Graphen und Algorithmen

Vorlesung #1: Grundlagen

Dr. Armin Fügenschuh
Technische Universität Darmstadt
WS 2007/2008

Übersicht

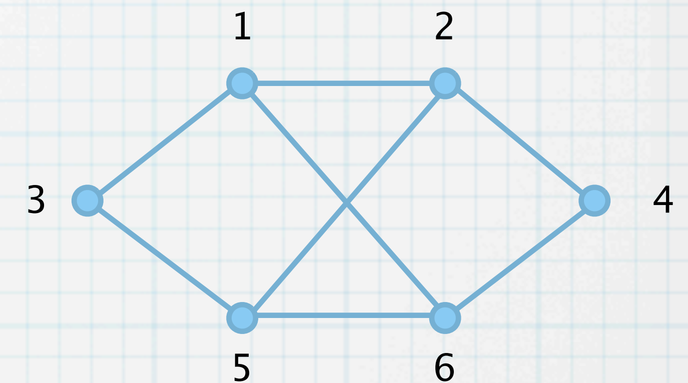
* Graphen: Grundbegriffe und -bezeichnungen
* Planare Graphen
* Darstellung von Graphen (im Rechner)
* Algorithmen und deren Komplexität
* Tiefen- und Breitensuche
* Topologisches Sortieren
* Test-Algorithmus für stark zusammenhänge Graphen

Grundbegriffe und Bezeichnungen

- **Definition 1**:
  Sei V eine endliche, nicht-leere Menge und E eine Mengen von zweielementigen Teilmengen von V. Dann heißt das Tupel $G = (V, E)$ **ungerichteter Graph** (oder auch nur **Graph**). Die Elemente von V heißen **Knoten**, **Ecken** oder **Punkte**, die Elemente von E **Kanten**. Für die Knoten- bzw. Kantenmenge eines Graphen G schreiben wir auch $V(G)$ bzw. $E(G)$.
- Beispiel:

$$V := \{1, 2, 3, 4, 5, 6\}$$

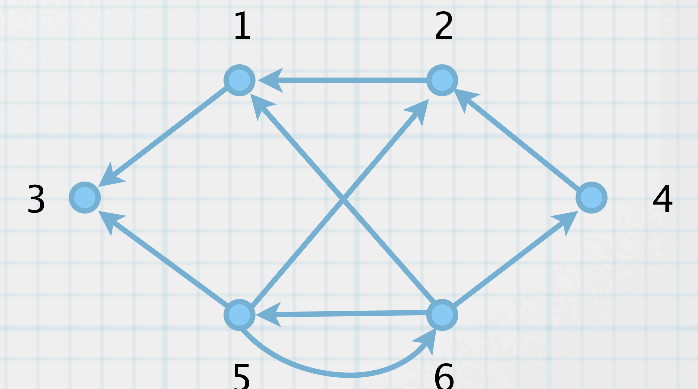
$$E := \{\{1, 2\}, \{1, 3\}, \{1, 6\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 6\}, \{5, 6\}\}$$



- **Definition 2**:
  Sei V eine endliche, nicht-leere Menge und $A \subseteq \{(i, j) \in V \times V : i \neq j\}$ eine (endliche) Menge. Dann heißt das Tupel $D = (V, A)$ **gerichteter Graph** oder auch **Digraph**. Die Elemente von V heißen wiederum **Knoten**, die Elemente von A **Bögen**. Für die Bogenmenge eines Digraphen schreiben wir auch $A(G)$.
- Beispiel:

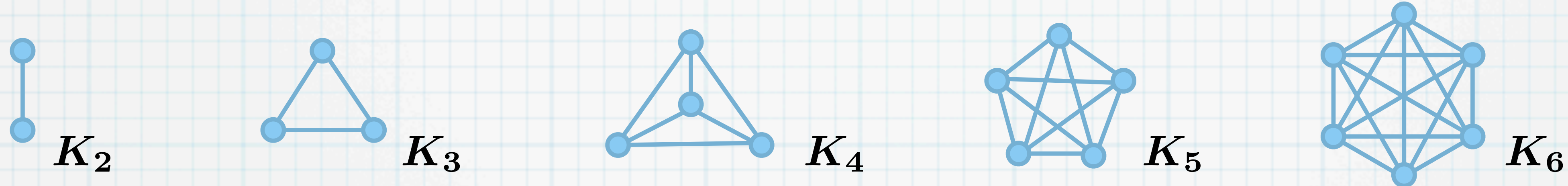
$$V := \{1, 2, 3, 4, 5, 6\}$$

$$A := \{(2, 1), (1, 3), (6, 1), (4, 2), (5, 2), (5, 3), (6, 4), (5, 6), (6, 5)\}$$

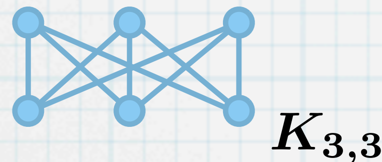


Vollständige und bipartite Graphen

- Beispiel & **Definition 3**:
  Der **vollständige Graph** K_n hat als Kanten alle 2-elementigen Teilmengen von $V := \{1, \ldots, n\}$.

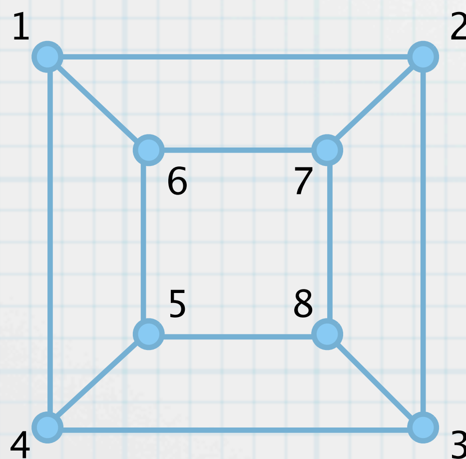


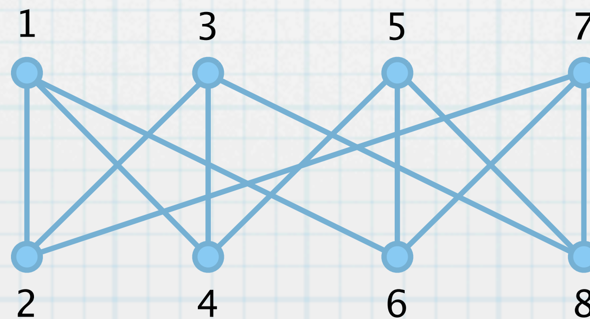
Der **vollständige bipartite Graph** $K_{m,n}$ hat als Knotenmenge die disjunkte Vereinigung der m-elementigen Menge V_1 und der n-elementigen Menge V_2. Kanten sind genau die $\{i, j\}$ mit $i \in V_1$ und $j \in V_2$.



$K_{3,3}$

Allgemein ist ein **bipartiter Graph** (auch **paarer Graph**) ein Graph $G = (V, E)$, dessen Knotenmenge V in zwei nicht-leere Untermengen X, Y geteilt werden kann, d.h. $X \cup Y = V$ und $X \cap Y = \emptyset$, so dass jede Kante e von der Form $e = \{x, y\}$ mit $x \in X$ und $y \in Y$ ist. Die Zerlegung $X \cup Y = V$ heißt **Zweiteilung** oder **Bipartition**.



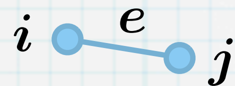


$X = \{1, 3, 5, 7\}$

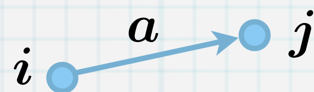
$Y = \{2, 4, 6, 8\}$

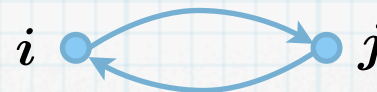
Grundbegriffe und Bezeichnungen (2)

* **Definition 4**:
Für eine Kante $e = \{i, j\}$ eines Graphen heißen i, j **Endpunkte** von e. Man sagt, i, j sind mit e **inzident** und i, j sind **adjazent** oder auch i, j sind **Nachbarn**.



Für einen Bogen $a = (i, j)$ eines Digraphen heißt i **Anfangsknoten** und j **Endknoten**. i, j sind mit a inzident. i ist **Vorgängerknoten** von j und j ist **Nachfolgerknoten** von i. Die Kanten (i, j) und (j, i) heißen **gegenläufig** oder **antiparallel**.





* **Definition 5**:
Für einen Knoten i eines Graphen ist der **Grad** $\deg(i)$ definiert als die Anzahl der mit i inzidenten Kanten.

Für einen Knoten i eines Digraphen definieren wir den **Ausgangsgrad** $\deg_{out}(i)$ als die Anzahl der Kanten, die mit i inzident sind und i als Anfangsknoten haben, sowie den **Eingangsgrad** $\deg_{in}(i)$ als die Anzahl der Kanten, die mit i inzident sind und i als Endknoten haben. Der **Grad** des Knotens ist dann definiert als $\deg(i) := \deg_{in}(i) + \deg_{out}(i)$.

