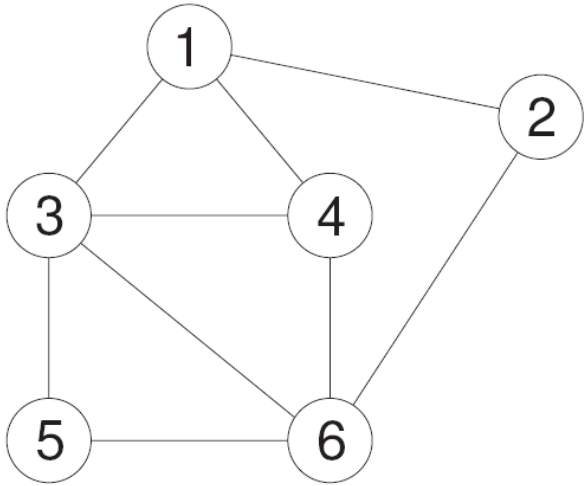
Algorithmen und Datenstrukturen

9. GRAPHEN 1
GRUNDLAGEN

WeST

Graphen

Ein Graph ist das mathematische Modell eines Netzwerks bestehend aus Knoten und Kanten



- Vielfältiger Einsatz
  - Verbindungsnetzwerke: Bahnnetz, Flugverbindungen, Strassenkarten,…
  - Verweise: WWW, Literaturverweise, Wikipedia, symbolische Links,…
  - Technische Modelle: Platinen-Layout, finite Elemente, Computergrafik
- Bäume und Listen sind spezielle Graphen

Algorithmen und Datenstrukturen

9.1. EINFÜHRUNG GRAPHEN

Algorithmen und Datenstrukturen

9.1.1 GRAPHENTHEORIE

Ungerichtete Graphen

Definition (Ungerichteter Graph)

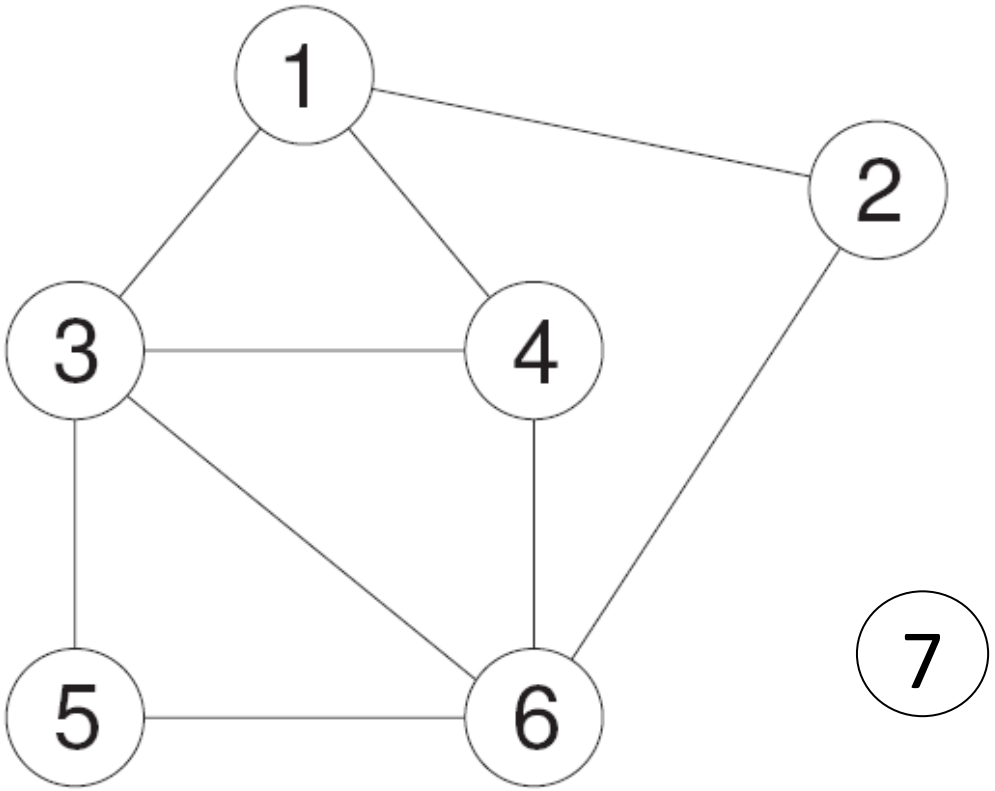
Ein ungerichteter Graph G ist ein Tupel G=(V,E) mit

1. V ist eine Menge von Knoten
2. E ist eine Menge von ungeordneten Paaren aus V, d.h. jedes $e \in E$ ist eine zweielementige Teilmenge der Knotenmenge V ($e=\{a,b\}$ mit $a,b \in V$)

Anmerkungen:

- Wir betrachten *simple* Graphen: keine Schleifen, d.h. Kanten von einem Knoten zu sich selbst
- Keine mehrfachen Kanten zwischen zwei Knoten (Parallelkanten)

Beispiel: Ungerichteter Graph



- V={1,2,3,4,5,6,7}
- E={{1,2},{1,3},{1,4},{2,6},{3,4},{3,5},{3,6},{4,6},{5,6}}

Adjazenz, Inzidenz, Grad, Weg

Sei G=(V,E) ein Graph.

- Zwei Knoten $v,w \in V$ heißen *adjazent*, falls $\{v,w\} \in E$ (v heißt dann auch *Nachbar* von w)
- Eine Kante $\{v,w\} \in E$ ist *inzident* zu einem Knoten $z \in V$, falls v=z oder w=z
- Der *Grad* (engl. *degree*) eines Knotens $v \in V$ ist die Anzahl seiner inzidenten Kanten, d.h.

$$degree(v)=|\{\{w,x\} \in E | w=v \text{ oder } x=v\}|$$

- Ein *Weg* W ist eine Sequenz von Knoten $W=(v_1,...,v_n)$ mit $v_1,...,v_n \in V$, für die gilt:

$$\{v_i,v_{i+1}\} \in E \text{ für alle } i=1,...,n-1$$

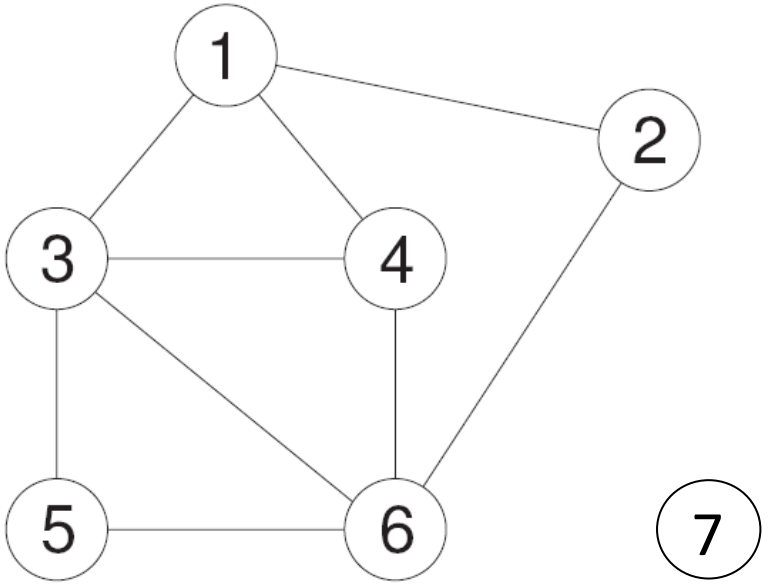
Pfad, Kreis, Länge

Sei G=(V,E) ein Graph.

- 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)
- Ein Weg P heisst *Kreis*, falls $v_1 = v_n$
- Ein Kreis K ist *elementar*, falls $v_i \neq v_j$ für alle i,j=1,...,n-1 mit i≠j (keine doppelten Knoten bis auf Anfangs- und Endpunkt)
- Die *Länge* eines Weges ist die Anzahl der durchlaufenen Kanten (die Länge eines Pfades ist also n-1)

Beispiel

- Knoten 1 und 3 sind adjazent
- (1,3,5,6,3,4) ist ein Weg
- (1,4,6,5) ist ein Pfad
- (1,3,4,6,3,4,1) ist ein Kreis
- Der Grad von Knoten 4 ist 3
- (3,4,6,3) ist ein elementarer Kreis
- Die Länge von (3,4,6,3,4,1) ist 4
- Die Länge von (1,3,6) ist 2



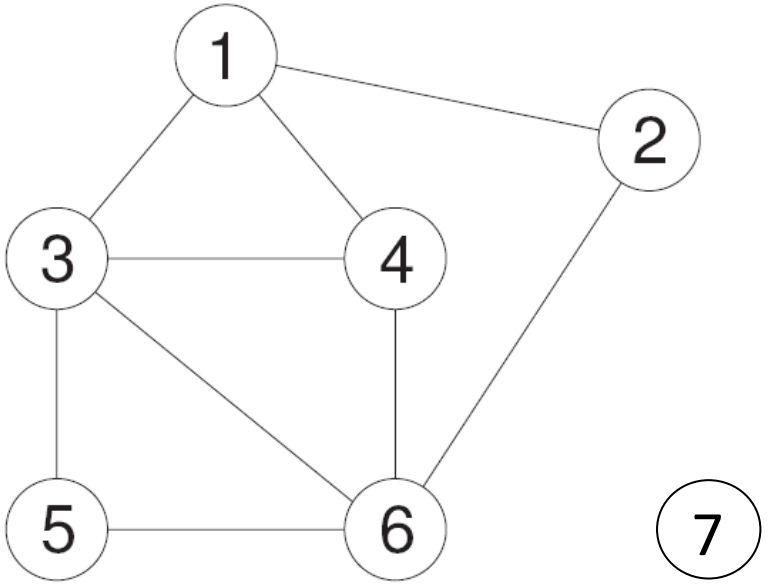
Teilgraph, Erreichbarkeit, Zusammenhang

Sei G=(V,E) ein Graph.

