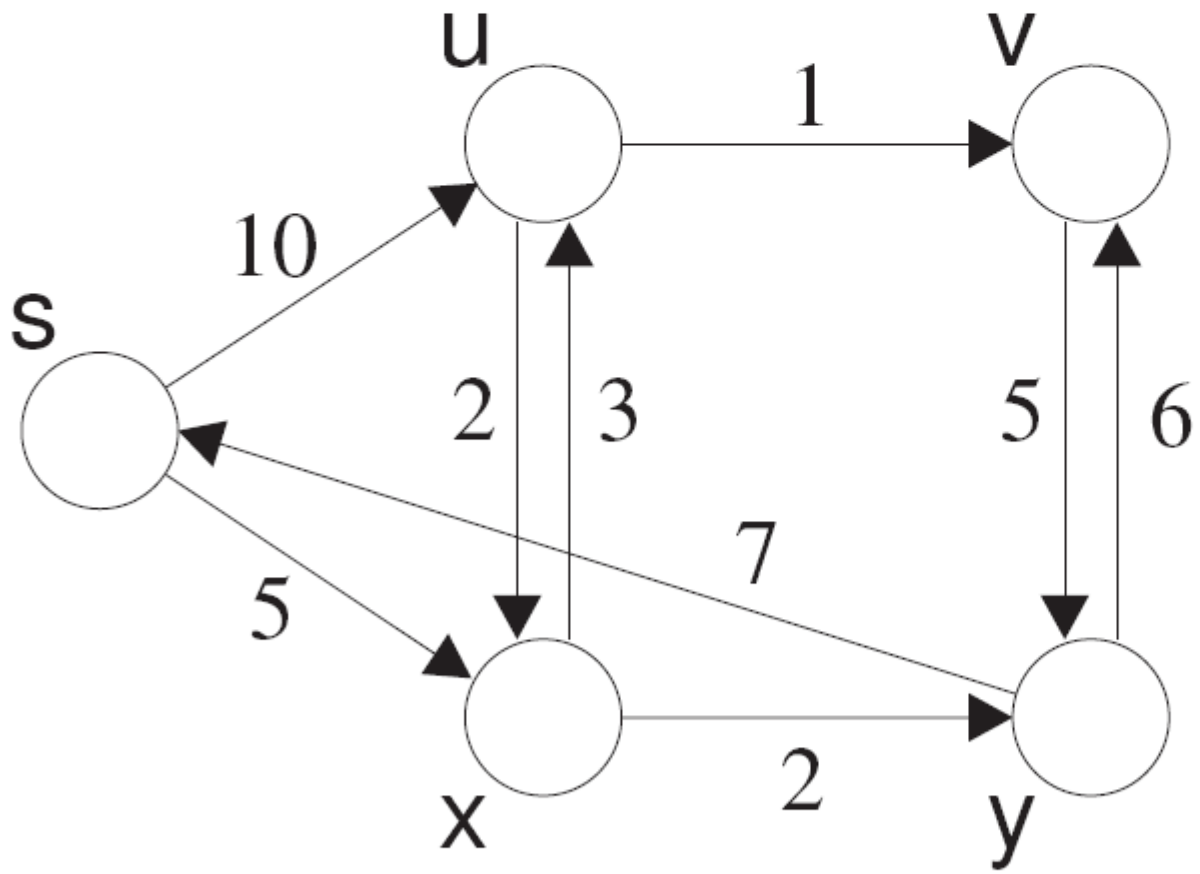
Algorithmen und Datenstrukturen

10. GRAPHEN 2
KÜRZESTE WEGE

Erinnerung: Gewichtete Graphen



Kürzeste Wege 1/2

Gegeben:

(Di-)Graph G=(V,E, γ) mit einer Gewichtsfunktion:

$$\gamma : E \to \mathbb{N}$$

Erinnerung:

- Ein (gerichteter) *Weg* W ist eine Sequenz von Knoten $W=(v_1,...,v_n)$ mit $v_1,...,v_n \in V$, für die gilt:
  $(v_i,v_{i+1}) \in E$ für alle i=1,...,n-1
- Ein Weg W heisst *Pfad*, falls zusätzlich gilt: $v_i \neq v_j$ für alle i,j=1,...,n mit i≠j (keine doppelten Knoten)

Kürzeste Wege 2/2

Gewicht eines Pfades P:

Aufsummierung der einzelnen Kantengewichte

$$w(P) = \sum_{i=1}^{n-1} \gamma((v_i, v_{i+1}))$$

Distanz zweier Knoten d(u,v) ist Gewicht eines günstigsten Pfades von u nach v

Kürzeste Wege Probleme

SPSP: Single pair shortest path

Eingabe: Graph G, Startknoten s, Endknoten t

Ausgabe: Distanz d(s,t)

SSSP: Single source shortest paths

Eingabe: Graph G, Startknoten s

Ausgabe: Distanzen d(s,v) für alle Knoten v

APSP: All-pairs shortest paths

Eingabe: Graph G

Ausgabe: Distanzen d(v,w) für alle Knoten v,w

Gewichtete Graphen

Ergänzung zum Graph-Interface:

```
public interface Graph{
    public int addNode();
    public boolean addEdge(int orig, int dest);
    public Collection<Integer> getChildren(int node);
    public Collection<Integer> getNodes();
    public int getWeight(int orig, int dest);
}
```

Spezialfall: homogene Gewichte

- Falls alle Gewichte identisch sind (wir also eigentlich einen ungewichteten Graphen haben), kann mit leichten Modifikationen schon die Breitensuche zur Lösung von SSSP genutzt werden
- Idee (sei Gewicht aller Kanten 1):
  1. Vom Startknoten ausgehend betrachte alle direkten Nachfolger (diese haben Distanz 1 zum Startknoten)
  2. Jeder Nachfolger eines direkten Nachfolgers wird eine um 1 erhöhte Distanz haben
  3. Da die Knoten entsprechend ihrer Distanz vom Startknoten aus abgearbeitet werden, müssen wir uns zu jedem Startknoten nur seinen direkten Vorgänger merken

Breitensuche für kürzeste Wege

```
public int[] ssspBfs(Graph g, int s){
    int[] d = new int[g.getNodes().size()];
    int[] pred = new int[g.getNodes().size()];
    Set<Integer> visited = new HashSet<Integer>();
    Queue<Integer> q = new LinkedList<Integer>();
    d[s] = 0;
    for(Integer m: g.getChildren(s)){
        q.add(m);
        pred[m] = s;
    }
    visited.add(s);
    while(!q.isEmpty()){
        int node = q.poll();
        d[node] = d[pred[node]] + 1;
        visited.add(node);
        for(Integer m: g.getChildren(node))
            if(!visited.contains(m) && !q.contains(m)){
                q.add(m);
                pred[m] = node;
            }
    }
    return d;
}
```

Analyse

Theorem (Terminierung)

Der Algorithmus `ssspBfs` terminiert nach endlicher Zeit.

Beweis:
Folgt aus dem entsprechenden Resultat für die Breitensuche □

Theorem (Korrektheit)

Falls alle Kantengewichte gleich 1 sind, löst der Algorithmus `ssspBfs` das Problem SSSP

Beweis:

Folgt aus dem entsprechenden Resultat für den Algorithmus von Dijkstra (siehe nächstes Kapitel) □

Theorem (Laufzeit)

Ist sowohl die Laufzeit von `getChildren` linear in der Anzahl der Kinder als auch `getNodes` linear in der Anzahl der Knoten, so hat der Algorithmus `ssspBfs` eine Laufzeit von O(|V|+|E|).

Beweis: Folgt aus dem entsprechenden Resultat für die Breitensuche □

Algorithmen und Datenstrukturen

10.1 DER ALGORITHMUS VON DIJKSTRA

Berechnung kürzester Wege: Dijkstra

Dijkstra Algorithmus zur Berechnung kürzester Wege (SSSP):

- Dijkstra 1959
- Iterative Erweiterung einer Menge von „günstig“ erreichbaren Knoten
- Greedy-Algorithmus
  - Ähnlich Breitensuche
  - Nur für nichtnegative Gewichte
  - Berechnet (iterativ verfeinernd) Distanzwerte d(v,w)
  - Prioritätswarteschlange zum Herauslesen des jeweils minimalen Elements

Priority-Queues

Eine Priority-Queue P ist eine dynamische Datenstruktur, die (mindestens) die folgenden Operationen unterstützt:

- P.add(Element): Element hinzufügen
- P.poll(): Minimalstes Element zurückgeben
- P.contains(Element): Enthält P das Element?

Die Ordnung zur Sortierung muss dabei vorab definiert sein.

Ein Heap kann beispielweise zur Implementierung einer Priority-Queue benutzt werden (add-Operation ist dann O(log n), poll-Operation O(log n), und contains-Operation ist O(n))

Benutzt man zusätzlich zum Heap noch einen binären Suchbaum auf denselben Elementen so ist auch contains in O(log n) realisierbar.

Priority-Queue in Java

```
class DijkstraComparator implements Comparator<Integer>{
    Map<Integer,Integer> d = new HashMap<Integer,Integer>();

    public DijkstraComparator(Map<Integer,Integer> d){
        this.d = d;
    }

    public int compare(Integer o1, Integer o2) {
        return d.get(o1).compareTo(d.get(o2));
    }
}
```

Ist d eine Map “Knoten”->”Aktueller Distanzwert von s aus”, so ist

```
PriorityQueue<Integer> queue =
new PriorityQueue<Integer>(g.getNumberOfNodes(),new DijkstraComparator(d));
```

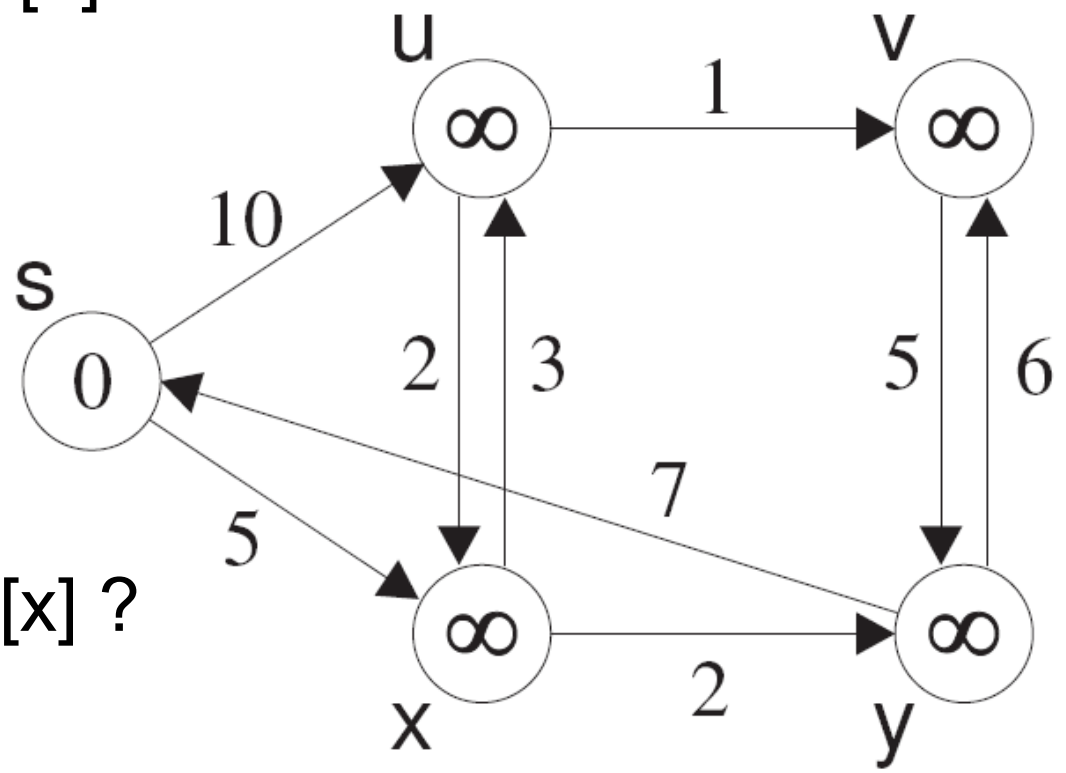
eine Priority-Queue, die bei iterativem Aufruf queue.poll() immer das Element mit dem minimalsten d-Wert zurückliefert