* **Lemma 6**:
Die Anzahl der Knoten ungeraden Grades in einem (Di-) Graphen ist gerade.

* Beweis:

Summiert man $\deg(i)$ über alle Knoten, so kommt jede Kante genau zwei Mal vor.

Damit gilt: $\sum_{i \in V} \deg(i) = 2|E|$.

Rechts steht eine gerade Zahl, also ist links die Anzahl ungerader Summanden gerade.

Grundbegriffe und Bezeichnungen (3)

* **Definition 7**:
  Gilt $\deg(i) = \deg(j)$ für alle Knotenpaare i, j, so heißt der Graph **regulär**.
  Gilt $\deg(i) = k$ für alle Knoten i, so heißt der Graph k–**regulär**.
* Bemerkung: Für manche Anwendungen sind **Mehrfachkanten** bzw. **–bögen** sowie **Schleifen** (auch **Schlinge** oder **Schlaufe** genannt) relevant. In dieser Vorlesung spielen sie jedoch keine Rolle. Unsere Definitionen 1 und 2 schließen sie sogar explizit aus. Graphen ohne Mehrfachkanten und Schleifen werden auch **einfache Graphen** oder **schlichte Graphen** genannt.

* **Definition 8**:
  Seien G und T Graphen. T ist **Teilgraph** (**Untergraph**, **Subgraph**) von G, wenn $V(T) \subseteq V(G)$ und $E(T) \subseteq E(G)$. In diesem Fall ist G Obergraph von T.
  T ist **induzierter Teilgraph** von G, wenn zudem gilt $E(T) = \{\{i, j\} \in E(G) : i, j \in V(T)\}$.
* Beispiel: Teilgraph (links) und induzierter Teilgraph (rechts)

Wege und Kreise in Graphen

Definition 9:
Sei $(e_1, \ldots, e_n)$ eine Sequenz von Kanten in einem Graphen G. Wenn es Knoten $v_0, \ldots, v_n$ gibt mit $e_i = \{v_{i-1}, v_i\}$ für alle $i = 1, \ldots, n$, so heißt die Sequenz **Kantenzug**.

Im Falle von $v_0 = v_n$ spricht man von einem **geschlossenen Kantenzug**.

Sind die e_i paarweise verschieden, liegt ein **Weg** (oder **Pfad**) vor.

Ein geschlossener Weg ist ein **Kreis**.

Ein Weg ist **einfach**, wenn die v_i paarweise verschieden sind.

Ein Kreis ist **einfach**, wenn die v_i paarweise verschieden sind (mit Ausnahme von $v_0 = v_n$).

Ein **Zyklus** ist ein einfacher Kreis.

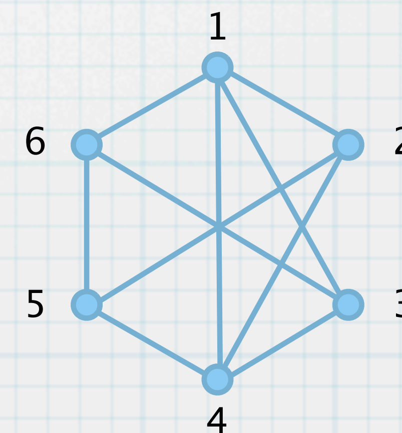
n wird als **Länge** des Kantenzugs, des Weges oder des Kreises bezeichnet.

Ein (**un-**)**gerader Kreis** ist ein Kreis, dessen Länge n eine (un-)gerade Zahl ist.

Die Knoten v_0, v_n heißen **Anfangs-** bzw. **Endknoten**.

Ein Graph, der keine Kreise enthält, ist **kreisfrei**.

* Bemerkung: In einfachen Graphen ist jede Kantensequenz eineindeutig abbildbar auf eine entsprechende Knotensequenz (mit mind. zwei Knoten).
* Beispiele:
  (a) (6,1,2,4,1,2): Kantenzug, aber kein Weg
  (b) (6,1,2,4,1,3): nicht-einfacher Weg
  (c) (6,1,2,4,1,3,6): nicht-einfacher Kreis
  (d) (6,1,2,5,4,3,6): einfacher Kreis



Wege und Kreise in Digraphen

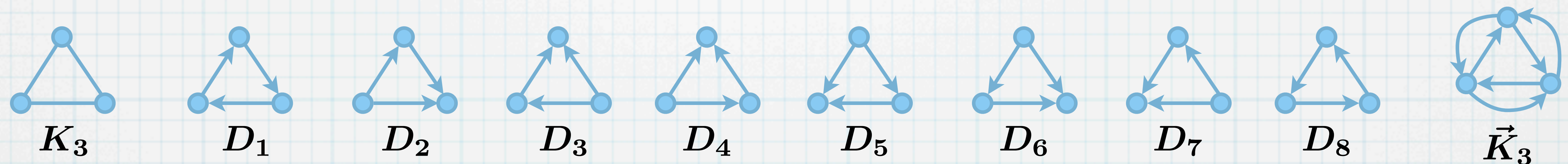
* **Definition 10**:
Sei $D = (V, A)$ ein Digraph. Setzt man $E := \{\{i, j\} : (i, j) \in A \vee (j, i) \in A\}$, so bezeichnet man $|D| := (V, E)$ als den **zugehörigen Graphen**.

  Sei $G = (V, E)$ ein Graph. Jeder Digraph D mit $|D| = G$ wird **Orientierung** von G genannt.

  Setzt man $A := \{(i, j), (j, i) : \{i, j\} \in E\}$, so ist $\vec{G} := (V, A)$ der **zugehörige Digraph**, auch **vollständige Orientierung** von G genannt.

  Die vollständige Orientierung des K_n wird **vollständiger Digraph** auf n Knoten genannt.

* Beispiel: Die acht Orientierungen des K_3 und seine vollständige Orientierung



* **Definition 11**:
Eine Sequenz von Bögen $(a_1, \ldots, a_n)$ in einem Digraphen $D = (V, A)$ heißt **Kantenzug**, (**einfacher**) **Weg** oder (**einfacher**) **Kreis**, wenn die entsprechende Sequenz von Kanten in $|D|$ die jeweilige Eigenschaft hat.

  Ist $(v_0, v_1, \ldots, v_n)$ die zugehörige Punktfolge, so ist entweder $a_i = (v_{i-1}, v_i)$ oder $a_i = (v_i, v_{i-1})$. Im ersten Fall spricht man von einer **Vorwärtskante**, im zweiten von einer **Rückwärtskante**.

  Sind alle Kanten des Weges nur Vorwärts- bzw. nur Rückwärtskanten, so ist es ein **vorwärts-** bzw. **rückwärtsgerichteter Kantenzug** (oder **Weg**) bzw. **gerichteter Kreis**.

* Beispiel: Die Kantenzüge in $D_2, D_3, D_4, D_5, D_6, D_7$ sind einfache Kreise, die Kantenzüge in D_1, D_8 sind einfache gerichtete Kreise.

Zusammenhangsbegriffe in Graphen und Digraphen

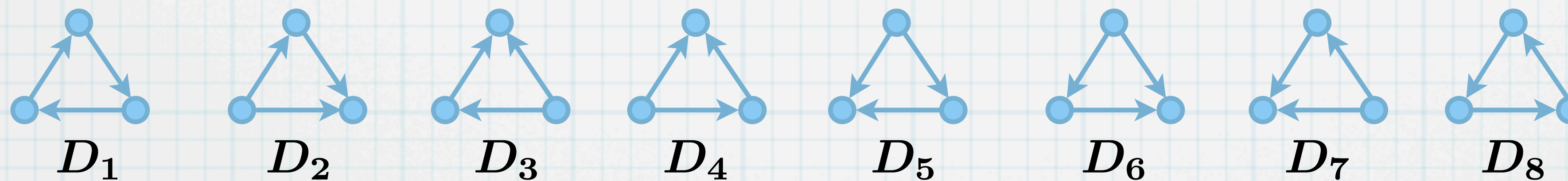
* **Definition 12**:
  Ein Knoten i eines Graphen heißt **verbindbar** mit einem Knoten j, wenn es einen Weg im Graphen gibt, der i, j als Endknoten hat.

  Ein Graph ist **zusammenhängend**, wenn je zwei seiner Knoten verbindbar sind.

  Für einen Knoten i eines Graphen bezeichne $C(i)$ die Menge aller Knoten, die mit i verbindbar sind. Dann heißt der durch $C(i)$ induzierte Untergraph die **Zusammenhangskomponente** von i.
* **Definition 13** (Starker Zusammenhang):
  Zwei Knoten i, j eines Digraphen heißen **verbindbar**, wenn es einen vorwärtsgerichteten Weg im Digraphen mit Anfangsknoten i und Endknoten j gibt.

  Ein Digraph D heißt (**schwach**) **zusammenhängend**, wenn Graph $|D|$ zusammenhängend ist.

