1 Graphentheorie

Wir werden Graphen und graphentheoretische Konzepte als Modellierung von theoretisch und praktisch relevanten Fragestellungen (speziell algorithmischen) kennenlernen.
Im Mittelpunkt steht die Frage der effizienten algorithmischen Lösbarkeit verschiedener Probleme unter Verwendung geeigneter Datenstrukturen.

1.1 Grundlegende Begriffe

1.1.1 Definition und Darstellung von Graphen

Ein *Graph* beschreibt Beziehungen (eine binäre Relation) zwischen den Elementen einer Menge von Objekten. Die Objekte werden als Knoten des Graphen bezeichnet; besteht zwischen zwei Knoten eine Beziehung, so sagen wir, dass es zwischen ihnen eine Kante gibt.

Definition Ein endlicher *ungerichteter* Graph G ist ein Paar (V, E) bestehend aus einer endlichen Knotenmenge V und einer Kantenmenge E von Knotenpaaren $e = \{u, v\}$, mit $u, v \in V$.
Im Gegensatz dazu ist endlicher *gerichteter Graph* G ist ein Paar (V, E) bestehend aus einer endlichen Knotenmenge V und einer Kantenmenge E von geordneten Knotenpaaren $e = (u, v)$, mit $u, v \in V$.

Man beachte, dass für eine ungerichtete Kante gilt $\{u, v\} = \{v, u\}$, während im gerichteten Fall $(u, v) \neq (v, u)$.
Eine Kante von einem Knoten zu sich selbst wird als *Schleife (loop)* bezeichnet. Meistens werden wir schleifenlose Graphen betrachten und zusätzlich auch ausschließen, dass es zwischen einem Knotenpaar mehrere Kanten gibt (dann wären die Kanten durch eine *Multimenge* zu beschreiben). Solche Graphen ohne loops und Mehrfachkanten heißen *schlicht (simple)*.

Wie werden Graphen dargestellt?

1. Graphische Darstellung: Die Knoten werden als Punkte (meistens in der Ebene) gezeichnet, die Kanten als ungerichtete bzw. gerichtete Strecken,

Streckenzüge bzw. Kurvenstücke, die die entsprechenden Punkte verbinden. Es gibt vielfältige andere Darstellungsformen mit dem Ziel, Graphen "schön" zu zeichnen; dies ist Gegenstand eines aktuellen Forschungsgebietes der Informatik: dem *Graph Drawing*.

2. Adjazenzliste: Ein Knoten v ist *adjazent (benachbart)* zu einem Knoten u, wenn es eine Kante von u nach v gibt.
   Wir beschreiben den Graphen, in dem wir für jeden Knoten alle seine adjazenten Knoten in einer Liste (als Datenstruktur) zusammenfassen.

3. Adjazenzmatrix: Wir nummerieren die Knoten aus V mit 1 bis n und bilden eine $n \times n$–Matrix A. Der Eintrag $A_{i,j}$ ist 1 wenn es eine Kante von i nach j gibt und 0 sonst.
   Man beachte, dass ungerichtete Graphen immer eine symmetrische Adjazenzmatrix haben.

Für die Behandlung algorithmischer Probleme sind in den allermeisten Fällen Adjazenzlisten die Datenstruktur der Wahl, weil sie die wahre Größe (d.h., den Speicherbedarf) eines Graphen (Anzahl der Kanten + Knoten) widerspiegeln, während Adjazenzmatrizen per Definition $|V|^2$ Einträge haben.

1.1.2 Beispiele für graphentheoretische Aufgabenstellungen

Wir wollen beispielhaft einige graphentheoretische und graphenalgorithmische Aufgaben skizzieren, die die Entwicklung der Graphentheorie stark beeinflusst haben.

1. Das 4-Farben-Problem: Man stelle sich die Welt mit einer beliebigen politischen Landkarte vor. Wir definieren einen Graphen, indem wir jedem Land einen Knoten zuordnen und zwei Knoten mit einer Kante verbinden, wenn sie einen gemeinsamen Grenzabschnitt haben. Wie viele Farben braucht man, um die Länder so einzufärben, dass benachbarte Länder verschiedene Farben haben. Appel und Haken (1978) haben (mit Computerhilfe, 1200 CPU-Stunden) "bewiesen", dass vier Farben immer ausreichen! Einen Beweis völlig ohne Computer gibt es bis heute nicht.

