Lektion 3

In Kapitel 3, das mit dieser Lektion beginnt, lernen Sie die wichtigsten Grundbegriffe der Graphentheorie kennen. Graphen sind ein mathematisches Konzept, das in der Informatik an vielen Stellen verwendet wird, sei es zur abstrakten Beschreibung von Anwendungsproblemen oder auch als Implementierungsansatz für effiziente Datenstrukturen.

Kap. 3 Grundbegriffe der Graphentheorie

Inhalt

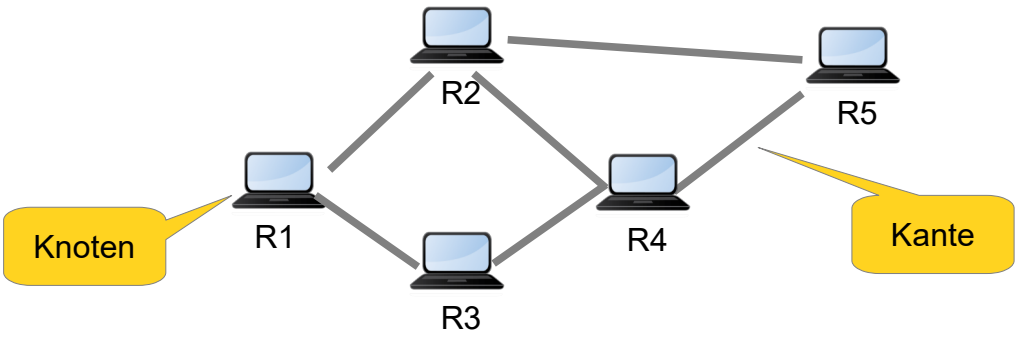
- Motivation: Anwendung von Graphen in der Informatik
- Gerichtete und ungerichtete Graphen
- Wege, Zyklen und Zusammenhang
- Bäume

3.1 Motivation: Anwendung von Graphen

Eine Art mathematischer Strukturen, die häufig in der Informatik zur abstrakten Formulierung von Anwendungsproblemen verwendet werden, sind sog. Graphen.

Durch Graphen wird formalisiert, welche Zweierbeziehungen (repräsentiert als sog. **Kanten**, engl. **edges**) zwischen Elementen (bezeichnet als **Knoten**, engl. **vertices** oder **nodes**) bestehen. Im ersten Semester haben Sie vermutlich an der einen und anderen Stelle schon Graphen verwendet.

Beispiel 3.1: Repräsentation eines Rechnernetzes als Graph



In diesem Kapitel werden Sie vor allem wichtige Grundbegriffe aus der Graphentheorie lernen, die man als Informatiker kennen sollte, um richtig „mitreden“ zu können. Im nächsten Semester lernen Sie dann in der Vorlesung „Algorithmen und Datenstrukturen“ eine Reihe wichtiger Graphalgorithmen kennen.

Bevor Sie die formalen Definitionen verschiedener Arten von Graphen und der zugehörigen Begriffe vorgestellt bekommen, möchte ich Ihnen an einem Beispiel aus dem Bereich des Compilerbaus zeigen, welche Rolle abstrakte formale Konzepte (hier Graphen) in der Informatik bei der Lösung von Problemen spielen.

Ausblick: Registervergabe im Compiler

Eine Teilaufgabe eines Compilers bei der Codegenerierung ist die Registervergabe. Ein Prozessor besitzt eine bestimmte Anzahl von Registern und Operationen, wie z.B. Addition oder Multiplikation, werden üblicherweise auf den Registern ausgeführt und nicht direkt mit Operanden, die im Hauptspeicher abgelegt sind. Der Vorteil von Registern ist der, dass der Zugriff darauf um ein Vielfaches schneller ist als der Zugriff auf den Hauptspeicher. Das Problem dabei ist, dass ein Prozessor nur eine relativ kleine Anzahl an Registern hat, im Vergleich zur Anzahl der Speicherplätze im Hauptspeicher. Die Konsequenz für die Codegenerierung ist, dass die überschaubare Anzahl von Registern möglichst gut genutzt werden muss.

In frühen Zeiten der Informatik waren Prozessoren üblich, deren Architekturansatz als **CISC** (**Complex Instruction Set Computer**) bezeichnet wird. Bei der CISC-Architektur hat ein Prozessor nur sehr wenige Register (z.B. 4 - 6), aber dafür einen sehr mächtigen Befehlssatz mit sehr komplexen Befehlen. Übliche Intel- und AMD-Prozessoren für PCs gehören im Wesentliche zur Klasse der CISC-Prozessoren.

In den 1970er Jahren war die Hardware-Entwicklung durch die zunehmende Integration dann so weit, dass es möglich wurde, Prozessoren mit einer relativ großen Anzahl von Registern zu realisieren. Bei IBM wurde dazu dann passend das Konzept der **RISC-Architektur** (**Reduced Instruction Set Computer**) entwickelt. Solche RICS-Prozessoren haben viele Register (z.B. Registerbänke mit 256 oder 512 Registern), aber statt komplexer Befehle einen einfachen Befehlssatz, so dass die Hardware und Steuerlogik simpel, aber schnell zu realisieren ist. Beispiele für RISC-Architekturen finden sich heutzutage z.B. bei Prozessoren von ARM für Smartphones und Embedded Systems.

