

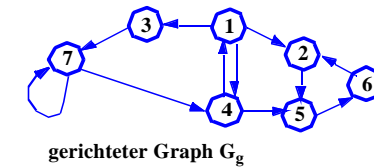
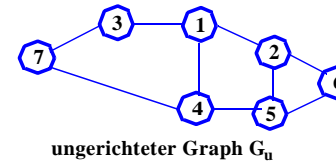
3. Graphen

- Definitionen
- Implementierungsalternativen
 - Kantenliste, Knotenliste
 - Adjazenzmatrix, Adjazenzliste
 - Vergleich
- Traversierung von Graphen
 - Breitensuche
 - Tiefensuche
- Topologisches Sortieren
- Transitive Hülle (Warshall-Algorithmus)
- Kürzeste Wege (Dijkstra-Algorithmus etc.)
- Minimale Spannbäume (Kruskal-Algorithmus)
- Maximale Flüsse (Ford-Fulkerson)
- Maximales Matching



Einführung

- Graphen sind zur Repräsentation von Problemen vielseitig verwendbar, z.B.
 - Städte: Verbindungswege
 - Personen: Relationen zwischen ihnen
 - Rechner: Verbindungen
 - Aktionen: zeitliche Abhängigkeiten
- Graph: Menge von Knoten (Vertices) und Kanten (Edges)
 - ungerichtete Graphen
 - gerichtete Graphen (Digraph, Directed graph)
 - gerichtete, azyklische Graphen (DAG, Directed Acyclic Graph)



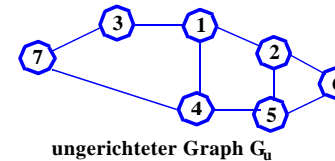
Definitionen

- $G = (V, E)$ heißt **ungerichteter Graph** : \Leftrightarrow
 - $V \neq \emptyset$ ist eine endliche, nichtleere Menge. V heißt Knotenmenge, Elemente von V heißen *Knoten*
 - E ist eine Menge von ein- oder zweielementigen Teilmengen von V . E heißt Kantenmenge, ein Paar $\{u,v\} \in E$ heißt *Kante*
 - Eine Kante $\{u\}$ heißt *Schlinge*
 - Zwei Knoten u und v heißen *benachbart* (adjazent): $\Leftrightarrow \{u,v\} \in E$ oder $(u=v) \wedge \{u\} \in E$.
 - Sei $G = (V,E)$ ein ungerichteter Graph. Wenn E keine Schlinge enthält, so heißt G *schlingenlos*.
- Bem.* Im weiteren werden wir Kanten $\{u,v\}$ als Paare (u,v) oder (v,u) und Schlingen $\{u\}$ als Paar (u,u) schreiben, um spätere gemeinsame Definitionen für ungerichtete und gerichtete Graphen nicht differenzieren und notationell unterscheiden zu müssen.
- Seien $G = (V_G, E_G)$ und $H = (V_H, E_H)$ ungerichtete Graphen.
 - H heißt *Teilgraph* von G ($H \subset G$): $\Leftrightarrow V_G \supset V_H$ und $E_G \supset E_H$
 - H heißt *vollständiger Teilgraph* von G : $\Leftrightarrow H \subset G$ und $[(u,v) \in E_G \text{ mit } u,v \in V_H \Rightarrow (u,v) \in E_H]$.



Beispiele ungerichteter Graphen

- Beispiel 1
 - $G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$,
 - $E_G = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$
- Beispiel 2



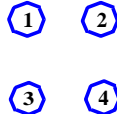
Definitionen (2)

■ $G = (V, E)$ heißt gerichteter Graph (Directed Graph, Digraph) : \Leftrightarrow

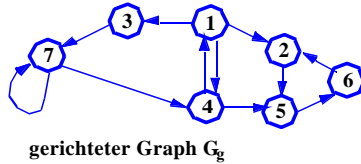
- $V \neq \emptyset$ ist endliche Menge. V heißt Knotenmenge, Elemente von V heißen Knoten.
- $E \subseteq V \times V$ heißt Kantenmenge, Elemente von E heißen Kanten.
Schreibweise: (u, v) oder $u \rightarrow v$. u ist die Quelle, v das Ziel der Kante $u \rightarrow v$.
- Eine Kante (u, u) heißt Schlinge.

■ Beispiel

- $G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$ und $E_G = \{1 \rightarrow 2, 1 \rightarrow 3, 1 \rightarrow 4, 2 \rightarrow 3, 2 \rightarrow 4, 3 \rightarrow 4\}$



■ Beispiel 2

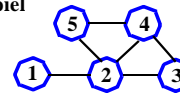


Definitionen (3)

■ Sei $G = (V, E)$ ein (un)gerichteter Graph und $k = (v_0, \dots, v_n) \in V^{n+1}$.