Dijkstras Algorithmus: Idee

1. Initialisiere alle Distanzwerte von s zu v mit ∞ (und von s zu s mit 0)
2. Initialisiere eine Priority-Queue Q mit allen v
3. Extrahiere das minimale Element w_{min} aus Q
4. Aktualisiere alle Distanzwerte der Nachfolger von w_{min} in Q:
   - Ist es günstiger über w_{min} zu einem Knoten w zu kommen?
   - Falls ja setze $d(s,w) = d(s,w_{min})+\gamma(w_{min},w)$
5. Wiederhole bei 3 solange Q noch Elemente hat

Dijkstras Algorithmus: Initialisierung

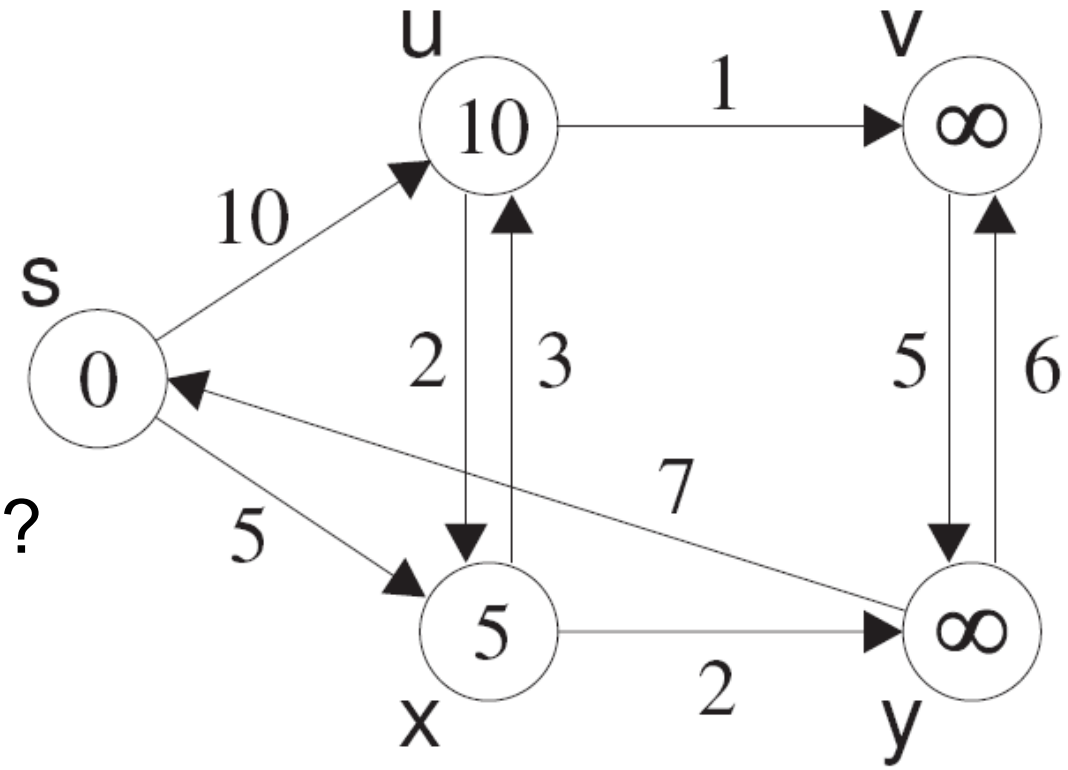
D[s]+γ(s,u) <D[u] ?
0 + 10 < ∞
=> D[u] = 10



D[s]+γ(s,x) <D[x] ?
0 + 5 < ∞
=> D[x] = 5

$$Q = \langle (s:0), (u:\infty), (v:\infty), (x:\infty), (y:\infty) \rangle$$

Dijkstras Algorithmus: Schritt 1

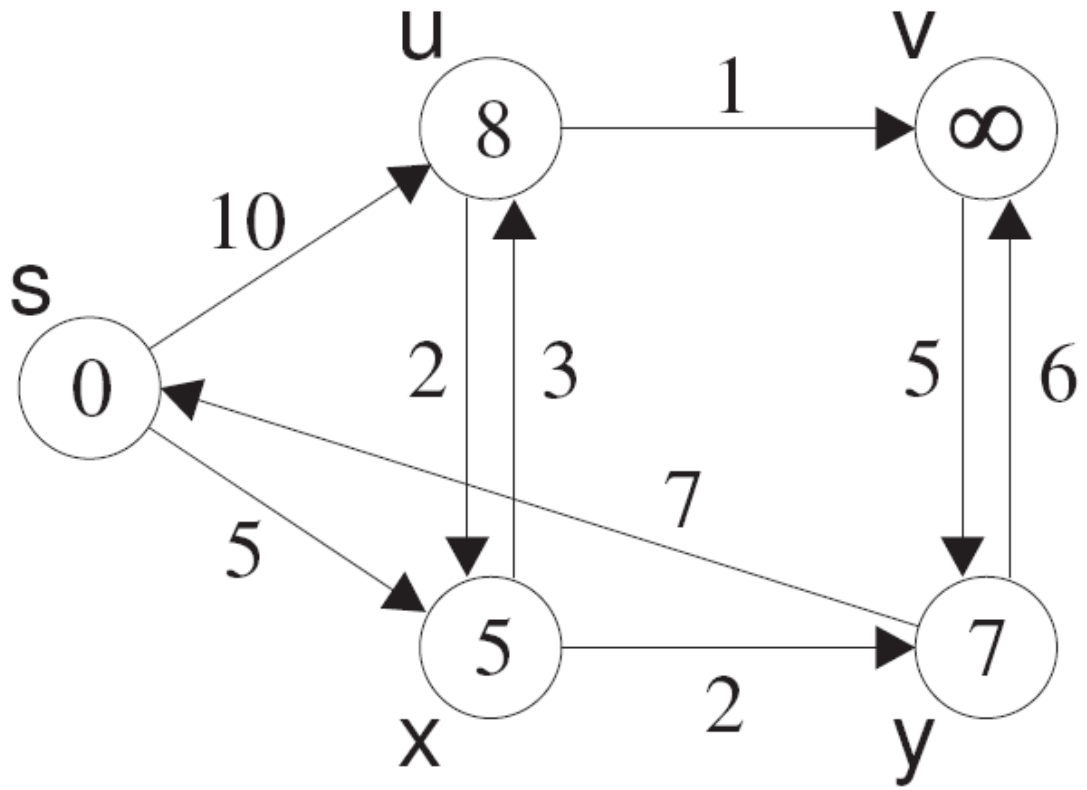


D[x]+γ(x,u) <D[u] ?
5 + 3 < 10
=> D[u] = 8

D[y] analog

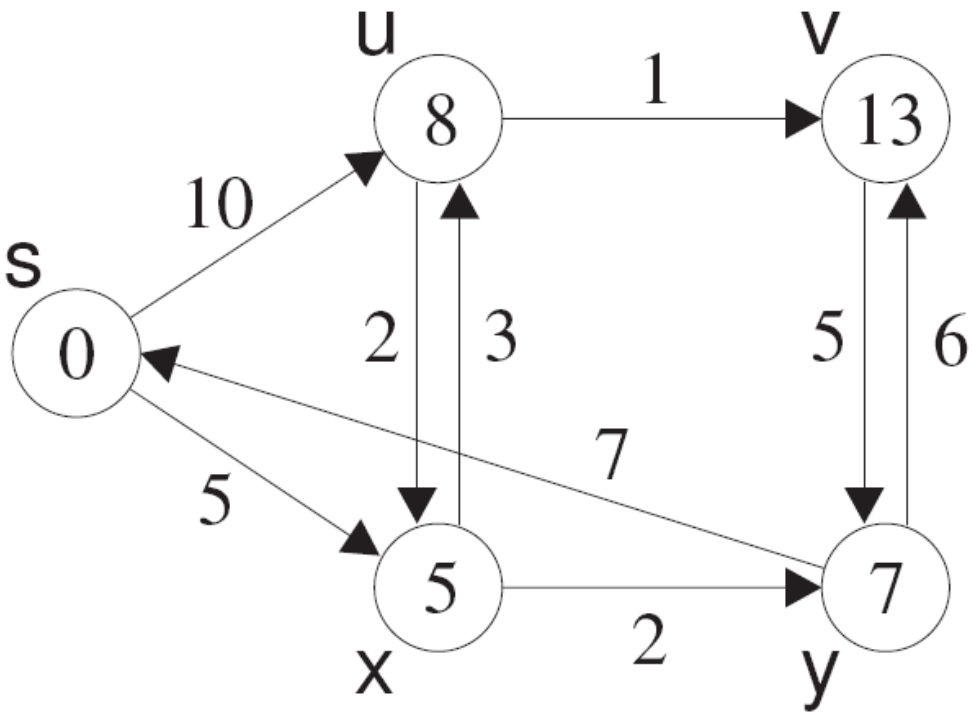
$$Q = \langle (x:5), (u:10), (v:\infty), (y:\infty) \rangle$$

Dijkstras Algorithmus: Schritt 2



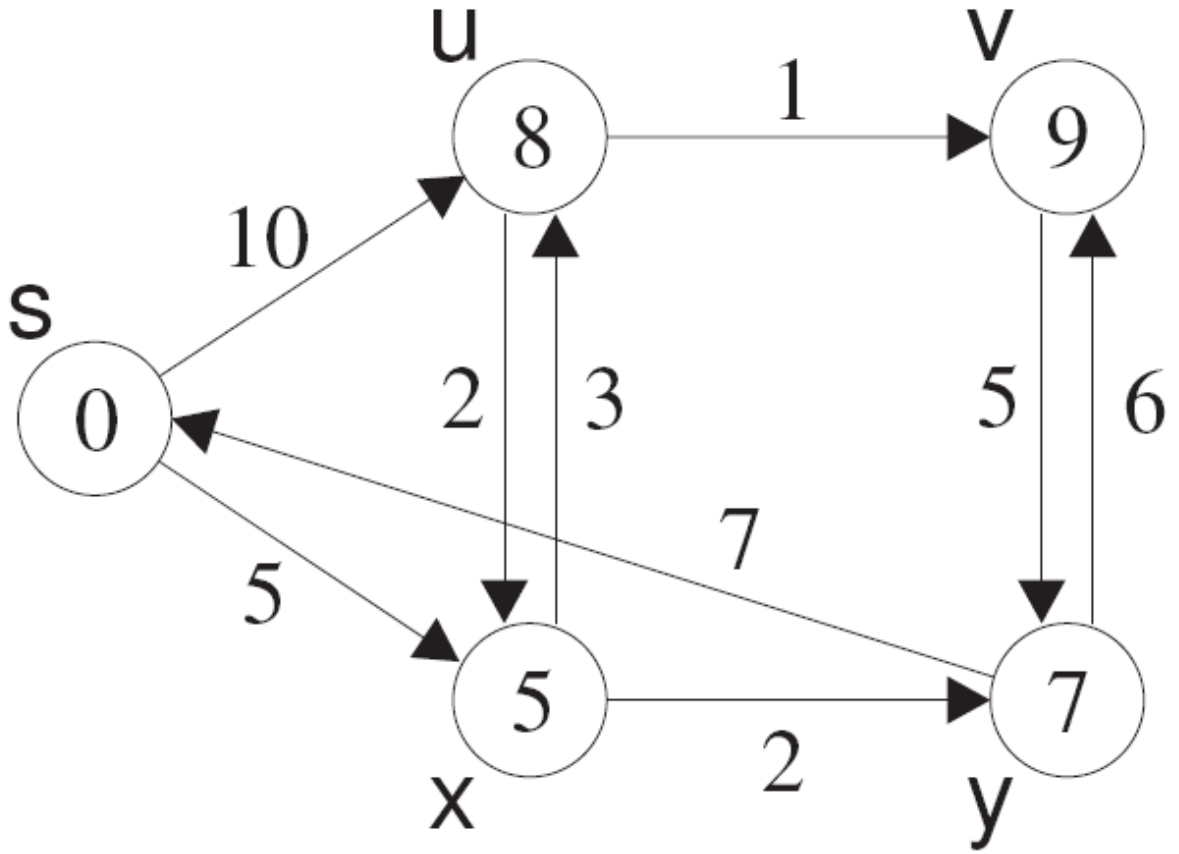
$$Q = \langle (y, 7), (u : 8), (v : \infty) \rangle$$

Dijkstras Algorithmus: Schritt 3



$$Q = \langle (u : 8), (v : 13) \rangle$$

Dijkstras Algorithmus: Schritt 4



$$Q = \langle (v : 9) \rangle$$

Dijkstras Algorithmus in Java

```
Map<Integer,Integer> dijkstra(Graph g, int s){
     Map<Integer,Integer> d = new HashMap<Integer, Integer>();
     PriorityQueue<Integer> queue = //Initialisiere Priority-Queue entsprechend
     for(Integer n: g){
          if(!n.equals(s)){
               d.put(n, Integer.MAX_VALUE);
               queue.add(n);
          }
     }
     d.put(s, 0);
     queue.add(s);

     while(!queue.isEmpty()){
          Integer u = queue.poll();
          for(Integer v: g.getChildren(u)){
               if(queue.contains(v)){
                    if(d.get(u) + g.getWeight(u,v) < d.get(v)){
                         d.put(v, d.get(u) + g.getWeight(u,v));
                         //refresh queue
                         queue.remove(v);
                         queue.add(v);
                    }
               }
          }
     }

     return d;
}
```

Analyse

Theorem (Terminierung)

Der Algorithmus von Dijkstra terminiert für eine endliche Eingabe nach endlicher Zeit.