Für Compilerbauer ergab sich damit die Herausforderung, diese große Anzahl an verfügbaren Registern möglichst optimal zu verwenden. Register werden typischerweise für die Speicherung von Zwischenergebnissen von Berechnungen verwendet. Während bei der Codegenerierung für CISC-Prozessoren die wenigen Register meist nur auf Ebene einzelner Anweisungen vergeben wurde, z.B. bei der Berechnung der rechten Seite folgender Anweisung,

```
b = 2 * a + 7;
```

ergab sich durch die RISC-Architektur mit vielen Registern die Möglichkeit, die

Registervergabe für komplette Methoden durchzuführen und damit auch viele Hauptspeicherzugriffe zu vermeiden. Den Durchbruch für eine geeignete Codegenerierung brachte das Verfahren der *Registervergabe durch Graphfärbung*, dessen Grundidee ich Ihnen kurz vorstellen möchte.

Wir betrachten dazu folgendes Codebeispiel:

```
m(a) {
    b = 2 * a + 7;
    c = a + b;
    d = a + 1;
    return c - d;
}
```

Die Variablen `b`, `c`, und `d` seien lokale Variablen der Methode `m`. Parameter `a` der Methode ist auch wie eine lokale Variable zu behandeln.

Man könnte beispielsweise jede Variable `a`, `b`, `c` und `d` in einem Register speichern, würde also hier vier Registern brauchen und hätte alle Hauptspeicherzugriffe vermieden. Bei realistischeren, komplexeren Methoden würde man aber schnell an die Grenze der verfügbaren Register kommen. Deswegen sollte die Registervergabe optimiert werden. Sehen Sie schon, mit wie vielen Registern man bei diesem Beispiel auskommt? (Vermutlich nicht.)

Welche elementare Grundregel ist bei der Registervergabe generell zu beachten? Ganz einfach: Zwei Zwischenergebnisse können nicht im gleichen Register gespeichert werden, wenn sie zur gleichen Zeit gebraucht werden. Die Zeit, wie lange ein einmal zugewiesener Wert einer Variable (das nennt man eine Variablendefinition) gebraucht wird, nennt man die Lebensdauer. Wenn also zwei Variablendefinitionen überschneidende Lebensdauern haben, muss man unterschiedliche Register dafür vergeben. Haben sie aber keine überschneidende Lebensdauer, kann das gleiche Register verwendet werden.

Registervergabe durch Graphfärbung

Für die Registervergabe sind folgende Teilaufgaben zu erledigen:

(1) Lebensdauer der Variablendefinitionen bestimmen.

(2) sog. *Interferenzgraph* aufstellen: Beschreibt, welche Variablendefinitionen gleichzeitig lebendig sind und somit zueinander im Konflikt stehen (interferieren).

(3) Die Registerzuordnung entspricht nun einer sog. Färbung des Interferenz-graphen:

- jede Farbe entspricht einem Register
- Variablen, die gleichzeitig lebendig sind, d.h. zueinander in Konflikt stehen, müssen unterschiedliche Farben (Register) erhalten.

Die dazu notwendige Graphfärbung ist im allgemeinen ein algorithmisch sehr schwieriges, nicht effizient lösbares Problem, sofern man es optimal lösen will (in der

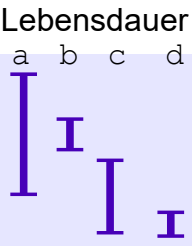
Komplexitätstheorie als *NP-vollständig* bezeichnet).

Es wurde dann aber ein heuristisches Verfahren für die Graphfärbung entwickelt, das zwar nicht immer optimal ist, aber in der Praxis sehr gut und schnell funktioniert.

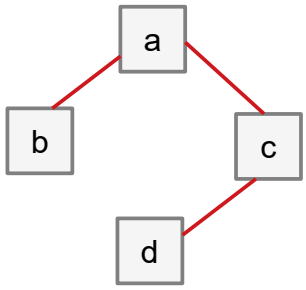
Wir vollziehen die Registervergabe durch Graphfärbung nun an unserem Beispiel nach:

(1) **Lebensdauern bestimmen**: Wir machen das hier einfach durch genaues Hinschauen. Dabei ist daran zu denken, dass die rechte Seite einer Wertzuweisung berechnet wird, bevor das Ergebnis an die linke Seite zugewiesen wird. Im Compiler muss dazu eine sog. *Datenflussanalyse* ausgeführt werden.