  Ein Digraph heißt **stark zusammenhängend**, wenn je zwei seiner Knoten verbindbar sind.
* Beispiel: Die Digraphen $D_2, D_3, D_4, D_5, D_6, D_7$ sind schwach zusammenhängend, D_1, D_8 sind stark zusammenhängend.

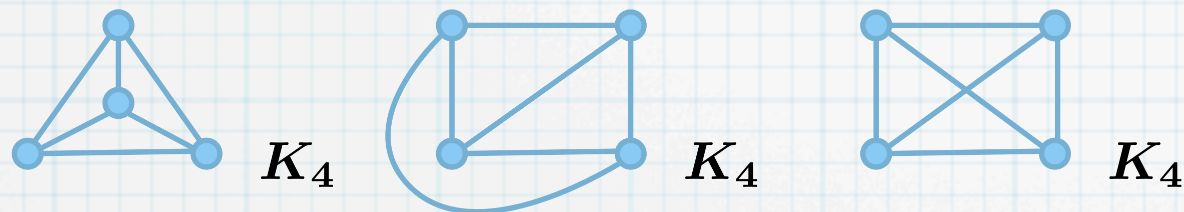


Charakterisierung bipartiter Graphen mittels Kreisen

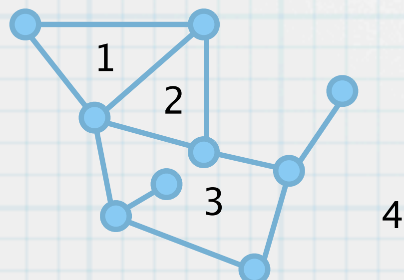
* **Satz 14** (König, 1916):
  Ein Graph ist genau dann bipartit, wenn er keinen ungeraden Kreis enthält.
* Beweis:
  ($\Rightarrow$): Sei $G = (V, E)$ bipartit mit $X \cup Y = V$. Sei $(v_0, v_1, \ldots, v_n, v_0)$ ein Kreis. O.B.d.A. $v_0 \in X$.
  Dann ist $v_1 \in Y, v_2 \in X$, usw.: Knoten mit geradem Index sind $\in X$, ungerade $\in Y$.
  $v_n \in Y$, weil Nachfolger $v_0 \in X$.
  Also ist n ungerade.
  ($\Leftarrow$): Sei $G = (V, E)$ ein (o.B.d.A. zusammenhängender) Graph ohne ungerade Kreise.
  Sei $u \in V$. Sei $X \subset V$ die Menge aller Knoten v, so dass der kürzeste Weg von u nach v gerade Länge hat. Sei $Y \subset V$ die Menge aller Knoten v, deren kürzester Weg von u nach v ungerade Länge hat. Klar: $u \in X$.
  Angenommen, es gibt adjazente Knoten $v, w \in X$.
  Betrachte kürzesten u–v–Weg $P := (u_1, \ldots, u_{2n+1})$ und kürzesten u–w–Weg $Q := (w_1, \ldots, w_{2m+1})$, wobei $u = u_1 = w_1, v = u_{2n+1}$ und $w = w_{2m+1}$. (Da $u, v, w \in X$ sind die Wege gerade, haben also ungerade viele Knoten.)
  Sei w' der letzte gemeinsame Knoten. (Ein solcher existiert, da zumindest $u \in P, Q$).
  Dann ist der Teil von P von u_1 bis w' ein kürzester u–w'–Weg und der Teil von Q von w_1 bis w'ebenfalls ein kürzester u–w'–Weg.
  Also haben beide Wege die gleiche Länge. Es gibt einen Index i mit $w' = u_i = w_i$.
  Betrachte die Knotensequenz $(u_i, u_{i+1}, \ldots, u_{2n+1}, w_{2m+1}, w_{2m}, \ldots, w_i)$.
  Diese ist ein Kreis, da $u_i = w_i = w'$ und $v = u_{2n+1}, w = w_{2m+1}$ adjazent.
  Länge des Kreises ist ungerade (Fallunterscheidung: i gerade oder ungerade, Kantenzahl jedenfalls ungerade), im Widerspruch zur Annahme.

Planare und ebene Graphen

- **Definition 15**:
  Ein **planarer Graph** ist ein Graph, der in der Ebene (auf dem Papier, der Tafel, usw.) so gezeichnet werden kann, dass sich jedes Paar von Kanten entweder nur an seinen Endknoten trifft oder gar nicht. Wird ein planarer Graph auf diese Weise gezeichnet, spricht man von einem **ebenen Graphen**.
- Beispiel:

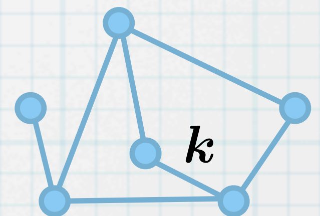


- **Definition 16**:
  Ein ebener Graph G teilt die Ebene in endlich viele Gebiete auf, die als **Flächen** von G bezeichnet werden. Die Flächen, die durch einen Kreis in G begrenzt werden, heißen **innere Flächen** von G. Die einzige Fläche, die nicht durch einen Kreis begrenzt wird, wird **äußere Fläche** von G genannt.
- Beispiel:

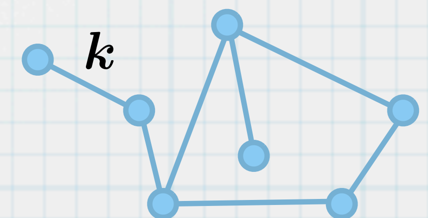


Eulers Formel

* **Satz 17** (Euler):
  Sei G ein zusammenhängender ebener Graph. Seien n, e, f die Anzahl der Knoten, Kanten bzw. Flächen von G. Dann gilt $n - e + f = 2$.
* Beweis (Induktion über die Anzahl der Kanten von G):
  Induktionsanfang, $e = 0$. Da G zusammenhängend, gibt es genau einen Knoten, $n = 1$.
  Damit gibt es eine Fläche, die äußere Fläche, $f = 1$.
  Es gilt also $n - e + f = 1 - 0 + 1 = 2$.
  Behauptung: Formel gilt für zusammenhängende planare Graphen G mit $e - 1$ Kanten.
  Induktionsschritt: Füge weitere Kante k zu G hinzu, so dass der neue Graph G' planar ist.
  Seien n', e', f' die Anzahl der Knoten, Kanten bzw. Flächen von G'. Es ist $e' = e + 1$.
  1. Fall, k verbindet zwei Knoten von G.
     Dann wird eine Fläche von G in zwei Flächen aufgeteilt, $f' = f + 1$.
     Knotenzahl bleibt unverändert, $n' = n$.
     Also gilt $n' - e' + f' = n - (e + 1) + (f + 1) = n - e + f = 2$.



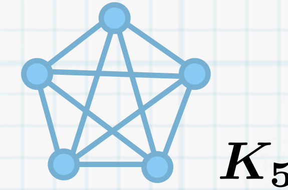
  2. Fall, k ist nur mit einem Knoten von G inzident.
     Dann muss ein weiterer Knoten hinzugefügt werden, $n' = n + 1$.
     Flächenzahl bleibt unverändert, $f' = f$.
     Also gilt $n' - e' + f' = (n + 1) - (e + 1) + f = n - e + f = 2$.



  (Bemerke, dass jeder zusammenhängende ebene Graph mit $e + 1$ Kanten aus einem geeigneten zusammenhängenden ebenen Graphen mit e Kanten durch Hinzufügen einer weiteren Kante und ggfs. eines weiteren Knoten erzeugt werden kann.)
* Folgerung: Jede ebene Zeichnung eines planaren Graphen hat gleich viele Flächen.

Schranken für planare Graphen

- **Satz 18**:
K_5 ist nicht planar.



- Beweis:
  - Angenommen, K_5 wäre planar.
  - K_5 hat $n = 5$ Knoten und $e = 10$ Kanten.
  - Nach Eulers Formel gibt es dann $f = 10 + 2 - 5 = 7$ Flächen.
  - Anderseits wird eine Fläche (in beliebigem Graphen) durch mindestens drei Kanten erzeugt.
  - Jede Kante grenzt dabei an höchstens zwei Flächen.
  - Um sieben Flächen abzugrenzen, braucht man daher $3 \cdot 7/2 = 10{,}5$ Kanten. Widerspruch.
- **Satz 19**:
Ein planarer Graph mit $n \geq 3$ Knoten hat höchstens $3n - 6$ Kanten und $2n - 4$ Flächen.
- Beweis:
  - Jede Fläche wird durch mindestens drei Kanten erzeugt.
  - Jede Kante grenzt an höchstens zwei Flächen.
  - Also ist die Anzahl Kanten $e \geq 3 \cdot f/2$.
  - Aus Eulers Formel folgt dann $e + 2 = n + f \leq n + 2/3e$, d.h. $e \leq 3n - 6$.
  - Ferner folgt aus Eulers Formel $n + f = e + 2 \geq 3 \cdot f/2 + 2$, d.h. $f \leq 2n - 4$.
- Anders gesagt: Die Anzahl der Kanten und Flächen in einem planaren Graphen wächst höchstens linear mit der Anzahl der Knoten.
- Folgerung: Ein Graph mit $n = 5$ kann höchstens $e = 9$ Kanten haben.

