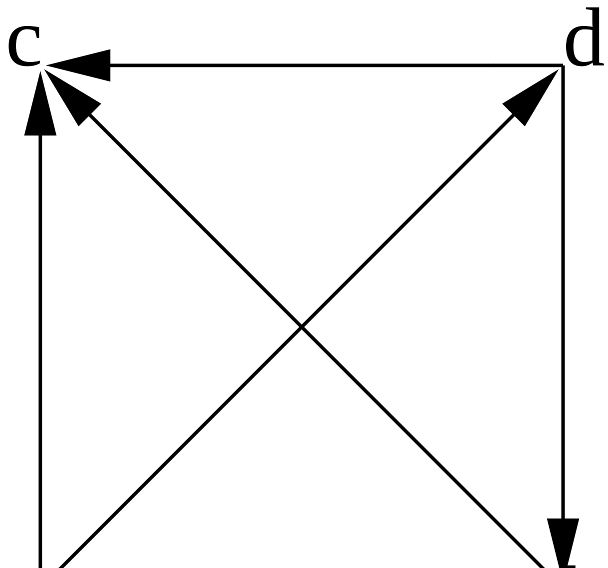
Graphen

- räumliche Beziehungen – Weg von Knoten a nach Knoten b
- zeitliche Beziehungen – Knoten a *und dann* Knoten b
- kausale Beziehungen – Knoten a *verursacht* Knoten b

2-stellige Relation:
{(a,b), (a,c), (a,d), (b,c), (d,b), (d,c)}



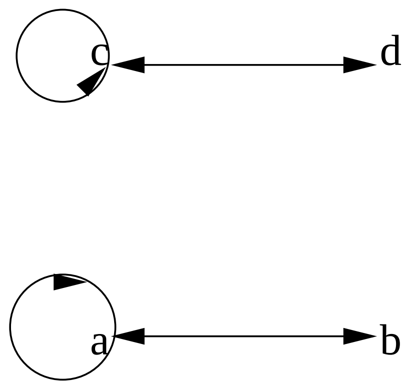
Gerichteter Graph

Gegeben sei eine Grundmenge von Knoten V (engl. vertex, vertices). Ein gerichteter Graph $G = (V, E)$ besteht aus V und Kanten $E \subseteq V \times V$ (engl. edge(s)). Ist V endlich, so ist auch der Graph G endlich.

$V = \{a, b, c, d\}$, so ist $V \times V$:

$\{$(a,a), (a,b), (a,c), (a,d), (b,a), (b,b), (b,c), (b,d), (c,a), (c,b), (c,c), (c,d), (d,a), (d,b), (d,c), (d,d)$\}$. Aus diesem kartesischen Produkt greift E eine Teilmenge heraus, z.B. die oben angeführte. Unsere Definition umfaßt aber auch etwa diese Teilmenge:

$\{$(a,b), (b,a), (a,a), (c,d), (d,c), (c,c)$\}$.



Zusammenhang

Ein gerichteter Graph heißt **stark zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist. Erreichbar ist ein Knoten v_i von einem anderen Knoten v_1 aus, wenn es eine Kante (v_1, v_i), zwei Kanten $(v_1, v_2), (v_2, v_i)$ oder eine Folge von Kanten gibt $(v_1, v_2), (v_2, v_3), ..., (v_{i-1}, v_i)$.

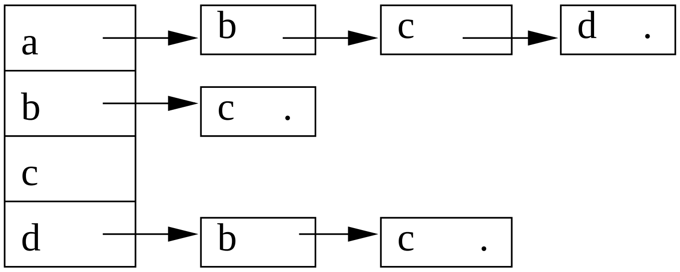
Ein gerichteter Graph heißt **schwach zusammenhängend**, wenn es zwischen zwei Knoten immer einen *Semiweg* gibt. Ein Semiweg zwischen zwei Knoten ist eine Folge von Kanten, die die Knoten verbindet, wobei man von der Richtung der Kanten absehen darf.

Ein Teilgraph heißt **Zusammenhangskomponente**, wenn er bezüglich der Zusammenhangseigenschaft (stark oder schwach) maximal ist, d.h. der Teilgraph kann nicht durch einen weiteren Knoten einer eine weitere Kante des Graphen erweitert werden, ohne eben diese Eigenschaft zu verlieren.