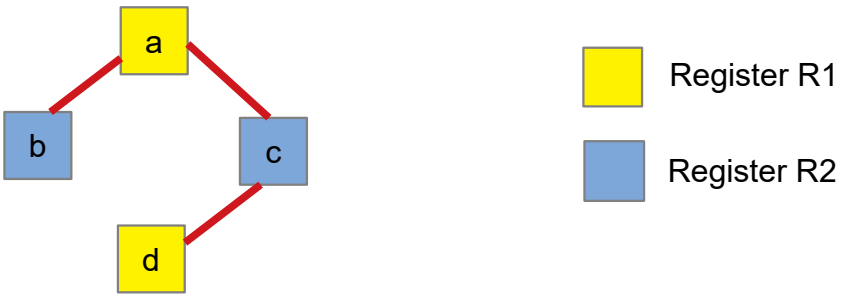
```
m(a) {
    b = 2 * a + 7;
    c = a + b;
    d = a + 1;
    return c - d;
}
```



(2) **Interferenzgraph aufstellen**: Die Variablen sind die Knoten des Graphen. Sind zwei Variablen gleichzeitig lebendig, gibt es eine Konflikt-Kante dazwischen. In unserem Beispiel ergibt sich folgender Graph.



(3) **Graphfärbung (= Registervergabe**), hier auch einfach durch Hinschauen bewerkstelligt):



Man kommt als in diesem Beispiel mit nur zwei Registern aus. Die Variablen `a` und `d` können in Register R1 und `b` und `c` in R2 abgelegt werden.

Gerichtete und ungerichtete Graphen

Die wesentliche Ausgangsinformation für die Registervergabe ist ein sog. *Graph*, eine mathematische Struktur aus *Knoten* (oben z.B. als Kästchen dargestellt) und *Kanten* (die jeweils zwei Knoten verbinden).

Bei Graphen unterscheidet man zwei grundsätzliche Arten:

- *gerichtete* Graphen
- *ungerichtete* Graphen.

Bei gerichteten Graphen repräsentiert eine Kante eine gerichtete Beziehung, bei ungerichteten Graphen eine symmetrische, ungerichtete Beziehung. Beim Interferenzgraph im Beispiel oben handelt es sich also um einen ungerichteten Graphen.

Wir werde zunächst gerichtete Graphen genauer behandeln und kommen später auf ungerichtete Graphen zurück.

3.2 Gerichtete Graphen

Definition 3.2 - Gerichteter Graph

- Ein **gerichteter Graph** $G = (V, E)$ besteht aus
  V Menge von **Knoten** (engl. ***vertex***)
  $E \subseteq V \times V$ Menge von **Kanten** (engl. ***edge*** oder ***node***).
- Für eine Kante $e = (u, v)$ ist
  u der **Ausgangs**- und
  v der **Zielknoten**.
- Existiert eine Kante $e = (u, v)$, dann ist Knoten v ein **Nachbar** von u.
  v heißen dann **benachbart** (**adjazent**) zu u.
- Eine Kante mit gleichem Ausgangs- und Zielknoten heißt **Schlinge** (oder auch **Schleife**)

Die Richtung einer Kante wird mathematisch dadurch ausgedrückt, dass eine Kante ein *Paar* ist, und bei einem Paar wird zwischen erster und zweiter Komponente unterschieden. Die erste Komponente ist der Ausgangsknoten, die zweite der Zielknoten.

Beispiel 3.3 - Graph als Menge von Knoten und Kanten

Gerichteter Graphen $G = (V, E)$ mit

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

$E = \{(1,2), (1,5), (2,4), (3,1), (5,3), (5,5)\}$

Hier ist beispielsweise (1,2) eine Kante von Ausgangsknoten 1 zu Zielknoten 2. Kante (5,5) ist eine Schlinge, da Ausgangs- und Zielknoten gleich sind.

3.2.1 Diagrammdarstellung von Graphen

Die Beschreibung auf Mengenebene ist für Menschen nicht gerade anschaulich, um zu sehen, wer mit wem verbunden ist. Deswegen wird oft eine verständlichere, gezeichnete Darstellung von Graphen in Diagrammform verwendet.

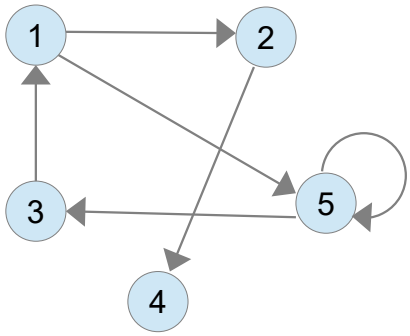
Diagrammdarstellung von gerichteten Graphen

- **Knoten** werden als **Kreise** (oder **Rechtecke**, **Ovale**,…) dargestellt
- **Kanten** werden als **Pfeil** vom Ausgangs- zum Zielknoten dargestellt.
- Die Positionierung der Knoten in der Ebene ist irrelevant. Kanten müssen keine geraden Linien sein, sie können auch beliebig gebogen/kurvig sein.

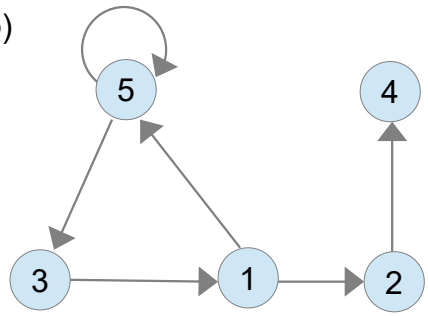
Beispiel 3.4- Graphen in Diagrammdarstellung

Welche Graphen, mathematisch beschrieben als Menge von Knoten und Menge von Kanten, werden durch folgende Diagramme dargestellt?