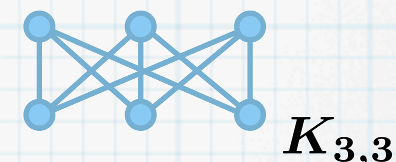
Knotengrad und Planarität

* **Satz 20**:
  Sei G ein planarer Graph. Dann hat G einen Knoten, dessen Grad kleiner als sechs ist.
* Beweis:
  Wenn G nur einen Knoten enthält, ist sein Grad Null.
  Wenn G aus zwei Knoten besteht, haben beide einen Grad von höchstens eins.
  Betrachte im Folgenden Graphen mit drei oder mehr Knoten.
  Angenommen, der Grad jedes Knotens ist mindestens sechs.
  Es gilt dann $\sum_{i \in V(G)} \deg(i) \geq 6n$.
  Es gilt jedoch auch (vgl. Lemma 6): $\sum_{i \in V(G)} \deg(i) = 2e$.
  Folglich $2e \geq 6n$, also $e \geq 3n$.
  Nach Satz 19 gilt jedoch $e \leq 3n - 6$. Widerspruch.
* **Satz 21**:
  In einem planaren Graphen ist der durchschnittliche Knotengrad kleiner als sechs.
* Beweis:
  Aus Satz 19 folgt für planare Graphen mit $n \geq 3$:

$$\frac{\sum_{i \in V} \deg(i)}{|V|} = \frac{2e}{n} \leq \frac{6n - 12}{n} < 6.$$

Schranken für planare Graphen (2)

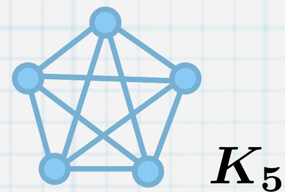
* **Satz 22**:
  $K_{3,3}$ ist nicht planar.
* Beweis:



  Angenommen, $K_{3,3}$ ist planar.
  $K_{3,3}$ hat $n = 6$ Knoten und $e = 9$ Kanten.
  Nach Eulers Formel daher $f = 9 + 2 - 6 = 5$ Flächen.
  Anderseits enthält der Graph keine Kreise der Länge drei.
  Also hat jede Fläche (mindestens) vier Kanten, jede Kante ist in (höchstens) zwei Flächen.
  Um fünf Flächen abzugrenzen, braucht es daher (mind.) $4 \cdot 5/2 = 10$ Kanten. Widerspruch.
* **Satz 23**:
  Ein planarer Graph mit n Knoten, der keine Kreise der Länge drei enthält, hat höchstens $2n - 4$ Kanten und $n - 2$ Flächen.
* Beweis:
  Jede Fläche wird durch mindestens vier Kanten erzeugt.
  Jede Kante grenzt an höchstens zwei Flächen.
  Also ist die Anzahl Kanten $e \geq 4 \cdot f/2 = 2 \cdot f$.
  Aus Eulers Formel folgt dann $e + 2 = n + f \leq n + 1/2e$, d.h. $e \leq 2n - 4$.
  Ferner folgt aus Eulers Formel $n + f = e + 2 \geq 2 \cdot f + 2$, d.h. $f \leq n - 2$.

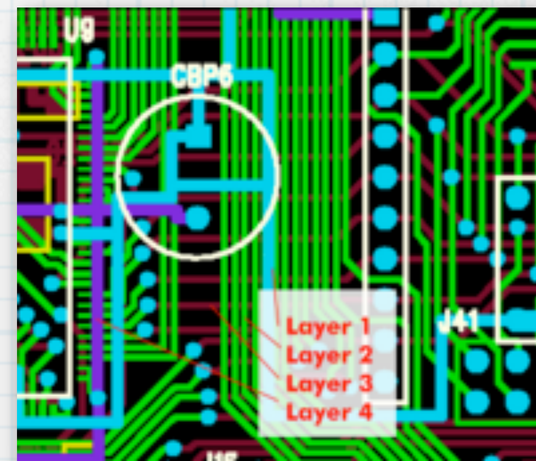
Dicke eines Graphen

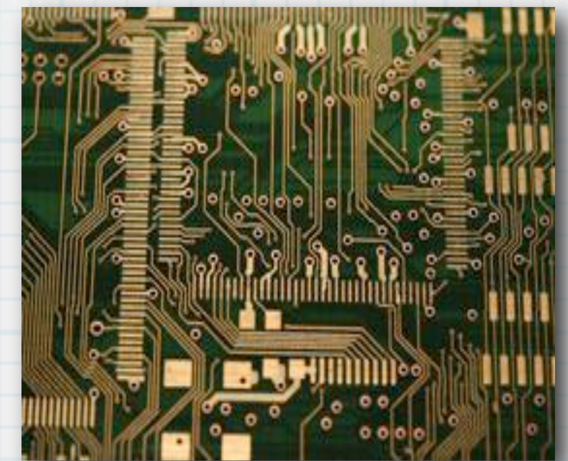
* **Definition 24**:
  Die Dicke t eines Graphen G ist definiert als die kleinste Anzahl von planaren Teilgraphen von G, die zusammen alle Kanten von G enthalten.
* Beispiel: $t(K_5) = 2$



K_5

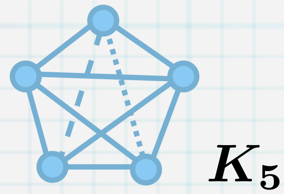
* **Satz 25**:
  Sei G ein Graph mit $n \geq 3$ Knoten und m Kanten. Dann gilt für die Dicke $t(G)$ folgende Abschätzung:
  $$t(G) \geq \max\left\{ \left\lceil \frac{m}{3n-6} \right\rceil, \left\lfloor \frac{m+3n-7}{3n-6} \right\rfloor \right\}.$$
* Beweis:
  Das linke Argument von „max“ folgt aus Satz 19.
  Das rechte Argument von „max“ folgt aus dem linken und $\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor$.
* Anwendung:
  * Schaltungsentwurf
  * Packen von Leitungen auf unterschiedliche Verdrahtungsebenen

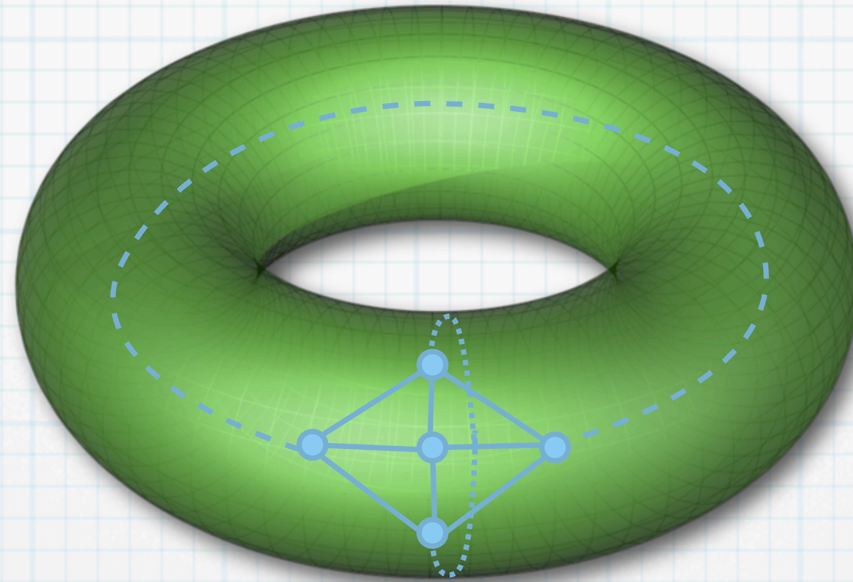




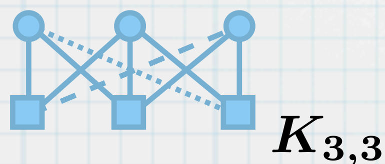
Graphen zeichnen auf anderen Oberflächen

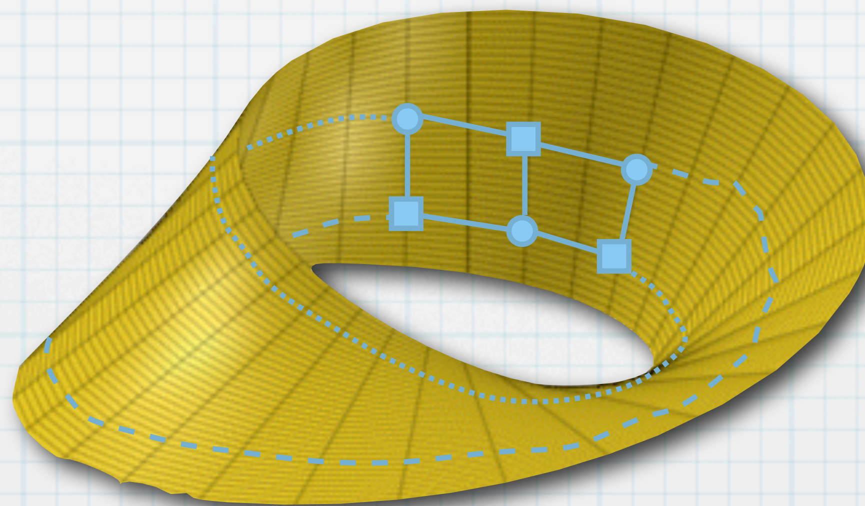
* Der Graph K_5 kann überschneidungsfrei (eben) auf einen Torus gezeichnet werden.