- Ein Graph G‘=(V‘,E‘) heißt *Teilgraph* von G, falls V‘$\subseteq$ V und E‘$\subseteq$ E $\cap$(V‘xV‘)
- Ein Knoten w$\in$V heißt *erreichbar* von einem Knoten v$\in$V, falls ein Weg W=(v_1,...,v_n) existiert mit v_1=v und v_n=w
- G heißt (einfach) *zusammenhängend*, falls für alle v,w$\in$V gilt, dass w von v erreichbar ist
- Ein Teilgraph G’=(V’,E’) von G heißt *Zusammenhangskomponente* von G, falls G’ zusammenhängend ist und kein Teilgraph G’’=(V’’,E’’) von G existiert mit V’ $\subset$ V’’

Beispiel

- G‘=({3,4,6},{{3,4},{4,6}}) ist ein Teilgraph von G
- Knoten 6 ist erreichbar von Knoten 1
- Knoten 7 ist nicht erreichbar von Knoten 1
- G ist nicht zusammen-hängend



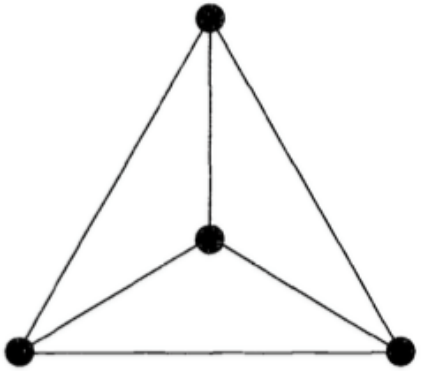
- Der Teilgraph G“=(V“,E“) mit
  V“ = {1,2,3,4,5,6}
  E“ = {{1,2},{1,3},{1,4},{2,6},{3,4},{3,5},{3,6},{4,6},{5,6}}
  ist eine Zusammenhangskomponente von G
- Der Teilgraph G“‘=({7}, ∅) ist eine Zusammenhangskomponent von G

Spezielle Graphen

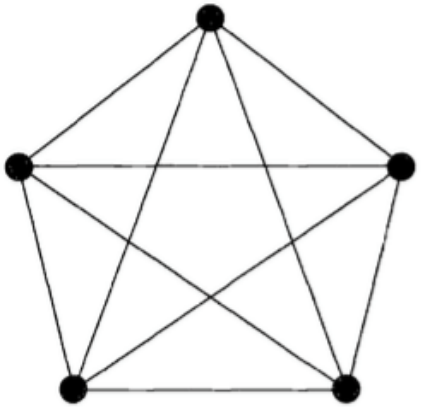
Sei G=(V,E) ein Graph.

- G heißt planar, falls er ohne Überschneidungen der Kanten in der Ebene gezeichnet werden kann (formale Definition ist etwas komplizierter)
- G heißt vollständig, falls E=VxV
- G heißt regulär, falls alle Knoten denselben Grad haben
- G heißt bipartit, falls $V=V_1 \cup V_2$ und
  - keine zwei Knoten in V_1 sind adjazent
  - Keine zwei Knoten in V_2 sind adjazent

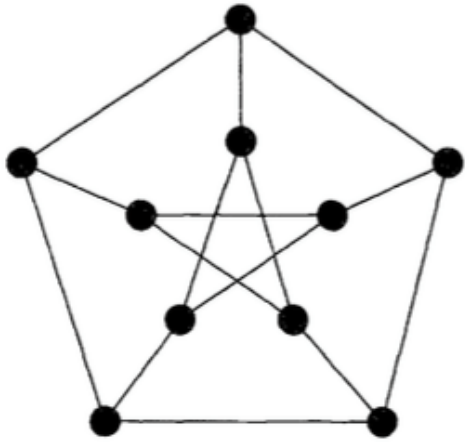
Beispiel



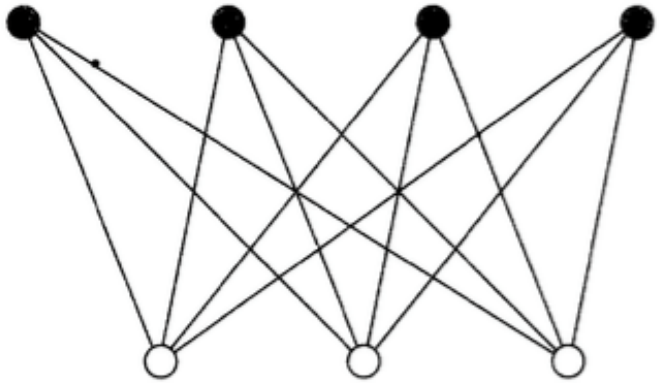
planar, regulär, vollständig



regulär, vollständig



regulär



bipartit

Gerichtete Graphen

Definition (Gerichteter Graph)

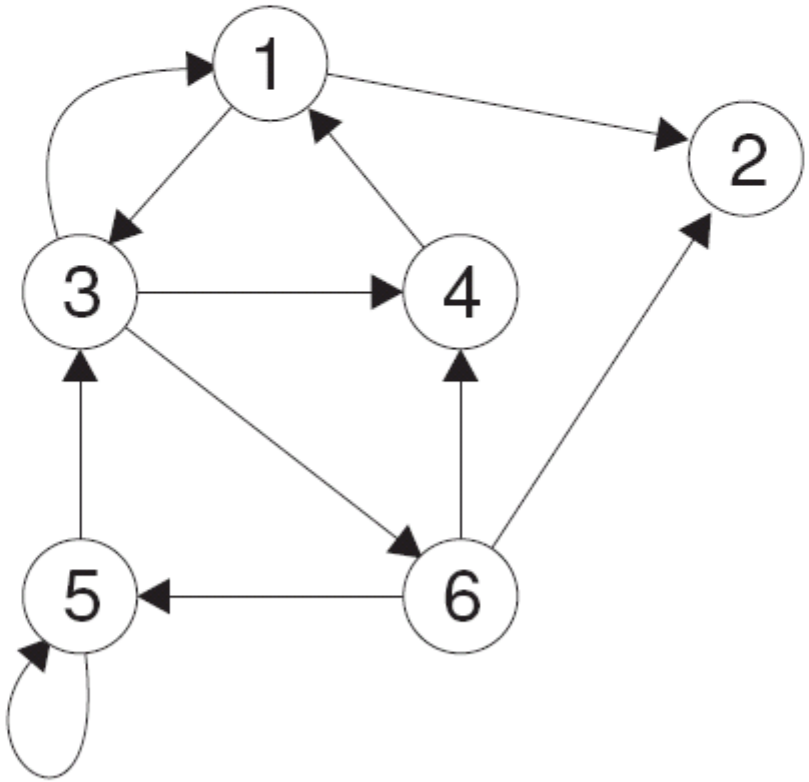
Ein gerichteter Graph G (auch *Digraph*) ist ein Tupel G=(V,E) mit

1. V ist eine Menge von Knoten
2. E ist eine Menge von *geordneten* Paaren aus V, d.h. jedes $e \in E$ ist ein Tupel e=(a,b) mit $a,b \in V$

Anmerkungen:

- (a,b) ist eine andere Kante als (b,a) (Unterscheide (a,b) und {a,b})
- Schleifen (a,a) sind erlaubt

Beispiel: Gerichteter Graph



- $G_g = (V_g, E_g)$
- $V_g = \{1,2,3,4,5,6\}$
- $E_g = \{(1,2),(1,3),(3,1),(3,4),(3,6),(4,1),(5,3),$ $(5,5),(6,2),\ (6,4),(6,5)\}$

Adjazenz, Inzidenz, Grad

Sei G=(V,E) ein Digraph.

- Zwei Knoten v,w∈V heißen *adjazent*, falls (v,w) ∈ E oder (w,v) ∈ E
- Für (v,w) ∈ E heißt w *Nachfolger* von v und v *Vorgänger* von w
- Eine Kante (v,w) ∈E ist *inzident* zu einem Knoten z∈V, falls v=z oder w=z
- Der *Eingangsgrad* (engl. *in-degree*) eines Knotens v∈V ist die Anzahl seiner Vorgänger

  $$indeg(v)=|\{(w,v) \in E\}|$$
- Der *Ausgangsgrad* (engl. *out-degree*) eines Knotens v∈V ist die Anzahl seiner Nachfolger

  $$outdeg(v)=|\{(v,w) \in E\}|$$

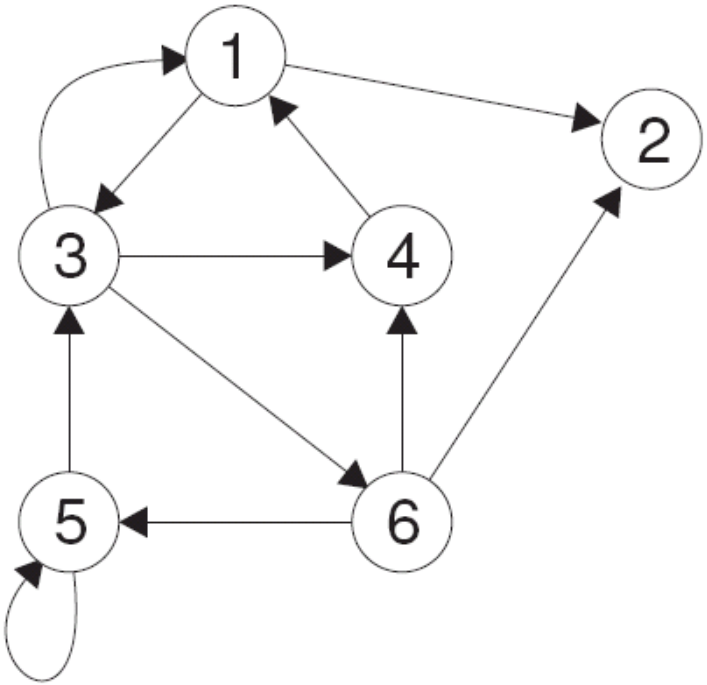
Weg

Sei G=(V,E) ein Digraph.

- 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
- Die Definitionen von Pfad, (elementarer) Kreis und Länge sind analog zu ungerichteten Graphen

Beispiel

- Knoten 1 ist Vorgänger zu Knoten 3
- Knoten 4 ist Nachfolger zu Knoten 6
- Der Eingangsgrad von Knoten 3 ist 2
- Der Ausgangsgrad von Knoten 3 ist 3
- (1,3,6,5,5,3,1) ist ein (gerichteter) Weg
- (1,3,6) ist ein Pfad
- (6,5,5,3,6) ist ein Kreis
- (6,5,3,6) ist ein elementarer Kreis



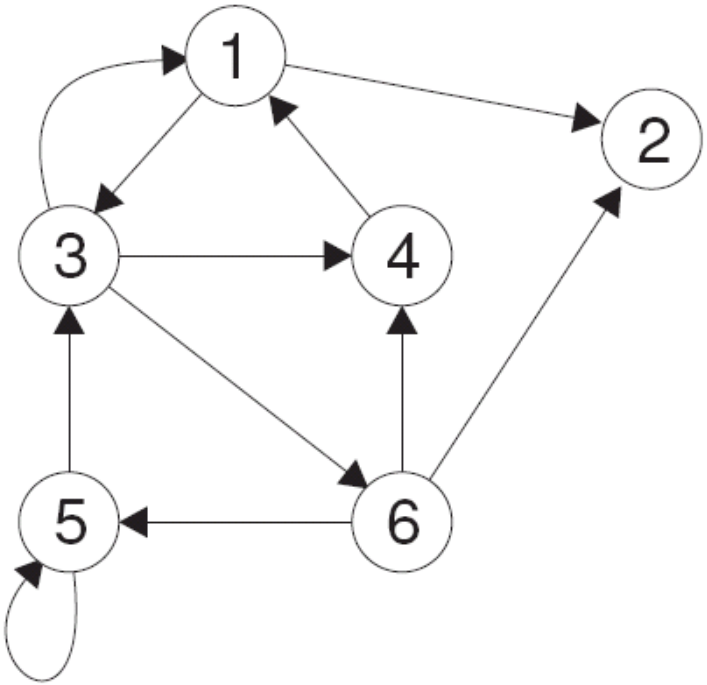
Teilgraph, Erreichbarkeit, Zusammenhang

Sei G=(V,E) ein Digraph.