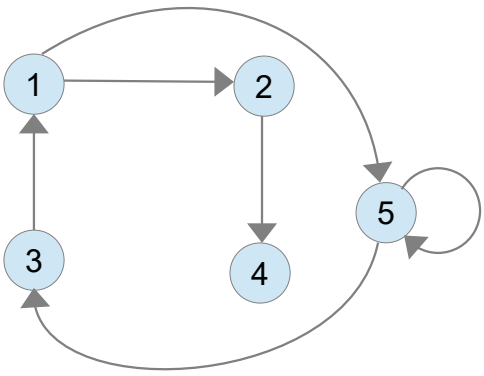
a)



b)



c)



- Bei a) wir der Graph $G = (V, E)$ aus Beispiel 3.3 dargestellt mit

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

  $E = \{(1,2), (1,5), (2,4), (3,1), (5,3), (5,5)\}$

  An der Darstellung ist gut zu sehen, dass es bei Knoten 5 eine Schlinge gibt (Kante (5,5)) und dass Knoten 1 die beiden Knoten 2 und 5 als Nachbarn hat. Knoten 3 ist aber keine Nachbar, da es keine Kante von 1 nach 3 gibt.

- Wenn Sie genau hinsehen, dann stellen Sie fest, dass auch bei b) und c) genau der gleiche Graph wie bei a) dargestellt wird.

Wichtig

- Ein Graph gibt nur die Information an welche Knoten es gibt und welche Knoten miteinander mittels Kante verbunden sind.
- Ein Graph kann auf ganz unterschiedliche Weise, d.h. in verschiedenen Diagrammdarstellungen (Graphlayouts), gezeichnet werden.

Ausblick: Graphdrawing

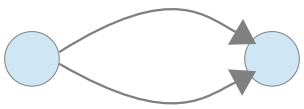
Das Thema Graphlayout (graph drawing) ist ein spannendes Forschungsgebiet der Informatik. Eine gute, übersichtliche Darstellung für eine gegebene Menge von Knoten und Kanten zu finden, ist alles andere als trivial.

Falls Sie später einmal für Projekte, Abschlussarbeit, etc., Tools für das Erstellen von Graphen oder für die Berechnung von Graphlayouts brauchen, hier zwei gängige, frei verfügbare Tools:

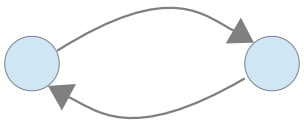
- Graphviz - Graph Visualization Software (www.graphviz.org): Bietet viele verschiedenen Algorithmen und Darstellungsoptionen für das Layout verschiedener Arten von Graphen.
- yEd Graph Editor (http://www.yworks.com/en/products_yed_about.html): Editor um Graphen einfach erstellen und darstellen zu können.

Anmerkungen

- Die hier verwendete Definition gerichteter Graphen erlaubt keine parallelen Mehrfachkanten zwischen zwei Knoten. So etwas geht also nicht:



Aber zwei Kanten in Gegenrichtung sind möglich:



- Ihnen ist vielleicht schon die Ähnlichkeit zum mathematischen Konzept der Relation aufgefallen. Ein gerichteter Graphen kann als eine zweistellige Relation über der Knotenmenge gesehen werden. Das Konzept der Relationen ist allerdings in der Mathematik erst viel später als das Konzept der gerichteten Graphen entstanden – so existiert beides parallel.

3.2.2 Speicherung von Graphen

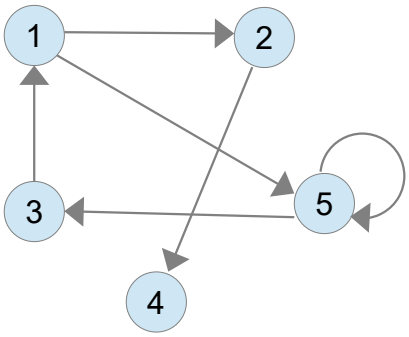
Für Algorithmen ist die mathematische Beschreibung eines Graphen als Menge von Knoten und Kanten nicht gut geeignet. Als Vorausschau auf das nächste Semester stelle ich Ihnen hier schon die zwei gängigsten Repräsentationen für Graphen vor, die Speicherung als sog. *Adjazenzmatrix* und die Speicherung mittels *Adjazenzlisten*.

Speicherung als Adjazenzmatrix

Durch eine Matrix wird beschreiben, welche Knoten miteinander durch eine Kante verbunden sind. Ein Eintrag 1 an Position [i,j] bedeutet, dass eine Kante von i nach j existiert, sonst steht dort 0. (Es kann natürlich auch eine boolesche Matrix mit true/false sein.)