- k heißt **Kantenfolge** der Länge n von v_0 nach v_n , wenn für alle $i \in \{0, \dots, n-1\}$ gilt: $(v_i, v_{i+1}) \in E$. Im gerichteten Fall ist v_0 der Startknoten und v_n der Endknoten, im ungerichteten Fall sind v_0 und v_n die Endknoten von k .
 v_1, \dots, v_{n-1} sind die **inneren Knoten** von k . Ist $v_0 = v_n$, so ist die Kantenfolge **geschlossen**.
- k heißt **Kantenzug** der Länge n von v_0 nach v_n , wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle $i, j \in \{0, \dots, n-1\}$ mit $i \neq j$ gilt: $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$.
- k heißt **Weg** (Pfad) der Länge n von v_0 nach v_n , wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle $i, j \in \{0, \dots, n\}$ mit $i \neq j$ gilt: $v_i \neq v_j$.
- k heißt **Zyklus** oder Kreis der Länge n , wenn k geschlossene Kantenfolge der Länge n von v_0 nach v_n und wenn $k' = (v_0, \dots, v_{n-1})$ ein Weg ist. Ein Graph ohne Zyklus heißt kreisfrei oder **azyklisch**. Ein gerichteter azyklischer Graph heißt auch **DAG (Directed Acyclic Graph)**.
- Graph ist **zusammenhängend**, wenn zwischen je 2 Knoten ein Kantenzug existiert

Beispiel



Kantenfolge:
Kantenzug:
Weg:
Zyklus:

■ Sei $G = (V, E)$ (un)gerichteter Graph, k Kantenfolge von v nach w . Dann gibt es einen Weg von v nach w .



Definitionen (4)

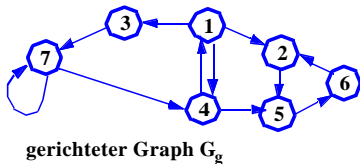
■ Sei $G = (V, E)$ ein gerichteter Graph

- **Eingangsgrad**: $eg(v) = |\{v' \mid (v', v) \in E\}|$
- **Ausgangsgrad**: $ag(v) = |\{v' \mid (v, v') \in E\}|$
- G heißt **gerichteter Wald**, wenn G zyklensfrei ist und für alle Knoten v gilt $eg(v) \leq 1$. Jeder Knoten v mit $eg(v)=0$ ist eine **Wurzel** des Waldes.
- **Aufspannender Wald** (Spannwald) von G : gerichteter Wald $W=(V, F)$ mit $F \subseteq E$

■ **Gerichteter Baum (Wurzelbaum)**: gerichteter Wald mit genau 1 Wurzel

- für jeden Knoten v eines gerichteten Baums gibt es genau einen Weg von der Wurzel zu v
- **Erzeugender / aufspannender Baum (Spannbaum)** eines Digraphen G : Spannwald von G mit nur 1 Wurzel

■ zu jedem zusammenhängenden Graphen gibt es (mind.) einen Spannbaum



Definitionen (5)

■ Markierte Graphen

Sei $G = (V, E)$ ein (un)gerichteter Graph, M_V und M_E Mengen und $\mu : V \rightarrow M_V$ und $g : E \rightarrow M_E$ Abbildungen.

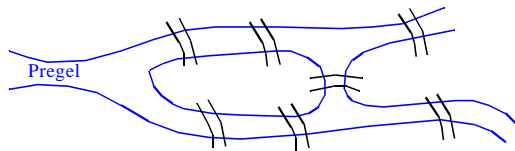
- $G' = (V, E, \mu)$ heißt **knotenmarkierter Graph**
 - $G'' = (V, E, g)$ heißt **kantenmarkierter Graph**
 - $G''' = (V, E, \mu, g)$ heißt **knoten- und kantenmarkierter Graph**
- M_V und M_E sind die Markierungsmengen (z.B. Alphabete oder Zahlen)



Algorithmische Probleme für Graphen

Gegeben sei ein (un)gerichteter Graph $G = (V, E)$

- Man entscheide, ob G zusammenhängend ist
- Man entscheide, ob G azyklisch ist
- Man finde zu zwei Knoten, $v, w \in V$ einen *kürzesten Weg* von v nach w (bzw. „günstigster“ Weg bzgl. Kantenmarkierung)
- Man entscheide, ob G einen *Hamiltonschen Zyklus* besitzt, d.h. einen Zyklus der Länge $|V|$
- Man entscheide, ob G einen *Eulerschen Weg* besitzt, d.h. einen Weg, in dem jede Kante genau einmal verwendet wird, und dessen Anfangs- und Endpunkte gleich sind (*Königsberger Brückenproblem*)



Algorithmische Probleme (2)