Beweis:

In jedem Schritt der while-Schleife wird ein Element aus *queue* entfernt und die Schleife endet sobald *queue* leer ist. Jeder Knoten hat nur endliche viele Kinder, deswegen ist auch die Laufzeit der inneren for-Schleife endlich. □

Analyse

Theorem (Korrekheit)

Sind alle Kantengewichte nicht-negativ, so enthält d am Ende die Distanzwerte von s zu allen anderen Knoten.

Beweis:

Beachte, dass sobald ein Knoten *v* aus *queue* entfernt wird, der Wert für *v* in *d* nicht mehr geändert wird.

Zeige nun, dass gilt:

Wird *v* aus *queue* entfernt, so enthält *d* den Distanzwert von *s* nach *v*.

Analyse

Zeige dies durch Induktion nach i=„Anzahl bisher aus queue entfernter Knoten“:

- i=0: Am Anfang hat *queue* nur für *s* einen endlichen Wert gespeichert, alle anderen Werte sind ∞. Der Knoten *s* wird auch stets zuerst entfernt und der Distanzwert ist 0. Dies ist auch korrekt, da *s* zu sich selbst Distanz 0 hat und alle anderen Knoten keine geringere Distanz von *s* aus haben können (da alle Kanten nicht-negative Gewichte haben).

Analyse

- i → i+1: Sei v der (i+1)te Knoten, der aus queue entfernt wird.
  - Da die bisher bearbeiteten Knoten den korrekten Distanzwert haben, ist der neue Distanzwert durch den „günstigsten“ aus dem bisher bearbeiteten Teilgraphen um genau eine Kante hinausgehenden Pfad bestimmt.
  - Jeder Pfad zum Zielknoten dieses Pfades, der um mehr als eine Kante aus dem bearbeiteten Bereich hinausgeht, ist teurer als die gewählte, da Kosten mit zusätzlich hinzugenommenen Kanten nicht sinken können. □

Analyse

Anmerkung:

- Sind alle Kantengewichte identisch (beispielsweise =1), so arbeitet der Algorithmus von Dijkstra die Knoten in derselben Reihenfolge ab wie die Breitensuche
- Die PriorityQueue enthält dann stets nur Knoten mit Distanzwerten, die sich um maximal 1 unterscheiden
- Daraus folgt direkt die Korrektheit der Breitensuche für die Berechnung kürzester Wege (für identische Gewichte)

Analyse

Theorem (Laufzeit)

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand von Dijkstras Algorithmus für einen beliebigen Knoten s in G ist O((|E| + |V|) log |V|).

Beweis:

Beachte: Wird für die Priority-Queue beispielsweise ein Heap verwendet, so hat die Operation poll() einen Aufwand von O(log k) (mit k=„Anzahl Elemente in Queue“).

Sei |V|=n und |E|=m.

Analyse

```
Map<Integer,Integer> dijkstra(Graph g, int s){
     Map<Integer,Integer> d = new HashMap<Integer, Integer>();
     PriorityQueue<Integer> queue = //Initialisiere Priority-Queue entsprechend
     for(Integer n: g){
          if(!n.equals(s)){
               d.put(n, Integer.MAX_VALUE);
               queue.add(n);
          }
     }
     d.put(s, 0);
     queue.add(s);

     while(!queue.isEmpty()){
          Integer u = queue.poll();
          for(Integer v: g.getChildren(u)){
               if(queue.contains(v)){
                    if(d.get(u) + g.getWeight(u,v) < d.get(v)){
                         d.put(v, d.get(u) + g.getWeight(u,v));
                         //refresh queue
                         queue.remove(v);
                         queue.add(v);

                    }
               }
          }
     }

     return d;
```

O(n)

O(log n)

n-mal

O(log n)

O(log n) (pro Kante im Graph)

Jede Kante im Graph wird hier irgendwann genau einmal betrachtet, Schleifenrumpf wird also m-mal ausgeführt

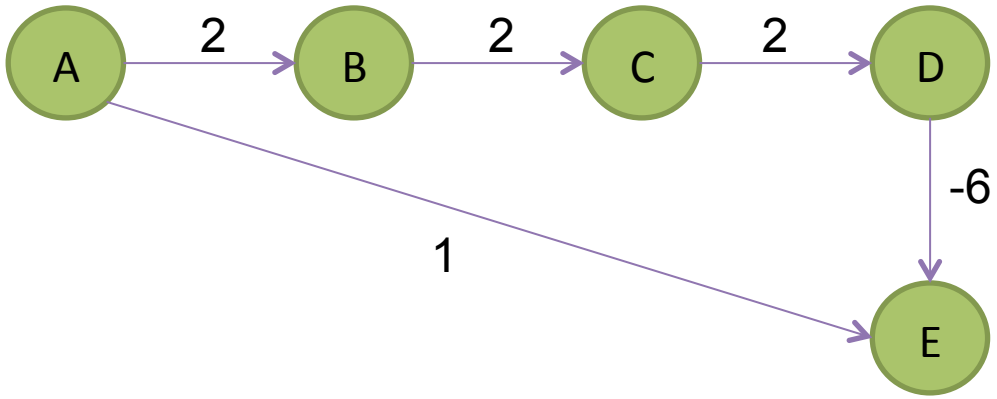
Insgesamt:
O(n log n) + O(n)
+ n* O(log n) + m *O(log n)
= O((m + n) log n)

Anmerkung

- Durch Benutzung sog. *Fibonacci-Heaps* (anstatt *normaler* Heaps) kann die Laufzeit von $O((m + n) \log n)$ verbessert werden zu $O(m + n \log n)$

Nachteil des Algorithmus von Dijkstra

- Kürzester Weg wird immer gefunden
- Aber: viele unnötige, sinnlose Wege werden gegangen
- Bei negativen Kanten auch falsche Ergebnisse



Algorithmen und Datenstrukturen

10.2 DER ALGORITHMUS VON BELLMANN-FORD

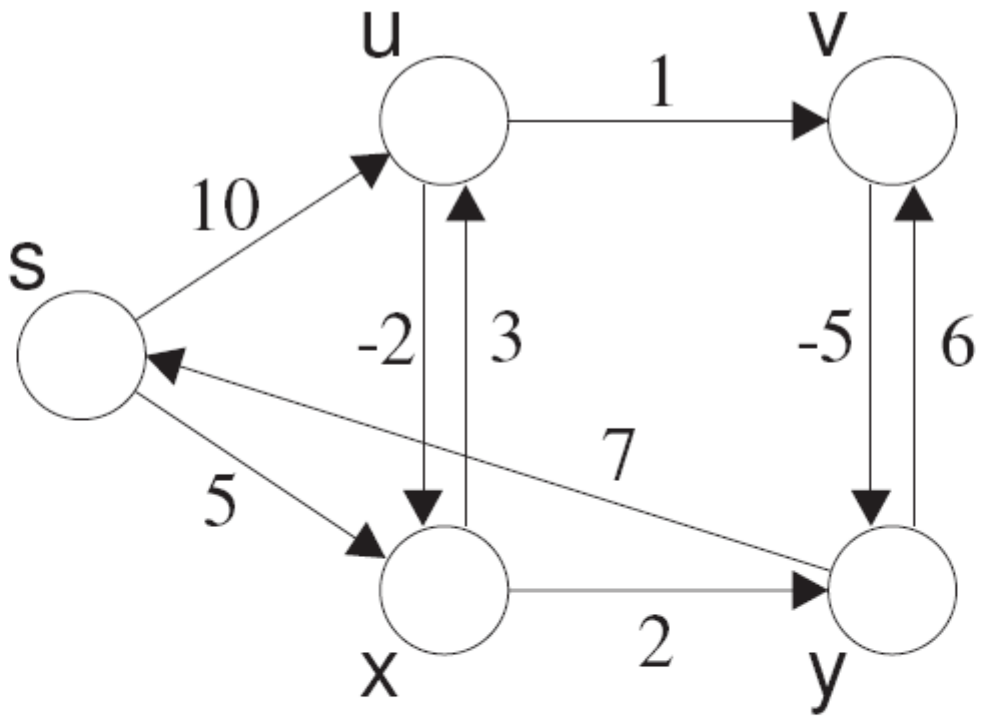
Berechnung kürzester Wege: Bellmann-Ford

- Dijkstra: nur nichtnegative Gewichte
- Variante mit negativen Gewichten?
  - Macht nur bei gerichteten Graphen Sinn
  - Problem: Zyklen mit negativem Gesamtgewicht
- Beispiel für Gewinne statt Kosten:
  - Verbindungsnetze mit Bonus-Gewinnen für bestimmte Verbindungen, um Auslastung zu erhöhen (z.B. bei Fluggesellschaften sind Flüge mit Zwischenstopp oft billiger aus diesem Grund)

- **Konkret: Bellman-Ford-Algorithmus**
- **Löst ebenfalls das SSSP Problem**

Kürzeste Wege mit negativen Kantengewichten

- Beispielgraph



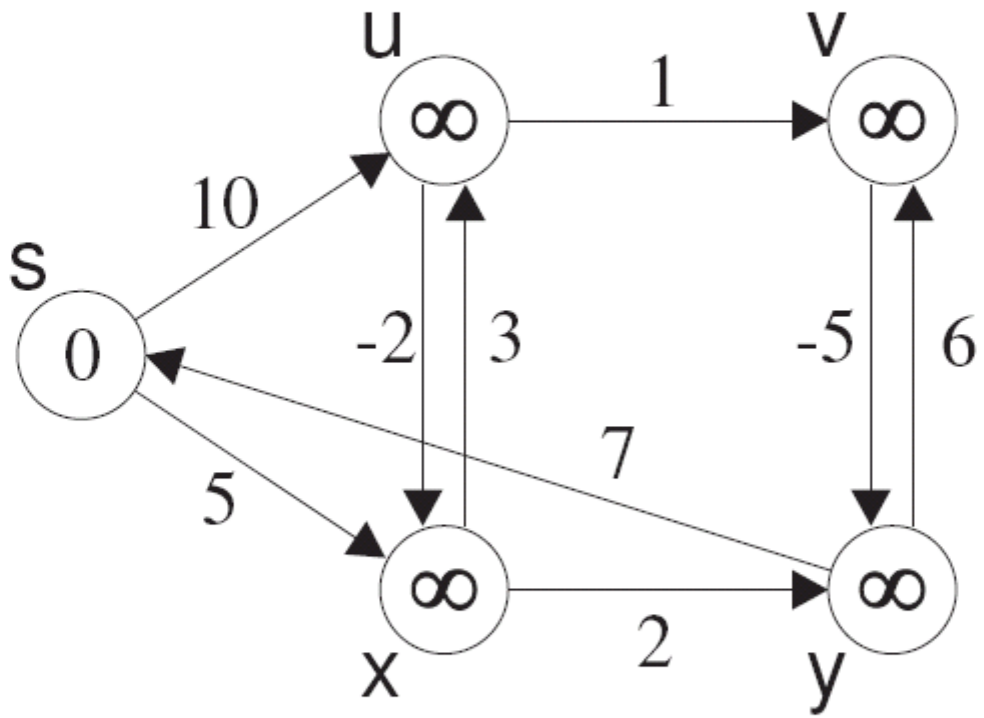
Bellman-Ford-Algorithmus: Prinzip

- Mehrere Durchläufe
  - Bestimme jeweils beste bisher mögliche (um eine Kante längere) Verbindung
  - Der i-te Durchlauf berechnet korrekt alle Pfade vom Startknoten der Länge i
  - Längster Pfad ohne Zyklus hat Länge kleiner als |V|-1
  - Ergebnis also spätestens nach |V|-1 Durchläufen stabil
  - Prinzip der *Dynamischen Programmierung*
- Ist Ergebnis nach |V|-1 Durchläufen nicht stabil, ist negativ bewerteter Zyklus enthalten