* Der Graph $K_{3,3}$ kann eben auf ein Möbiusband gezeichnet werden.





Algorithmen

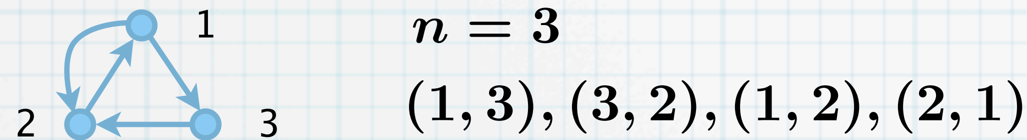
* Ein Algorithmus ist ein Verfahren zur Lösung von Problemen.
* Ein Problem besteht aus unendlich vielen Einzelfällen (Instanzen), die eine gemeinsame Struktur haben.
* Ein Algorithmus hat nach Bauer und Wössner (1982) folgende Eigenschaften:
  * **Endliche Beschreibbarkeit**: Das Verfahren kann mit einem endlichen Text beschrieben werden.
  * **Effektivität**: Jeder einzelne Schritt des Verfahrens muss mechanisch durchführbar sein.
  * **Endlichkeit**: Das Verfahren muss für jeden Einzelfall nach endlich vielen Schritten abbrechen.
  * **Determinismus**: Für jeden Einzelfall muss die Reihenfolge der Schritte eindeutig festliegen.
  * **Korrektheit**: Das Verfahren muss für jeden Einzelfall das Problem tatsächlich lösen.
* Daneben sollte ein Algorithmus möglichst **effizient** sein, d.h. mit wenig Bedarf an Ressourcen (Zeit, Platz) auskommen.
* Algorithmus ist von einem (Computer-) Programm zu unterscheiden.
* Ein Programm ist eine Abfolge von Befehlen in einer Programmiersprache, die zur Ausführung eines Algorithmus auf einem Rechner benötigt werden.

Ein Bipartit-Test-Algorithmus

* Eingabe: Graph $G = (V, E)$
* Ausgabe: ungerader Kreis, oder Meldung „Graph ist bipartit“
  (1) Untersuche für jede Teilmenge von Kanten, ob sie einen ungeraden Kreis bilden.
  (2) Falls ja, gibt den Kreis aus.
  (3) Falls keine solche Teilmenge gefunden wurde, gib Meldung „Graph ist bipartit“ aus.
* Aufgrund des Satzes von König ist dieser „Algorithmus“ korrekt.
* Trotzdem gibt es Bedenken:
  * Es ist nicht genau festgelegt, in welcher Weise die Teilmengen gebildet werden.
  * Da es sehr viele Teilmengen einer Obermenge gibt, ist dieser Algorithmus nicht effizient.

Darstellung von (Di-) Graphen als Kantenliste

- Kantenliste: Ein Digraph $D = (V, A)$ mit $V := \{1, \ldots, n\}$ wird spezifiert durch
  - die Angabe von n,
  - die Angabe der Liste seiner Kanten als Folge geordneter Paare $a_i = (u_i, v_i)$
- Beispiel:



$n = 3$

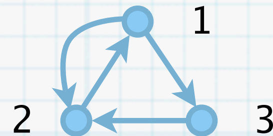
$(1, 3), (3, 2), (1, 2), (2, 1)$

- Mögliche Implementierung in C/C++:
  - ```
    int head[m]; // zwei getrennte Arrays für Head und Tail
    int tail[m]; // m ist Anzahl Kanten
    tail[0] = 1; head[0] = 2; // 0 ist erste Kante, m-1 ist letzte
    ```
  - ```
    struct Arc {
        int head;
        int tail;
    };
    Arc arc[m];
    arc[0].tail = 1; arc[0].head = 2;
    ```
  - ```
    struct Arc {
        int head;
        int tail;
    };
    Arc* arc = (Arc*) malloc(m * sizeof(Arc)); // Speicher für Kantenliste
    arc[0].tail = 1; arc[1].head = 2; // speichere erste Kante
    ... // hier weitere Zeilen Programmcode
    free(arc); // Speicher wieder freigeben, wenn nicht mehr benötigt
    ```
    ```

Einlesen eines Graphen aus einer Datei

- Gegeben sei eine Datei `mygraph.dat` mit folgendem Inhalt:

```
3
4
1   3
3   2
1   2
2   1
```



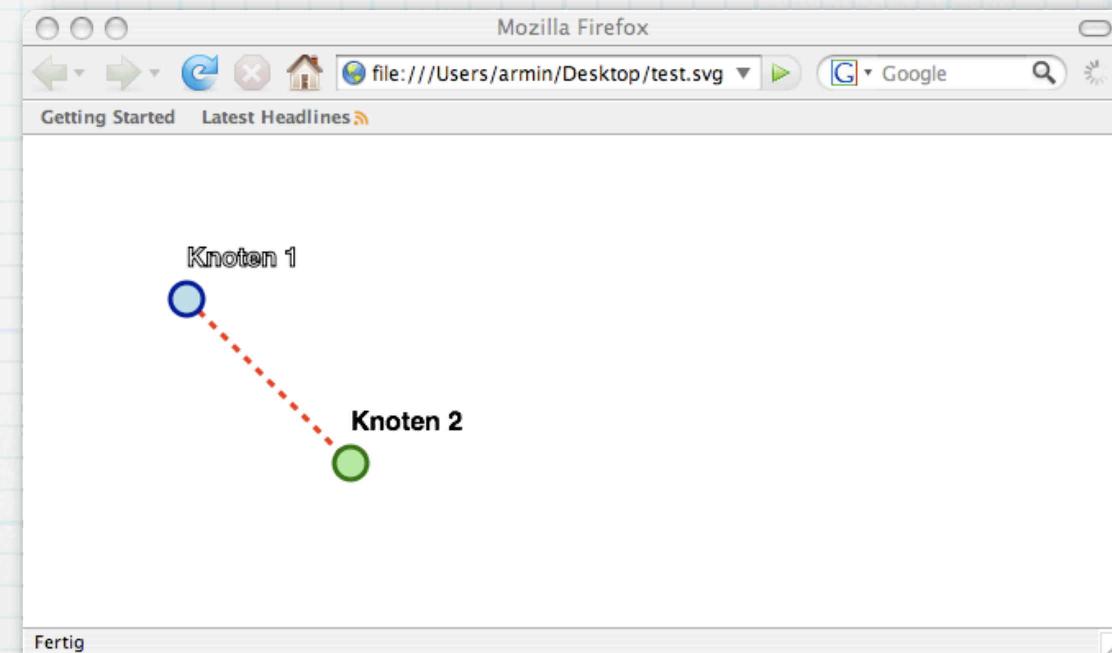
- Folgender Programmteil liest den Graphen in den Speicher (in C++):

```
#include <iostream.h>
#include <fstream.h>
struct Arc {
    int head;
    int tail;
};
int i,m,n;
ifstream file; // „ofstream“ für Ausgabe
file.open(„mygraph.dat“); // Datei zum Lesen öffnen
file >> n; // lese Anzahl Knoten
file >> m; // lese Anzahl Bögen
Arc* arc = (Arc*) malloc(m * sizeof(Arc)); // Speicher reservieren
for (i=0; i<m; i++) { // einer nach dem Anderen
    file >> arc[i].tail >> arc[i].head; // lese Bögen, (<< für Ausgabe)
}
file.close(); // Datei wieder schließen
free(arc); // wenn nicht mehr benötigt, Speicher freigeben
```

Grafische Ausgabe eines Graphen mit SVG

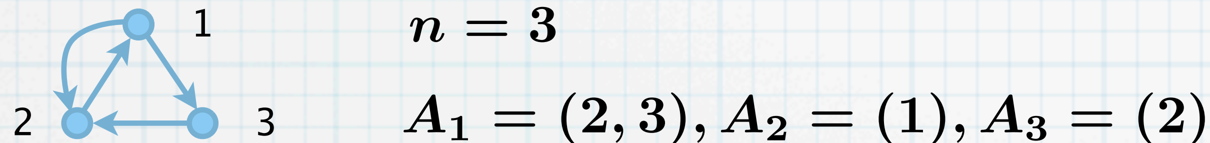
* Scalable Vector Graphic (SVG) ist eine offene Beschreibungssprache für 2D-Grafik.
* SVG basiert auf der Extensible Markup Language (XML) (ähnlich HTML).
* Die meisten Internet-Browser können SVG darstellen, z.B. Mozilla Firefox.
* Vollständige Dokumentation siehe `http://www.w3.org/Graphics/SVG/`
* Beispiel:

```
<?xml version="1.0"?>
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="300">
    <path d="M 100 100 L 200 200"
          style="stroke:red; stroke-width:3;
                 fill:none; stroke-dasharray:5,5"/>
    <circle cx="100" cy="100" r="10"
          style="stroke:darkblue;
                 stroke-width:3;
                 fill:lightblue"/>
    <circle cx="200" cy="200" r="10"
          style="stroke:darkgreen;
                 stroke-width:3;
                 fill:lightgreen"/>
    <text x="100" y="80"
          style="stroke:black;
                 stroke-width:1;
                 fill:none">Knoten 1</text>
    <text x="200" y="180"
          style="stroke:none;
                 stroke-width:1;
                 fill:black">Knoten 2</text>
</svg>
```



Darstellung eines (Di-) Graphen als Adjazenzliste

* Adjazenzliste: Ein Digraph $D = (V, A)$ mit $V := \{1, \ldots, n\}$ wird spezifiert durch
  * die Angabe von n,
  * die Angabe von n Listen $A_1, \ldots, A_n$, wobei A_i die Knoten j mit $(i, j) \in A$ enthält.
* Nachteil (im Falle von Graphen): jede Kante wird doppelt gespeichert.
* Vorteil: inzidente Knoten bearbeiten wird schneller.
* Für alle folgenden Algorithmen wird angenommen, dass (Di-) Graph als Adjazenzliste vorliegt.
* Beispiel:



$n = 3$

$A_1 = (2, 3), A_2 = (1), A_3 = (2)$

* Graph aus Datei in Adjazenzliste einlesen (mögliche Implementierung in C/C++):

```
int* deg = (int*) calloc(n, sizeof(int)); // Speicher für Ausgangsgrade
int** A = (int**) malloc(n * sizeof(int*)); // Speicher für Listen
for (i=0; i<n; i++) { A[i] = NULL; } // alle Zeiger auf NULL
for (i=0; i<m; i++) { // einer nach dem Anderen
    file >> tail >> head; // lese Bögen aus Datei
    int degree = deg[tail]; // hole Grad (Länge der Liste)
    A[tail] = (int*) realloc(A[tail], (degree+1) * sizeof(int));
      // Speicherplatz in der Liste um 1 erhöhen
    A[tail][degree] = head; // neuer Nachfolgerknoten am Ende einfügen
    deg[tail]++; // Grad um 1 erhöhen
}
... // hier weiter im Programm
for (i=0; i<m; i++) { free(A[i]); } // Speicherfreigabe in ...
free(A); // ... umgekehrter Reihenfolge
```

* Alternative Implementierung: einfach oder doppelt verketteten Listen.

Algorithmen-Schreibweise („Pseudo-Code")

* Wir spezifizieren anfangs die Eingabe und die Ausgabe des Algorithmus.
* Die Schlüsselworte

  (1) **algorithm** [name] ... **end algorithm**

  (2) **sub** [name](Parameter) ... **end sub**

  definieren den äußeren Rahmen des Algorithmus bzw. einer Unterroutine.
* Verzweigungen:

  (2) **if** B **then** P1; P2; P3;...; Pn **else** Q1; Q2; Q3; ...; Qm **end if**

  bedeutet: wenn Bedingung B erfüllt ist, führe P1, ..., Pn aus; wenn B nicht erfüllt ist, führe Q1, ..., Qm aus. Die Alternative (**else**) kann fehlen.
* Schleifen:

  (3) **for** i = 1 to n **do** P1; P2; ...; Pm **end for**

  bedeutet: P1, ..., Pm werden der Reihe nach für i = 1, i = 2, ..., i = n ausgeführt.
* Iterationen:

  (4) **while** B **do** P1; P2; ...; Pn **end while**

  bedeutet: wenn Bedingung B erfüllt ist, führe P1, ..., Pn aus; wiederhole dieses so lange, bis B nicht mehr erfüllt ist.

  (5) **repeat** P1; P2; ...; Pn **until** B

  bedeutet: führe P1, ..., Pn aus; ist B nicht erfüllt, wiederhole die Ausführung, bis B erfüllt ist.

  (6) **for** s in S **do** P1; P2; ...; Pn **end for**

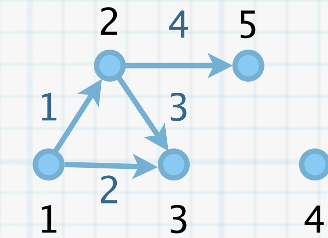
  bedeutet: P1, ..., Pn werden für alle Elemente s in S ausgeführt. Die Reihenfolge ist nicht spezifiziert.
* Sprünge:

  (7) **goto** (Schrittnummer)

  bedeutet: gehe zum Schritt mit der angegebenen Nummer.

Alternative Speicherung von Adjazenzlisten

- Eingabe: Digraph $D = (V, A)$, tail und head Listen
- Ausgabe: Listen mit firstTail, firstHead, nextTail, nextHead



$Bogen$	1	2	3	4
$tail$	1	1	2	2
$head$	2	3	3	5