2. Welche Graphen treten als Ecken-Kanten-Gerüst von Polyedern auf? Polyeder sind Schnittmengen von Halbräumen im R^3, man denke an Würfel oder Prismen.

3. Eulersche Graphen: Man entscheide für Graphen, ob man die Kanten so durchlaufen kann, dass man jede Kante genau einmal benutzt und man am Schluss wieder am Startknoten steht. Der Ausgangspunkt für diese Frage war das von Euler 1736 gelöste sogenannte Königsberger Brückenproblem.

4. Hamiltonsche Graphen: Dies sind solche Graphen, die man so durchlaufen kann, dass man jeden Knoten genau einmal besucht bis man zum Ausgangsknoten zurückkehrt. Während man für das vorherige Entscheidungsproblem eine effiziente algorithmische Lösung kennt, ist dieses hier algorithmisch schwer (NP-vollständig).

5. Travelling Salesperson Problem (TSP): Oft hat man es mit bewerteten Graphen zu tun, das heißt Kanten und/oder Knoten haben zusätzliche Informationen wie Gewichte, Längen, Farben etc.
   Ein Beispiel ist das TSP. Wir haben n Städte gegeben als Punkte in der Ebene. Für jedes Paar (u, v) von Städten kennt man die Kosten, um von u nach v zu kommen. Man entwerfe für den Handelsreisenden eine geschlossene Tour, die alle Städte besucht und minimale Gesamtkosten hat. Auch dies ist ein algorithmisch schweres (NP–vollständiges) Problem.

6. Planare Graphen: Welche Graphen lassen sich so in der Ebene zeichnen, dass sich Kanten nicht schneiden, also sich höchstens in Knoten berühren? Wie kann man sie charakterisieren und algorithmisch schnell erkennen? Dazu lieferte Kuratowski 1931 eine wunderschöne Charakterisierung planarer Graphen durch verbotene Teilgraphen. Interessanterweise kann man planare Graphen immer so kreuzungsfrei zeichnen, dass die Kanten sogar Strecken sind (Fáry, 1948).

7. Netzwerke: Welchen Graphen sollte man der Architektur eines Rechnernetzes zu Grunde legen, wenn die Kanten beschränkte Kapazität haben, aber trotzdem schneller Informationsaustausch gewährleistet werden soll? (Würfelarchitekturen, Butterfly–Netzwerke u.a.)

1.1.3 Definitionen und wichtige Graphen

Sei im folgenden $G = (V, E)$ ein schlichter ungerichteter Graph.

Definition: Der *Grad* eines Knoten v in einem ungerichteten Graphen ist die Anzahl $deg_G(v)$ seiner Nachbarn.

In einem gerichteten Graphen bezeichnet $indeg_G(v)$ die Anzahl der in v ankommenden Kanten, der *Ingrad*; während $outdeg_G(v)$ die Anzahl der in v startenden Kanten angibt.

Es gilt der folgende Sachverhalt, auch als Handschlag-Lemma bekannt.

Satz: Für einen ungerichteten Graphen $G = (V, E)$ haben wir

$$\sum_{v \in V} deg_G(v) = 2 \cdot |E|$$

Der Satz hat ein einfaches aber überraschendes Korollar.

Korollar: In jedem ungerichteten Graphen ist die Anzahl der Knoten mit ungeradem Grad gerade.

Beweis: Die Summe aller Knotengrade is gerade. Die Teilsumme der geraden Knotengrade ist auch gerade, also ist der Rest, die Summe der ungeraden Knotengrade, auch gerade. Das heißt dieser Rest muss eine gerade Anzahl von Summanden haben.

Definition: Ein Graph $G' = (V', E')$ ist *Untergraph* von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$ ist. G' heißt *induzierter Untergraph*, falls außerdem gilt $E' = E \cap \{\{u, v\} | u, v \in V'\}$.

