightarrow \text{han}, w(v,x))$ ;  
16         $x \rightarrow \text{key} = w(v,x)$ ;  
17         $x \rightarrow \text{par} = v$ ;  
18  }
```

Minimaler Spannbaum (Prim)



Negative Kantengewichte

Dijkstra funktioniert nicht:



Negative Kantengewichte

Lemma 2

Für jeden Knoten x mit $d(s, x) \neq \pm\infty$, existiert ein kürzester Weg mit höchstens $n - 1$ Kanten zwischen s und x .

Beweis:

- ▶ sei x ein Knoten mit $d(s, x) \neq \pm\infty$; dann existiert kürzester Pfad; sei P solch ein Pfad mit wenigsten Kanten
- ▶ angenommen, P hat $\geq n$ Kanten (d.h. $\geq n + 1$ Knoten)
- ▶ dann enthält P einen gerichteten Zyklus Z
- ▶ wenn Z negatives Gewicht hat gilt $d(s, x) = -\infty$ (\neq)
- ▶ andernfalls entfernen wir Z aus P und erhalten ein Pfad der Länge höchstens $w(P)$ hat mit weniger hops (\neq)

Bellman-Ford

Für $n - 1$ Phasen relaxiert man alle Kanten.

```
1 Input: weighted graph  $G=(V,E,w)$ ; start vertex  $s$ ;  
2 Output: key-field of every node contains distance from  $s$   
3  
4 NaiveBellmanFord( $s$ )  
5     foreach  $v \in V$   
6          $v \rightarrow \text{key} = \infty$ ;  
7      $s \rightarrow \text{key} = 0$ ;  
8     for  $i = 1$  to  $n-1$   
9         foreach  $(x,y) \in E$   
10             $y \rightarrow \text{key} = \min(y \rightarrow \text{key}, x \rightarrow \text{key} + w(x,y))$ 
```

Für hop-minimalen kürzesten s - x Pfad P existiert Teilfolge von Relaxierungen, die P von s nach x abarbeitet.

⇒ alle Distanzen sind am Ende korrekt, falls man keine negativen Kreise hat

Wie finden wir Knoten, die von s aus über einen Pfad erreichbar sind der einen Knoten eines negativen Kreises enthält? (Das sind **genau** die Knoten die Distanz $-\infty$ haben)

Wir fügen eine weitere Phase hinzu.

Lemma 3

Jeder von s erreichbare negative Zyklus enthält einen Knoten, der in Phase n sein Distanzlabel ändert.

Beweis:

- ▶ Sei Z negativer Zyklus (erreichbar von s). Nach Phase $n - 1$ haben alle Knoten des Zyklus endliches Label.
- ▶ Angenommen kein Knoten in Z ändert sein Label in Phase n . Das bedeutet

$$v_{i+1} \rightarrow \text{key} \leq v_i \rightarrow \text{key} + w(v_i, v_{i+1})$$

Bellman-Ford

D.h. wir müssen nur all Knoten markieren die in Runde n ihr Label ändern (und alle Knoten, die von diesen erreichbar sind).

Dies geschieht durch folgende Routine:

```
1 mark(x)
2   if (x->key != -∞)
3       x->key = -∞;
4       for ((x,y) ∈ E) mark(y)
```

Falls wir $\text{mark}(x)$ für alle Knoten $x \in S \subseteq V$ (für beliebige Teilmenge S) aufrufen benötigt dies Zeit $\mathcal{O}(|S| + m)$, da jede Kante nur einmal geprüft wird.

Bellman-Ford