Beispiel 3.5 - Adjazenzmatrix

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



Die Adjazenmatrix-Speicherung hat den Vorteil, dass sie sehr einfach als zweidimensionales Array realisiert werden kann und dass eine Kante zwischen zwei Knoten sehr effizient gesetzt, geprüft oder gelöscht werden kann.

Der Nachteil ist der Speicherbedarf für die n^2 Einträge bei n Knoten, unabhängig von der Anzahl der Kanten. Bei Anwendungen ist es oft so, dass die Knoten nur wenig verbunden sind, dass also relativ wenig Kanten existieren und folglich die Matrix weitgehend nur mit Nullen gefüllt ist. Ein weiterer Nachteil besteht auch darin, dass es schlecht möglich ist neue Knoten zu ergänzen.

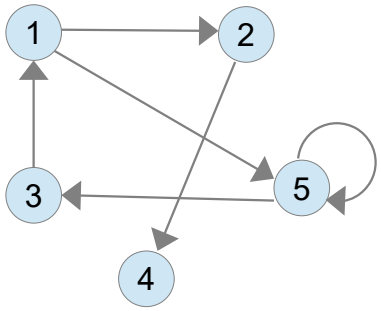
Speicherung mittels Adjazenzlisten

Eine Alternative zur Speicherung ist die Adjazenzlisten-Repräsentation. Dabei wird zu jedem Knoten die Menge der Nachbarknoten gespeichert (es muss keine geordnete Liste sein).

Beispiel 3.6 - Adjazenzlisten

Der gleiche Graph wie oben würde so mittels Adjazenzlisten gespeichert:

Knoten	Adjazenzliste
1	[2, 5]
2	[4]
3	[1]
4	[]
5	[3, 5]



Die Speicherung mittels Adjazenzlisten hat den Vorteil, dass der Speicherbedarf von der Anzahl der Kanten abhängt. Bei nur schwach verbundenen Graphen ist das günstig.

Ein weiterer Vorteil besteht darin, dass viel Graphalgorithmen „nachbarschafts-orientiert“ arbeiten, d.h. es wird dabei in einzelnen Bearbeitungsschritten jeweils die Menge der Nachbarn eines Knotens gebraucht. Die Nachbarn stehen bei der Adjazenzlistenspeicherung direkt zur Verfügung, während sie bei einer Adjazenzmatrix erst berechnet werden müssen.

Nachdem nun das Grundkonzept für Graphen klar sein sollte, werden in den folgenden Abschnitten einige dazugehörige Begriffe eingeführt.

3.2.3 Knotengrad

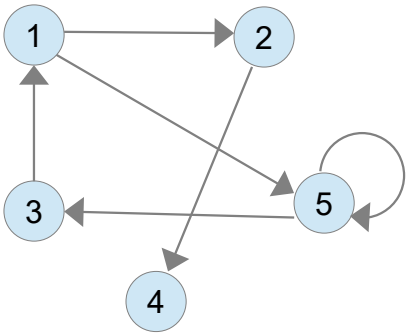
Der Grad eines Knotens gibt an, mit wie vielen Kantenenden der Knoten verbunden ist.

Definition 3.7 - Grad eines Knotens

Für einen Knoten v eines gerichteten Graphen $G = (V, E)$ ist

- der **Eingangsgrad** die Anzahl der Kanten mit Zielknoten v
- der **Ausgangsgrad** die Anzahl der Kanten mit Ausgangsknoten v
- der **Grad** die Summe aus Eingangs- und Ausgangsgrad von v

Beispiel 3.8



Für Knoten 1 gilt:

- Eingangsgrad: 1 (Kante von 3 nach 1)
- Ausgangsgrad: 2 (Kanten von 1 nach 2 und nach 5)
- Grad: 1+2 = 3

Für Knoten 5 gilt:

- Eingangsgrad: 2 (Kante von 1 nach 5 und von 5 nach 5)
- Ausgangsgrad: 2 (Kante von 5 nach 3 und von 5 nach 5)
- Grad: 2+2 = 4

Anmerkung

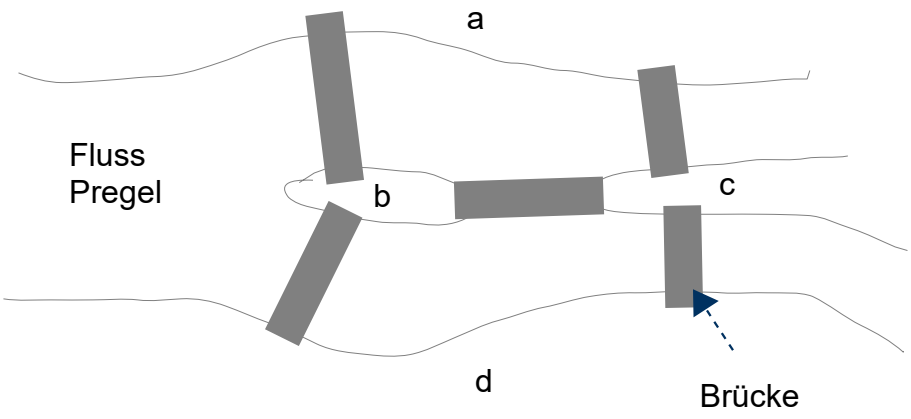
- Nach der in dieser Vorlesung verwendeten Definition zählen wir für den Grad die Enden einer Kante, die an einem Knoten beginnen oder enden. Eine Schlinge zählt 1 für den Eingangsgrad, 1 für den Ausgangsgrad und somit 2 für den Grad.