Definition: Zwei Graphen $G = (V, E)$ und $G' = (V', E,)$ heißen *isomorph*, wenn es eine Bijektion $\phi : V \to V'$ gibt mit der Eigenschaft $\forall u, v \in V : \{u, v\} \in E \iff \{\phi(u), \phi(v)\} \in E'$.

Das Testen, ob zwei Graphen isomorph sind ist im allgemeinen algorithmisch schwer, für eingeschränkte Klassen, wie z.B. die planaren Graphen kennt man effiziente Algorithmen.

Es gibt verschiedene Operationen, die aus Graphen neue Graphen erzeugen, wie Vereinigung aber auch zum Beispiel die Komplementbildung.
Definition: $G^c = (V, E^c)$ ist das Komplement von $G = (V, E)$ falls für jedes $u, v \in V$ gilt, dass $\{u, v\} \in E$ genau dann wenn $\{u, v\} \notin E^c$.

Wir lernen im Folgenden einige wichtige, weil zumindestens in der Theorie häufig auftretende Graphen kennen.

1. Der *vollständige Graph* K_n, $n \geq 1$ eine natürliche Zahl, besteht aus n Knoten und allen möglichen $\binom{n}{2}$ Kanten.
   Jeder Graph ist natürlich Untergraph eines vollständigen Graphen.

2. Der *vollständige bipartite Graph* $K_{n,m}$, mit $n, m \geq 1$, besteht aus $n + m$ Knoten und allen $n \cdot m$ Kanten, die jeweils einen der n Knoten mit einem der m Knoten verbinden.
   Untergraphen eines vollständigen bipartiten Graphen heißen *bipartit*.

3. Der *Hyperwürfel* Q_n hat als Knoten alle 0-1-Folgen der Länge n, zwei Folgen werden durch eine Kante verbunden, wenn sie sich genau an einer Stelle unterscheiden, also Hamming–Abstand 1 haben.
   Der Q_n hat 2^n Knoten und $n \cdot 2^{n-1}$ Kanten, was sofort aus dem Handschlag–Lemma folgt.

4. Der *Weg* P_n, $n \geq 0$ besteht aus $n+1$ Knoten $v_1, \ldots, v_{n+1}$ und den n Kanten $\{v_i, v_{i+1}\}$ für $1 \leq i \leq n$.

5. Der *Kreis* C_n, $n \geq 3$, entsteht aus dem Weg P_{n-1} durch das Hinzufügen der Kante $\{v_1, v_n\}$.

6. Ein Graph heißt *k-regulär*, wenn alle Knoten den Grad k haben.
   Alle Kreise sind also 2–regulär, der K_n ist $(n-1)$–regulär und der Q_n ist n–regulär.

7. *Bäume*, wie wir sie schon in Inf A zur Genüge kennen gelernt haben, sind die in der Informatik am häufigsten auftretenden Graphen.

1.2 Zusammenhang und Abstand in ungerichteten Graphen

Definition: Seien u und v Knoten in einem ungerichteten Graphen $G = (V, E)$. Wir sagen, dass v von u *erreichbar* ist, wenn es in G einen Weg als Untergraphen gibt, der u mit v verbindet.

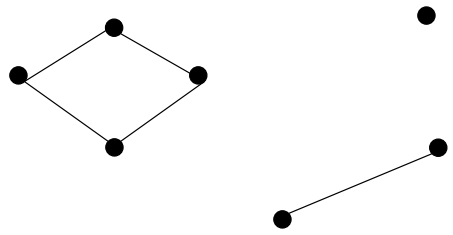
Erreichbarkeit ist eine binäre Relation über der Knotenmenge V. Offensichtlich ist diese Relation eine Äquivalenzrelation, denn sie ist:

- **reflexiv**: u ist mit sich selbst verbunden (durch den P_0);
- **symmetrisch**: der Weg, der u mit v verbindet, verbindet auch v mit u;
- **transitiv**: seien u, v, w Knoten und $u = u_1, \ldots, u_k = v$ die Knoten eines Weges von u nach v sowie $v = v_1, \ldots, v_l = w$ die Knoten eines Weges von v nach w. Sei i der kleinste Index mit $u_i \in \{u_1, \ldots, u_k\} \cap \{v_1, \ldots, v_l\}$. Wir haben $u_i = v_j$ für ein $1 \leq j \leq l$ und die Knoten $u_1, \ldots u_i, v_{j+1}, \ldots, v_l$ definieren einen Weg von u nach w.