- Ein Graph G‘=(V‘,E‘) heißt *Teilgraph* von G, falls V‘$\subseteq$ V und E‘$\subseteq$ E $\cap$(V‘xV‘)
- Ein Knoten w$\in$V heißt *erreichbar* von einem Knoten v$\in$V, falls ein Weg W=(v_1,...,v_n) existiert mit v_1=v und v_n=w
- G heißt (stark) *zusammenhängend*, falls für alle v,w$\in$V gilt, dass w von v und v von w erreichbar ist
- Ein Teilgraph G’=(V’,E’) von G heißt *Starke Zusammenhangskomponente* von G, falls G’ stark zusammenhängend ist und kein Teilgraph G’’=(V’’,E’’) von G existiert mit V’ $\subset$ V’’

Beispiel

- G‘=({1,3,4},{(1,3),(4,1)}) ist Teilgraph von G_g
- Knoten 6 ist erreichbar von Knoten 1
- Knoten 5 ist nicht erreichbar von Knoten 2
- Der Teilgraph G“=(V“,E“) mit
  V“ = {1,3,4,5,6}
  E“ = { (1,3),(3,1),(3,4),(3,6),(4,1),(5,3),(5,5),(6,4),(6,5)}
  ist eine starke Zusammenhangskomponente von G_g



Gerichtete und ungerichtete Graphen

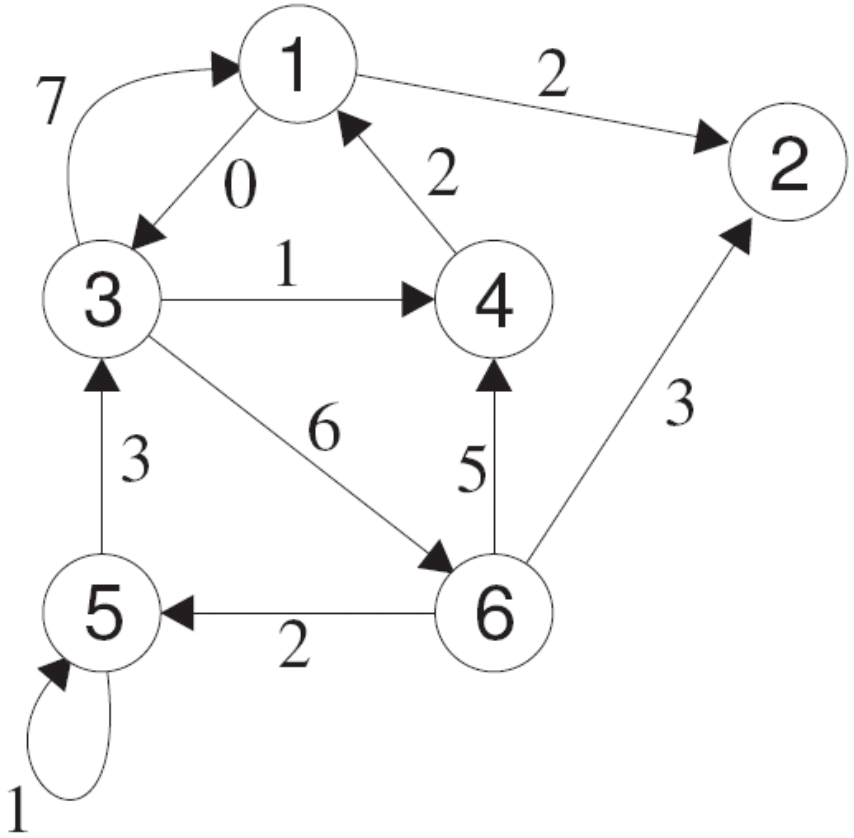
- Ein ungerichteter Graph kann in einen gerichteten Graphen transformiert werden, indem jede ungerichtete Kante {v,w} durch zwei gerichtete Kanten (v,w) und (w,v) ersetzt wird
- Beispielsweise ist dann Zusammenhang identisch mit starkem Zusammenhang
- Normalerweise werden wir uns wegen der größeren Ausdrucksstärke auf gerichtete Graphen konzentrieren, d.h. im folgenden benutzen wir den Term “Graph” meist als Synonym für Digraph

Definition: Gewichtete Graphen

- Graph G=(V,E) und
- Kantengewichtsfunktion g
- G=(V,E,g) mit

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

- gerichtet oder ungerichtet
- Kantengewichte müssen nicht notwendigerweise natürliche Zahlen sein



Algorithmen und Datenstrukturen

9.1.2 REPRÄSENTATION VON GRAPHEN

Repräsentation von Graphen

Fragestellung:

Was ist eine effiziente Datenstruktur für Graphen?

Überblick:

- Kanten- und Knotenlisten
- Matrixdarstellung
- Adjazenzlisten

Kanten- und Knotenlisten

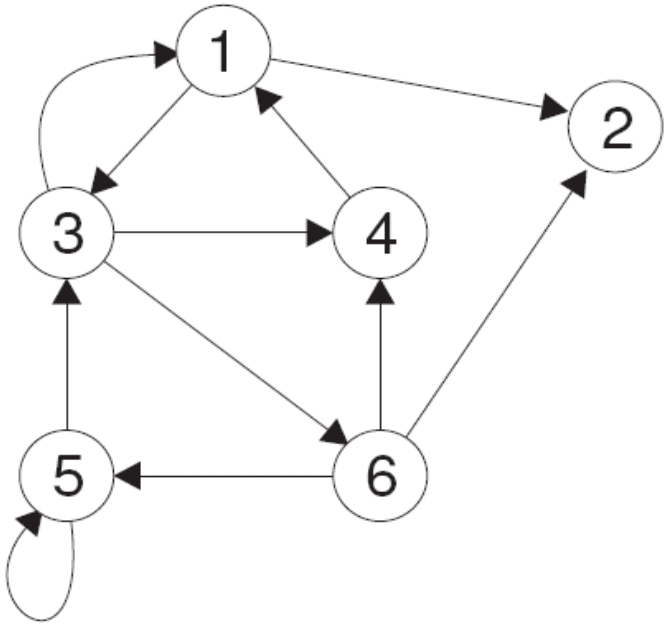
- Einfache Realisierung bei durchnummerierten Knoten
- Historisch erste verwendete Datenstruktur
- Als Austauschformat geeignet
- Auflistung nach Knoten oder nach Kanten sortiert

Beispiel: Kantenlisten

- Kantenliste für G_g:

Knotenzahl Kantenzahl Kante1 Kante2….

6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4



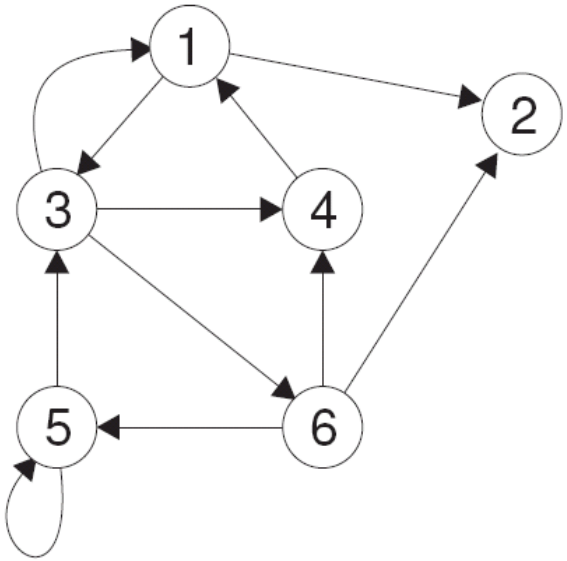
Beispiel: Knotenlisten

- G_g als Knotenliste:

$$6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5$$

Teilfolge 2, 2, 3 bedeutet

- *„Knoten 1 hat Ausgangsgrad 2 und herausgehende Kanten zu den Knoten 2 und 3“*



Vergleich: Kanten- und Knotenlisten

Falls ein Graph mehr Kanten als Knoten hat (=„Normalfall“), benötigen Knotenlisten weniger Speicherbedarf als Kantenlisten:

- Kantenlisten: 2+2|E|
- Knotenlisten: 2+|V|+|E|

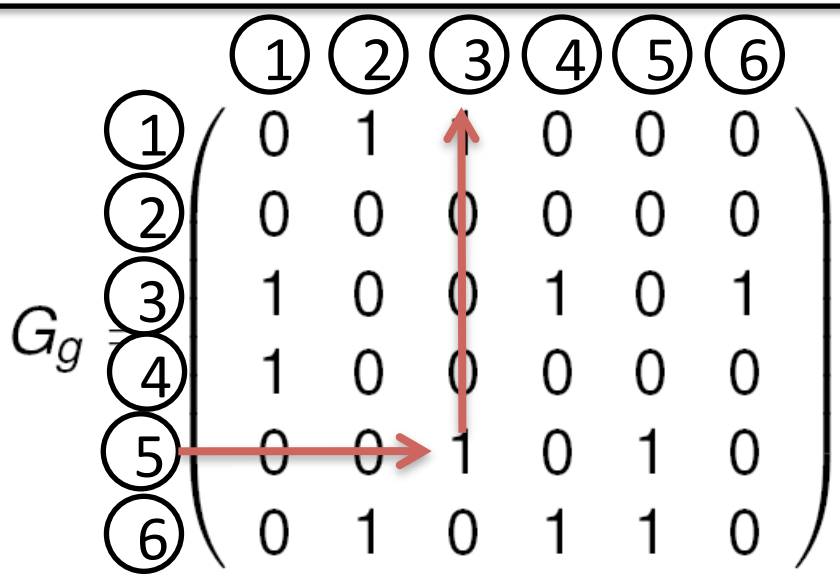
Definition: Adjazenzmatrix

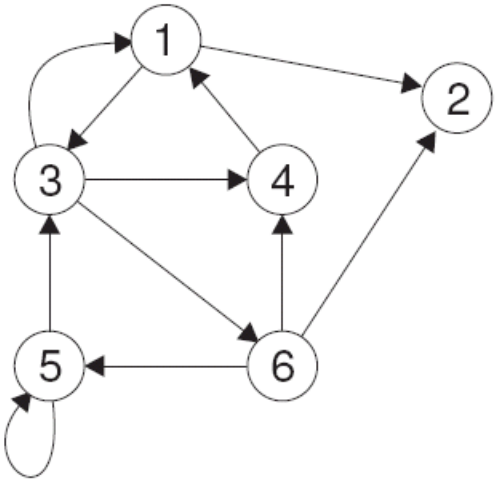
- Adjazenz bedeutet berühren, aneinandergrenzen
- Darstellung des Graphen als Boole‘sche Matrix
- 1-Einträge für direkte Nachbarschaft

