Algorithmen und Datenstrukturen 2

Sommersemester 2007
4. Vorlesung

Peter F. Stadler

Universität Leipzig
Institut für Informatik
studla@bioinf.uni-leipzig.de

Traversierung

Durchlaufen eines Graphen, bei dem jeder Knoten (bzw. jede Kante) genau 1-mal aufgesucht wird

- Beispiel 1: Aufsuchen aller Verbindungen (Kanten) und Kreuzungen (Knoten) in einem Labyrinth
- Beispiel 2: Aufsuchen aller Web-Server durch Suchmaschinen-Roboter

Generische Lösungsmöglichkeit für Graphen G=(V, E)

for each Knoten $v \in V$ **do** { markiere v als unbearbeitet};

B = {s}; // Initialisierung der Menge besuchter Knoten B mit Startknoten $s \in V$;

markiere s als bearbeitet;

while es gibt noch unbearbeitete Knoten v' mit $(v,v') \in E$ und $v \in B$ **do** {

  $B = B \cup \{v'\}$;

  markiere v' als bearbeitet;

  };

}

Realisierungen unterscheiden sich bezüglich Verwaltung der noch abzuarbeitenden Knotenmenge und Auswahl der jeweils nächsten Kante

Breiten- und Tiefendurchlauf

Breitendurchlauf (Breadth First Search, BFS)

- ausgehend von Startknoten werden zunächst alle direkt erreichbaren Knoten bearbeitet
- danach die über mindestens zwei Kanten vom Startknoten erreichbaren Knoten, dann die über drei Kanten usw.
- es werden also erst die Nachbarn besucht, bevor zu den Söhnen gegangen wird
- kann mit FIFO-Datenstruktur für noch zu bearbeitende Knoten realisiert werden

Tiefendurchlauf (Depth First Search, DFS)

- ausgehend von Startknoten werden zunächst rekursiv alle Söhne (Nachfolger) bearbeitet; erst dann wird zu den Nachbarn gegangen
- kann mit Stack-Datenstruktur für noch zu bearbeitende Knoten realisiert werden
- Verallgemeinerung der Traversierung von Bäumen

Algorithmen nutzen "Farbwert" pro Knoten zur Kennzeichnung des Bearbeitungszustandes

- weiß: noch nicht bearbeitet
- schwarz: abgearbeitet
- grau: in Bearbeitung

Breitensuche I

Bearbeite einen Knoten, der in n Schritten von u erreichbar ist, erst, wenn alle Knoten, die in n-1 Schritten erreichbar sind, abgearbeitet wurden.

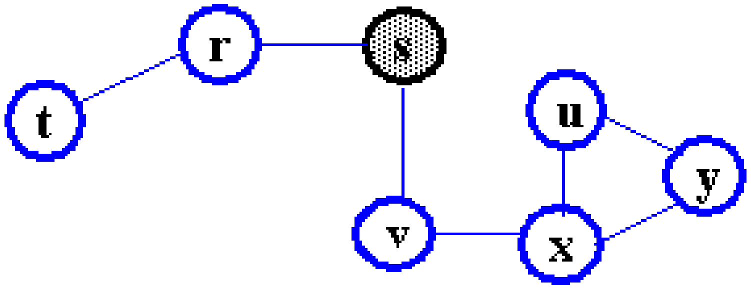
- ungerichteter Graph G = (V,E); Startknoten s; Q sei FIFO-Warteschlange.
- zu jedem Knoten u wird der aktuelle Farbwert, der Abstand *d* zu Startknoten s, und der Vorgänger *pred*, von dem aus u erreicht wurde, gespeichert
- Funktion *succ*(u) liefert die Menge der direkten Nachfolger von u
- *pred*-Werte liefern nach Abarbeitung für zusammenhängende Graphen einen aufspannenden Baum (Spannbaum), ansonsten Spannwald