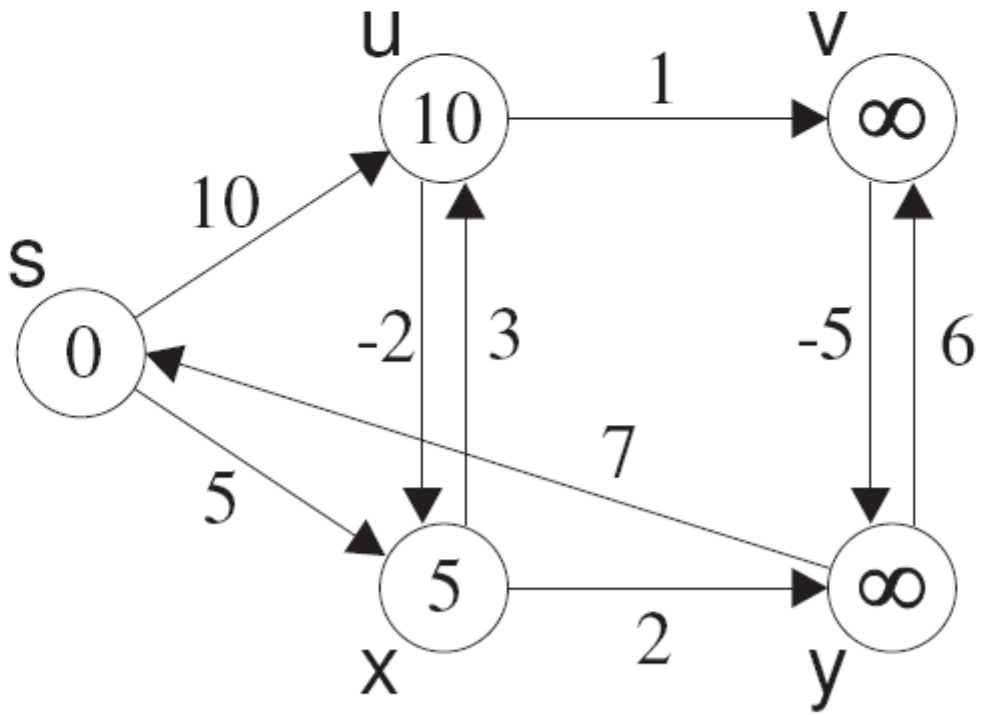
Bellman-Ford-Algorithmus (Pseudocode)

```
algorithm BF(G, s)
    Eingabe: ein Graph G mit Startknoten s

    D[0][s] = 0
    D[0][t] = ∞ for all other t
    for i := 1 to |V|-1 do
        for each v∈V do
            D[i][v] := D[i-1][v]
        od
        for each (u,v)∈ E do
            if D[i-1][u]+γ((u,v)) < D[i][v] then
                D[i][v] := D[i-1][u] + γ((u,v))
            fi
        od
    od
```

Bellman-Ford: Initialisierung

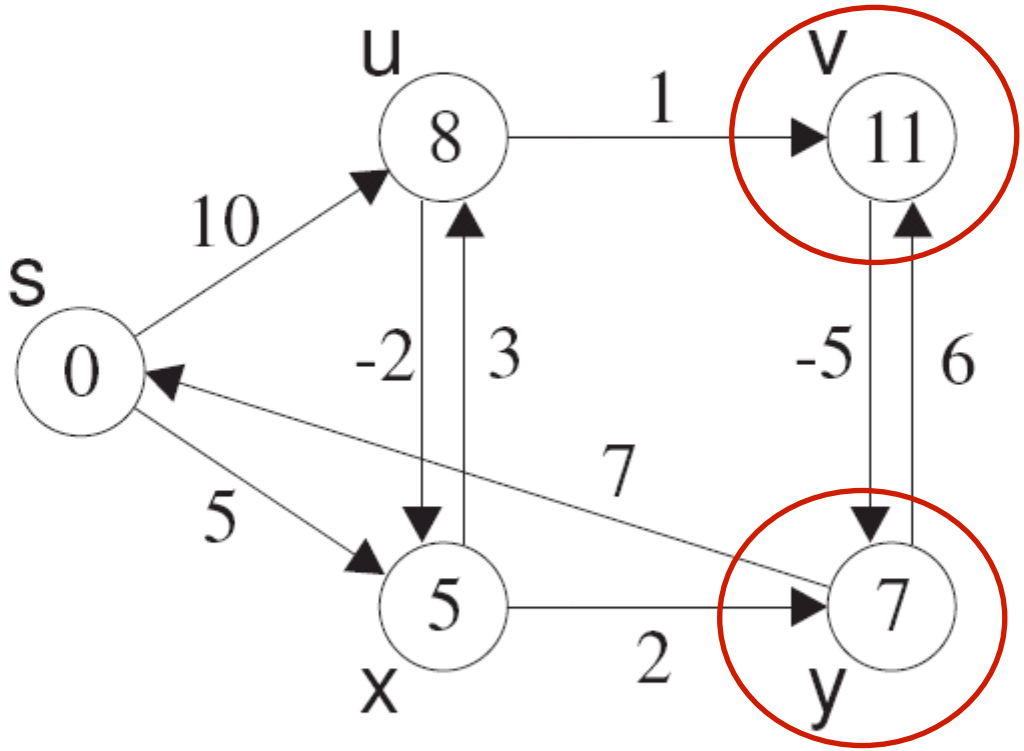


Bellman-Ford: Schleife i=1



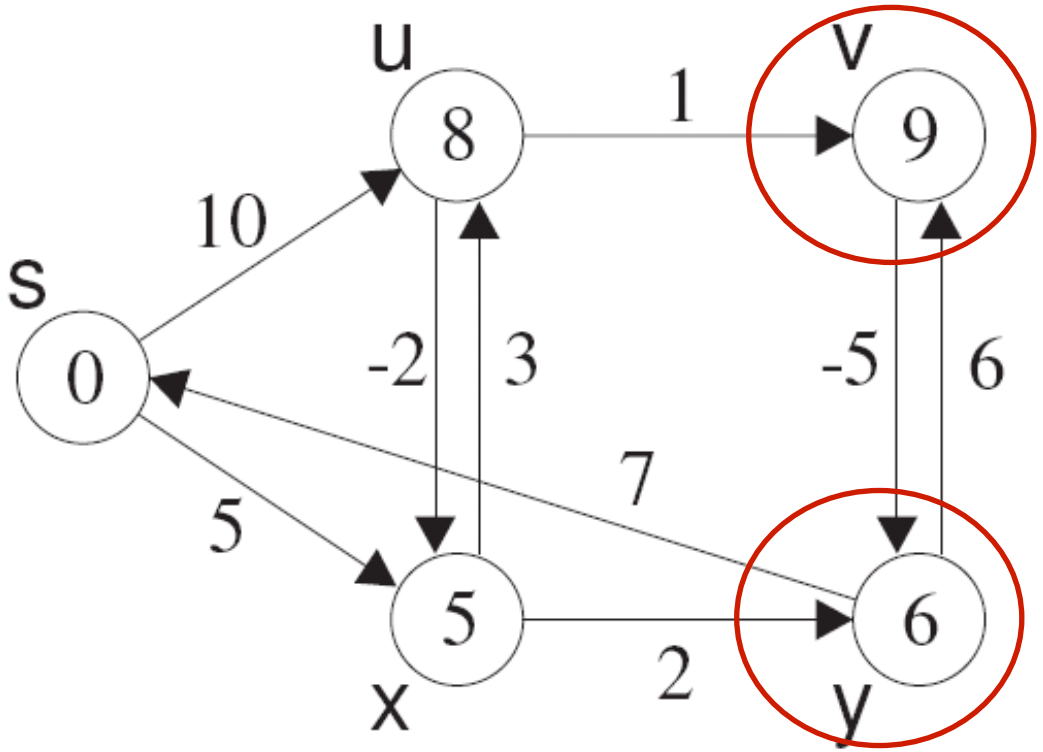
- u hat den Wert 10 und x den Wert 5 bekommen

Bellman-Ford: Schleife i=2



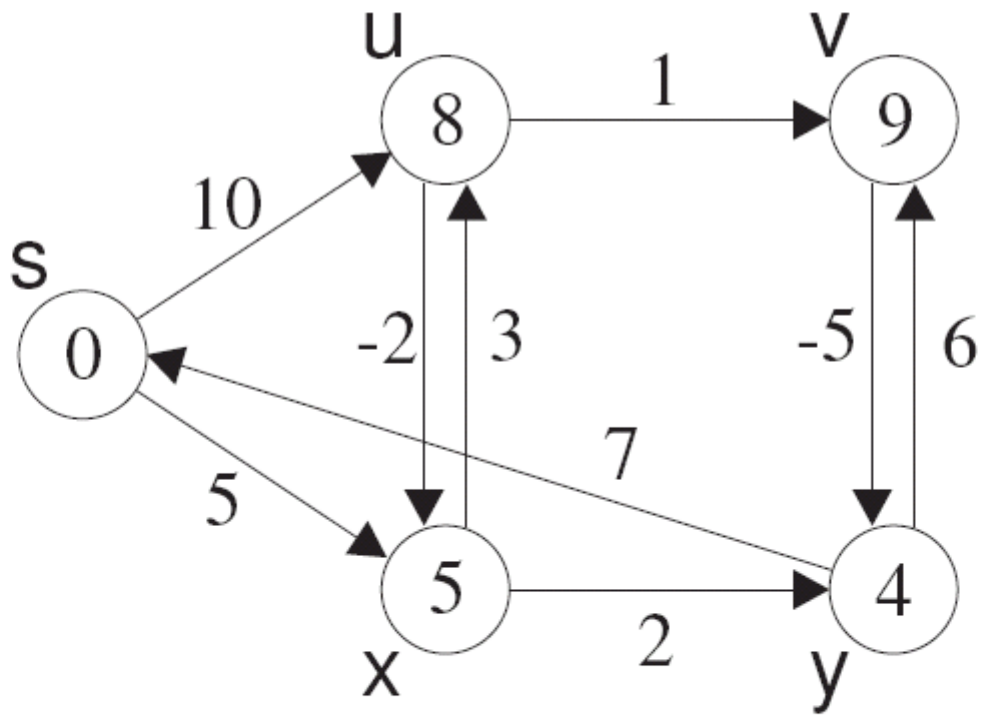
- Knoten u wurde aktualisiert (Pfad über x)
- Änderung an u wird erst im nächsten Schritt an v propagiert

Bellman-Ford: Schleife i=3



- Negativ gewichtete Kante (v,y) zur Berechnung von D[y] genutzt

Bellman-Ford: Schleife i=4



- Negativ gewichtete Kante (v,y) zur Berechnung von D[y] genutzt
- Greedy-Verfahren das jeden Knoten nur einmal besucht, hätte für y den in jedem Schritt lokal optimalen Pfad <s,x,y> gewählt und nicht das beste Ergebnis geliefert

Analyse

Theorem (Terminierung)

Der Algorithmus BF(G,s) terminiert für eine endliche Eingabe G in endlicher Zeit.

Beweis:

Alle Schleifen sind endlich.