- A ist eine Adjanzmatrix für Graph G=(V,E):

  $(A_{ij}) = 1$ gdw. $(i,j) \in E$

Beispiel: Adjazenzmatrix gerichtet





Adjazenzmatrix: Eigenschaften

- A ist eine Adjanzmatrix für Graph G=(V,E):
  $(A_{ij}) = 1$ gdw. $(i,j) \in E$

Besonderheiten für bestimmte Graphentypen:

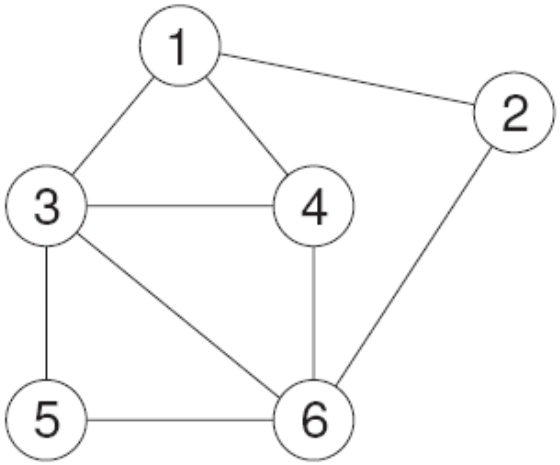
- Ungerichtete Graphen: Halbmatrix (Dreieck) reicht aus
- Gewichtete Graphen: Gewichte statt Boole‘sche Werte

Vorteile der Adjazenzmatrix:

- Einige Graphoperationen als Matrixoperationen möglich
  - Erreichbarkeit durch iterierte Matrixmultiplikation
  - Schöne Eigenschaften für die mathematische Analyse

Adjazenzmatrix: ungerichtet

$$G_u = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$



Adjazenzmatrix: ungerichtet als Dreiecksmatrix

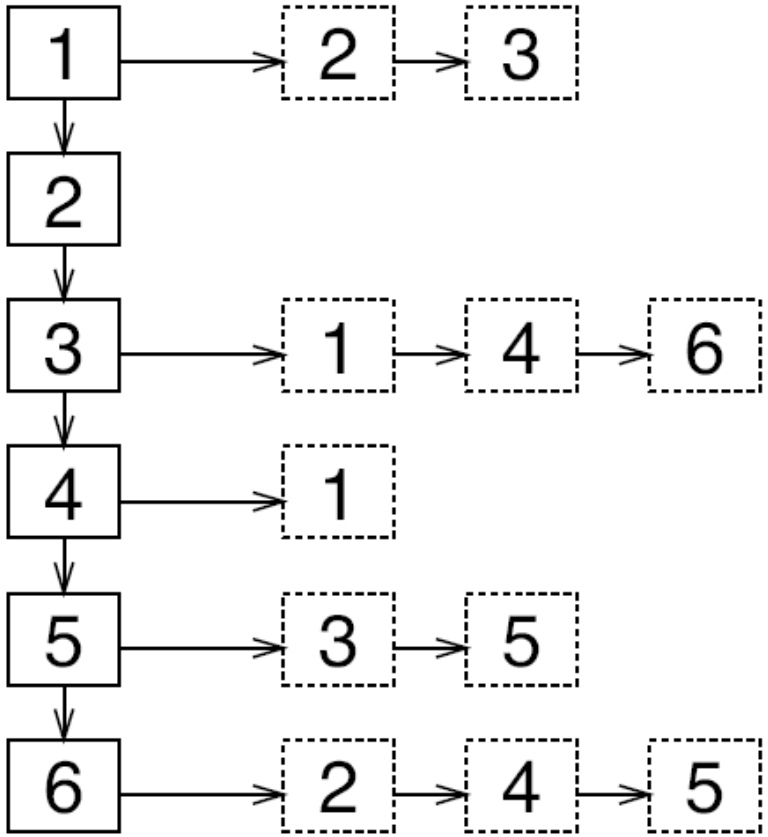
$$G_u = \begin{pmatrix} 0 & & & & & \\ 1 & 0 & & & & \\ 1 & 0 & 0 & & & \\ 1 & 0 & 1 & 0 & & \\ 0 & 0 & 1 & 0 & 0 & \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

- Diagonale kann ebenfalls weggelassen werden, wenn Schleifen verboten sind

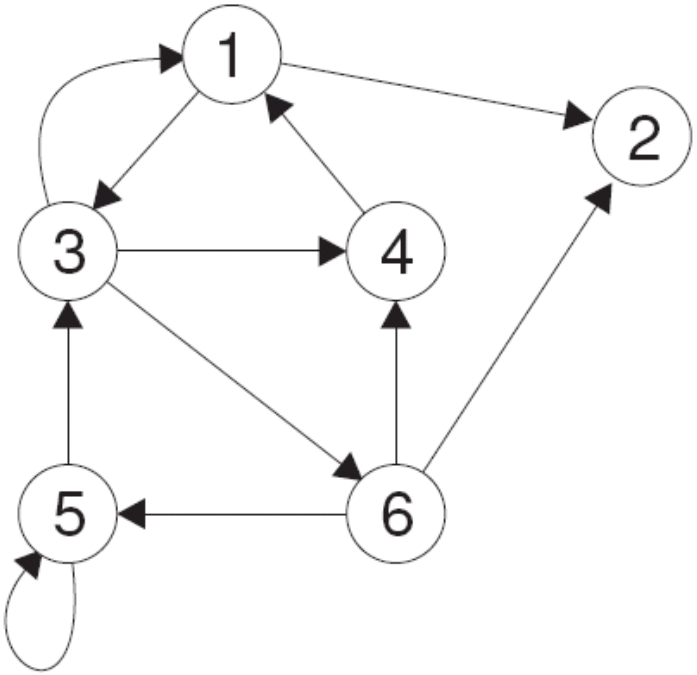
Definition: Adjazenzliste

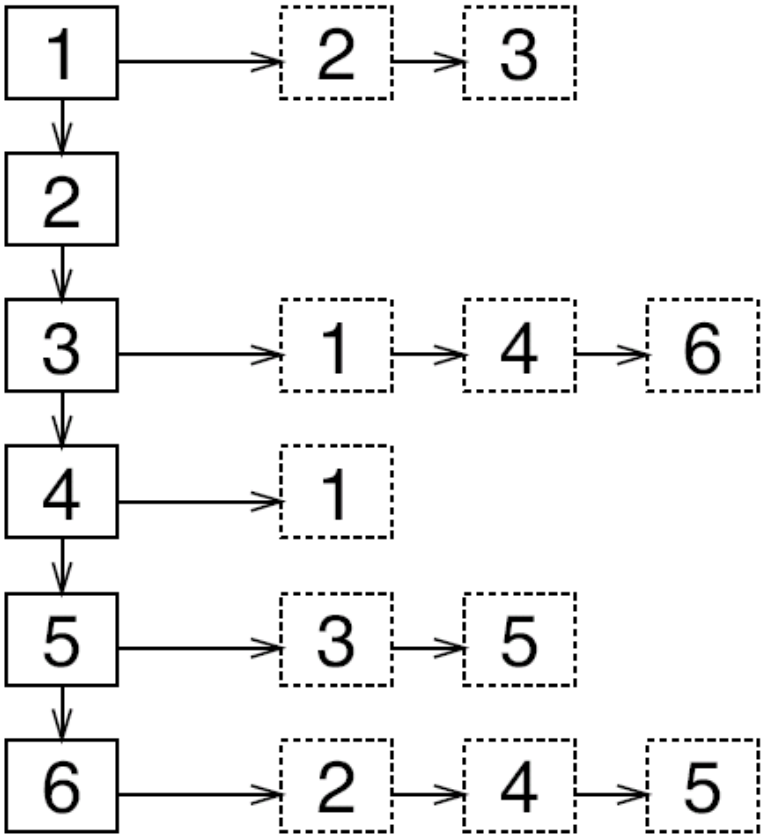
- Liste der Knoten (alternativ: Array)
- pro Knoten, die von ihm ausgehenden Kanten
  - als Liste (besonders geeignet für dünn besetzte Matrizen) oder Array von Zeigern
- Graph durch |V|+1 verkettete Listen realisiert

- Erlaubt dynamische Erweiterungen im Sinne verketteter Listen (Knotenlisten können natürlich auch als verkettete Listen realisiert werden)



Beispiel: Adjazenzliste





Speicherbedarf bei Adjazenzlisten

- Seien n=|V| und m=|E|
- Benötigt werden insgesamt

$$n + \sum_{i=1}^{n} ag(i) = n + m$$

Listenelemente

ag(i) = Anzahl Nachfolger von i

Transformation zwischen den Darstellungen

- Die vorgestellten Realisierungsvarianten sind äquivalent
- Jede Darstellung kann in jede andere ohne Informationsverlust transformiert werden
- Auslesen der einen Darstellung und anschließend das Erzeugen der jeweils anderen Darstellung
- Aufwand dieser Transformationen variiert von $O(n+m)$ bis $O(n^2)$, wobei im schlechtesten Fall $m=n^2$ gilt
- n^2 tritt immer auf, wenn eine naive Matrixdarstellung beteiligt ist
- Nicht naive Darstellungen für sehr dünn besetzte Matrizen nötig

Komplexitätsbetrachtungen

- Kantenlisten
  - Einfügen von Kanten (Anhängen zweier Zahlen) und von Knoten (Erhöhung der ersten Zahl um 1): besonders günstig
  - Löschen von Kanten: Verschiebung nachfolgender Kanten
  - Löschen von Knoten: Nummerierung der Knoten aktualiseren
- Knotenlisten
  - Einfügen von Knoten (Erhöhung der ersten Zahl und Anhängen einer 0): günstig
- Matrixdarstellung
  - Manipulieren von Kanten sehr effizient ausführbar
  - Aufwand bei Knoteneinfügung hängt von Realisierung ab
    - Worst case: Kopieren der Matrix in eine größere Matrix
- Adjazenzliste
  - Unterschiedlicher Aufwand, je nachdem, ob die Knotenliste als Array (mit Direktzugriff) oder als verkettete Liste (mit sequenziellem Durchlauf) realisiert wird

Komplexitätsbetrachtungen

Operation	Kanten-liste	Knoten-liste	Adjazenz-matrix	Adjazenz-liste
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	$O(m)$	$O(n+m)$	$O(n^2)$	$O(n+m)$