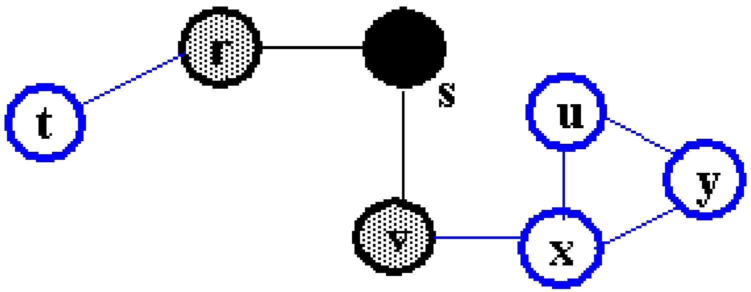
Algorithmus BFS(G,s):

```
for each Knoten v ∈ V - s do { farbe[v]= weiß; d[v] = ∞; pred [v] = null };
farbe[s] = grau; d[s] = 0; pred [s] = null; Q = emptyQueue; Q = enqueue(Q,s);
while not isEmpty(Q) do { v = front(Q);
            for each u ∈ succ (v) do {
                        if farbe(u) = weiß then
                        { farbe[u] = grau; d[u] = d[v]+1; pred[u] = v; Q = enqueue(Q,u); };
            };
            dequeue(Q); farbe[v] = schwarz;
}
```

Breitensuche II

Beispiel





Komplexität: ein Besuch pro Kante und Knoten: O(n + m)

- falls G zusammenhängend gilt |E| > |V|-1 -> Komplexität O(m)

Breitensuche unterstützt Lösung von Distanzproblemen, z.B. Berechnung der Länge des *kürzesten Wegs* eines Knoten s zu anderen Knoten

Tiefensuche I

Bearbeite einen Knoten v erst dann, wenn alle seine Söhne bearbeitet sind
(außer wenn ein Sohn auf dem Weg zu v liegt)

- (un-)gerichteter Graph G = (V,E); *succ*(v) liefert Menge der direkten Nachfolger von Knoten v
- zu jedem Knoten v wird der aktuelle Farbwert, die Zeitpunkte *in* bzw. *out*, zu denen der Knoten im Rahmen der Tiefensuche erreicht bzw. verlassen wurden, sowie der Vorgänger *pred*, von dem aus v erreicht wurde, gespeichert
- die *in*- bzw. *out*-Zeitpunkte ergeben eine Reihenfolge der Knoten analog zur Vor- bzw. Nachordnung bei Bäumen

Tiefensuche II

Algorithmus:

DFS(G):

for each Knoten v $\in$ V **do** { farbe[v]= weiß; pred [v] = null };

zeit = 0; **for each** Knoten v $\in$ V **do** { **if** farbe[v]= weiß **then** DFS-visit(v) };

DFS-visit (v): **// rekursive Methode zur Tiefensuche**

farbe[v]= grau; zeit = zeit+1; in[v]=zeit;