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

Analyse

Theorem (Korrektheit)

Ist G ein Graph, der keinen Zyklus mit negativem Gewicht hat, so enthält D nach Aufruf BF(G,s) die Distanzwerte von s zu allen Knoten.

Beweis:

Wir zeigen, dass die folgenden Aussagen Schleifeninvariante der for-Schleife (Schleifenvariable i) sind:

1. Ist D[i][v] < ∞, so ist D[i][v] der Wert *eines* Pfades von s nach v
2. Ist D[i][v] < ∞, so ist D[i][v] der kleinste Wert eines Pfades von s nach v mit *maximal* i Kanten
3. D[i][v] < ∞ gdw. es einen Pfad von s nach v mit gleich oder weniger als i Kanten gibt

Da G keine Zyklen mit negativem Gewicht hat, ist die Länge des längsten kürzesten Pfades maximal |Anzahl Knoten|-1 (jeder Knoten wird auf diesem Pfad einmal besucht). Also gilt nach dem letzten Schleifendurchlauf nach 2 und 3. die Aussage des Theorems.

Analyse

Noch zu zeigen:

1. Ist D[i][v] < ∞, so ist D[i][v] der Wert *eines* Pfades von s nach v
2. Ist D[i][v] < ∞, so ist D[i][v] der kleinste Wert eines Pfades von s nach v mit *maximal* i Kanten
3. D[i][v] < ∞ gdw. es einen Pfad von s nach v mit gleich oder weniger als i Kanten gibt

Wir zeigen diese Aussagen durch Induktion nach i (=#Schleifendurchläufe).

- i=0: Vor dem ersten Schleifendurchlauf gilt nur D[0][s]=0 < ∞. Daraus folgt direkt 1., 2., 3.

Analyse

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

- i → i+1:
  Zunächst 3.:
  - Ist D[i][v] endlich, so ist D[i+1][v] endlich und die Aussage gilt nach IV
  - Ist D[i+1][v] in diesem Schritt auf einen endlichen Wert gesetzt worden, so gab es ein u, so dass D[i][u] vorher schon endlich war und D[i+1][v]=D[i][u]+γ(u,v). Nach IV gibt es einen Pfad von s nach u der Länge i. Damit gibt es einen Pfad der Länge i+1 von s nach v.
  - Umgekehrt wird bei Existenz eines Pfades der Länge i+1 dieser auch gefunden und D[i+1][v] auf einen endlichen Wert gesetzt.

Analyse

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

- i → i+1:
  Zeige 1. „Ist D[i+1][v] < ∞, so ist D[i+1][v] der Wert *eines* Pfades von s nach v“:

  Nach IV is D[i][u] der Wert eines Pfades von s nach u. Wird D[i+1][v]=D[i][u]+γ(u,v) gesetzt so ist somit D[i+1][v] der Wert des Pfades von s nach v über u.

Analyse

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

- i → i+1:
  Zeige 2. „Ist D[i][v] < ∞, so ist D[i][v] der kleinste Wert eines Pfades von s nach v mit *maximal* i Kanten“:

  Nach IV is D[i][v] der kleinste Wert eines Pfades von s nach v mit *maximal* i Kanten“. Mache folgende Fallunterscheidung:
  - 1.Fall: Es existiere ein Pfad P1 von s nach v mit i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat. Betrachte den vorletzen Knoten u auf diesem Pfad und den Teilpfad P2 von P1 von s nach u. Dieser Teilpfad hat minimalen Wert unter allen Pfaden der maximalen Länge i von s nach u (ansonsten wäre P1 kein Pfad mit minimalem Wert). Nach IV ist D[i][u] genau dieser Wert und D[i][u]+γ(u,v) der Wert von P1, der dann im i+1ten Durchgang aktualisiert wird.

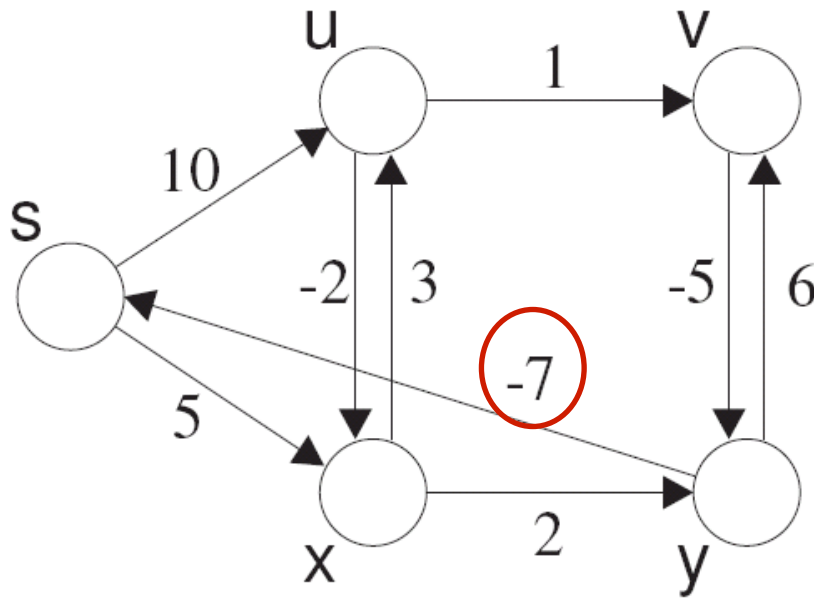
Analyse

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

– 2.Fall: Es existiere *kein* Pfad von s nach v mit i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat.
  - 1. Unterfall: Es existiert kein Pfad von s nach v mit maximal i+1 Kanten. Dann bleibt nach 3. D[i+1][v]=∞.
  - 2. Unterfall: Es existiert ein Pfad von s nach v mit k<i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat. Dann ist nach IV D[i][v] genau dieser Wert und wird im i+1ten Durchgang auch nicht aktualisiert.

Graph mit negativ gewichtetem Zyklus

- Betrachten wir die Situation nach |V|-1 Iterationen
  - Eine Kante könnte noch verbessert werden *gdw*. der Graph einen Zyklus mit **negativer Länge** enthält
- Beispiel:



- Zyklus s, x, u, v, y, s hat die Kosten 5+3+1-5-7 = -3
- Jeder Durchlauf durch den Zyklus erzeugt also einen Gewinn, es gibt hier keinen günstigsten Pfad endlicher Länge!

Analyse

Theorem (Laufzeit)

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand vom Algorithmus von Bellmann-Ford für einen beliebigen Knoten s in G ist O(|V||E|).

Beweis:

Einfache Schleifenanalyse.

```
algorithm BF(G, s)
      Eingabe: ein Graph G mit Startknoten s

      D[0][s] = 0
      D[0][t] = ∞ for all other t
      for i := 1 to |V|-1 do
            for each v∈V do
                  D[i][v] := D[i-1][v]
            od
            for each (u,v)∈ E do
                  if D[i-1][u]+γ((u,v)) < D[i][v] then
                        D[i][v] := D[i-1][u] + γ((u,v))
                  fi
            od
      od
```

Algorithmen und Datenstrukturen

10.3 DER ALGORITHMUS VON FLOYD-WARSHALL

All Pairs Shortest Paths: Floyd-Warshall

- Dijkstras Algorithmus und Bellman-Ford berechnen zu einem gegebenen Startknoten die kürzesten Wege zu allen anderen Knoten (Single Source Shortest Paths – SSSP)
- Berechnung aller kürzesten Wege zwischen allen Knoten v und w?
  - Aufruf obiger Algorithmen für jeden Startknoten einzeln
  - Geht das geschickter?

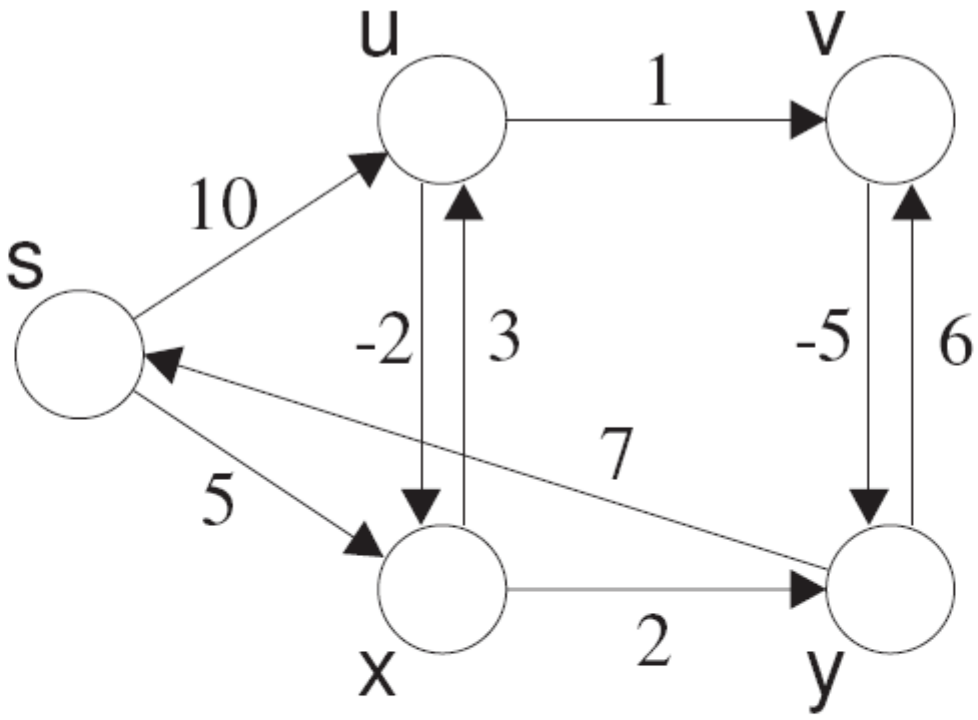
Antwort: Der Algorithmus von Floyd-Warschall löst das All Pairs Shortest Paths (APSP) Problem (nicht unbedingt effizienter aber eleganter),
Prinzip: *Dynamische Programmierung*

All Pairs Shortest Paths: Problemdefinition (Wdh.)

- Gegeben Graph G=(V,E), finde für jedes Paar (v,w)∈VxV den Wert D(v,w) eines kürzesten Pfades

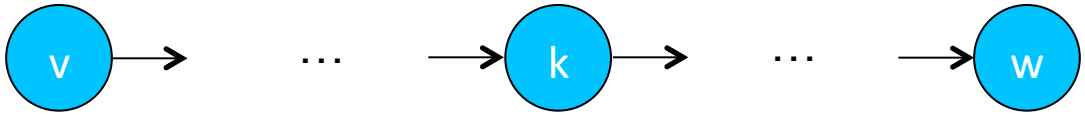
- Annahme: keine negativen Kreise!

D	s	u	v	x	y
s	0	8	9	5	4
u	3	0	1	-2	-4
v	2	10	0	7	-5
x	6	3	4	0	-1
y	7	15	6	12	0



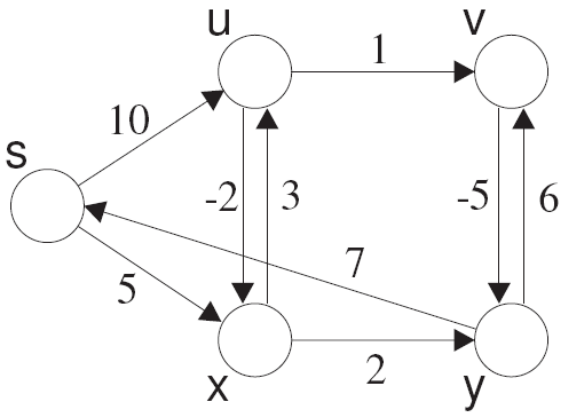
Floyd-Warshall: Idee 1/3

- Die Grundidee des Floyd-Warshall Algorithmus ist die folgende Beobachtung:



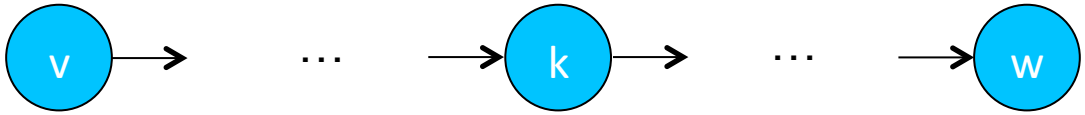
  - Geht ein kürzester Weg $\{(v,a_1),\ldots,(a_n,k), (k,a_{n+1}),\ldots, (a_m,w)\}$ von v nach w über k, dann gilt
    - $\{(v,a_1),\ldots,(a_n,k)\}$ ist ein kürzester Weg von v nach k
    - $\{(k,a_{n+1}),\ldots,(a_m,w)\}$ ist ein kürzester Weg von k nach w

- s → y: {(s,x),(x,u),(u,v),(v,y)}
- s → u: {(s,x),(x,u)}
- u → y: {(u,v),(v,y)}



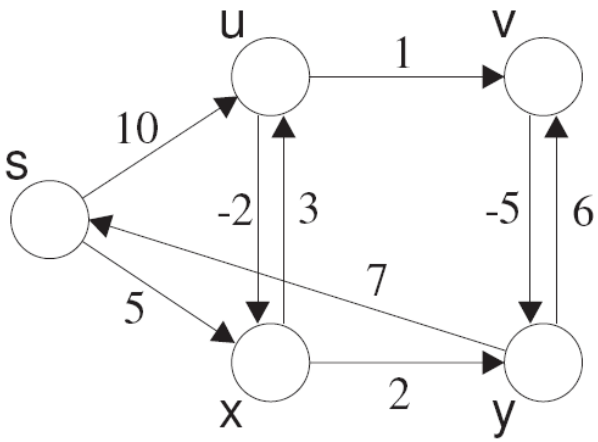
Floyd-Warshall: Idee 2/3

- Beachte, dass die Umkehrung nicht gilt!