- Löschen eines Knotens impliziert für gewöhnlich Löschen der zugehörigen Kanten

Algorithmen und Datenstrukturen

9.1.3 DATENSTRUKTUREN FÜR GRAPHEN

Graphen in Java

- Kein hauseigene Graphenimplementierung in Java
- Diverse Pakete für verschiedene Anwendungen:
  - Jung (`http://jung.sourceforge.net`)

```
Graph<Integer, String> g = new SparseMultigraph<Integer, String>();
g.addVertex((Integer)1);
g.addVertex((Integer)2);
g.addEdge("Edge1", 1, 2);
```

  - Neo4j (`http://www.neo4j.org`)

```
GraphDatabaseService= new
  GraphDatabaseFactory().newEmbeddedDatabase("PATH");
Transaction tx = graphDb.beginTx();
try{
  Node firstNode = graphDb.createNode();
  Node secondNode = graphDb.createNode();
  Relationship relationship = firstNode.createRelationshipTo(secondNode,
            … );
  tx.success();
}finally{
  tx.finish();
}
```

Datenstrukturen für Graphen

- Allgemeine Schnittstelle (für die Vorlesung)

```
public interface Graph {
   public int addNode();
   public boolean addEdge (int orig, int dest);
}
```

Adjazenzliste: Rahmenstruktur

```
public class AdjacencyListGraph implements Graph {
  private int [][] adjacencyList = null;
  …
}
```

Adjazenzliste: Knoten hinzufügen

```
public int addNode(){
   int nodeNumber = (adjacencyList == null)?
          0:adjacencyList.length;
   int [][] newadjacencyList =
      new int[nodeNumber+1][];
   for (int i=0; i< nodeNumber; i++)
      newAdjacencyList [i]=adjacencyList[i];
   newAdjacencyList[nodeNumber]=null;
   adjacencyList=newAdjacencyList;
   return nodeNumber+1;
}
```

alte adjacencyList kopieren

Neuer Knoten hat noch keine Kanten

Adjazenzliste: Kante hinzufügen

```
public boolean addEdge(int orig, int dest){
  int nodeNumber = (adjacencyList ==null)?
        0: adjacencyList.length;
  if (orig > nodeNumber || dest > nodeNumber
     || orig < 1 || dest <1)
     return false;
  if (adjacencyList[orig-1] != null)
      for (int n : adjacencyList[orig-1])
         if (n==dest) return false;
  if (adjacencyList[orig-1] ==null){
     adjacencyList[orig-1]=new int [1];
     adjacencyList[orig-1][0]=dest;
  }
…

}
```

Kante bereits vorhanden?

Erste Kante am Knoten orig?

Adjazenzliste: Kante hinzufügen

```
    …
    else {
        int[] newList =
            new int[adjacencyList[orig-1].length+1];
        System.arraycopy(adjacencyList[orig-1],0,
                newList,0,adjacencyList[orig-1]-length);
        newList[adjacencyList[orig-1].length]=dest;
        adjacencyList[orig-1]=newList;
    }
    return true;
}
```

Algorithmen und Datenstrukturen

9.1 EINFÜHRUNG GRAPHEN

ZUSAMMENFASSUNG

Zusammenfassung

- Graphentheorie
  - ungerichtete, gerichtete, gewichtete Graphen
- Repräsentation von Graphen
  - Knoten- und Kantenlisten
  - Adjazenzlisten
  - Adjazenzmatrix
- Datenstrukturen für Graphen

Algorithmen und Datenstrukturen

9.2. BREITENSUCHE

Breitensuche

Fragestellung:

Wie kann man die Knoten eines Graphen effizient aufzählen?

$\rightarrow$ Breitensuche (Breadth-First-Search, BFS)

- Knoten eines Graphen werden nach Entfernung vom Zielknoten aufgezählt

$\rightarrow$ Tiefensuche (später)

Breitensuche

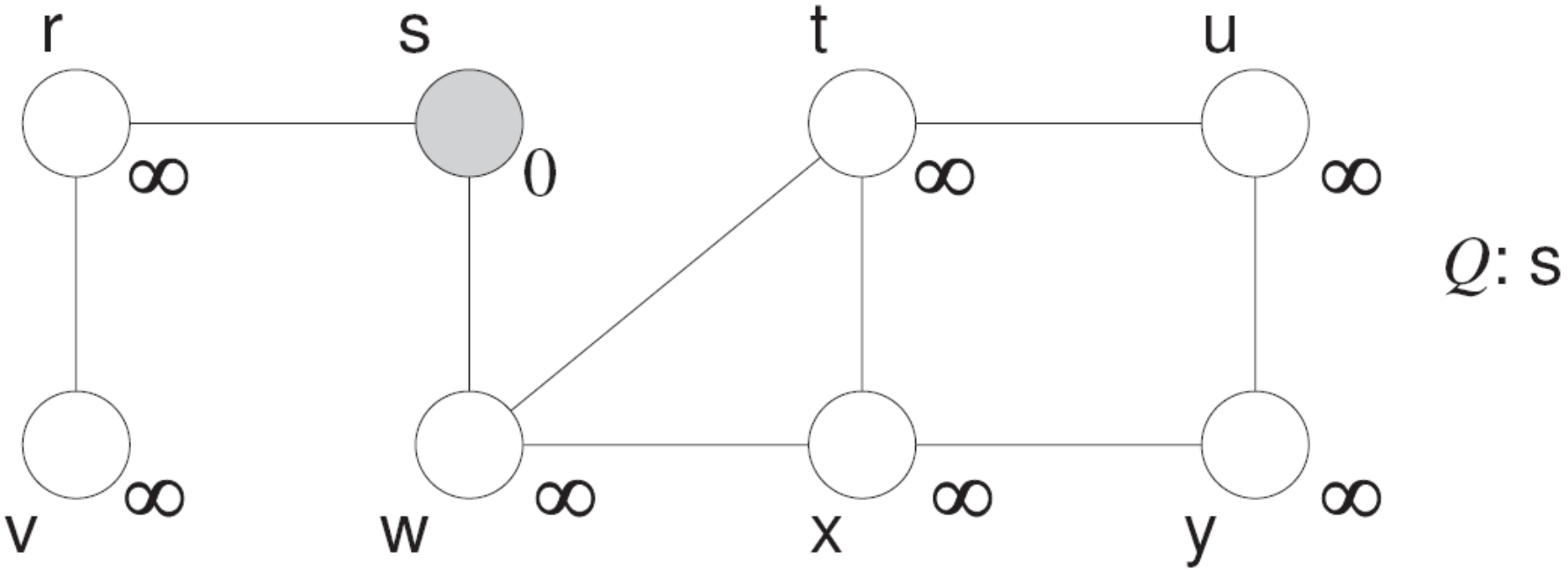
Breitendurchlauf für ungerichtete Graphen

- Warteschlange als Zwischenspeicher
- Farbmarkierungen beschreiben Status der Knoten
  - weiss: unbearbeitet
  - grau: in Bearbeitung
  - schwarz: abgearbeitet
- Pro Knoten wird Entfernung zum Startknoten berechnet

Breitensuche: Initialisierung

- Startknoten
  - In Warteschlange einfügen
  - Farbe grau
  - Entfernung 0
- Andere Knoten
  - Entfernung unendlich
  - Farbe weiß

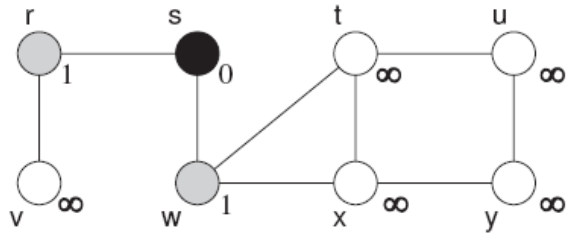
Breitensuche: Erster Schritt



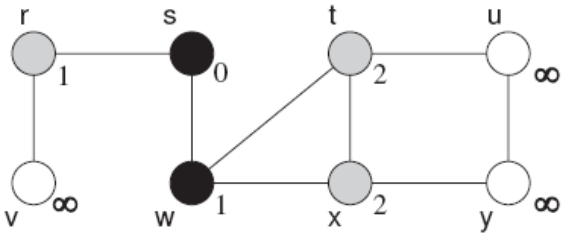
Breitensuche: Einzelschritt

- Aktuellen Knoten k aus Warteschlange entnehmen
- k wird schwarz
- Alle von k aus erreichbaren weissen Knoten:
  - Grau färben
  - Entfernung ist Entfernungswert von k plus 1
  - in die Warteschlange aufnehmen