- Färbungsproblem:** Man entscheide zu einer vorgegebenen natürlichen Zahl k („Anzahl der Farben“), ob es eine Knotenmarkierung $\mu : V \rightarrow \{1, 2, \dots, k\}$ so gibt, daß für alle $(v, w) \in E$ gilt: $\mu(v) \neq \mu(w)$ [G azyklisch]
- Cliquenproblem:** Man entscheide für ungerichteten Graphen G zu vorgegebener natürlicher Zahl k , ob es einen Teilgraphen G' („ k -Clique“) von G gibt, dessen Knoten alle paarweise durch Kanten verbunden sind
- Matching-Problem:** Sei $G = (V, E)$ ein Graph. Eine Teilmenge $M \subseteq E$ der Kanten heißt Matching, wenn jeder Knoten von V zu höchstens einer Kante aus M gehört. Problem: finde ein maximales Matching
- Traveling Salesman Problem:** Bestimme optimale Rundreise durch n Städte, bei der jede Stadt nur einmal besucht wird und minimale Kosten entstehen

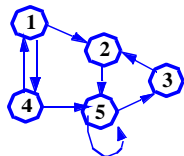
Hierunter sind bekannte NP-vollständige Probleme, z.B. das Cliquenproblem, das Färbungsproblem, die Hamilton-Eigenschaftsprüfung und das Traveling Salesman Problem



Speicherung von Graphen

- Knoten- und Kantenlisten**
 - Speicherung von Graphen als Liste von Zahlen (z.B. in Array oder verketteter Liste)
 - Knoten werden von 1 bis n durchnummeriert; Kanten als Paare von Knoten
- Kantenliste**
 - Liste: Knotenzahl, Kantenzahl, Liste von Kanten (je als 2 Zahlen)
 - Speicherbedarf: $2 + 2m$ (m = Anzahl Kanten)
- Knotenliste**
 - Liste: Knotenzahl, Kantenzahl, Liste von Knoteninformationen
 - Knoteninformation: Ausgangsgrad und Zielknoten $ag(i), v_1 \dots v_{ag(i)}$
 - Speicherbedarf: $2 + n + m$ (n = Anzahl Knoten, m = Anzahl Kanten)

Beispiel



Kantenliste:

Knotenliste:



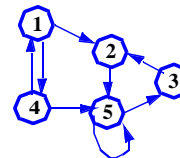
Speicherung von Graphen (2)

Adjazenzmatrix

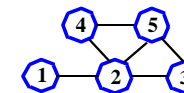
Ein Graph $G = (V, E)$ mit $|V| = n$ wird in einer Boole'schen $n \times n$ -Matrix

$$A_G = (a_{ij}), \text{ mit } 1 \leq i, j \leq n \text{ gespeichert, wobei } a_{ij} = \begin{cases} 0 & \text{falls } (i, j) \notin E \\ 1 & \text{falls } (i, j) \in E \end{cases}$$

Beispiel:



A_G	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	0	1
3	0	1	0	0	0
4	1	0	0	0	1
5	0	1	0	1	1



Speicherplatzbedarf $O(n^2)$

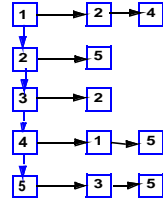
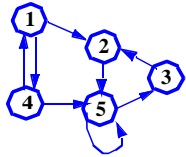
- jedoch nur 1 Bit pro Position (statt Knoten/Kantennummern)
- unabhängig von Kantenmenge
- für ungerichtete Graphen ergibt sich symmetrische Belegung (Halbierung des Speicherbedarfs möglich)



Speicherung von Graphen (3)

■ Adjazenzlisten

- verkettete Liste der n Knoten (oder Array-Realisierung)
- pro Knoten: verkettete Liste der Nachfolger (repräsentiert die von dem Knoten ausgehenden Kanten)
- Speicherbedarf: n+m Listenelemente



- Variante: doppelt verkettete Kantenlisten (doubly connected arc list, DCAL)



Speicherung von Graphen (4)

```

/** Repräsentiert einen Knoten im Graphen. */
public class Vertex {
    Object key = null; // Knotenbezeichner
    LinkedList edges = null; // Liste ausgehender Kanten

    /** Konstruktor */
    public Vertex(Object key) { this.key = key; edges = new LinkedList(); }

    /** Ueberschreibe Object.equals-Methode */
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (obj instanceof Vertex) return key.equals(((Vertex) obj).key);
        else return key.equals(obj); }

    /** Ueberschreibe Object.hashCode-Methode */
    public int hashCode() { return key.hashCode(); } ... }

/** Repraesentiert eine Kante im Graphen. */
public class Edge {
    Vertex dest = null; // Kantenzielknoten
    int weight = 0; // Kantengewicht

    /** Konstruktor */
    public Edge(Vertex dest, int weight) {
        this.dest = dest; this.weight=weight; } ... }
    
```



```

/** Graphrepräsentation. */
public class Graph {

    protected Hashtable vertices = null; // enthaelt alle Knoten des Graphen

    /** Konstruktor */
    public Graph() { vertices = new Hashtable(); }

    /** Fuegt einen Knoten in den Graphen ein. */
    public void addVertex(Object key) {
        if (vertices.containsKey(key))
            throw new GraphException("Knoten existiert bereits!");
        vertices.put(key, new Vertex(key)); }

    /** Fuegt eine Kante in den Graphen ein. */
    public void addEdge(Object src, Object dest, int weight) {
        Vertex vsrc = (Vertex) vertices.get(src);
        Vertex vdest = (Vertex) vertices.get(dest);
        if (vsrc == null)
            throw new GraphException("Ausgangsknoten existiert nicht!");
        if (vdest == null)
            throw new GraphException("Zielknoten existiert nicht!");
        vsrc.edges.add(new Edge(vdest, weight)); }

    /** Liefert einen Iterator ueber alle Knoten. */
    public Iterator getVertices() { return vertices.values().iterator(); }

    /** Liefert den zum Knotenbezeichner gehoerigen Knoten. */
    public Vertex getVertex(Object key) {
        return (Vertex) vertices.get(key); } }
    
```