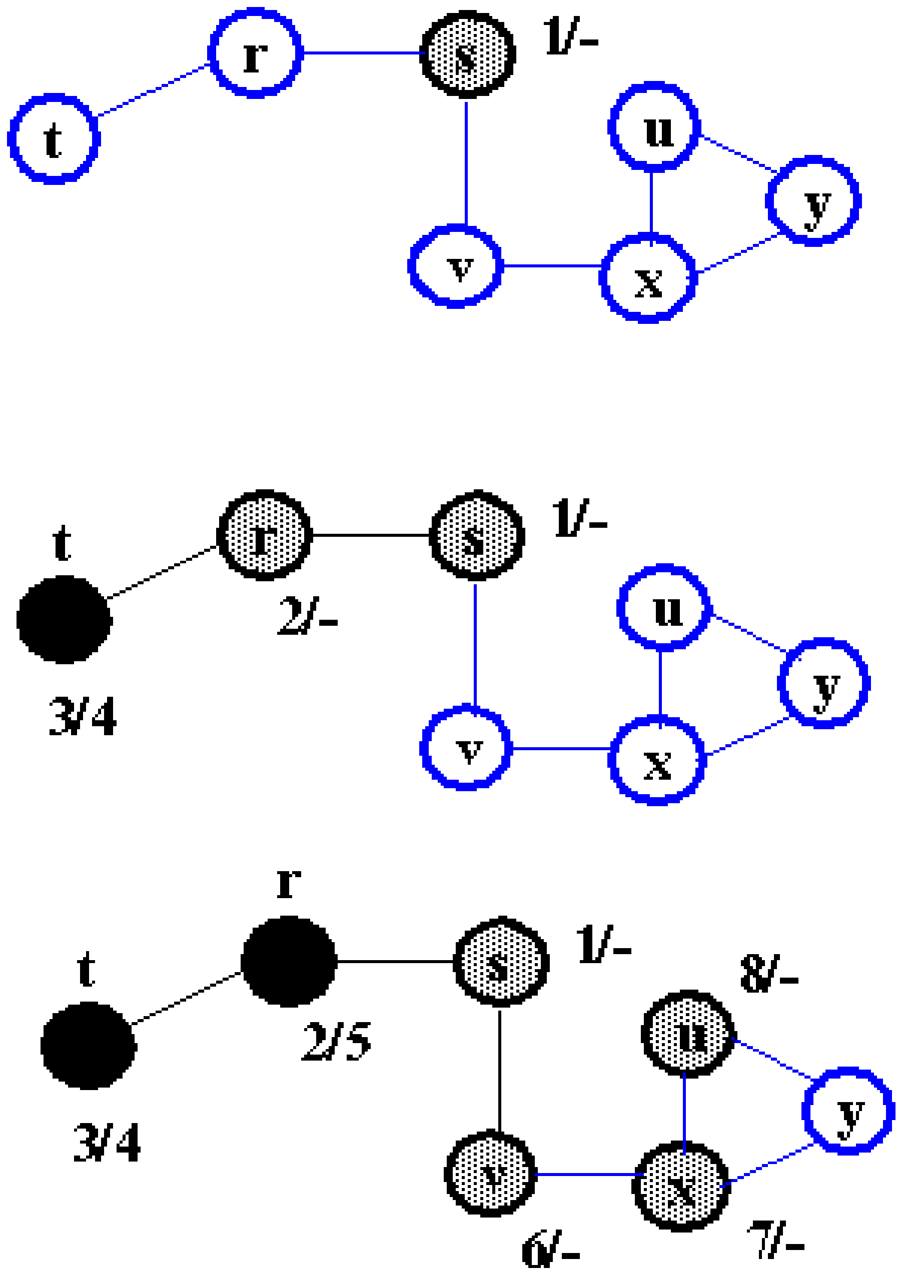
for each u $\in$ succ (v) **do** { **if** farbe(u) = weiß **then** { pred[u] = v; DFS-visit(u); }; };

farbe[v] = schwarz; zeit = zeit+1; out[v]=zeit;

lineare Komplexität O(n+m)

- DFS-visit wird genau einmal pro (weißem) Knoten aufgerufen
- pro Knoten erfolgt Schleifendurchlauf für jede von diesem Knoten ausgehende Kante