```
(1)  algorithm createLists
(2)      for alle Knoten v in V do
(3)          firstTail(v) := firstHead(v) := previousIn(v) := previousOut(v) := „kein Bogen“ end for
(4)      for alle Bögen e in A do
(5)          nextTail(e) := nextHead(e) := „kein Bogen“ end for
(6)      for alle Bögen e=(v,w)=(tail(e), head(e)) in A do
(7)          e‘ := previousOut(v)
(8)          if e‘ ist kein Bogen then
(9)              firstTail(v) = e
(10)         else
(11)             nextTail(e‘) = e
(12)         end if
(13)         previousOut(v) := e
(14)         e‘ := previousIn(w)
(15)         if e‘ ist kein Bogen then
(16)             firstHead(w) = e
(17)         else
(18)             nextHead(e‘) = e
(19)         end if
(20)         previousIn(w) := e
(21)     end for
(22) end algorithm
```

$e_1 = (v, w) = (1, 2)$

$e' = kB$

$Knoten$	1	2	3	4	5
$prevIn$	kB	kB	kB	kB	kB
$prevOut$	kB	kB	kB	kB	kB

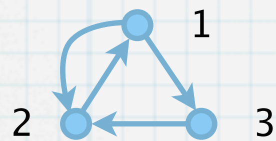
$e' = kB$

$Knoten$	1	2	3	4	5
$firstTail$	kB	kB	kB	kB	kB
$firstHead$	kB	kB	kB	kB	kB

$Bogen$	1	2	3	4
$nextTail$	kB	kB	kB	kB
$nextHead$	kB	kB	kB	kB

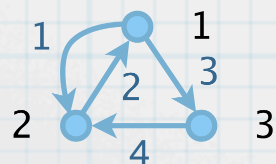
Adjazenz- und Inzidenzmatrizen

* Ein Digraph $D = (V, A)$ mit $V := \{1, \ldots, n\}$ wird spezifiert durch die Angabe einer $n \times n$ **Adjazenzmatrix** $M = (a_{i,j})_{1 \leq i,j \leq n}$, wobei $a_{i,j} = 1$, wenn $(i, j) \in A$, und $a_{i,j} = 0$ sonst.
* Nachteile:
  * Immer gleicher, hoher Speicherbedarf, auch wenn Digraph nur wenigen Kanten hat.
  * Im Falle von Graphen wird jede Kante wird doppelt gespeichert.
* Vorteil: man kann sehr schnell testen, ob eine Kante existiert.
* Beispiel:



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

* Ein Digraph $D = (V, A)$ mit $V := \{1, \ldots, n\}$ und $A := \{1, \ldots, m\}$ wird spezifiziert durch die Angabe einer $n \times m$ **Inzidenzmatrix** $M = (a_{i,j})_{1 \leq i \leq n, 1 \leq j \leq m}$, wobei $a_{i,j} = 1$ wenn Knoten i der Anfangsknoten von Bogen j ist, $a_{i,j} = -1$ wenn Knoten i Endknoten von Bogen j ist und $a_{i,j} = 0$ sonst. (Für Graphen speichert man stets +1).
* Nachteil: sehr hoher Speicheraufwand.
* Trotzdem wichtig für die Theorie:
  * Matrix ist Nebenbedingungsmatrix des Minimalkosten-Fluss-Problems.
  * Matrix hat interessante Eigenschaften (Stichwort: „total unimodular“).
* Beispiel:

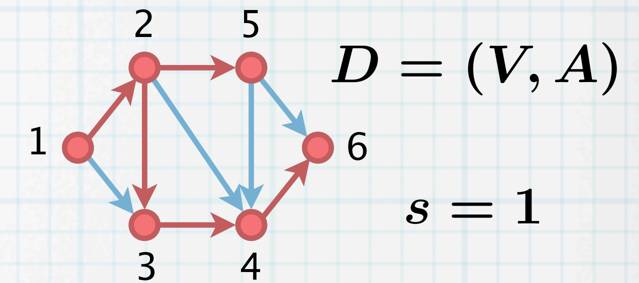


$$M = \begin{pmatrix} 1 & -1 & 1 & 0 \\ -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

Komplexität von Algorithmen

* Wieviel Platz braucht ein Algorithmus (für zusätzliche Daten während der Berechnung)?
* Wie schnell ist er (in Abhängigkeit der Eingabedaten)?
* **Definition 25**:
  Die **Kodierungslänge** (oder **Länge**) $[I]$ der Instanz I eines Problems ist die Länge einer Sequenz aus Nullen und Einsen (Bits), die zur Beschreibung der Instanz benötigt werden.
* Beispiele:
  * Kodierung von nichtnegativen ganzen Zahlen: 711 lässt sich binär schreiben als 1011000111. Also ist $[711] = 10$. Allgemein: $[k] = \lceil \log k \rceil$.
  * Kodierung von ganzen Zahlen: $[k] = \lceil \log k \rceil + 1$ (ein Bit mehr fürs Vorzeichen).
  * Kodierung eines Digraphen $D = (V, A)$ mit $V := \{1, \ldots, n\}$ und $A := \{1, \ldots, m\}$:
    $[D] = n^2$ (Größe der 0–1–Adjazenzmatrix) oder $[D] = \lceil \log n \rceil + \lceil \log n \rceil \cdot m + \lceil \log m \rceil \cdot n$ (Länge der Bitsequenz einer Adjazenzliste).
* **Definition 26**:
  Die **Zeitkomplexität** eines Algorithmus A ist eine Funktion f, welche die maximale Anzahl an Berechnungsschritten $f(n)$ angibt, die A zur Lösung einer Instanz der Länge n benötigt.
* Diese Definition orientiert sich am „ungünstigsten Fall“. Andere Maße, die sich z.B. am durchschnittlichen Fall orientieren, werden verwendet, sind aber schwieriger zu analysieren. Man braucht dazu u.A. eine Wahrscheinlichkeitsverteilung für die Eingabedaten.
* **Definition 27**:
  Wir schreiben $f(n) = O(g(n))$, wenn es eine Konstante $c > 0$ gibt mit $f(n) \leq c \cdot g(n)$ für alle hinreichend großen n, und sagen, die Zeitkomplexität des Algorithmus ist $O(g(n))$.
  Ein Algorithmus mit Komplexität $O(n^k)$ für ein geeignetes k ist **polynomial** oder **effizient**.
  Gibt es kein solches k, ist die Zeitkomplexität des Algorithmus **exponentiell**.
  Probleme, für die es polynomiale Algorithmen gibt, heißen **leicht**.

(Di-) Graphen durchsuchen, Tiefen- und Breitensuche



- Eingabe: Digraph $D = (V, A)$, Knoten $s \in V$
- Ausgabe: Teilmenge aller von s erreichbaren Knoten $R \subseteq V$

(1) **algorithm** scanGraph
(2) $R := \{s\}, Q := (s), T := \emptyset$
(3) **repeat**
(4) wähle $v \in Q$
(5) **if** es gibt ein $a = (v, w) \in A$ mit $w \in V \backslash R$ **then**
(6) $R := R \cup \{w\}, Q := Q \cup (w), T := T \cup \{a\}$
(7) **else**
(8) $Q := Q \backslash (v)$
(9) **end if**
(10) **until** $Q = \emptyset$
(11) **end algorithm**

$v = 1$

$a = (v, w) = (1, 2)$

$R = \{1\}$

$Q = (1)$

$T = \{\}$

- Bemerkung: scanGraph kann auch für Graphen verwendet werden. In Schritt (5) muss es dazu heißen: „... es gibt ein $e = \{v, w\} \in E$ mit..."
- In Schritt (4) und (5) gibt es Wahlmöglichkeit bzgl. der Reihenfolge der Knoten/Kanten.
  - Wählt man stets dasjenige $v \in Q$, welches zuletzt zu Q hinzugefügt wurde, spricht man von einer **Tiefensuche**. Q wird als LIFO-Stapel (last-in-first-out Stack) implementiert.
  - Wählt man stets dasjenige $v \in Q$, welches zuerst zu Q hinzugefügt wurde, spricht man von einer **Breitensuche**. Q wird als FIFO-Warteschlange (first-in-first-out Queue) implementiert.
- Bemerkung: Die Menge T ist, falls G zusammenhängend, ein „Spannbaum" (Definition später). scanGraph kann verwendet werden, um den Zusammenhang eines Graphen zu prüfen.

Korrektheit und Laufzeit des Suchalgorithmus

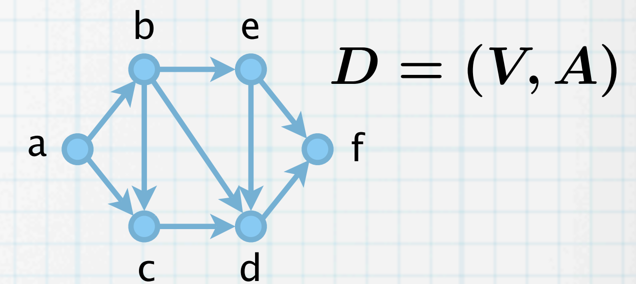
* **Satz 28**:
  Der (Di-)Graphen-Suchalgorithmus scanGraph arbeitet korrekt. Er kann so implementiert werden, dass er $O(|A| + |V|)$ Komplexität hat. Die (schwachen) Zusammenhangskomponenten eines (Di-)Graphen können in linearer Zeit ermittelt werden.
* Beweis:
  * Korrektheit: Angenommen, es gibt einen Knoten $w \in V \backslash R$, der von s aus erreichbar ist.
    Dann gibt es auf dem s-w-Pfad einen Bogen (x, y) (eine Kante $\{x, y\}$) mit $x \in R, y \notin R$.
    Da $x \in R$, war auch irgendwann $x \in Q$.
    Der Algorithmus terminiert erst, wenn Q leer, d.h. insbesondere, wenn x entfernt wurde.
    Dieses geschieht in Schritt (8), wenn es keinen Bogen (x, y) mit $y \notin R$ gibt. Widerspruch.
  * Laufzeit: Wir nehmen an, dass $D = (V, A)$ als Adjazenzliste gegeben ist.
    Ferner wird angenommen, dass alle elementaren Rechen- und Vergleichsoperationen konstante Zeit benötigen.
    Jeder Knoten des Graphen wird höchstens einmal zu R hinzugefügt.
    Jeder Knoten wird höchstens einmal zu Q hinzugefügt (und dann auch wieder entfernt).
    Jeder Bogen (bzw. jede Kante) wird höchstens einmal in Schritt (5) betrachtet und in Schritt (6) zu T hinzugefügt.
    Um die Laufzeit linear zu bekommen, wird für Suchschritt (5) ein Zähler implementiert, der die Position innerhalb der jeweiligen Bogenliste eines Knotens angibt. Erreicht der Zähler das Ende der Liste, kann der Knoten aus Q entfernt werden.