Speicherung von Graphen: Vergleich

■ Komplexitätsvergleich

Operation	Kantenliste	Knotenliste	Adjazenzmatrix	Adjazenzliste
Einfügen Kante	O(1)	O(n+m)	O(1)	O(1) / O(n)
Löschen Kante	O(m)	O(n+m)	O(1)	O(n)
Einfügen Knoten	O(1)	O(1)	O(n ²)	O(1)
Löschen Knoten	O(m)	O(n+m)	O(n ²)	O(n+m)
Speicherplatzbedarf	O(m)	O(n+m)	O(n ²)	O(n+m)

- Löschen eines Knotens löscht auch zugehörige Kanten
- Änderungsaufwand abhängig von Realisierung der Adjazenzmatrix und Adjazenzliste

■ Welche Repräsentation geeigneter ist, hängt auch vom Problem ab:

- Frage: Gibt es Kante von a nach b: *Matrix*
- Durchsuchen von Knoten in durch Nachbarschaft gegebener Reihenfolge: *Listen*

■ Transformation zwischen Implementierungsalternativen möglich



Traversierung

- 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 ∈ V do { markiere v als unbearbeitet;
B = {s}; // Initialisierung der Menge besuchter Knoten B mit Startknoten s ∈ V;
markiere s als bearbeitet;
while es gibt noch unbearbeitete Knoten v' mit (v,v') ∈ E und v ∈ B do {
    B = B ∪ {v'};
    markiere v' als bearbeitet;
};
}
```
- Realisierungen unterscheiden sich bezüglich Verwaltung der noch abzuarbeitenden Knotenmenge und Auswahl der jeweils nächsten Kante



Traversierung (2)

- 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

- 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*
- 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;
}
```



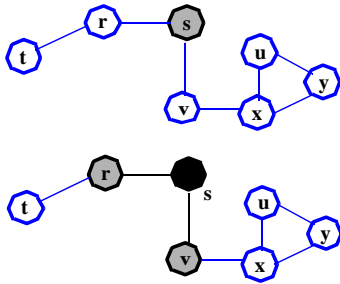
Breitensuche (2)

```
/** Liefert die Liste aller erreichbaren Knoten in Breitendurchlauf. */
public List traverseBFS(Object root, Hashtable d, Hashtable pred) {
    LinkedList list = new LinkedList();
    Hashtable color = new Hashtable();
    Integer gray = new Integer(1);
    Integer black = new Integer(2);
    Queue q = new Queue();
    Vertex v, u = null;
    Iterator eIter = null;
    v = (Vertex)vertices.get(root);
    color.put(v, gray);
    d.put(v, new Integer(0));
    q.enqueue(v);
    while (! q.empty()) {
        v = (Vertex) vertices.get(((Vertex)q.front()).key);
        eIter = v.edges.iterator();
        while(eIter.hasNext()) {
            u = ((Edge)eIter.next()).dest;
            if (color.get(u) == null) {
                color.put(u, gray);
                d.put(u, new Integer(((Integer)d.get(v)).intValue() + 1));
                pred.put(u, v);
                q.enqueue(u);
            }
        }
        q.dequeue();
        list.add(v);
        color.put(v, black);
    }
    return list;
}
```



Breitensuche (3)

■ Beispiel:



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

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

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



Tiefensuche

■ 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 $G = (V, E)$; $\text{succ}(v)$ 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

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

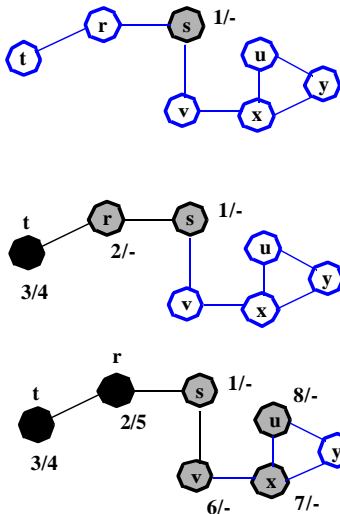
```
farbe[v]= grau; zeit = zeit+1; in[v]=zeit;
for each  $u \in \text{succ}(v)$  do { if farbe[u] = weiß then { pred[u] = v; DFS-visit(u); }; };
farbe[v] = schwarz; zeit = zeit+1; out[v]=zeit;
```