Tiefensuche: Beispiel



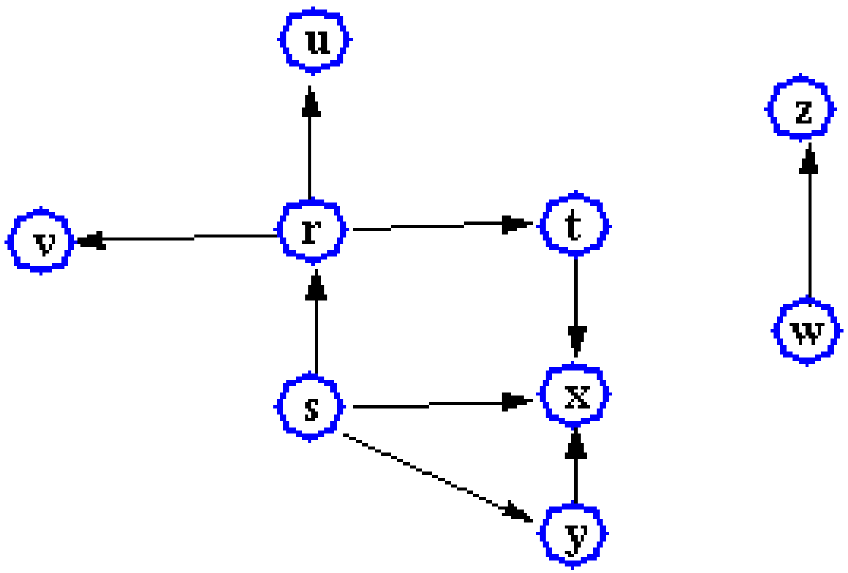
Topologische Sortierung I

- gerichtete Kanten eines zyklenfreien Digraphs (DAG) beschreiben Halbordnung unter Knoten
- topologische Sortierung erzeugt vollständige Ordnung, die nicht im Widerspruch zur partiellen Ordnung steht
- Topologische Sortierung eines Digraphen G = (V,E):

Abbildung ord: $V \rightarrow \{1, ..., n\}$ mit $|V| = n$,

so dass mit $(u,v) \in E$ auch $ord(u) < ord(v)$ gilt.

Beispiel:



Topologische Sortierung II

Satz: Digraph $G = (V,E)$ ist zyklenfrei <=> für G existiert eine topologische Sortierung

Beweis:

<= klar

=> Induktion über $|V|$.

Induktionsanfang: $|V|=1$, keine Kante, bereits topologisch sortiert

Induktionsende: $|V|=n$.

- Da G azyklisch ist, muss es einen Knoten v ohne Vorgänger geben. Setze $ord(v) = 1$
- Durch Entfernen von v erhalten wir einen azyklischen Graphen G' mit $|V'| = n-1$, für den es nach Induktionsvoraussetzung topologische Sortierung ord' gibt
- Die gesuchte topologische Sortierung für G ergibt sich durch $ord(v') = ord'(v') + 1$, für alle $v' \in V'$

Topologische Sortierung III

Korollar: Zu jedem DAG gibt es eine topologische Sortierung

Beweis liefert einen Algorithmus zur topologischen Sortierung

Bestimmung einer Abbildung *ord* für gerichteten Graphen G = (V,E) zur topologischen Sortierung und Test auf Zyklenfreiheit

TS (G):

```
i=0;
while G hat wenigstens einen Knoten v mit eg (v) = 0 do {
        i = i+1; ord(v) := i; G = G -{v};   };
 if G = {} then „G ist zyklenfrei“ else „G hat Zyklen“;
```

- (Neu-)Bestimmung des Eingangsgrades kann sehr aufwendig werden
- Effizienter ist daher, den jeweils aktuellen Eingangsgrad zu jedem Knoten zu speichern

Effiziente Alternative: Verwendung der Tiefensuche

- Verwendung der *out*-Zeitpunkte, in umgekehrter Reihenfolge
- Realisierung mit Aufwand O(n+m)
- Mit denselben Kosten O(n+m) kann die Zyklenfreiheit eines Graphen getestet werden (Zyklus liegt dann vor, wenn bei der Tiefensuche der Nachfolger eines Knotens bereits grau gefärbt ist!)

Topologische Sortierung IV

Anwendungsbeispiel: Der zerstreute Professor legt die Reihenfolge beim Ankleiden fest.

- Unterhose vor Hose
- Hose vor Gürtel
- Hemd vor Gürtel
- Gürtel vor Jackett
- Hemd vor Krawatte
- Krawatte vor Jackett
- Socken vor Schuhen
- Unterhose vor Schuhen
- Hose vor Schuhen
- Uhr: egal