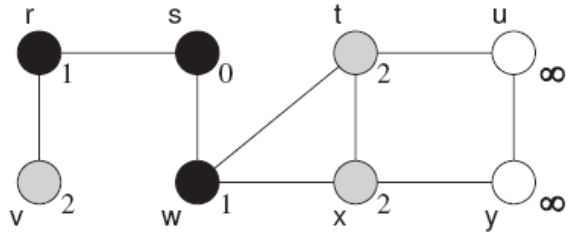
Breitensuche: Folgeschritte



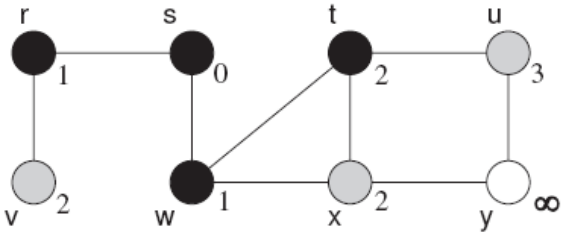
Q: w r



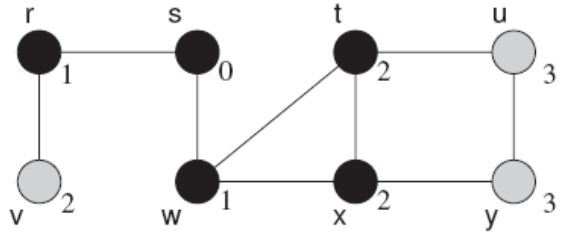
Q: r t x



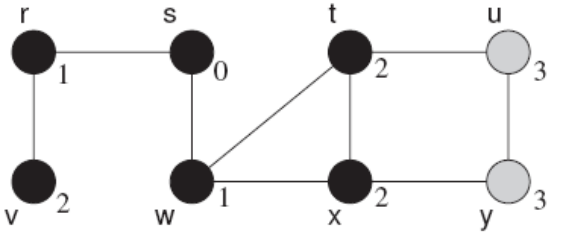
Q: t x v



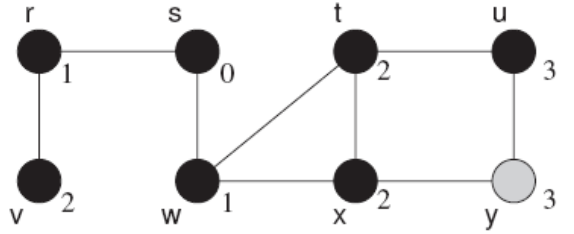
Q: x v u



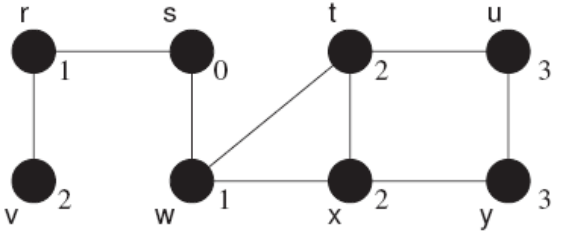
Q: v u y



Q: u y

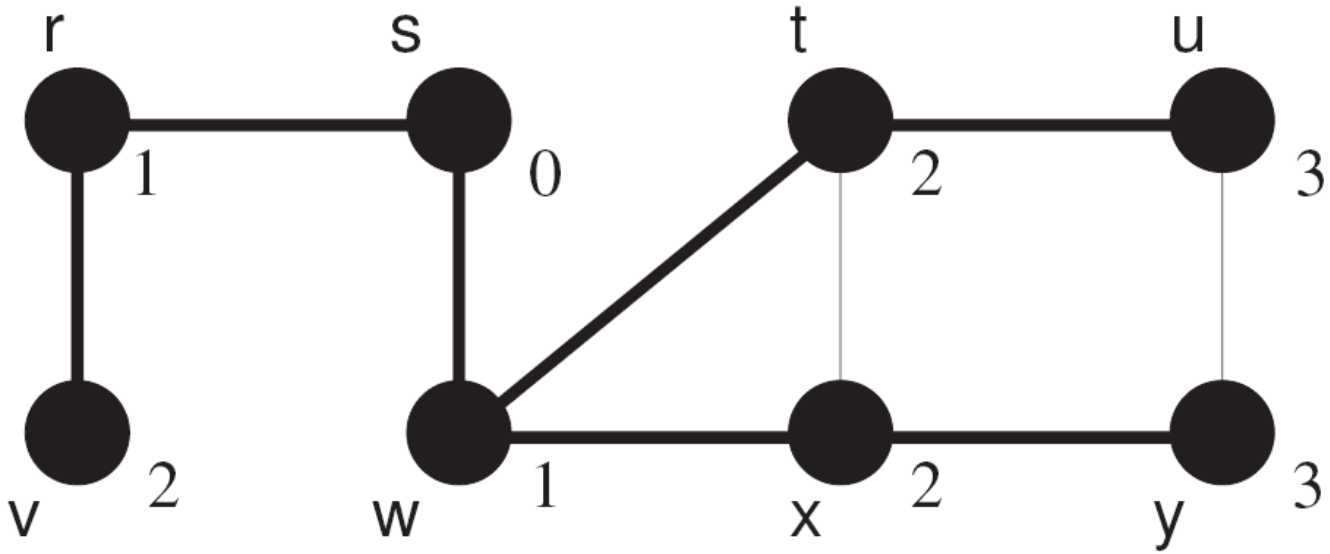


Q: y



Q:

Breitensuche: Ergebnis



Breitensuche: Algorithmus

Ergänzung zum Graph-Interface:

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

Breitensuche: Algorithmus 1/2

Breitendurchlauf als Iterator:

```
public class BfsIterator implements Iterator<Integer>{
    private Graph g;
    private Queue<Integer> q;
    private Set<Integer> visited;

    public BfsIterator(Graph g, int s){
        this.g = g;
        this.q = new LinkedList<Integer>();
        q.add(s);
        this.visited = new HashSet<Integer>();
    }

    public boolean hasNext() { return !this.q.isEmpty(); }

    public Integer next() {
        Integer n = this.q.poll();
        this.visited.add(n);
        for(Integer m: this.g.getChildren(n))
            if(!this.visited.contains(m) && !this.q.contains(m))
            this.q.add(m);
        return n;
    }
}
```

Breitensuche: Algorithmus 2/2

Ausgabe aller Knoten:

```
// Sei g ein Graph
Iterator<Integer> it = new BfsIterator(g,1);
while(it.hasNext())
    System.out.println(it.next());
```

Analyse

Theorem (Terminierung)

Die Breitensuche terminiert nach endlicher Zeit.

Beweis: Übung □

Theorem (Korrektheit)

Ist G zusammenhängend, so werden alle Knoten von G genau einmal besucht.

Beweis: Übung □

Theorem (Laufzeit)

Ist G=(V,E) zusammenhängend und ist die Laufzeit von `getChildren` linear in der Anzahl der Kinder, so hat die Breitensuche eine Laufzeit von O(|V|+|E|).

Beweis: Übung □

Algorithmen und Datenstrukturen

9.2 BREITENSUCHE
ZUSAMMENFASSUNG

Zusammenfassung

- Breitensuche als Verfahren um alle Knoten eines Graphen aufzuzählen

Algorithmen und Datenstrukturen

9.3. TIEFENSUCHE

Tiefensuche

Tiefendurchlauf (Depth-First-Search, DFS)

- Knoten werden aufgezählt indem
  - vom Startknoten aus ein Pfad soweit wie möglich verfolgt wird und
  - bei Bedarf ein Backtracking durchgeführt wird

- Farbmarkierungen für den Bearbeitungsstatus eines Knotens
  - weiß markiert: noch nicht bearbeiteter Knoten
  - graue Knoten: in Bearbeitung
  - schwarze Knoten: bereits fertig abgearbeitet

Tiefensuche: Algorithmus

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();
}
```

Tiefensuche: Algorithmus

```
enum Color {WHITE, GRAY, BLACK};

Map<Integer,Color> color  = new HashMap<Integer,Color>();
Map<Integer,Integer> pi   = new HashMap<Integer,Integer>();
Map<Integer,Integer> b    = new HashMap<Integer,Integer>();
Map<Integer,Integer> e    = new HashMap<Integer,Integer>();

int time = 0;
```

- color: speichert die Farbe (=Bearbeitungszustand) eines Knotens:
  - Weiss: der Knoten wurde noch nicht bearbeitet
  - Grau: der Knoten wird gerade bearbeitet
  - Schwarz: die Bearbeitung des Knotens wurde abgeschlossen
- pi: speichert den Vorgänger eines Knotens beim Durchlauf
- b: speichert den Zeitpunkt des Bearbeitungsbeginns eines Knotens
- e: speichert den Zeitpunkt des Bearbeitungsendes eines Knotens

Tiefensuche: Algorithmus

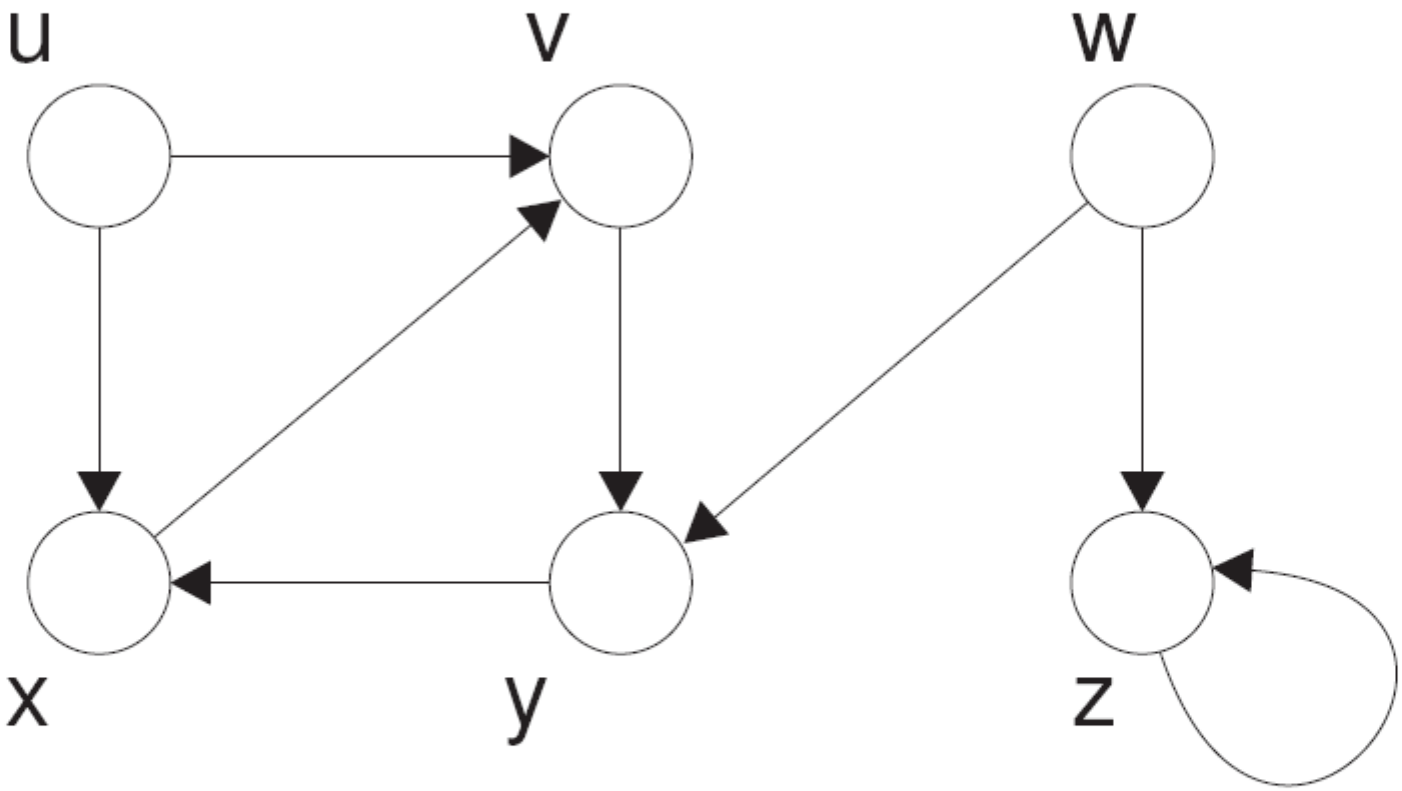
```
public void dfs(Graph g){
    for(Integer n: g.getNodes())
        color.put(n, Color.WHITE);
    for(Integer n: g.getNodes())
        if(color.get(n).equals(Color.WHITE))
            dfsVisit(g,n);
}