Darstellung von Graphen

Adjacent (engl) – benachbart (deutsch)

Adjazenzliste – Liste von Nachfolgern



Adjazenzmatrix – Nachfolgermatrix

Nachfolger

		a	b	c	d
	a	0	1	1	1
Knoten	b	0	0	1	0
	c	0	0	0	0
	d	0	1	1	0

Realisierung

Vertex mit den Eigenschaften eines Knoten:

- contents: Inhalt des Knoten – ein Objekt;
- successors: Nachfolger – eine verkettete Liste;
- alreadyVisited: Markierung, ob der Knoten schon besucht wurde.

und den Methoden:

- **Vertex(Object c)** wobei einem Knoten die Referenz auf irgendein Objekt zugewiesen wird, eine verkettete Liste für die Nachfolger erzeugt wird und die Markierung, ob der Knoten schon besucht wurde, mit `false` initialisiert wird.
- die Methoden **getContents**, **getSuccessors** und **visited** liefern die Ausprägung der jeweiligen Eigenschaft eines Knotens zurück.
- **addSuccessor** trägt in die verkettete Liste der Nachfolger eines Knotens einen neuen Knoten ein.

Dazu wid die Methode der verketteten Listen **add** aufgerufen.

- **setVisited** weist der Eigenschaft $alreadyVisited$ einen neuen Wert durch die Parameterübergabe *Wertübergabe* zu.

- **toString** gehört zum guten Ton einer JAVA-Klasse!

Problemlösung durch Suchen

- depthFirstSearch: Tiefensuche von einem bestimmten Knoten aus, wobei jeder Knoten, der bei der Tiefensuche besucht wird, markiert wird mit $alreadyVisited$ =`true`. Auf diese Weise wird ein Zyklus erkannt, wenn bei der Verfolgung der Nachfolger ein bereits besuchter Knoten als Nachfolger vorkommt.

- breadthFirstSearch: Breitensuche von einem bestimmten Knoten aus.

in JAVA

```
package LS8Tools;

import java.util.LinkedList;
import java.util.ListIterator;
import AlgoTools.IO;
public class Vertex {
    protected Object contents;           // Inhalt des Knotens
    protected LinkedList successors;     // Liste der Nachfolger
    protected boolean alreadyVisited;    // Markierung, ob der
                                         // Knoten schon besucht wurde
    private int value;                   // Wert des Knotens fuer die
        // Berechnung der stark verbundenen Komponenten

    public Vertex (Object _contents) {
        contents = _contents;
        successors = new LinkedList ();
        alreadyVisited = false;
        value = 0;
    }
```

```
public Object getContents () { return (contents); }
public LinkedList getSuccessors () {
    return (successors); }
public boolean visited () { return (alreadyVisited); }
public void setVisited (boolean visited) {
    alreadyVisited = visited; }
public int getValue () { return (value); }
public void setValue (int _value) { value = _value; }

public void addSuccessor (Vertex v) {
    successors.add (v); }

public String toString () {
    String s;                  // Ausgabezeichenkette
    ListIterator i;            // Iterator fuer die Nachfolger
    Vertex v;    // aktuell betrachteter Nachfolgerknoten
    s = new String (contents.toString () + ": ");
    i = successors.listIterator ();
    while (i.hasNext ()) {
        v = (Vertex) i.next ();
        s += (v.getContents ()).toString();
        if (i.hasNext ()) s += ", ";
    }
    return (s);
}
```

```
public Vertex depthFirstSearch (Object vertexContents) {
                              //Tiefensuche mit Zyklenerkennung;
        ListIterator i; // Iterator fuer den aktuellen Nachfolger
        Vertex v;     // aktuell betrachteter Nachfolgerknoten
        Vertex r;       // in tieferer Ebene gefundener Knoten
        this.setVisited (true);          //als besucht markieren
        if (this.contents.equals (vertexContents)) {
            return (this);
        }

        i = (this.successors).listIterator();
        while (i.hasNext ()) {
            v = (Vertex) i.next ();
            if (!(v.visited ())) {
                r = v.depthFirstSearch (vertexContents);
                if (r != null) return (r);
            }
        }
        return (null);                          // Suche erfolglos
    }
```

```
public Vertex breadthFirstSearch (Object vertexContents) {

                          //Breitensuche mit Zyklenerkennung
    ListIterator i;    // Iterator fuer den aktuellen Knoten
    Vertex v;                 // aktuell betrachteter Knoten
    Vertex successor;         // aktueller Nachfolgerknoten
    Schlange openVertices          // besuchte Knoten mit
        = new Schlange (100);               // Nachfolgern
    openVertices.enq (this );
    this.setVisited (true);

    while (!(openVertices.empty ())) {
        v = (Vertex) openVertices.front ();
        openVertices.deq ();

        if (vertexContents.equals (v.getContents())) {
            return (v);
        }
        i = (v.getSuccessors()).listIterator();
        while (i.hasNext()) {
            successor = (Vertex) i.next();
            if (!(successor.visited())) {
                successor.setVisited (true);
                openVertices.enq (successor);
          }}}
    return (null);
}}
```