■ 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

■ gerichtete Kanten eines zyklensfreien Digraphs (DAG) beschreiben Halbordnung unter Knoten

■ topologische Sortierung erzeugt vollständige Ordnung, die nicht im Widerspruch zur partiellen Ordnung steht

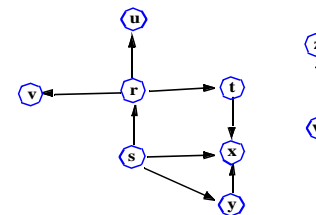
- d.h. falls eine Kante von Knoten i nach j existiert, erscheint i in der linearen Ordnung vor j

■ Topologische Sortierung eines Digraphen $G = (V, E)$:

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

so daß mit $(u, v) \in E$ auch $\text{ord}(u) < \text{ord}(v)$ gilt.

■ Beispiel:



Topologische Sortierung (2)

- Satz: Digraph $G = (V, E)$ ist zyklensfrei \Leftrightarrow für G existiert eine topologische Sortierung

Beweis: \Leftarrow klar

\Rightarrow Induktion über $|V|$.

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

Induktionsschluß: $|V| = n$.

- Da G azyklisch ist, muß es einen Knoten v ohne Vorgänger geben. Setze $\text{ord}(v) = 1$
- 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 $\text{ord}(v') = \text{ord}'(v') + 1$, für alle $v' \in V'$

- Korollar: zu jedem DAG gibt es eine topologische Sortierung



Topologische Sortierung (3)

- Beweis liefert einen Algorithmus zur topologischen Sortierung

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

TS (G):

$i=0$;

while G hat wenigstens einen Knoten v mit $\text{eg}(v) = 0$ **do** {

$i = i+1$; $\text{ord}(v) := i$; $G = G - \{v\}$; }

if $G = \{\}$ **then** „ G ist zyklensfrei“ **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 Zyklensfreiheit eines Graphen getestet werden (Zyklus liegt dann vor, wenn bei der Tiefensuche der Nachfolger eines Knotens bereits *grau* gefärbt ist!)



Topologische Sortierung (4)

- Anwendungsbeispiel

zerstreuter 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)



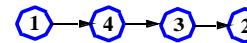
Transitive Hülle

- 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 $(v, v') \in E^*$ ist, wenn es einen Weg von v nach v' in G gibt.

Beispiel



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

- Algorithmus zur Berechnung von Pfeilen der reflexiven transitiven Hülle

boolean [] [] $A = \{ \dots \}$; **for** (int $i = 0$; $i < A.length$; $i++$) $A[i][i] = \text{true}$;

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** $A[j][k]$ $A[i][k] = \text{true}$;

- 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 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

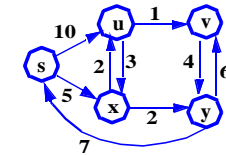
- innerste for-Schleife wird nicht notwendigerweise n^2 -mal ($n=|V|$) durchlaufen, sondern nur falls Verbindung von i nach j in E^* vorkommt, also $O(k)$ mit $k=|E^*|$ mal
- Gesamtkomplexität $O(n^2+k \cdot n)$.



Kürzeste Wege

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

- Weg/Pfad P der Länge n: $(v_0, v_1), (v_1, v_2), \dots, (v_{n-1}, v_n)$
- **Gewicht** (Länge) des Weges/Pfades
 $w(P) = \sum g((v_i, v_{i+1}))$



- **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 (2)

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

Bestimmung der von einem Knoten ausgehenden kürzesten Wege

- **gegeben:** kanten-bewerteter Graph $G = (V, E, g)$ mit $g: E \rightarrow \mathbb{R}^+$ (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

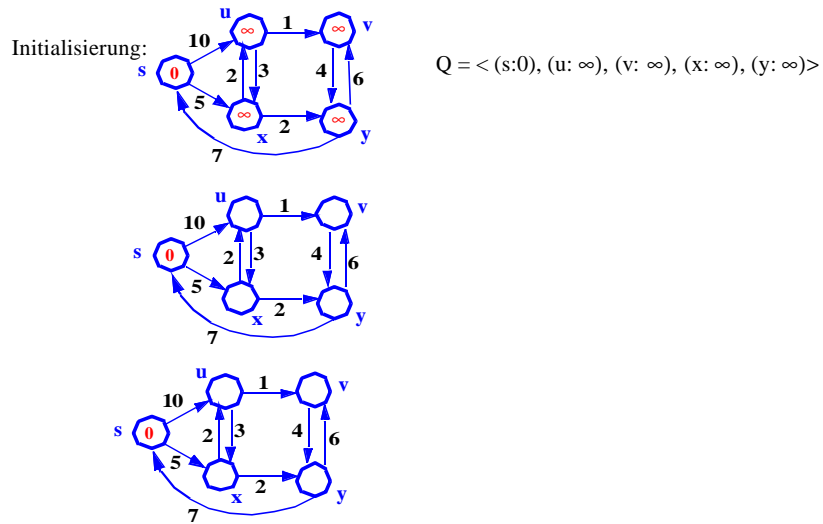
Dijkstra (G,s):