public void dfsVisit(Graph g, Integer n){
    color.put(n, Color.GRAY);
    time++;
    b.put(n, time);
    for(Integer m: g.getChildren(n)){
        if(color.get(m).equals(Color.WHITE)){
            pi.put(m, n);
            dfsVisit(g,m);
        }
    }
    color.put(n, Color.BLACK);
    time++;
    e.put(n, time);
}
```

Tiefensuche: Vorgehen

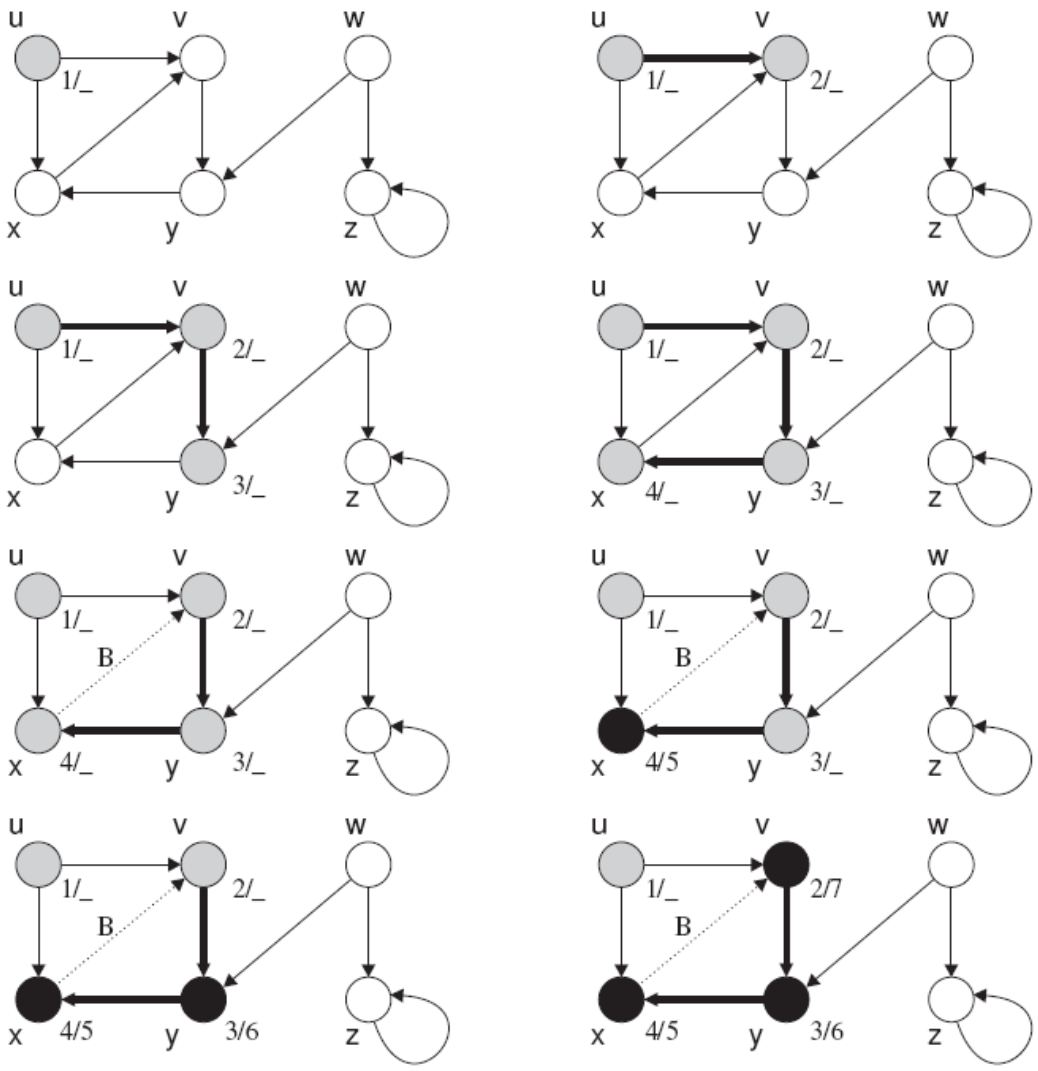
- Rekursiver Abstieg
- Pro Knoten zwei Werte plus Farbwerte
  - Beginn der Bearbeitung b
  - Ende der Bearbeitung e
- Rekursiver Aufruf nur bei weißen Knoten
  - Terminierung der Rekursion garantiert

- Die Ausführung von DFS resultiert in einer Folge von „DFS-Bäumen“:
  - Der erste Baum wird aufgebaut bis keine Knoten mehr hinzugefügt werden können
  - Anschliessend: wähle unbesuchten Knoten und fahre fort

Tiefensuche: Beispiel



Tiefensuche: Beispiel

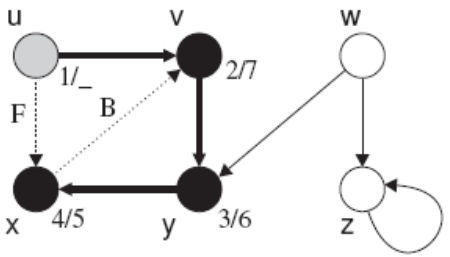


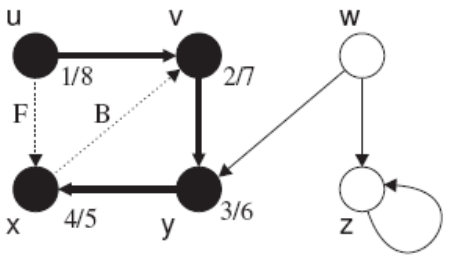
Notation:

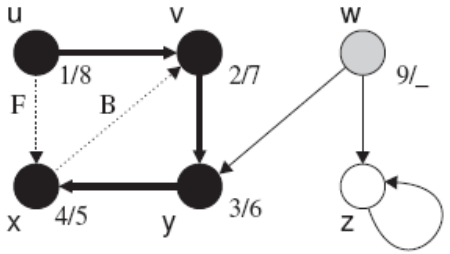
<Beginn der Bearbeitung b> / <Ende der Bearbeitung e>

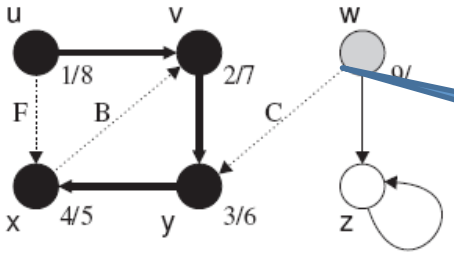
Tiefensuche: Beispiel

1. Durchlauf

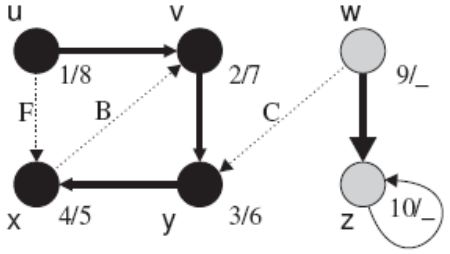


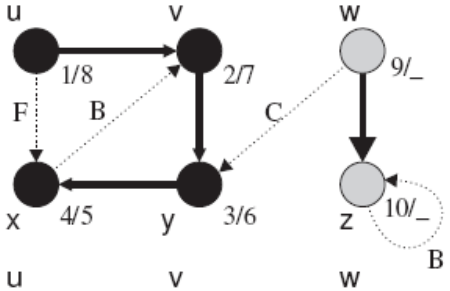


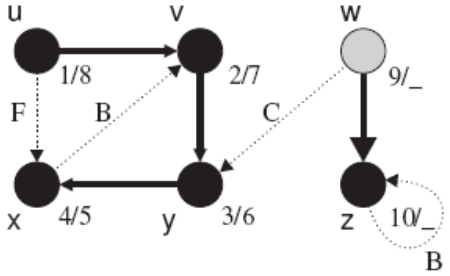


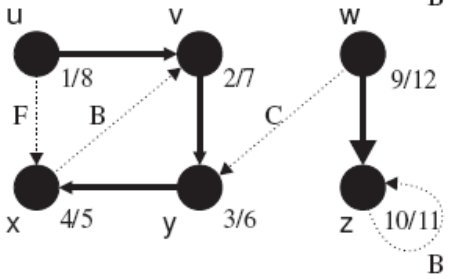


2. Durchlauf









WeST

Tiefensuche: gefundene Informationen

- Kanten des aufspannenden Baumes: **Zielknoten bei Test ist weiß**
- B-Kanten: **Zielknoten beim Test grau**
  - Back-Edges oder Rückkanten im aufgespannten Baum
  - Eine mit B markierte Kante zeigt einen Kreis im Graphen an!
- F-Kanten: **beim Test wird schwarzer Knoten** gefunden, dessen **Bearbeitungsintervall ins Intervall des aktuell bearbeiteten Knotens passt**:
  - Forward-Edge bzw. Vorwärtskante in den aufgespannten Baum
- C-Kanten: **Schwarzer Zielknoten** v, dessen **Intervall nicht ins Intervall passt** (b[u] > e[v]):
  - Cross-Edge, eine Kante, die zwei aufspannende Bäume verbindet

Analyse

Theorem (Terminierung)

Die Tiefensuche terminiert nach endlicher Zeit.

Beweis: Übung □

Theorem (Korrektheit)

Es werden alle Knoten von G genau einmal besucht.

Beweis: Übung □

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 die Tiefensuche eine Laufzeit von $O(|V|+|E|)$.