Die Erreichbarkeitsrelation zerlegt also die Knotenmenge V in Äquivalenzklassen $V_1, \ldots, V_k$. Die von diesen Mengen induzierten Untergraphen heißen *Zusammenhangskomponenten* des Graphen G.

Definition: Ein ungerichteter Graph heißt *zusammenhängend*, wenn er genau eine Zusammenhangskomponente hat.

Beispiel:

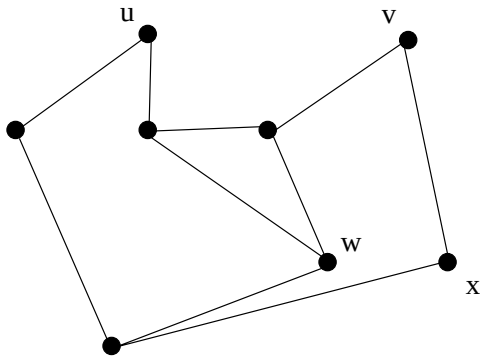


Dieser Graph hat 3 Zusammenhangskomponenten

Definition: Seien u, v Knoten in einem ungerichteten Graphen $G = (V, E)$. Sind u und v in einer gemeinsamen Zusammenhangskomponente von G, so definieren wir ihren *Abstand* $d(u, v)$ als Länge (Anzahl der Kanten) eines kürzesten Weges von u nach v.
Gehören sie zu verschiedenen Komponenten, so setzen wir $d(u, v) = \infty$.

Der *Durchmesser* $D(G)$ des Graphen ist definiert als das Maximum über alle paarweisen Abstände zwischen Knoten.

Beispiel:



d(u,w)=2 , d(u,v)=3 , D(G)=3

1.3 Charakterisierung bipartiter Graphen

Es erweist sich in vielen Fällen als nützlich, mehrere äquivalente Charakterisierungen ein und derselben Graphklasse zu haben. Im Falle der bipartiten Graphen, die wir als Untergraphen der vollständigen bipartiten Graphen eingeführt hatten, liefert dies der folgende Satz.

Satz: Ein Graph ist genau dann bipartit, wenn alle in ihm als Untergraph enthaltenen Kreise gerade Länge haben.

Beweis: Zunächst überlegt man sich, dass wir den Graphen als zusammenhängend voraussetzen können, ansonsten führt man den folgenden Beweis für jede Zusammenhangskomponente.
Sei $G = (V, E)$ bipartit, das heißt, $V = A \cup B$ mit $A \cap B = \emptyset$ und Kanten verlaufen nur zwischen Knoten aus A und Knoten aus B. Sei des weiteren C ein Kreis in G. C benutzt abwechselnd Knoten aus A und B und hat somit gerade Länge.

Wir zeigen die andere Richtung. Wir fixieren einen beliebigen Knoten $u \in V$.

Wir definieren:

$$A = \{v \in V |\ d(u,v) \ \text{ gerade }\}, B = V \setminus A$$

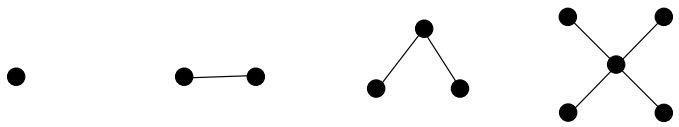
Zu zeigen, es gibt keine Kanten zwischen Knoten aus A (bzw. aus B). Wir führen einen indirekten Beweis.
Wir nehmen an, es gibt eine Kante $\{v,w\}, v,w \in B$ (für A analog) und finden einen Widerspruch zur Annahme, dass alle Kreise gerade Länge haben.
Wir betrachten kürzeste Wege von u zu v und zu w. Diese Wege haben gleiche Länge! (wegen der Kante zwischen v und w) Sei x der letzte gemeinsame Knoten auf beiden Wegen. Dann bilden die beiden Wegabschnitte von x nach v bzw. nach w zusammen mit der Kante $\{v,w\}$ einen Kreis ungerader Länge.