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

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



Dijkstra-Algorithmus: Beispiel



Dijkstra-Algorithmus (3)

■ Korrektheitsbeweis

- nach i Schleifendurchgängen sind die Längen von i Knoten, die am nächsten an s liegen, korrekt berechnet und diese Knoten sind aus Q entfernt.
- **Induktionsanfang:** s wird gewählt, $D(s) = 0$
- **Induktionsschritt:** Nimm an, v wird aus Q genommen. Der kürzeste Pfad zu v gehe über direkten Vorgänger v' von v . Da v' näher an s liegt, ist v' nach Induktionsvoraussetzung mit richtiger Länge $D(v')$ bereits entfernt. Da der **kürzeste Weg** zu v die Länge $D(v') + g((v',v))$ hat und dieser Wert bei Entfernen von v' bereits v zugewiesen wurde, wird v mit der richtigen Länge entfernt.
- erfordert nichtnegative Kantengewichte (steigende Länge durch hinzugenommene Kanten)

■ Komplexität $\leq O(n^2)$

- n -maliges Durchlaufen der **äußeren Schleife** liefert Faktor $O(n)$
- innere Schleife: Auffinden des Minimums begrenzt durch $O(n)$, ebenso das Aufsuchen der Nachbarn von v

■ Pfade bilden aufspannenden Baum (der die Wegstrecken von s aus gesehen minimiert)

■ Bestimmung des kürzesten Weges zwischen u und v : Spezialfall für Dijkstra-Algorithmus mit Start-Knoten u (Beendigung sobald v aus Q entfernt wird)



Kürzeste Wege mit negativen Kantengewichten

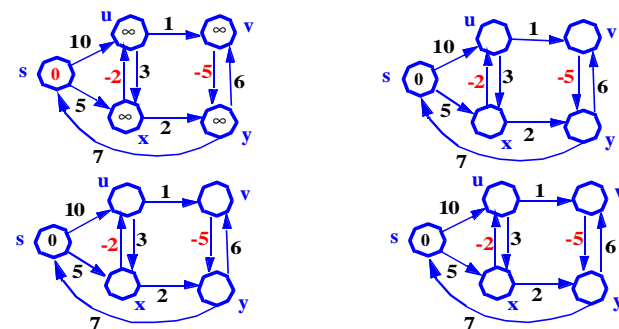
■ Bellmann-Ford-Algorithmus **BF** (G,s):

```

for each Knoten  $v \in V - s$  do {  $D[v] = \infty$ ;  $D[s] = 0$ ;
for  $i = 1$  to  $|E|-1$  do
  for each  $(u,v) \in E$  do {
    if  $D[u] + g((u,v)) < D[v]$  then  $D[v] = D[u] + g((u,v))$ ;
  }
};

```

Beispiel:



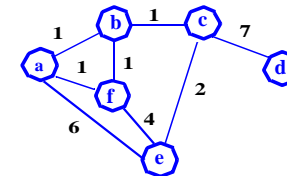
Minimale Spannäume

■ Problemstellung: zu zusammenhängendem Graph soll Spannbaum (aufspannender Baum) mit minimalem Kantengewicht (minimale Gesamtlänge) bestimmt werden

- relevant z.B. zur Reduzierung von Leitungskosten in Versorgungsnetzen
- zusätzliche Knoten können zur Reduzierung der Gesamtlänge eines Graphen führen

■ Kruskal-Algorithmus (1956)

- Sei $G = (V, E, g)$ mit $g: E \rightarrow \mathbb{R}$ (Kantengewichte) gegebener ungerichteter, zusammenhängender Graph. Zu bestimmen minimaler Spannbaum $T = (V, E')$
- $E' = \{\}$; sortiere E nach Kantengewicht und bringe die Kanten in PriorityQueue Q ; jeder Knoten v bilde eigenen Spannbaum(-Kandidat)
- solange Q nicht leer:
 - entferne erstes Element $e = (u,v)$
 - wenn beide Endknoten u und v im selben Spannbaum sind, verwerfe e , ansonsten nehme e in E' auf und fasse die Spannäume von u und v zusammen

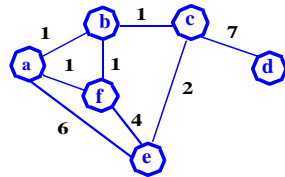


■ Analog: Bestimmung maximaler Spannäume (absteigende Sortierung)



Minimale Spannbäume (2)