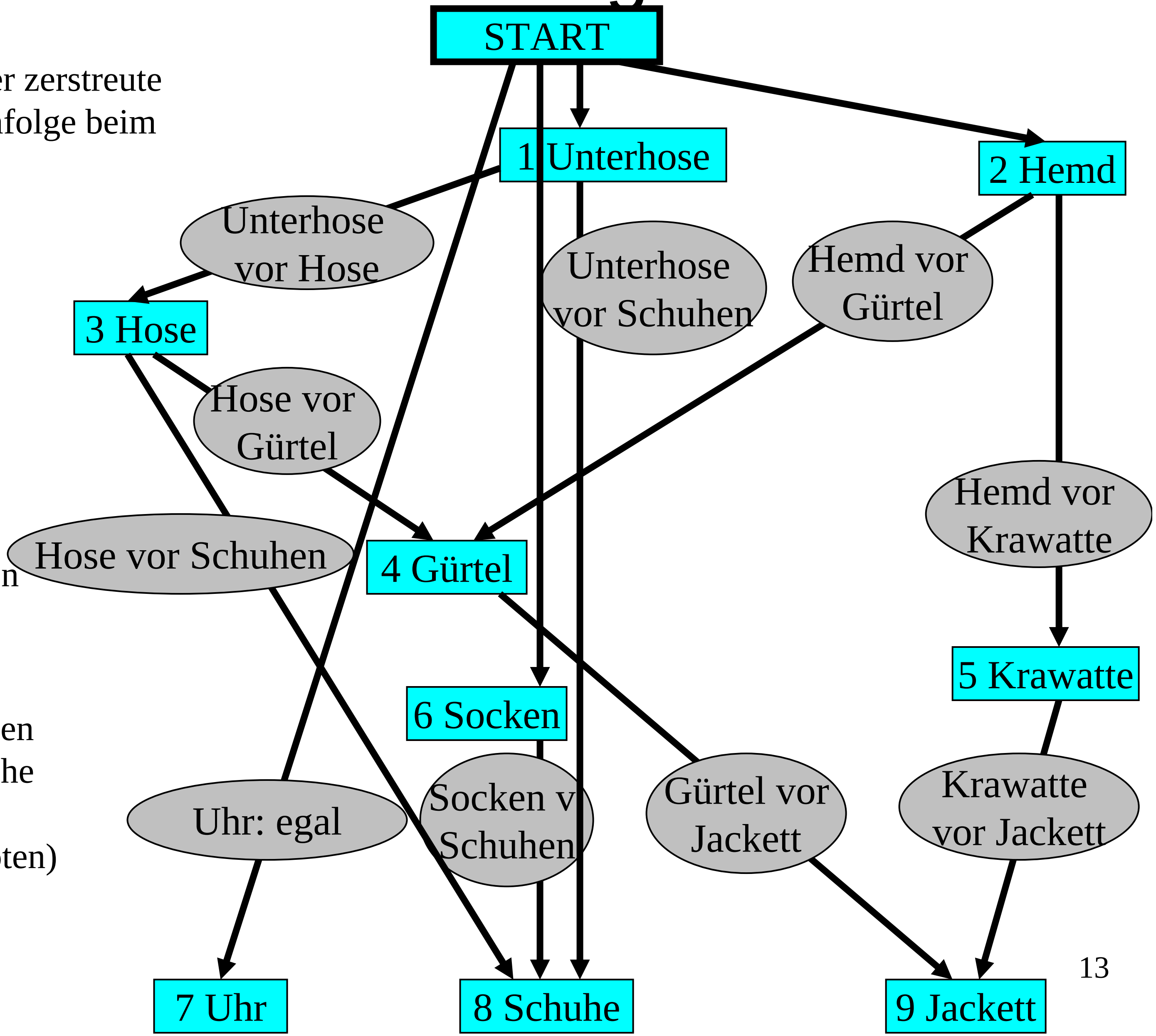
Ergebnis der topologischen Sortierung mit Tiefensuche abhängig von Wahl der Startknoten (weißen Knoten)

Topologische Sortierung V

Anwendungsbeispiel: Der zerstreute Professor legt die Reihenfolge beim Ankleiden fest.