Man beachte, dass der Satz insbesondere gilt für Graphen, die gar keine Kreise besitzen, also Bäume und Wälder.

1.4 Bäume und ihre Charakterisierung

Definition: Ein zusammenhängender ungerichteter Graph ist ein *Baum*, wenn er keinen Kreis enthält.
Ein ungerichteter Graph, dessen Zusammenhangskomponenten Bäume sind, heißt *Wald*.

Beispiel:



Baumbeispiele

Definition: Sei $G = (V,E)$ ungerichtet und zusammenhängend mit $|V| = n$. Ein Untergraph T auf allen n Knoten, der ein Baum ist, heißt *aufspannender Baum* von G.

Analog definiert man *aufspannende Wälder* für nicht zusammenhängende Graphen.

Beobachtung: Jeder zusammenhängende Graph hat einen aufspannenden Baum, dieser ist aber nur eindeutig, wenn der Graph selbst ein Baum ist. Dann ist T der Graph selbst.

Satz: Folgende Aussagen sind äquivalent für $G = (V, E)$.

(1) $G = (V, E)$ ist ein Baum.

(2) Je zwei Knoten sind durch genau einen Weg verbunden.

(3) G ist zusammenhängend und es gilt: $|E| = |V| - 1$.

Beweis:

(1) $\Rightarrow$ (2) und (2) $\Rightarrow$ (1) sind einfache indirekte Schlüsse.

Wir zeigen (1) $\Rightarrow$ (3):
Zunächst hat jeder Baum Knoten vom Grad 1, diese nennt man *Blätter*. Das sieht man wie folgt. Seien $u_1, u_2, \ldots, u_i$ die Knoten eines längsten Weges in G. Alle Nachbarn von u_1 liegen auf diesem Weg, sonst wäre er nicht längster Weg. Nur u_2 kann Nachbar sein, sonst gäbe es einen Kreis.
Wir entfernen u_1 und die Kante $\{u_1, u_2\}$ aus G und erhalten einen zusammenhängenden Restgraphen G'. Dieser hat genau einen Knoten und eine Kante weniger als G. Wenn wir dies iterieren, bleibt zum Schluss genau ein Knoten ohne Kanten übrig. Also $|E| = |V| - 1$.

(3) $\Rightarrow$ (1):
Sei $T = (V, E')$ aufspannender Baum von G, damit $|V| - |E'| = 1$. Aber nach Vorraussetzung gilt auch $|V| - |E| = 1$. Allerdings ist $E' \subseteq E$ und die einzige Möglichkeit hierfür ist $E = E'$.

2. Grundlegende graphentheoretische Algorithmen

Neben der Untersuchung struktureller Eigenschaften von Graphklassen ist es für praktische Anwendungen eine zentrale Aufgabe, möglichst effiziente algorithmische Lösungen zu finden zur Bestimmung graphentheoretischer Parameter und Eigenschaften.

Wie bestimmt man den Abstand zwischen Knoten eines Graphen, wie testet man, ob er zusammenhängend ist und welche Datenstrukturen eignen sich dafür?

2.1 Graphdurchmustern: Breitensuche und Tiefensuche

Im Folgenden wollen wir Graphen systematisch von einem Startknoten aus durchmustern, das heißt, alle Knoten und Kanten 'anschauen'.
Sei $G = (V, E)$ ein ungerichteter (oder gerichteter) Graph gegeben durch seine Adjazenzlistendarstellung.

2.1.1 Breitensuche BFS

Die Breitensuche (breadth first search) startet in $s \in V$. Wir schauen uns zuerst alle Nachbarn von s an, danach die Nachbarn der Nachbarn usw. bis wir alle Knoten und Kanten erreicht haben.