Anwendung des Kruskal-Algorithmus



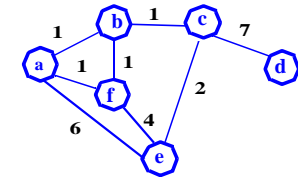
Komplexität $O(m \log n)$



Minimale Spannbäume (3)

Alternative Berechnung (Dijkstra)

- Startknoten s
- Knotenmenge B enthält bereits abgearbeitete Knoten



$s =$ an Kante mit minimalem Gewicht beteiligter Knoten

$B = \{s\}; E' = \{\};$

while $|B| < |V|$ do {

wähle $(u,v) \in E$ mit minimalem Gewicht mit $u \in B, v \notin B$;

füge (u,v) zu E' hinzu;

füge v zu B hinzu; }

- es wird nur 1 Spannb Baum erzeugt

- effiziente Implementierbarkeit mit PriorityQueue über Kantengewicht

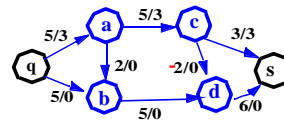


Flüsse in Netzen

Anwendungsprobleme:

- Wieviele Autos können durch ein Straßennetz fahren?
- Wieviel Abwasser fasst ein Kanalnetz?
- Wieviel Strom kann durch ein Leitungsnetz fließen?

■ **Def.:** Ein (Fluß-) *Netzwerk* ist ein gerichteter Graph $G = (V, E, c)$ mit ausgezeichneten Knoten q (*Quelle*) und s (*Senke*), sowie einer *Kapazitätsfunktion* $c: E \rightarrow \mathbb{Z}^+$.



Kantenmarkierung:
Kapazität $c(e)$ / Fluß $f(e)$

■ Ein *Fluß* für das Netzwerk ist eine Funktion $f: E \rightarrow \mathbb{Z}^+$, so daß gilt:

- *Kapazitätsbeschränkung:* $f(e) \leq c(e)$, für alle e in E .
- *Flußerhaltung:* für alle v in $V \setminus \{q, s\}$: $\sum_{(v',v) \in E} f((v',v)) = \sum_{(v,v') \in E} f((v,v'))$
- Der *Wert* von f , $w(f)$, ist die Summe der Flußwerte der die Quelle q verlassenden Kanten: $\sum_{(q,v) \in E} f((q,v))$

■ **Gesucht:** Fluß mit maximalem Wert

- begrenzt durch Summe der aus q wegführenden bzw. in s eingehenden Kapazitäten
- jeder weitere „Schnitt“ durch den Graphen, der q und s trennt, begrenzt max. Fluss



Flüsse in Netzen (2)

■ *Schnitt* (A, B) eines Fluß-Netzwerks ist eine Zerlegung von V in disjunkte Teilmengen A und B , so daß $q \in A$ und $s \in B$.

- Die *Kapazität des Schnitts* ist $c(A, B) = \sum_{u \in A, v \in B} c((u, v))$
- *minimaler Schnitt* (minimal cut): Schnitt mit kleinster Kapazität

■ *Restkapazität, Restgraph*

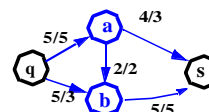
Sei f ein zulässiger Fluß für $G = (V, E)$. Sei $E' = \{(v, w) \mid (v, w) \in E \text{ oder } (w, v) \in E\}$

- Wir definieren die *Restkapazität einer Kante* $e = (v, w)$ wie folgt:

$$\text{rest}(e) = \begin{cases} c(e) - f(e) & \text{falls } e \in E \\ f((w, v)) & \text{falls } (w, v) \in E \end{cases}$$

- Der *Restgraph* von f (bzgl. G) besteht aus den Kanten $e \in E'$, für die $\text{rest}(e) > 0$

■ Jeder gerichtete Pfad von q nach s im Restgraphen heißt *zunehmender Weg*



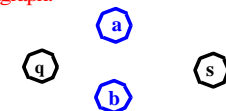
Kantenmarkierung: Kapazität $c(e)$ / Fluß $f(e)$

verwendete Wege:

- 1.) q, a, b, s (Kapaz. 2)
- 2.) q, b, s (3)
- 3.) q, a, s (3)

$w(f) = 8$, nicht maximal

Restgraph:



Kantenmarkierung: rest (e)



Flüsse in Netzen (3)

Theorem (Min-Cut-Max-Flow-Theorem):

Sei f zulässiger Fluß für G . Folgende Aussagen sind äquivalent:

- 1) f ist maximaler Fluß in G .
- 2) Der Restgraph von f enthält keinen zunehmenden Weg.
- 3) $w(f) = c(A,B)$ für einen Schnitt (A,B) von G .