Beweis: Übung □

Anwendungen von DFS

- Test auf Zyklenfreiheit
  - Ein Graph heißt zyklenfrei, wenn es keinen Kreis K in G gibt
  - Test basiert auf dem Erkennen von Back-Edges
  - Effizienter als beispielsweise Konstruktion der transitiven Hülle

- **Topologisches Sortieren**
  - „topologisch“ sortieren nach Nachbarschaft, nicht nach totaler Ordnung

Algorithmen und Datenstrukturen

9.3 TIEFENSUCHE
ZUSAMMENFASSUNG

Zusammenfassung

- Tiefensuche als Verfahren um alle Knoten eines Graphen aufzuzählen
- Kann dazu genutzt werden um Kreis im Graphen zu finden

Algorithmen und Datenstrukturen

9.4. TOPOLOGISCHES SORTIEREN

Topologisches Sortieren

Fragestellung:

Gegeben ein azyklischer gerichteter Graph, wie können Knoten unter Berücksichtigung von Abhängigkeiten aufgezählt werden?

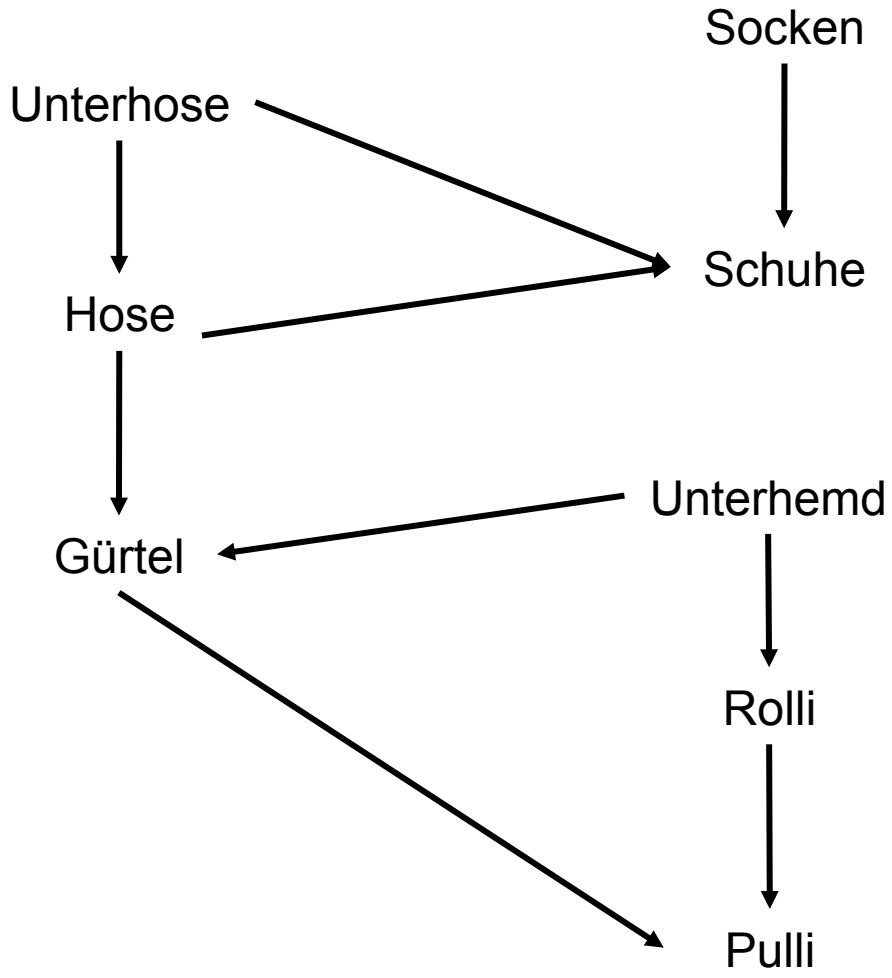
- Anwendung: Scheduling bei kausalen/zeitlichen Abhängigkeiten, z.B. Netzplantechnik
- Mathematisch: Konstruktion einer totalen Ordnung aus einer Halbordnung

Beispiel: Die sorgfältige Mutter

Die sorgfältige Mutter legt ihrem Kind morgens die Kleidungsstücke so auf einen Stapel, dass das Kind nur die Kleidungsstücke vom Stapel nehmen und anziehen muss und dann richtig gekleidet ist. Hierfür legt sie die Reihenfolgebedingungen fest:

Unterhose vor Hose
Hose vor Gürtel
Unterhemd vor Gürtel
Gürtel vor Pulli
Unterhemd vor Rolli
Rolli vor Pulli
Socken vor Schuhen
Hose vor Schuhen
Uhr: egal

Beispiel: Die sorgfältige Mutter

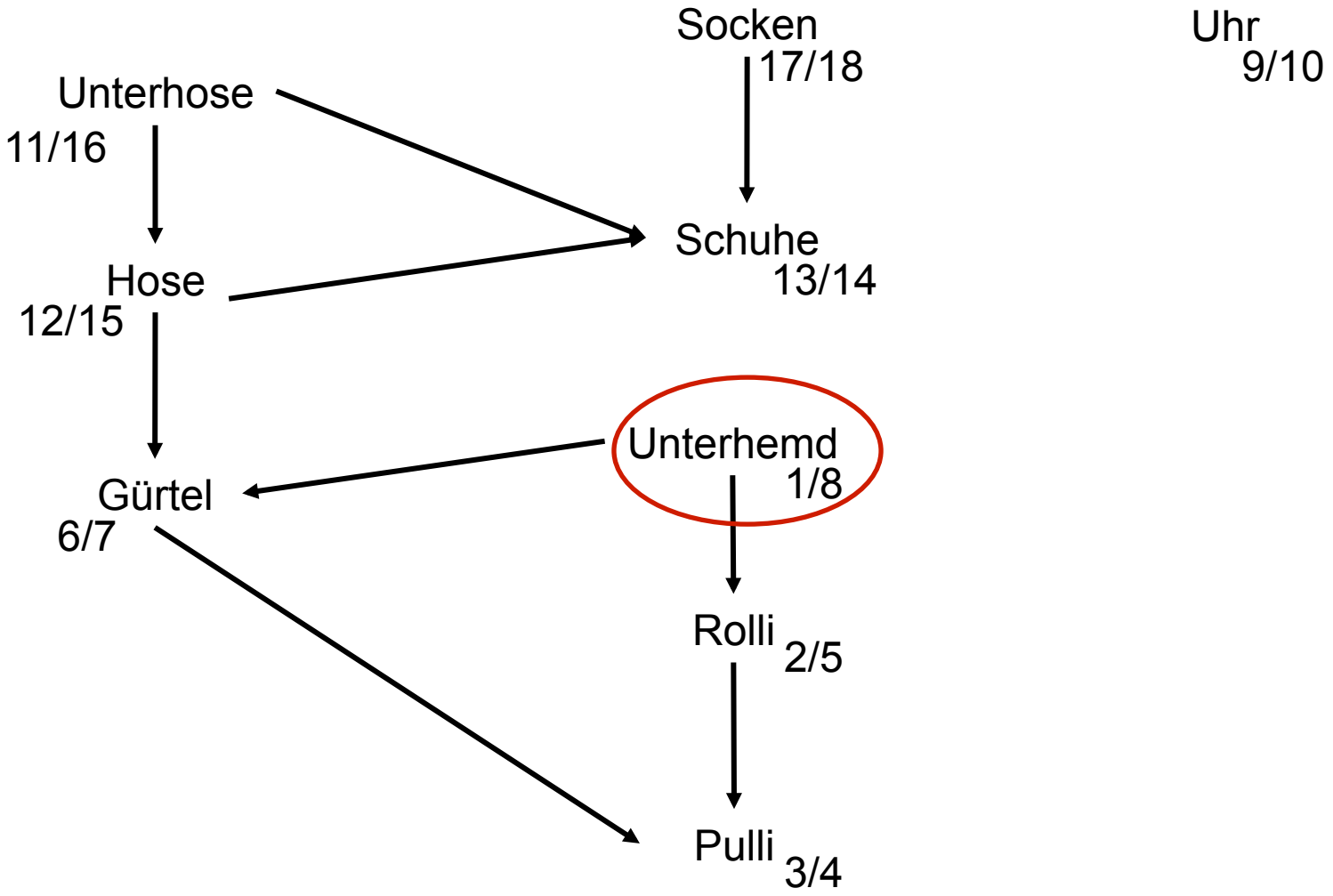


Uhr

Topologisches Sortieren und DFS

- DFS erstellt topologische Ordnung „on-the-fly“
- Sortierung nach e-Wert (invers) ergibt korrekte Reihenfolge
- Statt expliziter Sortierung nach e:
  - Knoten beim Setzen des e-Wertes vorne in eine verkettete Liste einhängen

Topologisches Sortieren mit DFS: Ergebnisgraph



Topologisches Sortieren: Ergebnisliste

18 Socken

16 Unterhose

15 Hose

14 Schuhe

10 Uhr

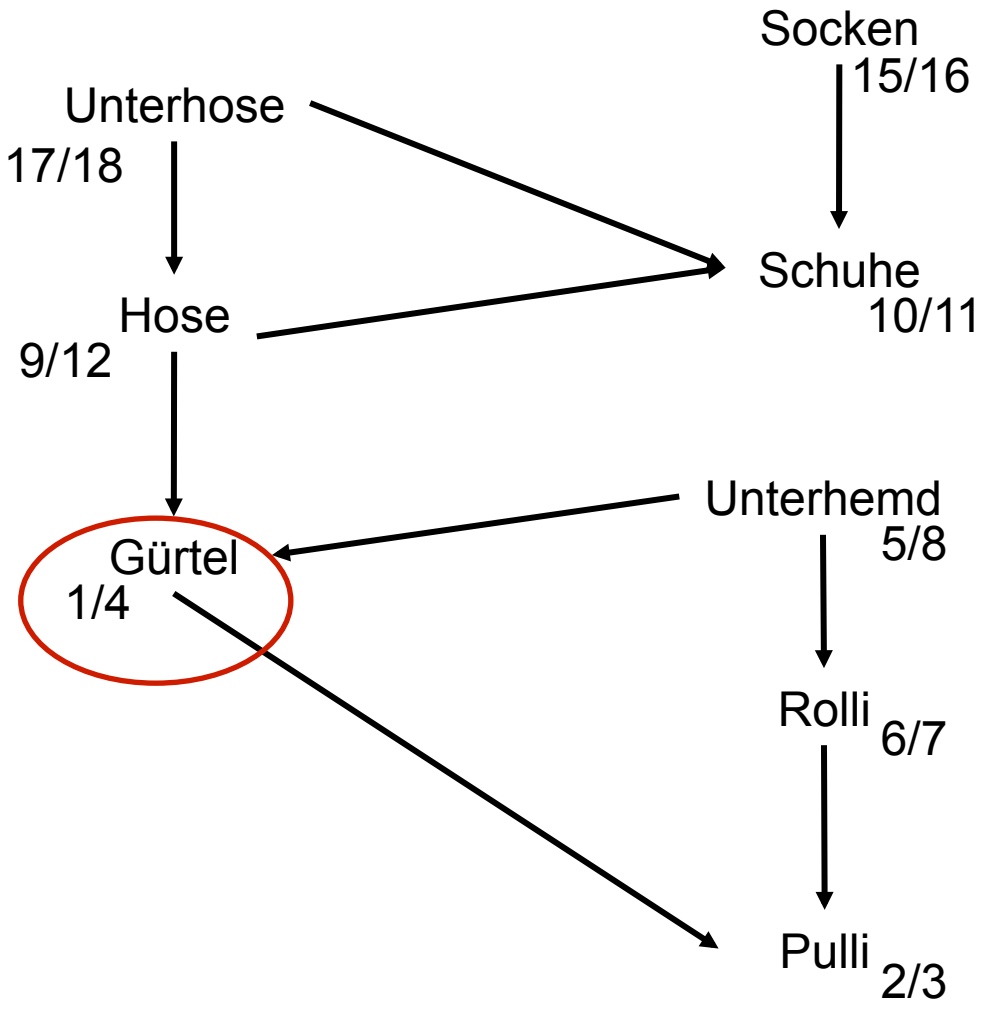
8 Unterhemd

7 Gürtel

5 Rolli

4 Pulli

Topologisches Sortieren: Alternativer Durchlauf



Wie sieht hier die Ergebnisliste aus?

Algorithmen und Datenstrukturen

9.4 TOPOLOGISCHES SORTIEREN

ZUSAMMENFASSUNG

Zusammenfassung

- Konstruktion einer totalen Ordnung aus einer Halbordnung