3.3 Ungerichtete Graphen

Die Graphentheorie wurde im 18. Jahrhundert von dem berühmten Mathematiker Leonhard Euler erfunden (der in Königsberg lebte), um das folgende „Königsberger Brückenproblem“ zu lösen.

Das Königsberger Brückenproblem (Leonhard Euler, 1736)

In Königsberg gibt es zahlreiche Gewässer und Brücken, z.B. wie hier angedeutet.



Die Frage, die sich Herr Euler gestellt hat, war folgende:

Gibt es einen Weg für den Sonntagsnachmittagsspaziergang, der jede Brücke genau einmal überquert und wieder am Ausgangspunkt endet?

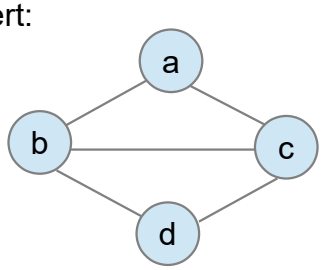
Für die Situation in Königsberg kann das relativ leicht herausgefunden werden. Als genialer Mathematiker hat er das Problem natürlich verallgemeinert:

> *Wie kann für eine beliebige „Stadt“ erkannt werden, ob es so einen Rundgang über alle Brücken gibt bzw. wann ist so eine Rundtour nicht möglich?*

Die erstaunlich einfache Lösung des Problems kommt später, erst brauchen wir die passenden Begriffe dafür.

Für die Lösung des Brückenproblems ist offensichtlich irrelevant, welche Form genau die Flüsse oder Teilgebiete haben. Wichtig ist nur, welches Gebiet mit welchem anderen durch eine Brücke verbunden ist. Dafür hat Euler das Konzept *Graph* als abstrahierende Beschreibung eingeführt. Die Verbindung über Brücken hat dabei keine vorgegebene Richtung. Somit hat man es hier mit *ungerichteten* Graphen zu tun

Die Situation oben würde durch folgenden ungerichteten Graphen mit den Knoten a, b, c und d repräsentiert:



Wie kann mathematisch formuliert werden, dass eine Kante bei einem ungerichteten Graphen keine Richtung hat? Die Lösung dafür ist, dass eine Kante als eine *Menge* repräsentiert wird. Bei einer Menge gibt es kein erstes und zweites Element, die Menge {a,b} ist die gleiche Menge wie {b,a}. Es kann also nicht zwischen Ausgangs- und Zielknoten unterschieden werden.

Definition 3.9 - Ungerichtete Graphen

- Ein **ungerichteter Graph** $G = (V, E)$ besteht aus
  - einer Knotenmenge V und
  - einer Kantenmenge E

$$E \subseteq \{ \{x, y\} \mid x, y \in V\}$$

Anmerkungen

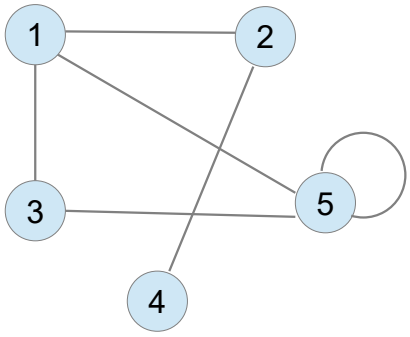
- Auch ein ungerichteter Graph kann Schlingen haben, d.h. Kanten, die einen Knoten mit sich selbst verbinden. Schlingen werden als einelementige Mengen repräsentiert.
- In der Diagrammdarstellung werden Kanten als Linien ohne Pfeil dargestellt.

Definition 3.10 - Knotengrad bei ungerichteten Graphen

Der **Grad eines Knotens** *v* ist die Anzahl der von *v* ausgehenden Kanten. Schlingen werden hierbei doppelt gezählt.

Aufgabe 3.11 - ungerichteter Graph als Menge von Knoten und Kanten

(1) Beschreiben Sie folgenden ungerichteten Graphen formal als Menge von Knoten und Kanten:



(2) Welchen Grad haben die Knoten 1 und 5?

Anmerkung

- Wir zählen auch bei ungerichteten Graphen für den Grad die Kantenenden, die bei einem Knoten liegen. Sie finden in der Literatur teilweise auch eine Definition für den Grad eines Knotens, bei dem die Kanten (nicht Kanten*enden*) gezählt werden, die mit dem Knoten verbunden sind. Der Unterschied ist der, dass eine Schlinge dann nur mit 1 zum Grad zählt.