Ford-Fulkerson-Algorithmus

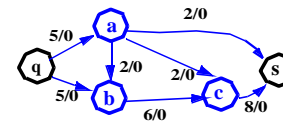
- füge solange zunehmende Wege zum Gesamtfluß hinzu wie möglich
- Kapazität erhöht sich jeweils um Minimum der verfügbaren Restkapazität der einzelnen Kanten des zunehmenden Weges

```

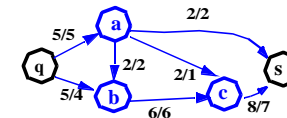
for each  $e \in E$  {  $f(e) = 0$ ; }
while ( es gibt zunehmenden Weg  $p$  im Restgraphen von  $f$  ) {
     $r = \min\{\text{rest}(e) \mid e \text{ liegt in } p\}$ ;
    for each  $e = (v,w)$  auf Pfad  $p$  {
        if (  $e$  in  $E$  )     $f(e) = f(e) + r$ ;
        else                 $f((w,v)) = f((w,v)) - r$ ; }
}
    
```



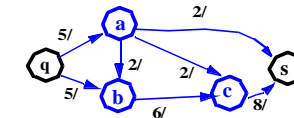
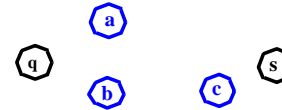
Ford-Fulkerson: Anwendungsbeispiel



Kantenmarkierung:
Kapazität $c(e)$ / Fluß $f(e)$



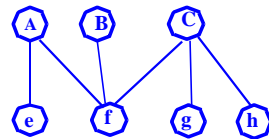
Restgraph:



Maximales Matching

Beispiel:

- Eine Gruppe von Erwachsenen und eine Gruppe von Kindern besuchen Disneyland.
- Auf der Achterbahn darf ein Kind jeweils nur in Begleitung eines Erwachsenen fahren.
- Nur Erwachsene/Kinder, die sich kennen, sollen zusammen fahren. Wieviele Kinder können maximal eine Fahrt mitmachen?



■ **Matching** (Zuordnung) M für ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, so daß jeder Knoten in V in höchstens einer Kante vorkommt

- $|M|$ = Größe der Zuordnung
- **Perfektes Matching**: kein Knoten bleibt „allein“ (unmatched), d.h. jeder Knoten ist in einer Kante von M vertreten

■ Matching M ist maximal, wenn es kein Matching M' gibt mit $|M| < |M'|$

■ Verallgemeinerung mit gewichteten Kanten: Matching mit maximalem Gewicht

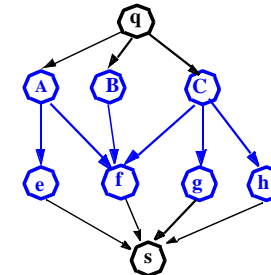


Matching (2)

■ **Def.:** Ein *bipartiter Graph* ist ein Graph, dessen Knotenmenge V in zwei disjunkte Teilmengen V_1 und V_2 aufgeteilt ist, und dessen Kanten jeweils einen Knoten aus V_1 mit einem aus V_2 verbinden

■ Maximales Matching kann auf maximalen Fluß zurückgeführt werden:

- Quelle und Senke hinzufügen.
- Kanten von V_1 nach V_2 richten.
- Jeder Knoten in V_1 erhält eingehende Kante von der Quelle.
- Jeder Knoten in V_2 erhält ausgehende Kante zur Senke.
- Alle Kanten erhalten Kapazität $c(e) = 1$.

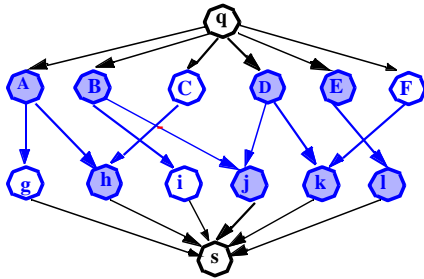


■ Jetzt kann Ford-Fulkerson-Algorithmus angewendet werden



Matching (3)

■ Weiteres Anwendungsbeispiel



- ist gezeigtes Matching maximal?



Zusammenfassung

- viele wichtige Informatikprobleme lassen sich mit gerichteten bzw. ungerichteten Graphen behandeln
- wesentliche Implementierungsalternativen: Adjazenzmatrix und Adjazenzlisten
- Algorithmen mit linearem Aufwand:
 - Traversierung von Graphen: Breitensuche vs. Tiefensuche
 - Topologisches Sortieren
 - Test auf Azyklität
- Weitere wichtige Algorithmen[†]:
 - Warshall-Algorithmus zur Bestimmung der transitiven Hülle
 - Dijkstra-Algorithmus bzw. Bellmann-Ford für kürzeste Wege
 - Kruskal-Algorithmus für minimale Spannbäume
 - Ford-Fulkerson-Algorithmus für maximale Flüsse bzw. maximales Matching
- viele NP-vollständige Optimierungsprobleme
 - Traveling Salesman Problem, Cliquesproblem, Färbungsproblem ...
 - Bestimmung eines planaren Graphen (Graph-Darstellung ohne überschneidende Kanten)

[†] Animationen u.a. unter <http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/main/index.shtml>