```
1 BellmanFord(s)
2   foreach v ∈ V
3     v->key = ∞;
4     v->par = NULL;
5   s->key = 0;
6   for i = 1 to n-1
7     foreach (x,y) ∈ E
8       if (y->key > x->key + w(x,y))
9         y->key = x->key + w(x,y);
10        y->par = x;
11   foreach (x,y) ∈ E
12     if (y->key > x->key + w(x,y))
13       y->par = x;
14   mark(y)
```

Laufzeit: $\mathcal{O}(mn)$

Bellman-Ford

Invariante:

Nach der ℓ -ten Runde zeigt parent-Zeiger von x auf den Vorgänger von x auf einem kürzesten s - x Pfad mit höchstens ℓ Kanten.

$x.\text{parent} = \emptyset$ bedeutet **entweder**, dass es keinen s - x Pfad mit höchstens ℓ Kanten gibt **oder** dass der kürzeste dieser Pfade leer ist ($s = x$).

Nach Terminierung:

Für Knoten v mit $d(s, v) \neq \pm\infty$ führen die parent pointer von x nach s (rückwärts entlang kürzestem s - v Pfad).

Wenn man den parent-Zeigern von Knoten v mit $d(s, v) = -\infty$ folgt gelangt man zu einem negativen Kreis (dies kann man nutzen um einen negativen Kreis zu finden).

Verbesserung

- ▶ Benutze Queue, die Knoten speichert zu denen ein kürzerer Pfad gefunden wurde (und deren ausgehende Kanten deshalb relaxiert werden müssen).
- ▶ Wiederhole:
Entferne Element x aus der Queue und relaxiere ausgehende Kanten.

Falls Relaxierung entlang Kante (x, y) erfolgreich wird y in die Queue aufgenommen (falls noch nicht enthalten).

Runde/Phase endet wenn wenn die zu Anfang der Phase in der Queue enthaltenen Knoten bearbeitet sind.

APSP – Floyd Warshall

gegeben:

- ▶ Graph mit beliebigen Kantengewichten; keine negativen Kreise

gesucht:

- ▶ Distanzen/kürzeste Weg zwischen **allen** Knotenpaaren.

Naive Strategie

- ▶ n -mal Bellman-Ford. Laufzeit $\mathcal{O}(m \cdot n^2)$.

Beobachtung:

geht der kürzeste u - w Pfad über v dann sind auch die Teile von u nach v und v nach w kürzeste Pfade.

Dynamisches Programmieren:

- ▶ Berechne kürzeste Wege $P_A(u, w)$ von u nach w , die nur Zwischenknoten aus Menge A benutzen (initial ist Menge A leer).
- ▶ Wenn man alle Pfade $P_A(u, w)$ kennt, kann man einfach die Pfade $P_{A \cup \{v\}}(u, w)$ berechnen.
- ▶ Am Ende möchten wir $P_V(u, w)$ für alle Paare u, w kennen.

APSP – Floyd Warshall

```
1 Input: weighted graph  $G=(V,E,w)$ ;  
2 Output:  $d[u,v]$  is distance from  $u$  to  $v$ ;  $pred[u,v]$   
3         is predecessor of  $v$  on shortest path tree from  $u$   
4  
5 FloydWarshall( $G$ )  
6   for  $u,v \in V$   
7      $d[u,v] = \infty$ ;  
8      $pred[u,v] = \text{NULL}$ ;  
9   for  $v \in V$   
10     $d[u,v] = 0$ ;  
11   for  $(u,v) \in E$   
12     $d[u,v] = w(u,v)$ ;  
13     $pred[u,v] = u$ ;  
14   for  $v \in V$   
15    for  $\{u,w\} \in V \times V$   
16      if  $(d[u,w] > d[u,v] + d[v,w])$   
17         $d(u,w) = d(u,v) + d(v,w)$ ;  
18         $pred(u,w) = pred(v,w)$ ;
```


APSP – Floyd Warshall

- ▶ Komplexität $\mathcal{O}(n^3)$
- ▶ funktioniert auch bei Kanten mit negativem Gewicht
- ▶ Kreise negativer Länge verfälschen das Ergebnis; können aber durch negative Diagonaleinträge in der Ergebnismatrix erkannt werden.