Wir geben den Knoten 'Farben', diese symbolisieren ihren aktuellen Zustand:

- weiß: Knoten wurde noch nicht gesehen; zu Beginn sind alle Knoten weiß
- grau: Knoten wurde schon gesehen, wir müssen aber noch überprüfen, ob er noch weiße Nachbarn hat
- schwarz: Knoten ist erledigt, Knoten selbst und alle seine Nachbarn wurden gesehen.

Die noch zu untersuchenden Knoten werden in einer Warteschlange Q verwaltet. Knoten, deren Farbe von weiß nach grau wechselt, werden ans Ende der Schlange eingefügt. Die Schlange wird vom Kopf her abgearbeitet. Ist die Schlange leer, sind alle von s erreichbaren Knoten erledigt.

Mit BFS kann der Abstand $d[u] = d(s, u)$ eines erreichbaren Knotens u von s

berechnet werden (Beweis später) und gleichzeitig wird ein Baum von kürzesten Wegen von s zu allen erreichbaren Knoten aufgebaut. Dieser ist dadurch beschrieben, dass man für jeden Knoten einen Zeiger $\pi[u]$ auf den Vorgängerknoten auf einem kürzesten Weg von s nach u aufrechterhält. Ist dieser noch nicht bekannt oder existiert gar nicht, so ist der Zeiger auf NIL gesetzt. Will man für einen Knoten u am Schluss nicht nur den Vorgänger $\pi[u]$ sondern den ganzen kürzesten Weg von s nach u, so iteriert man einfach die Vorgängerabfrage bis man bei s landet.

Breitensuche BFS(G, s)**:**

```
01 for jede Ecke u ∈ V(G)\{s}
02     do Farbe[u] ← weiß
03        d[u] ← ∞
04        π[u] ← NIL
05 Farbe[s] ← grau
06 d[s] ← 0
07 π[s] ← NIL
08 Q ← {s}
09 while Q≠ ∅
10       do u ← Kopf[Q]
11          for jeden Nachbarn v ∈ Adj[u]
12              do if Farbe[v] = weiß
13                    then Farbe[v] ← grau
14                         d[v] ← d[u]+1
15                         π[v] ← u
16                         Setze v ans Ende von Q
17          Entferne Kopf aus Q
18          Farbe[u] = schwarz
```

Die Komplexität des BFS–Algorithmus ist offensichtlich $O(|V|+|E|)$. Man beachte, dass der konstruierte Baum kürzester Wege von der Reihenfolge der Knoten in den Adjazenzlisten abhängt, der Abstand der Knoten natürlich nicht. Für ein und denselben Graphen kann es sogar in Abhängigkeit von diesen Reihenfolgen mehrere nichtisomorphe Kürzeste–Wege–Bäume geben, siehe 2. Übungszettel.

2.1.2 Tiefensuche DFS

Die Idee der *Tiefensuche* (depth first search) ist einfach. Hat ein Knoten, den man besucht, mehrere unentdeckte Nachbarn, so geht man zum ersten Nachbarn und von dort in die 'Tiefe' zu einem noch unentdeckten Nachbarn des Nachbarn, falls es ihn gibt. Das macht man rekursiv, bis man nicht mehr in die Tiefe gehen kann. Dann geht man solange zurück, bis wieder eine Kante in die Tiefe geht, bzw. alles Erreichbare besucht wurde.

Die Farben haben wieder dieselbe Bedeutung wie beim BFS, ebenso die π–Zeiger, die die entstehende Baumstruktur implizit speichern. Der DFS berechnet nicht die Abstände vom Startknoten.

Man kann aber jedem Knoten ein Zeitintervall zuordnen, indem er 'aktiv' ist, was für verschiedene Anwendungen interessant ist. Dazu läßt man eine globale Uhr mitlaufen, und merkt sich für jeden Knoten u den Zeitpunkt $d[u]$, bei dem u entdeckt wird und den Zeitpunkt $f[u]$, bei dem u erledigt ist, da der Algorithmus alles in der 'Tiefe' unter u gesehen hat. Daher gilt für beliebige zwei Knoten im Graphen, dass entweder eines der Zeitintervalle voll im anderen enthalten ist, oder beide disjunkt sind.