Klasse Graph

```
import java.util.LinkedList;            // ADT verzeigerte Liste
import java.util.ListIterator;                        // Iterator
import java.util.Stack;                 // ADT Keller (= Stack)
import Vertex;   // Klasse fuer die Knoten in einem Graphen
import AlgoTools.IO;    // Vornbergers AlgoTools-I/O-Klasse

public class Graph {
    private LinkedList vertices;               // Liste aller Knoten
    private String name;                       // Name des Graphen
    public Graph (String _name) {
        name = _name;
        vertices = new LinkedList ();
    }
    public String getName () {  return (name);  }
    public LinkedList getVertices() {
        return (vertices);  }
    public void addVertex (Vertex v) {
        vertices.add (v);  }
    public void addEdge (Vertex v1, Vertex v2) {
        v1.addSuccessor (v2);  }
```

```
public String toString () {
    String s;                  // Ausgabezeichenkette
    ListIterator i;            // Iterator f. aktuellen Knoten
    Vertex v;                  // aktuell betrachteter Nachfolger
    s = new String (name + ":\n");
    i = vertices.listIterator ();
    while (i.hasNext ()) {
        v = (Vertex) i.next ();
        s += ""+ v.toString () + "\n";
    }
    return (s);
}

public void markAllVerticesAsNotVisited () {
    ListIterator i;     // Iterator fuer den aktuellen Knoten
    Vertex v;           // aktuell betrachteter Nachfolgerknoten
    i = vertices.listIterator ();
    while (i.hasNext ()) {
        v = (Vertex) i.next ();
        v.setVisited (false);
        v.setValue (0);
    }
}
```

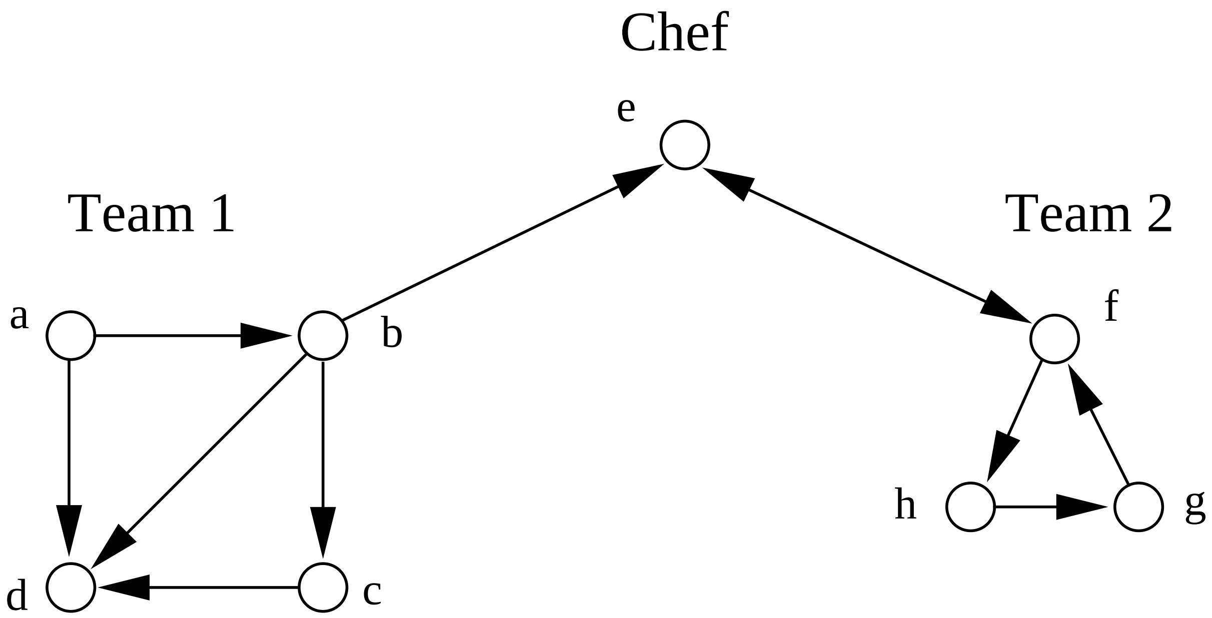
```
public Vertex depthFirstSearch (Object vertexContents)  {
        //Tiefensuche mit Zyklenerkennung (Top Level);
    ListIterator i;    // Iterator fuer den aktuellen Knoten
    Vertex v;                 // aktuell betrachteter Knoten
    Vertex r;                  // in tieferer Ebene gefunden
    markAllVerticesAsNotVisited ();
    i = vertices.listIterator ();
    while (i.hasNext ())  {
        v = (Vertex) i.next ();
        if (!(v.visited ()))  {
            r = v.depthFirstSearch (vertexContents);
            if (r != null) return (r);
        }
    }
    return (null);
}
```

```
private int depthFirstScan (Vertex startVertex,
            int vertexNumber, Stack vertexStack) {
    ListIterator i; // Iterator fuer den Nachfolger
    Vertex v; // aktuell betrachteter Nachfolger
    int min; // minimaler Wert eines besuchten Knotens
    int m; // Wert des aktuell besuchten Nachfolgers
    startVertex.setVisited (true); // als besucht markieren
    vertexNumber++; // Zaehler besuchter Knoten +1
    startVertex.setValue (vertexNumber);
    min = vertexNumber;
    vertexStack.push (startVertex);

    i = (startVertex.getSuccessors ()).listIterator();
    while (i.hasNext ()) {
        v = (Vertex) i.next ();
        if (!(v.visited ()))
        m = depthFirstScan (v, vertexNumber,
vertexStack);
        else
        m = v.getValue ();
        if (m < min) min = m;
    }

    if (min == startVertex.getValue ()) {
        IO.print ("Stark verbundene Komponente: ");
        do {
            v = (Vertex) vertexStack.pop ();
            IO.print ((v.getContents ()).toString() + "");
```

```
            v.setValue (vertices.size () + 1);
        } while (v != startVertex);
        IO.println ();
    }

    return (min);
}


public void printStrongComponents () {
    Stack vertexStack // Stack besuchter Knoten
    = new Stack ();
    ListIterator i; // Iterator fuer den Nachfolger
    Vertex v; // aktuell betrachteter Nachfolgerknoten
    markAllVerticesAsNotVisited ();
    // markiert alle Knoten als unbesucht und
    // setzt alle Knotenwerte (value) auf Null
    i = vertices.listIterator ();
    while (i.hasNext ()) {
        v = (Vertex) i.next ();
        if (!(v.visited ()))
                depthFirstScan (v, 0, vertexStack);
    }
}
```

```
public static void main (String[] args)  {
    // Testprogramm fuer die Klasse
    Vertex v1, v2; // 'Graph':
    Graph g1;

    v1 = new Vertex ("A");
    v2 = new Vertex ("B");
    v3 = new Vertex ("C");
    v4 = new Vertex ("D");
    g1 = new Graph ("G1");
    g1.addVertex (v1);
    g1.addVertex (v2);
    ...
    g1.addEdge (v1, v2);
    ...
    IO.println (g1.toString ());
    v1 = g1.depthFirstSearch ("C");
    IO.println ("C gefunden in "+ v1.toString ());
  }

}
```