- Ist $\{(v,a_1),\ldots,(a_n,k)\}$ ein kürzester Weg von v nach k und ist $\{(k,a_{n+1}),\ldots,(a_m,w)\}$ ein kürzester Weg von k nach w dann gilt **nicht notwendigerweise**, dass
  - $\{(v,a_1),\ldots,(a_n,k), (k,a_{n+1}),\ldots,(a_m,w)\}$ ein kürzester Weg von v nach w ist!

- $x \rightarrow y$: $\{(x,y)\}$
- $y \rightarrow v$: $\{(y,v)\}$
- $x \rightarrow v$: $\{(x,y),(y,v)\}$ ist nicht k. W.!



WeST

Floyd-Warshall: Idee 3/3

- Jedoch gilt:
  - Ist bekannt, dass ein kürzester Weg zwischen v und w nur Knoten aus V' $\subseteq$ V enthält so gilt entweder
    - kürzester Weg zwischen v und w benutzt nur Knoten aus V'\{k}, oder
    - kürzester Weg zwischen v und w ist Konkatenation aus
      - kürzestem Weg zwischen v und k und
      - kürzestem Weg zwischen k und w
      - und beide Wege enthalten nur Knoten aus V'\{k}

$$D^{V'}[i,j] = \begin{cases} \gamma(i,j) & \text{falls } V' = \emptyset \\ \min\{D^{V'\setminus\{k\}}[i,j], D^{V'\setminus\{k\}}[i,k] + D^{V'\setminus\{k\}}[k,j]\} & \text{für } k \in V' \end{cases}$$

Floyd-Warshall Algorithmus (Pseudocode)

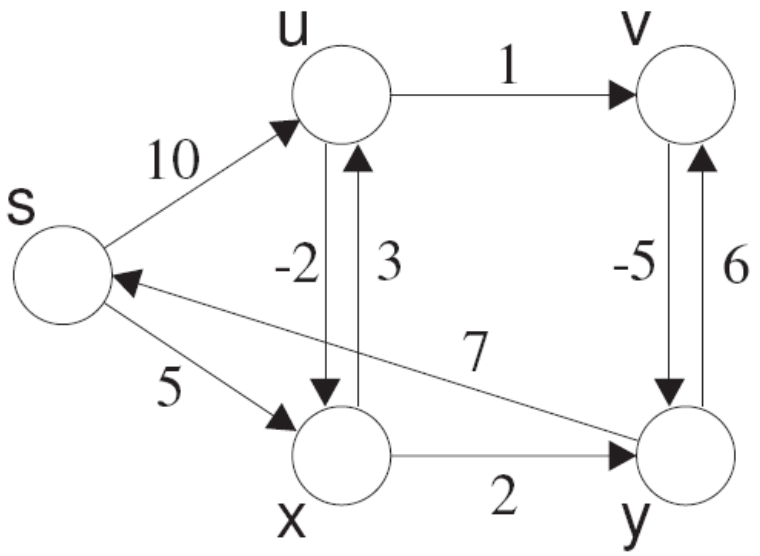
```
algorithm FW(G)
    Eingabe: ein Graph G

    for each v,v'∈V
            D[v,v'] = γ((v,v')) (or ∞)
    for each k ∈ V do
        for each i ∈ V do
            for each j ∈ V do
                if D[i,k]+D[k,j] < D[i,j] then
                    D[i,j] := D[i,k]+D[k,j]
                fi
            od
        od
    od
```

Beispiel

Initialisiere D mit Kantengewichten (nicht vorhandene Kante haben Gewicht ∞, Kantengewicht zum Knoten selber ist 0)

Anmerkung: Im Folgenden betrachten wir nur solche Schleifendurchgänge mit k≠i, k≠j, i≠j

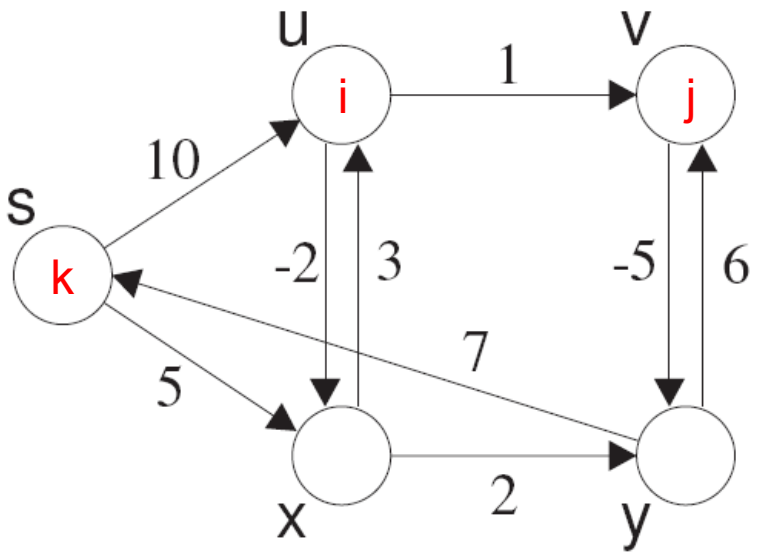
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[u,s] + D[s,v] < D[u,v] ?

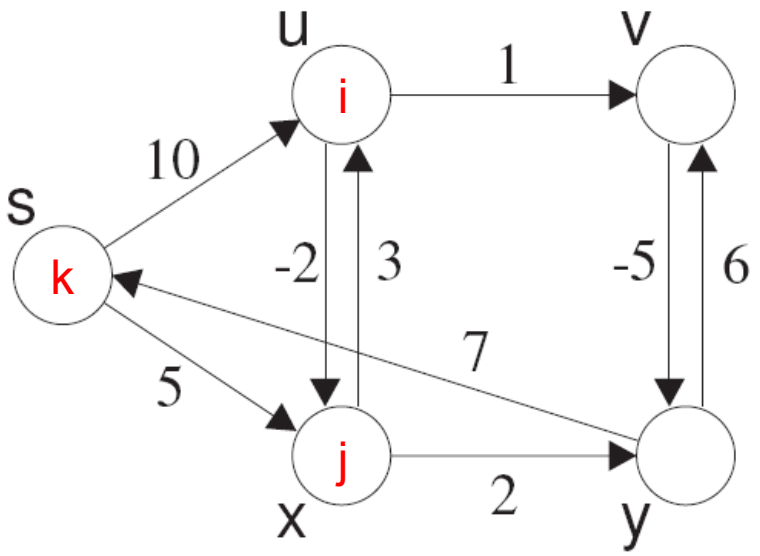
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[u,s] + D[s,x] < D[u,x] ?

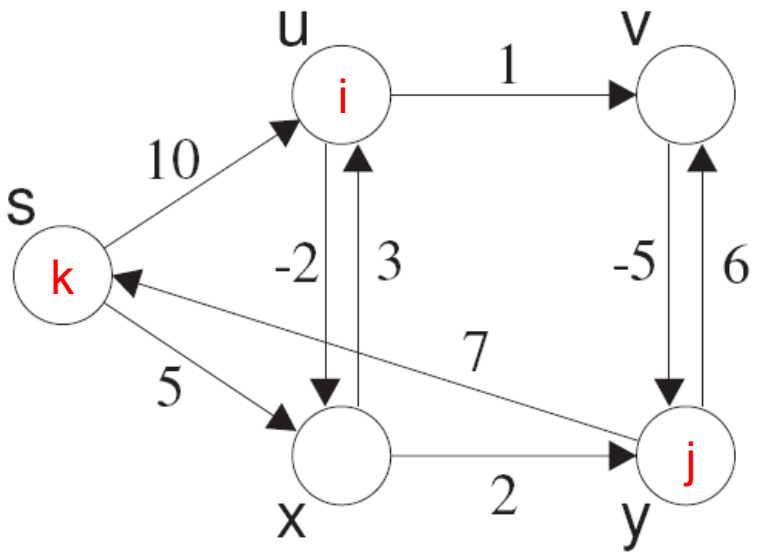
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[u,s] + D[s,y] < D[u,y] ?

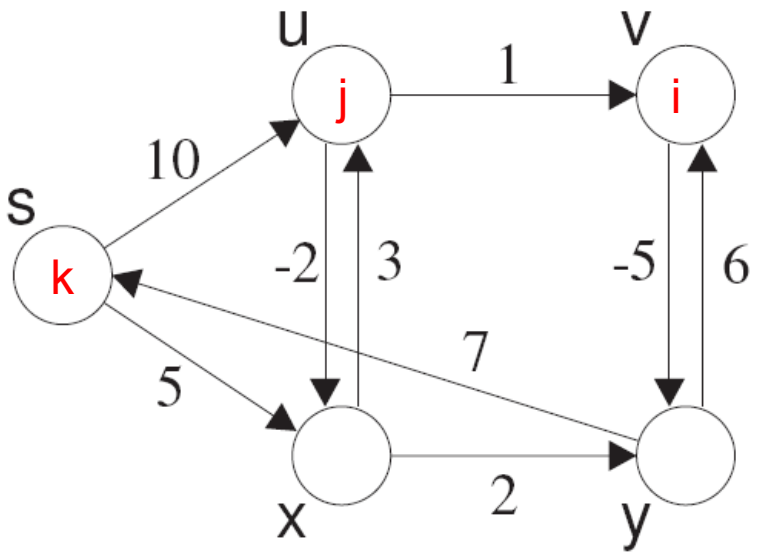
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[v,s] + D[s,u] < D[v,u] ?

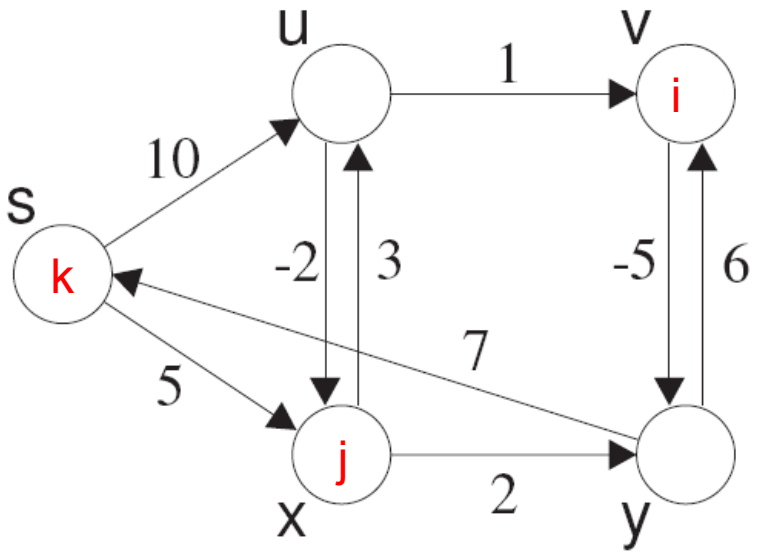
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[v,s] + D[s,x] < D[v,x] ?