- Unterhose vor Hose
- Hose vor Gürtel
- Hemd vor Gürtel
- Gürtel vor Jackett
- Hemd vor Krawatte
- Krawatte vor Jackett
- Socken vor Schuhen
- Unterhose vor Schuhen
- Hose vor Schuhen
- Uhr: egal

Ergebnis der topologischen Sortierung mit Tiefensuche abhängig von Wahl der Startknoten (weißen Knoten)



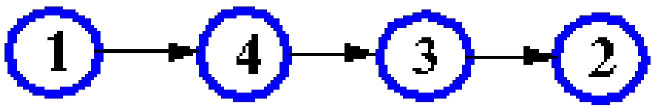
Transitive Hülle I

Erreichbarkeit von Knoten

- welche Knoten sind von einem gegebenen Knoten aus erreichbar?
- gibt es Knoten, von denen aus alle anderen erreicht werden können?
- Bestimmung der transitiven Hülle ermöglicht Beantwortung solcher Fragen

Ein Digraph G* = (V,E*) ist die reflexive, transitive Hülle (kurz: Hülle) eines Digraphen G = (V,E), wenn genau dann (v,v') E* ist, wenn es einen Weg von v nach v' in G gibt.

Beispiel



A	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

Transitive Hülle II

Naiver Algorithmus zur Berechnung von Pfeilen der reflexiven transitiven Hülle

```
boolean [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = true;
for (int i = 0; i < A.length; i++)
        for (int j = 0; j < A.length; j++)
                if A[i][j] for (int k = 0; k < A.length; k++) if A [j][k] A [i][k] = true;
```

- es werden nur Pfade der Länge 2 bestimmt!
- Komplexität $O(n^3)$

Transitive Hülle: Warshall-Algorithmus

Einfache Modifikation liefert vollständige transitive Hülle:

```
boolean [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = true;
for (int j = 0; j < A.length; j++)
            for (int i = 0; i < A.length; i++)
                        if A[i][j] for (int k = 0; k < A.length; k++) if A [j][k] A [i][k] = true;
```

Korrektheit des Warshall-Algorithmus

Korrektheit kann über Induktionsbeweis gezeigt werden:

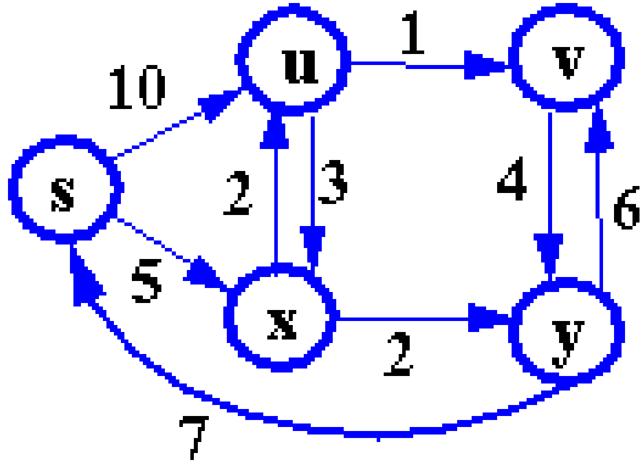
- Induktionshypothese P(j): gibt es zu beliebigen Knoten i und k einen Weg von i nach k, so dass alle Zwischenknoten aus der Menge {0, 1, ...j} sind, so wird in der j-ten Iteration A [i][k]=true gesetzt. Wenn P(j) für alle j gilt, wird keine Kante der transitiven Hülle vergessen
- Induktionsanfang: j=0: Falls A[i][0] und A[0][k] gilt, wird in der Schleife mit j=0 auch A[i][k] gesetzt
- Induktionsschluss: Sei P(j) wahr für 0 .. j. Sei ein Weg von i nach k vorhanden, der Knoten j+1 nutzt, dann gibt es auch einen solchen, auf dem j+1 nur einmal vorkommt. Aufgrund der Induktionshypothese wurde in einer früheren Iteration der äußeren Schleife bereits (i,j+1) und (j+1,k) eingefügt. In der (j+1)-ten Iteration wird nun (i,k) gefunden. Somit gilt auch P(j+1).