  * Um alle Zusammenhangskomponenten zu ermitteln, wendet man den Algorithmus auf einen beliebigen Startknoten an. Dann prüft man, ob $|R| = |V|$, d.h. der Digraph zusammenhängt. Falls nicht, wendet man den Algorithmus auf einen Knoten $s \in V \backslash R$ an, bis alle Zusammenhangskomponenten ermittelt sind. Da kein Knoten bzw. Bogen der bereits betrachteten Komponenten nochmals untersucht wird, bleibt die Laufzeit linear.

Topologisches Sortieren

* **Definition 29**:
  Ein Digraph $D = (V, A)$ heißt **azyklisch**, wenn er keinen gerichteten Kreis enthält.
* **Definition 30**:
  Eine **topologische Ordnung** eines Digraphen $D = (V, A)$ ist eine Abbildung $order : V \to \mathbb{N}$ mit $order(i) < order(j)$ für alle $(i, j) \in A$.
* **Satz 31**:
  Ein Digraph besitzt genau dann eine topologische Ordnung, wenn er azyklisch ist.
* **Lemma 32**:
  Ein azyklischer Digraph $D = (V, A)$ besitzt mindestens einen Knoten $v \in V$ mit $\deg_{in}(v) = 0$.
* Beweis:
  Wähle einen beliebigen Knoten $v_0 \in V$.
  Ist $\deg_{in}(v_0) = 0$, dann sind wir fertig.
  Andernfalls gibt es einen Bogen $(v_1, v_0) \in A$.
  Ist $\deg_{in}(v_1) = 0$, dann sind wir fertig.
  Andernfalls gibt es einen Bogen $(v_2, v_1) \in A$. Da D azyklisch ist, ist $v_2 \neq v_0$.
  Auf diese Weise konstruiert man eine Knotenfolge $v_0, v_1, v_2, \ldots$
  Da D nur endlich viele Knoten hat, erreicht man einen Knoten v mit $\deg_{in}(v) = 0$.
* Beweis von Satz 31:
  Klar: wenn eine topologische Ordnung vorhanden ist, gibt es keinen gerichteten Kreis.
  Umgekehrt: wähle einen Knoten $v_1 \in V$ mit $\deg_{in}(v_1) = 0$.
  Definiere $D_1 := (V_1, A_1)$ mit $V_1 := V \setminus \{v_1\}$ und $A_1 := \{(i, j) \in A : i, j \in V_1\}$.
  Digraph D_1 ist ebenfalls azyklisch. Wähle Knoten $v_2 \in V_1$ mit $\deg_{in}(v_2) = 0$... usw...
  Dann ist $v_1, v_2, \ldots$ die gesuchte topologische Anordnung.

Ein Algorithmus zum topologischen Sortieren

* Eingabe: Digraph $D = (V, A)$
* Ausgabe: entweder eine topologische Anordnung $ord : V \to \mathbb{N}$, oder Meldung, dass ein gerichteter Kreis existiert



(1) **algorithm** topologicalSorting
(2) $N := 0, L := \emptyset$
(3) **for** alle Knoten $v \in V$ **do** $ind(v) := 0$ **end for**
(4) **for** alle $(u, v) \in A$ **do** $ind(v) := ind(v) + 1$ **end for**
(5) **for** alle $v \in V$ mit $ind(v) = 0$ **do** $L := L \cup \{v\}$ **end for**
(6) **while** $L \neq \emptyset$ **do**
(7) wähle einen Knoten $v \in L$
(8) $L := L \backslash \{v\}$
(9) $N := N + 1$
(10) $ord(v) := N$
(11) **for** alle Knoten $w \in V$ mit $(v, w) \in A$ **do**
(12) $ind(w) := ind(w) - 1$
(13) **if** $ind(w) = 0$ **then** $L := L \cup \{w\}$ **end if**
(14) **end for**
(15) **end while**
(16) **if** $N = |V|$ **then** gibt ord aus
else gibt „gerichteter Kreis" aus **end if**
(17) **end algorithm**

$v = a$

$L = \{\}$

$N = 0$

$w = b$

v	a	b	c	d	e	f
$ind(v)$	0	0	0	0	0	0

v	a	b	c	d	e	f
$ord(v)$						

Algorithmus für starke Zusammenhangskomponenten

* Eingabe: Digraph $D = (V, A)$.
* Ausgabe: Funktion $\varphi : V \to \mathbb{N}$, welche die Zuordnung der Knoten zu Komponenten angibt.

(1) **sub** visit1(v)
(2) $R := R \cup \{v\}$
(3) **for** alle $w \in V \setminus R$ mit $(v, w) \in A$ **do** visit1(w) **end do**
(4) $N := N + 1, \psi(v) := N, \psi^{-1}(N) := v$
(5) **end sub**
(6) **sub** visit2(v)
(7) $R := R \cup \{v\}$
(8) **for** alle $w \in V \setminus R$ mit $(w, v) \in A$ **do** visit2(w) **end do**
(9) $\varphi(v) := K$
(10) **end sub**
(11) **algorithm** stronglyConnectedComponent
(12) $R := \emptyset, N := 0$
(13) **for** alle Knoten $v \in V$ **do**
(14) **if** $v \notin R$ **then** visit1(v) **end if**
(15) **end for**
(16) $R := \emptyset, K := 0$
(17) **for** i **from** $|V|$ **down to** 1 **do**
(18) **if** $\psi^{-1}(i) \notin R$ **then**
(19) $K := K + 1$
(20) visit2($\psi^{-1}(i)$)
(21) **end if**
(22) **end for**
(23) **end algorithm**

$v = a$

$w = g$

$N = 0$

$v = c$

$w = b$

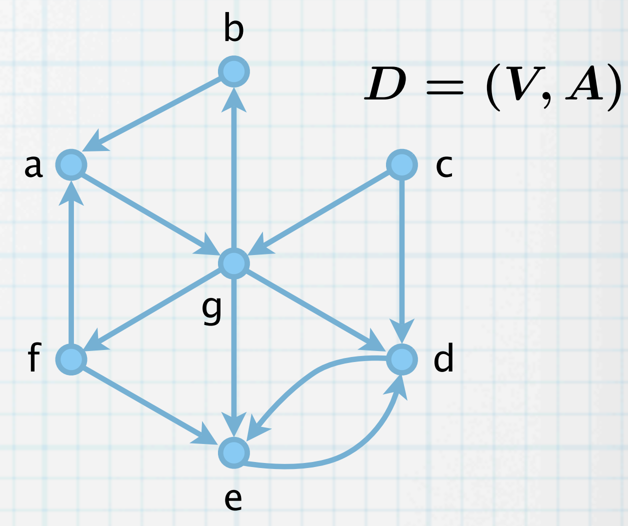
$R = \{\}$

$v = a$

$i = 7$

$\psi^{-1}(7) = c$

$K = 0$



v	a	b	c	d	e	f	g
ψ							

v	a	b	c	d	e	f	g
φ							

Beweis der Korrektheit und Laufzeitkomplexität

- **Satz 29**:
  Der Algorithmus stronglyConnectedComponent arbeitet korrekt und in linearer Zeit.
- Beweis:
  - Laufzeit: Analog zum Beweis von Satz 28 ist auch hier die Laufzeit $O(|V| + |A|)$.
  - Korrektheit: Seien Knoten u, v in einer stark zusammenhängenden Komponente C (d.h. es gibt gerichtete Wege in C in beide Richtungen). Sei o.B.d.A. u derjenige Knoten, den visit2 zuerst erreicht. Sei x der Knoten aus Schritt (20), von dem aus visit2 Knoten u erreicht.

    Da visit2 eine Tiefensuche ist, erreicht es nach Satz 28 auch Knoten v.

    Somit gilt $\varphi(x) = \varphi(u)$ und $\varphi(x) = \varphi(v)$, also insbesondere $\varphi(u) = \varphi(v)$.

    Zu zeigen: gilt $\varphi(u) = \varphi(v)$, sind u, v in einer stark zusammenhängenden Komponente.

    Sei x wieder derjenige Knoten, von dem aus visit2 Knoten u, v erreicht. Es gibt also einen rückwärts gerichteten Weg von x nach u, d.h. einen vorwärts gerichteten Weg von u nach x.

    Es gilt $\psi(x) > \psi(u)$, da x vor u von visit2 behandelt wird.

    Das bedeutet aber, dass visit1 die Untersuchung von u vor der von x abgeschlossen hatte.

    Da es einen Weg von u nach x gibt, wurde u durch einen rekursiven Aufruf von visit1 erreicht, wobei zuvor visit1 mit Übergabewert x aufgerufen wurde (rekursiv od. Schritt 14).

    Also gibt es auch einen vorwärts gerichteten Weg von x nach u.

    Mit genau der gleichen Argumentation zeigt man, dass es sowohl einen vorwärts gerichteten Weg von v nach x als auch von x nach v gibt.

    Somit gibt es gerichtete Wege zwischen u, v in beide Richtungen (jeweils über x).

    Also sind u, v in einer stark zusammenhängenden Komponente.

Literaturquellen

* R.K. Ahuja, T.L. Magnanti, J.B. Orlin, **Network Flows - Theory, Algorithms, and Applications**, Prentice Hall, Upper Saddle River, 1993. (Kapitel 2&3, Seite 23-92)
* J. Clark, D.A. Holton, **Graphentheorie**, Spektrum Akademischer Verlag, Heidelberg, 1994. (Kapitel 5, Seite 173-208)
* T.H. Cormen, C.E. Leiserson, **Introduction to Algorithms**, 2nd Edition, McGraw-Hill Book Company, Boston, 2001. (Kapitel 22, Seite 525-560)
* J. Erickson, **Algorithms**, Lecture Notes, University of Illinois at Urbana-Champaign, 2007. (Kapitel 11)
* D. Jungnickel: Graphen, **Netzwerke und Algorithmen**, BI Wissenschaftsverlag, Mannheim, 1994. (Kapitel 1&2, Seite 17-88)
* B. Korte, J. Vygen: **Combinatorial Optimization - Theory and Algorithms**, 2nd Edition, Springer Verlag, Berlin, 2001. (Kapitel 2, Seite 13-48)
* L. Lovasz, J. Pelikan, K. Vesztergombi, **Discrete Mathematics - Elementary and Beyond**, Springer Verlag, New York, 2003. (Kapitel 7&12, Seite 125-140&189-196)
* J. Matousek, J. Nesetril, **Invitation to Discrete Mathematics**, Clarendon Press, Oxford, 1999. (Kapitel 3&5, Seite 97-137&167-201)
* R.J. Wilson, **Einführung in die Graphentheorie**, Vandenhoeck&Ruprecht, Göttingen, 1976. (Kapitel 1-5, Seite 9-85)