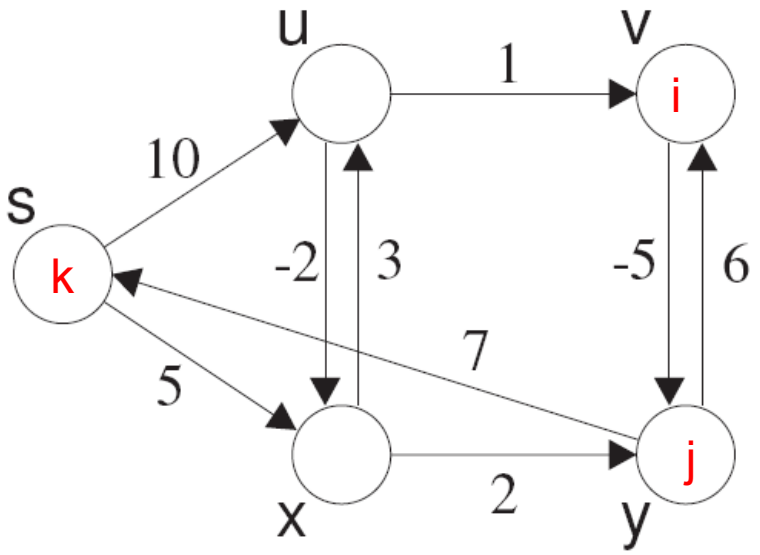
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[v,s] + D[s,y] < D[v,y] ?

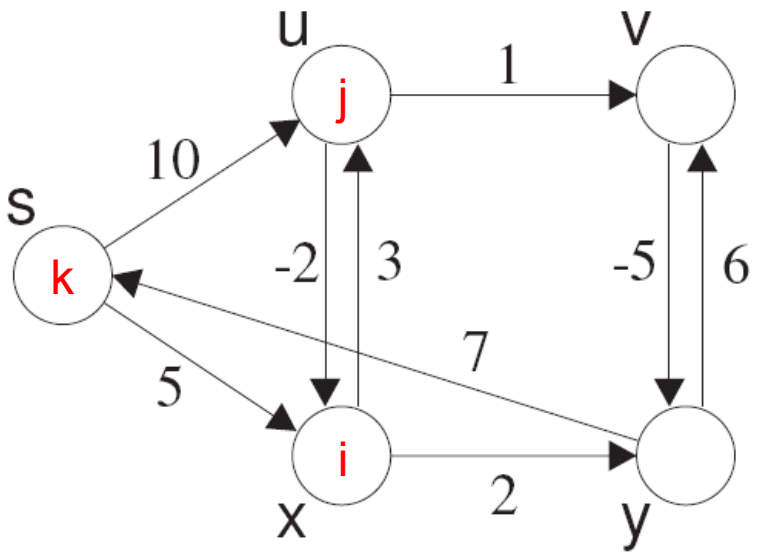
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[x,s] + D[s,u] < D[x,u] ?

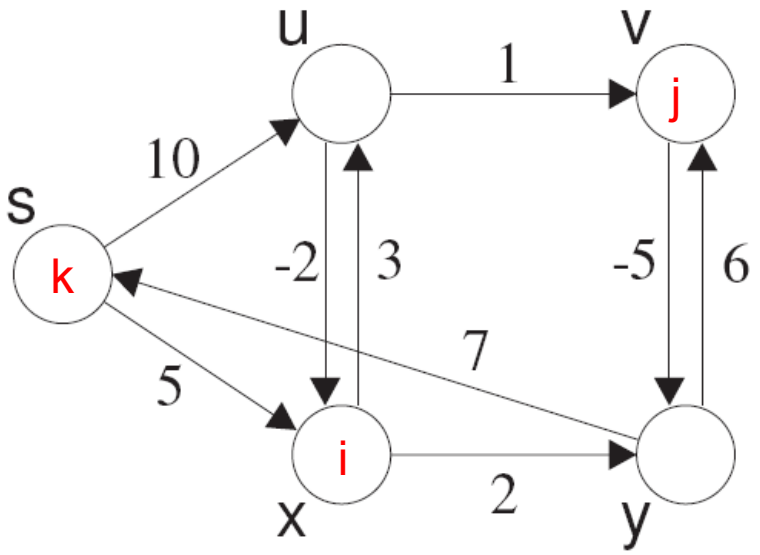
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[x,s] + D[s,v] < D[x,v] ?

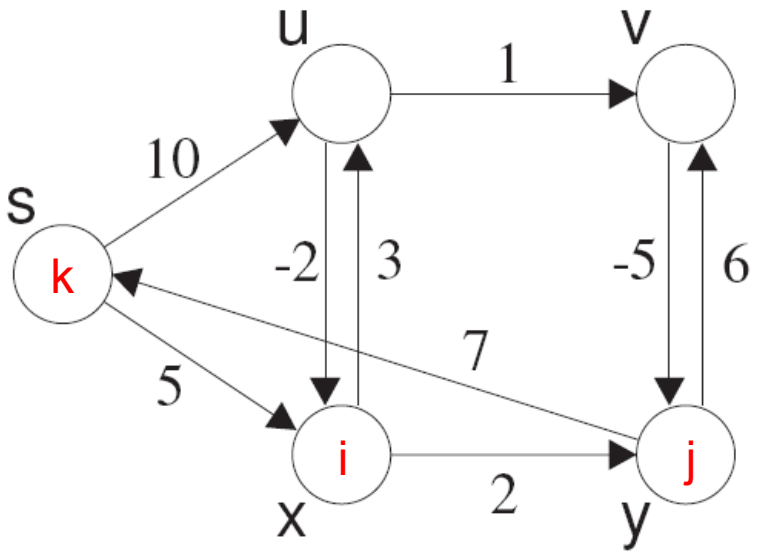
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[x,s] + D[s,y] < D[x,y] ?

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0

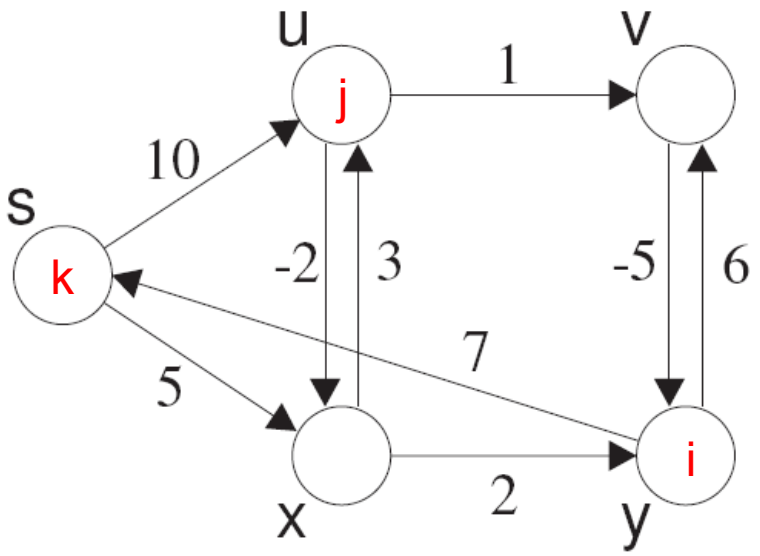


Beispiel

D[y,s] + D[s,u] < D[y,u] ?

➔ D[y,u] = D[y,s] + D[s,u] = 7 + 10 = 17

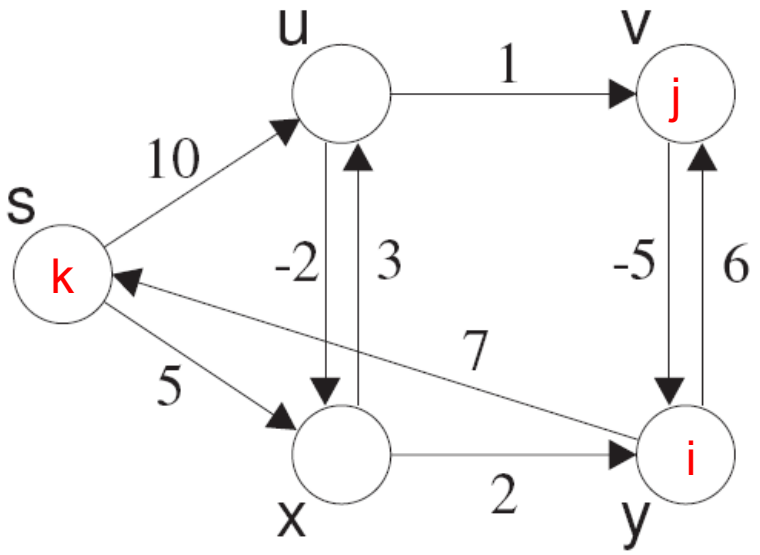
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0



Beispiel

D[y,s] + D[s,v] < D[y,v] ?

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	17	6	∞	0

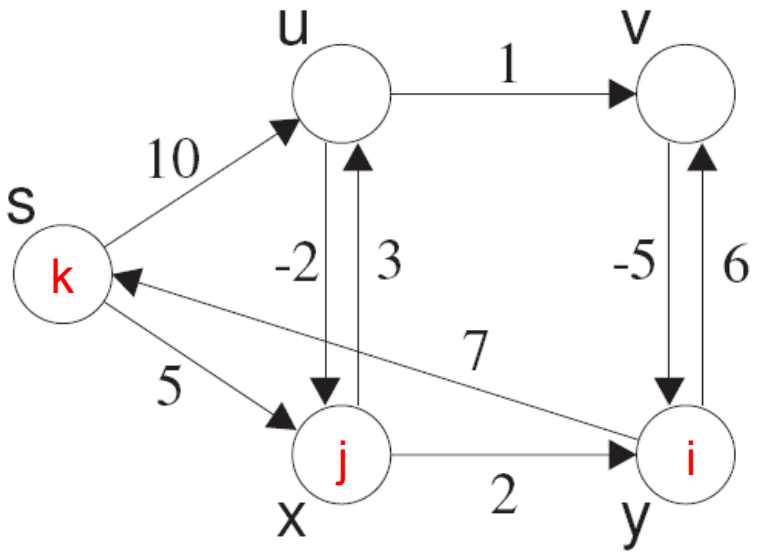


Beispiel

D[y,s] + D[s,x] < D[y,x] ?

➔ D[y,x] = D[y,s] + D[s,x] = 7 + 5 = 12

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	17	6	∞	0

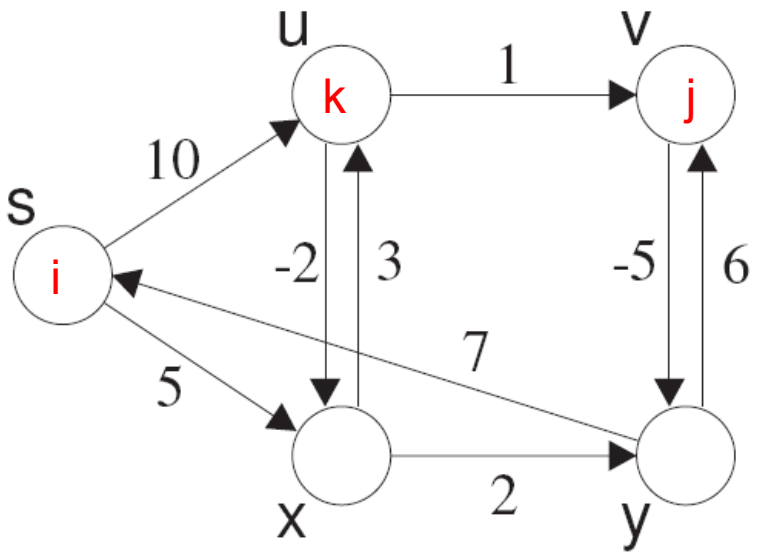


Beispiel

D[s,u] + D[u,v] < D[s,v] ?