Komplexität des Warshall-Algorithmus

Komplexität

- innerste for-Schleife wird nicht notwendigerweise n^2-mal ($n=|V|$) durchlaufen, sondern nur falls Verbindung von i nach j in E* vorkommt, also $O(k)$ mit $k=|E^*|$ mal
- Gesamtkomplexität $O(n^2+k\cdot n)$.

Kürzeste Wege I



kantenmarkierter (gewichteter) Graph G = (V, E, g)

- Weg/Pfad P der Länge n: $(v_0,v_1), (v_1,v_2), \ldots ,(v_{n-1}, v_n)$
- Gewicht (Länge) des Weges/Pfads

$$w (P) = g ((v_i, v_{i+1}))$$

- Distanz d(u,v): Gewicht des kürzesten Pfades von u nach v

Varianten

- nichtnegative Gewichte vs. negative und positive Gewichte
- Bestimmung der kürzesten Wege
  - a) zwischen allen Knotenpaaren,
  - b) von einem Knoten u aus
  - c) zwischen zwei Knoten u und v

Bemerkungen

- kürzeste Wege sind nicht immer eindeutig
- kürzeste Wege müssen nicht existieren:
  - es existiert kein Weg;
  - es existiert Zyklus mit negativem Gewicht

Kürzeste Wege II

Warshall-Algorithmus lässt sich einfach modifizieren, um kürzeste Wege zwischen allen Knotenpaaren zu berechnen

- Matrix A enthält zunächst Knotengewichte pro Kante, ∞ falls "keine Kante" vorliegt
- A[i,i] wird mit 0 vorbelegt
- Annahme: kein Zyklus mit negativem Gewicht vorhanden

```
int [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = 0;
for (int j = 0; j < A.length; j++)
        for (int i = 0; i < A.length; i++)
                for (int k = 0; k < A.length; k++)
                        if (A [i][j] + A [j][k] < A [i][k])
                                A [i][k]= A [i][j] + A [j][k];
```

Komplexität: $O(n^3)$

Kürzeste Wege: Dijkstra-Algorithmus I

Bestimmung der von einem Knoten ausgehenden kürzesten Wege

- gegeben: kanten-bewerteter Graph G = (V,E, g) mit g: E -> R^+ (Kantengewichte)
- Startknoten s; zu jedem Knoten u wird die Distanz zu Startknoten s in D[u] geführt
- Q sei Prioritäts-Warteschlange (sortierte Liste); Priorität = Distanzwert
- Funktion succ(u) liefert die Menge der direkten Nachfolger von u

- Verallgemeinerung der Breitensuche (gewichtete Entfernung)
- funktioniert nur bei nicht-negativen Gewichten
- Optimierung gemäß Greedy-Prinzip

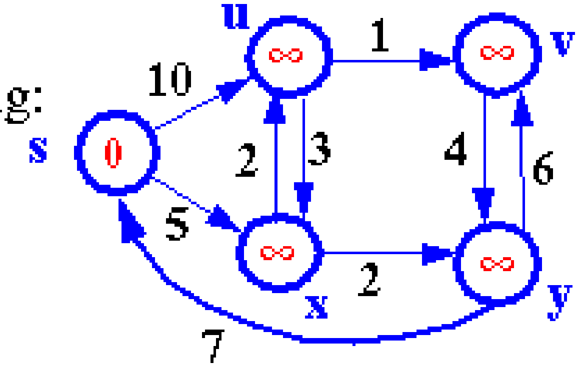
Kürzeste Wege: Dijkstra-Algorithmus II

Dijkstra (G,s):

```
for each Knoten v ∈ V - s do { D[v] = ∞;};
D[s] = 0; PriorityQueue Q = V;
while not isEmpty(Q) do { v = extractMinimum(Q);
        for each u ∈ succ (v) ∩ Q do {
                if D[v] + g ((v,u)) < D[u] then
                        { D[u] = D[v] + g ((v,u));
                          adjustiere Q an neuen Wert D[u]; };
        };
}
```

Dijkstra-Algorithmus: Beispiel

Initialisierung:



$Q = <(s:0), (u: \infty), (v: \infty), (x: \infty), (y: \infty)>$