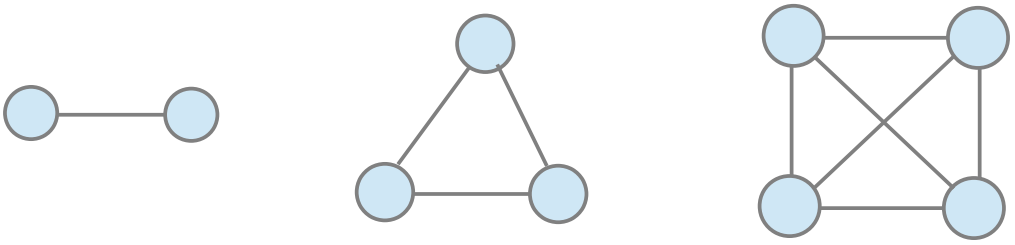
3.3.1 Vollständige Graphen

Definition 3.12 - Vollständiger Graph

Ein ungerichteter Graph heißt **vollständiger Graph**, wenn es zwischen je zwei verschiedenen Knoten eine Kante gibt (d.h. ohne Schlingen).

Beispiel 3.13 - vollständige Graphen

Vollständige Graphen mit 2, 3 und 4 Knoten (mit K_2, K_3 und K_4 bezeichnet):



K_2 K_3 K_4

Aufgabe 3.14 - vollständiger Graph

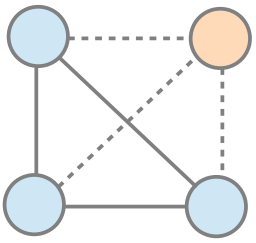
Geben Sie den vollständigen Graphen K_5 mit 5 Knoten an.

Eigenschaft 3.15 - Kantenzahl vollständiger Graphen

Ein **vollständiger Graph** mit *n* Knoten hat $\frac{n(n-1)}{2}$ **Kanten**.

Beweis:

- Der vollständige Graph aus 1 Knoten hat 0 Kanten
- Hat man einen vollständigen Graphen mit *n*-1 Knoten, kann man daraus einen vollständigen Graph mit *n* Knoten machen, indem man einen neuen Knoten dazu nimmt und durch *n*-1 Kanten mit jedem schon vorhandenen Knoten verbindet.



- Um einen vollständigen Graphen mit *n* Knoten aufzubauen, muss man also nach und nach 1 + 2 + 3 + … + (*n*-1) Kanten ergänzen. Das ergibt

$$\frac{n(n-1)}{2}$$

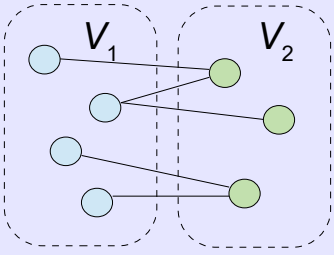
Kanten (arithmetische Reihe, Gauß'sche Summenformel).

3.3.2 Bipartite Graphen

Ein Graph wird *bipartit* genannt, d.h. in zwei Teile geteilt, wenn es zwei „Sorten" von Knoten gibt und eine Kante immer einen Knoten der einen Sorte mit einem Knoten der anderen Sorte verbindet.

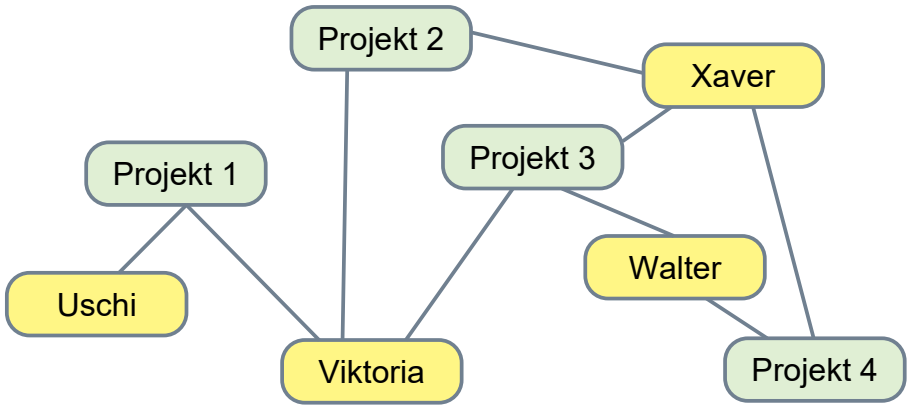
Definition 3.16 - bipartiter Graph / vollständiger bipartiter Graph

- Ein Graph $G = (V, E)$ heißt **bipartit**, wenn die Knotenmenge V in zwei disjunkte Teilmengen V_1, V_2 mit $V = V_1 \cup V_2$ aufgeteilt werden kann, so dass jede Kante zwei Knoten aus verschiedenen Teilmengen verbindet.



- Ein **bipartiter Graph** heißt **vollständig**, wenn es von jedem Knoten aus V_1 eine Kante zu jedem Knoten aus V_2 gibt.