Witze erzählen



in JAVA

```
import AlgoTools.IO;
import LS8Tools.Graph;
import LS8Tools.Vertex;

public class WitzFluss {

    public static void main (String[] argv) {
        Graph firma = new Graph ("Big money, much fun");

        Vertex a, b, c, d, e, f, g, h, chef;

        a = new Vertex ("a");                 // Erzeuge Team 1
        b = new Vertex ("b");
        c = new Vertex ("c");
        d = new Vertex ("d");

        firma.addVertex (a); firma.addVertex (b);          // In
Graphen aufnehmen
        firma.addVertex (c); firma.addVertex (d);

        firma.addEdge (a, b); firma.addEdge (a, d);          //
Verbindungen knuepfen
        firma.addEdge (b, c); firma.addEdge (b, d);
        firma.addEdge (c, d);
```

```
f = new Vertex ("f");                    // Erzeuge Team 2
g = new Vertex ("g");
h = new Vertex ("h");

firma.addVertex (f); firma.addVertex (g);          // In
Graphen aufnehmen
firma.addVertex (h);

firma.addEdge (f, h); firma.addEdge (h, g);          //
Verbindungen knuepfen
firma.addEdge (g, f);

e = new Vertex ("Chef"); // Der Chef ist auch dabei
chef = e;
firma.addVertex (chef);

firma.addEdge (chef, f); firma.addEdge (f, chef);
firma.addEdge (b, chef);

IO.println ("Hoert A Witze ? ");
if (firma.breadthFirstSearch (b, "a") != null)
IO.println ("A lacht auch ueber andere Witze.");
else
IO.println ("A muss ueber eigene Witze lachen.");

IO.println ("Hoert Team 1 Witze vom Chef bzw.
Team 2 ?");
if (firma.depthFirstSearch (chef, "d") == d)
```

```
        IO.println ("Team 1 lacht ueber den Chef.");
        else
        IO.println ("Team 1 lacht nur ueber eigene Witze.");

        IO.println ("Weiss der Chef, wie witzig 'c' ist ?");
        if (firma.breadthFirstSearch (c, "Chef") == null)
        IO.println ("Der Chef weiss nichts ueber c's
Witzigkeit.");
        else
        IO.println ("Der Chef schaetz 'c' wegen seinem
Humor.");

    }

}
```