➔ D[s,v] = D[s,u] + D[u,v] = 10 + 1 = 11

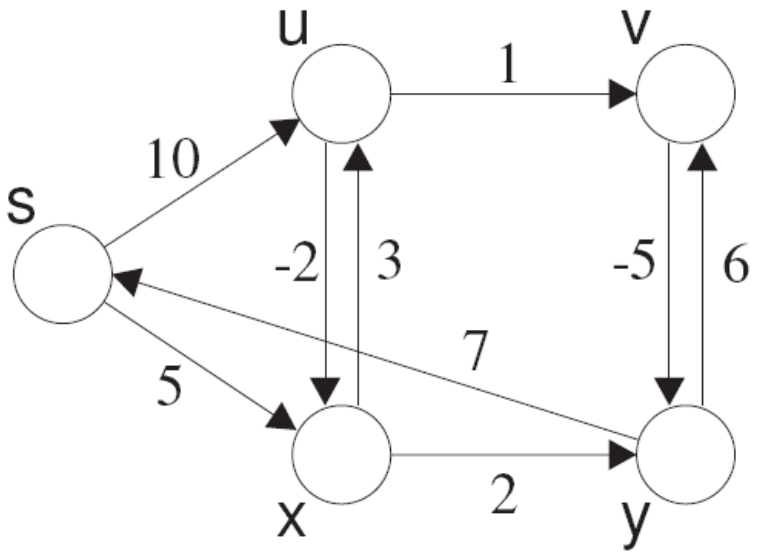
D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	17	6	12	0



Beispiel

- …

D	s	u	v	x	y
s	0	8	9	5	4
u	3	0	1	-2	-4
v	2	10	0	7	-5
x	6	3	4	0	-1
y	7	15	6	12	0



Analyse

Theorem (Terminierung)

Der Algorithmus FW(G) terminiert für eine endliche Eingabe G in endlicher Zeit.

Beweis:

Alle Schleifen sind endlich.

```
algorithm FW(G)
    Eingabe: ein Graph G

      for each v,v'∈V
            D[v,v'] = γ((v,v')) (or ∞)
      for k ∈ V do
          for each i ∈ V do
                    for each j ∈ V do
                         if D[i,k]+D[k,j] < D[i,j] then
                              D[i,j] := D[i,k]+D[k,j]
                              fi
               od
          od
      od
```

Analyse

Theorem (Korrektheit)

Ist G ein Graph, der keinen Zyklus mit negativem Gewicht hat, so enthält D nach Aufruf FW(G) die Distanzwerte von allen Knoten zu allen anderen Knoten.

Beweis:

Betrachte folgende Schleifeninvariante (äußerste for-Schleife mit Laufvariable k):

> Nach der k-ten Schleifeniteration gilt, dass D[v,w] (für alle v,w) der Wert eines kürzesten Pfades ist, der nur Knoten 1,…,k benutzt

Nach Beendigung des Algorithmus gilt damit die Aussage des Theorems.

Analyse

Noch zu zeigen:

> Nach der k-ten Schleifeniteration gilt, dass D[v,w] (für alle v,w) der Wert eines kürzesten Pfades ist, der nur Knoten 1,…,k benutzt

Zeige dies durch Induktion:

- k=0 (Initialisierung): Nach der Initialisierung gilt D[v,w]=∞ gdw. es keine Kante von v nach w gibt (dann muss jeder Pfad zwischen v und w mind. einen anderen Knoten enthalten). Ist D[v,w] endlich so ist dies genau der Wert der Kante (es gibt also einen Pfad, der keine weiteren Knoten beinhaltet).

Analyse

- k→ k+1: Nach Induktionsannahme ist D[v,w] der Wert eines kürzesten Pfades, der nur Knoten aus 1,…,k enthält. Im k+1-Schleifendurchgang wird überprüft, ob es einen kürzeren Weg über k+1 gibt und ggfs. aktualisiert. Es wird also genau folgende Gleichung ausgenutzt:

$$D^{V'}[i,j] = \begin{cases} \gamma(i,j) & \text{falls } V' = \emptyset \\ \min\{D^{V'\setminus\{k\}}[i,j], D^{V'\setminus\{k\}}[i,k] + D^{V'\setminus\{k\}}[k,j]\} & \text{für } k \in V' \end{cases}$$

Anschliessend ist also D[v,w] der Wert eines kürzesten Pfades ist, der nur Knoten 1,…,k+1 benutzt. □

Analyse

Theorem (Laufzeit)

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand vom Algorithmus von Floyd-Warshall auf G ist $O(|V|^3)$.

Beweis:

```
algorithm FW(G)
    Eingabe: ein Graph G

      for each v,v'∈V
            D[v,v'] = γ((v,v')) (or ∞)
      for each k ∈ V do
          for each i ∈ V do
                    for each j ∈ V do
                         if D[i,k]+D[k,j] < D[i,j] then
                              D[i,j] := D[i,k]+D[k,j]
                              fi
                od
          od
      od
```

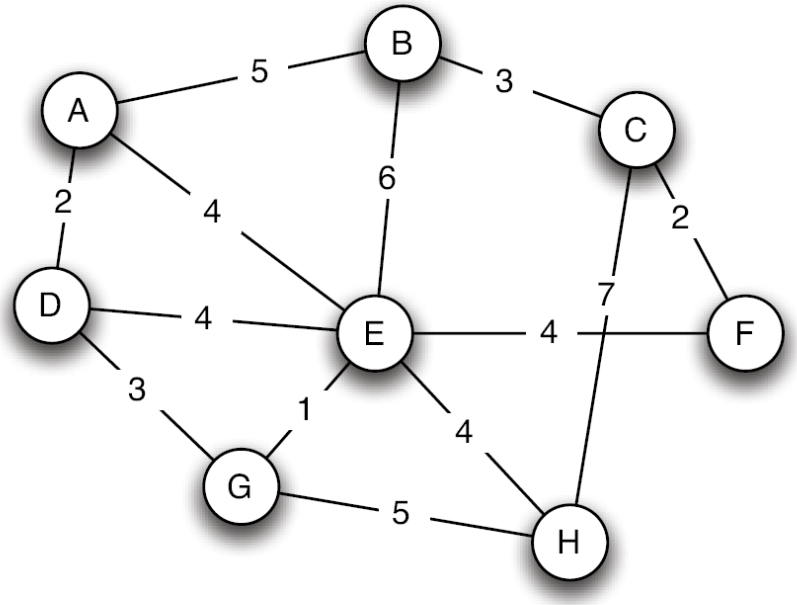
Einfache Schleifenanalyse. □

Algorithmen und Datenstrukturen

10.4 DAS TRAVELING SALESMAN PROBLEM

Traveling Salesman Problem (TSP)

- Gegeben: Graph mit *n* Knoten (Städten) und *m* Kanten (Straßen); Straßen sind mit Entfernung versehen



- Gesucht: kürzeste Route, die alle Städte besucht und an den Startpunkt zurückkehrt

TSP im Detail

- Gegeben:
  - *n* Städte
  - n *x n* Entfernungsmatrix $M = (m_{i,j})$
  - $m_{i,j}$ Entfernung von Stadt i nach Stadt j (evtl. ∞)
- Gesucht:
  - Rundreise über alle Städte mit insgesamt minimaler Länge
  - Permutationen π : {1,….,*n*} → {1,….,*n*}
    d.h. π(*i*) gibt an, welche Stadt im *i*-ten Schritt besucht wird
  - Gesucht wird Permutation π mit

$$C(\pi) = m_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} m_{\pi(i),\pi(i+1)} \text{ ist minimal.}$$

TSP mit dynamischer Programmierung (nach Held, Karp)

- betrachte g(i,S): Länge des kürzesten Weges von Stadt i über jede Stadt in S nach Stadt 1
  - Da Rundreise, kann Stadt 1 beliebig gewählt werden
- Es gilt:

$$g(i,S) = \begin{cases} m_{i,1} & \textbf{falls } S = \emptyset \\ \min_{j \in S}(m_{i,j} + g(j, S - \{j\})) & \textbf{sonst} \end{cases}$$

- Idee: baue Rundreisen von hinten schrittweise auf
- Lösung: g(1,{2,…,n})

TSP: HK-Algorithmus (Pseudocode)

```
for i = 2 to n do g[i,{}] = m_{i,1} od;
for k = 1 to n-2
    for all S with |S| = k and 1 ∉ S do
        berechne g[i,S] nach Formel;
    od;
berechne g[1,{2,...,n }] nach Formel
```

Rückwege

Teillösungen/-reisen

Gesamte Reise

TSP: Beispiel

- 4 Städte, symmetrische Entfernungen
  M=

i \ j	1	2	3	4
1	0	4	9	8
2	4	0	12	2
3	9	12	0	10
4	8	2	10	0

- Initialisierung
  (Länge 1 = Rückkehr von der letzten Station nach Station 1):

  g[2 , { }] = 4
  g[3 , { }] = 9
  g[4 , { }] = 8

TSP: Beispiel II

Die letzten 2 Schritte inklusive Rückkehr zur Station 1:

g[2 , { 3 }] = 12 + g[3, {}] = 12 + 9 = 21

g[2 , { 4 }] = 2 + 8 = 10

g[3 , { 2 }] = 12 + 4 = 16

g[3 , { 4 }] = 10 + 8 = 18

g[4 , { 2 }] = 2 + 4 = 6

g[4 , { 3 }] = 10 + 9 = 19

	1	2	3	4
1	0	4	9	8
2	4	0	12	2
3	9	12	0	10
4	8	2	10	0

TSP: Beispiel III

Lösung: 1,2,4,3,1

Die letzten 4 Schritte, d.h. komplette Rundreise

$g[\ 1,\{\ 2\ ,\ 3\ ,\ 4\ \}\ ] = \min(m_{1,2} + g[2,\{3,4\}],\ m_{1,3} + g[3,\{2,4\}],\ m_{1,4} + g[4,\{2,3\}]) = 25$

Die letzten 3 Schritte inklusive Rückkehr zu Station 1:

$g[\ 2\ ,\ \{\ 3\ ,\ 4\ \}\ ] = \min(m_{2,3} + g[3,\{4\}],\ m_{2,4} + g[4,\{3\}]) = \min(12+18,\ 2+19) = 21$

$g[\ 3\ ,\ \{\ 2\ ,\ 4\ \}\ ] = \min(m_{3,2} + g[2,\{4\}],\ m_{3,4} + g[4,\{2\}]) = \min(12+10,\ 10+6) = 16$

$g[\ 4\ ,\ \{\ 2\ ,\ 3\ \}\ ] = \min(m_{4,2} + g[2,\{3\}],\ m_{4,3} + g[3,\{2\}]) = \min(2+21,\ 10+16) = 23$

Die letzten 2 Schritte inklusive Rückkehr zur Station 1:

$g[\ 2\ ,\ \{\ 3\ \}\ ] = 12 + 9 = 21$

$g[\ 2\ ,\ \{\ 4\ \}\ ] = 2 + 8 = 10$

$g[\ 3\ ,\ \{\ 2\ \}\ ] = 12 + 4 = 16$

$g[\ 3\ ,\ \{\ 4\ \}\ ] = 10 + 8 = 18$

$g[\ 4\ ,\ \{\ 2\ \}\ ] = 2 + 4 = 6$

$g[\ 4\ ,\ \{\ 3\ \}\ ] = 10 + 9 = 19$

	1	2	3	4
1	0	4	9	8
2	4	0	12	2
3	9	12	0	10
4	8	2	10	0

Analyse

Theorem (Korrektheit)

Der HK-Algorithmus löst das TSP.

Ohne Beweis

Theorem (Laufzeit)

Der HK-Algorithms hat eine Laufzeit von $O(n^2 2^n)$.

Ohne Beweis

Anmerkung: TSP ist ein NP-vollständiges Problem, d.h. vermutlich existiert kein polynomieller Algorithmus.

Algorithmen und Datenstrukturen

10. KÜRZESTE WEGE
ZUSAMMENFASSUNG

Zusammenfassung

- Algorithmus von Djikstra
  - Löst SSSP
  - Laufzeit $O(|E| + |V| \log |V|)$ (optimiert)
- Algorithmus von Bellmann-Ford
  - Löst SSSP
  - Laufzeit $O(|V||E|)$
- Algorithmus von Floyd-Warshall
  - Löst APSP
  - Laufzeit $O(|V|^3)$
- Das Traveling Salesman Problem