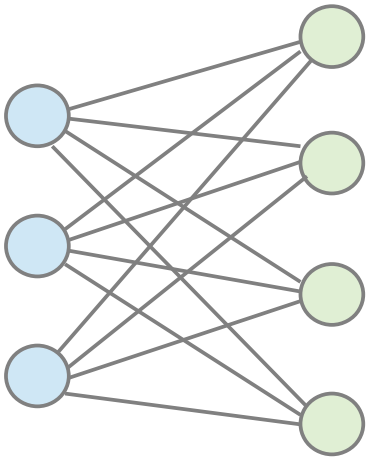
Beispiel 3.17 - Projektzuordnung als bipartiter Graph



- Eine Sorte Knoten für Projekte und die zweite Sorte Knoten für Personen
- Kanten verbinden immer einen Projektknoten mit einem Personenknoten

Beispiel 3.18 - Vollständiger bipartiter Graph $K_{3,4}$

Jeder blaue Knoten ist mit jedem grünen Knoten verbunden.



3.3.3 Planare Graphen

Planarität ist eine Eigenschaft von Graphen, die damit zusammenhängt, wie Graphen als Diagramm in der Ebene gezeichnet werden können.

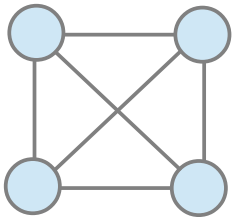
Definition 3.19 - Planare Graphen

Ein Graph G heißt **planar**, wenn er in der Ebene so gezeichnet werden kann, dass sich seine Kanten nicht kreuzen. Die Kanten müssen dabei *nicht* als gerade Linien gezeichnet werden, sondern können auch beliebig gebogen oder kurvig sein.

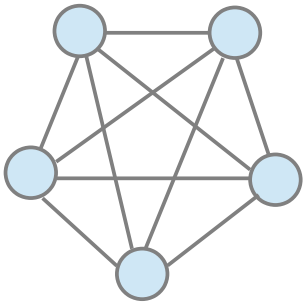
Aufgabe 3.20 - planare Graphen

Welche der folgenden Graphen sind planar? Können Sie die Graphen überschneidungsfrei zeichnen?

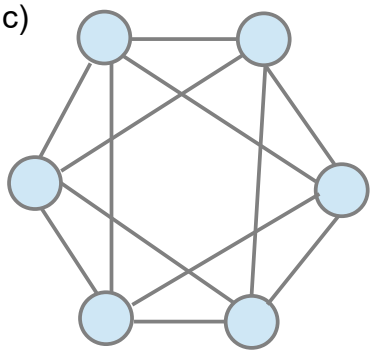
a)



b)



c)

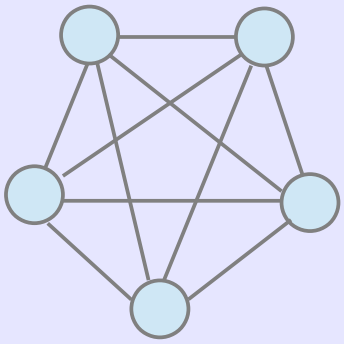


Woran kann man erkennen, ob ein Graph planar ist oder nicht? Vom polnischen Mathematiker Kazimierz Kuratowski (1896-1980) wurde folgendes bewiesen:

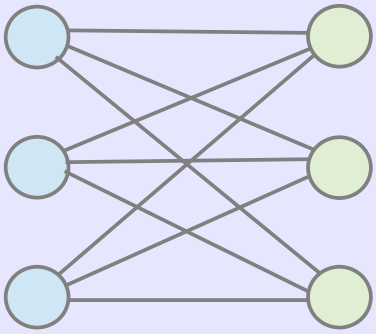
Eigenschaft 3.21 - Kuratowski-Graphen

(1) Der **vollständige Graph K_5** mit 5 Knoten und der **vollständige bipartite Graph $K_{3,3}$** mit zweimal drei Knoten sind **nicht planar**.

K_5:



$K_{3,3}$:



(2) Ein endlicher Graph ist genau dann planar, wenn er keinen Teilgraphen erhält, der durch Unterteilung von K_5 oder $K_{3,3}$ entstanden ist. Unterteilung bedeutet hier, dass beliebig oft (auch null mal) neue Knoten auf einer Kante eingefügt werden.

- Wenn in einem Graphen also irgendwie K_5 oder $K_{3,3}$ als Teil einhalten ist, ist der Graph nicht planar.
- Folglich sind also beispielsweise alle vollständigen Graphen mit mehr als 5 Knoten auch nicht planar, da immer K_5 als Teilgraph enthalten ist.

Fazit zu Lektion 3

Diese Fragen sollten Sie nun beantworten können

- Was sind Graphen?
- Wie sind gerichtete Graphen formal definiert?
- Was ist der Zusammenhang zwischen einem Graphen und der Diagrammdarstellung des Graphen?
- Wie können Graphen gespeichert werden?
- Wie sind Eingangsgrad, Ausgangsgrad und Grad eines Knotens definiert
- Was sind ungerichtete Graphen? Wie werden sie formal beschrieben?
- Was ist ein vollständiger Graph?
- Wie viele Kanten hat ein vollständiger Graph mit n Knoten?
- Wann ist ein Graph ein bipartiter Graph?
- Wann ist ein Graph planar? Welche Eigenschaften gelten für planare Graphen?