Die geeignete Datenstruktur, um DFS zu implementieren, ist ein *stack (Kellerspeicher)*.
Ist der Graph in Adjazenzlistenform gegeben, so läuft DFS ebenfalls in Zeit $O(|V| + |E|)$.

Wie beim BFS hängen die entstehenden DFS–Bäume von der Reihenfolge in den Adjazenzlisten ab.

DFS(G)

01 **for** *jede Ecke* $u \in V(G)$
02 **do** *Farbe[u]* $\leftarrow$ *weiß*
03 $\pi[u] \leftarrow$ *NIL*
04 *Zeit* $\leftarrow 0$
05 **for** *jede Ecke* $u \in V(G)$
06 **do if** *Farbe[u] = weiß*
07 *DFS-visit(u)*

DFS-visit(u)

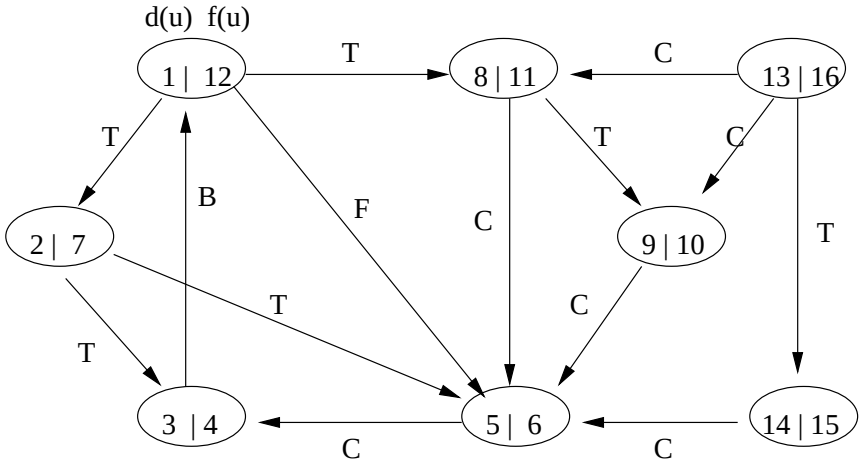
01 $Farbe[u] \leftarrow grau$
02 $d[u] \leftarrow Zeit \leftarrow Zeit+1$
03 **for** *jede Ecke* $v \in Adj[u]$
04 **do if** $Farbe[v] = weiß$
05 **then** $\pi[v] = u$
06 $DFS\text{-}visit(v)$
07 $Farbe[u] = schwarz$
08 $f[u] \leftarrow Zeit \leftarrow Zeit+1$

Der Stack ist eine Datenstruktur, die das LIFO–Prinzip (*last–in–first–out* umsetzt. Das heißt, die zu speichernden Daten (hier die grau werdenden Knoten) sind linear angeordnet und man kann ein neuen Eintrag nur als "oberstes" Element in den Stack einfügen (mit einer *push*–Operation) bzw. das oberste Element entfernen (mit einer *pop*–Operation, also beim DFS wenn der Knoten schwarz wird). Als weitere Funktionalität bietet ein Stack die Abfrage nach seiner Größe (*size*–Operation), die Boolesche Anfrage *isEmpty* und die Ausgabe des obersten Eintrages (*top*–Operation, ohne den Eintrag zu entfernen).
Wir werden verschiedene Realisierungen einer Stack–Datenstruktur noch kennen lernen und in Java implementieren.

Man kann die Kanten eines Graphen nach ihrer Rolle bei einer DFS–Durchmusterung klassifizieren. Wir unterscheiden:

- Tree-Kanten (T–Kanten): von grauen nach weißen Knoten
- Back–Kanten (B–Kanten): von grau nach grau
- Forward–Kanten (F–Kanten): von grau nach schwarz und zwar von Vorfahren zu Nachkommen im DFS–Baum
- Cross–Kanten (C–Kanten): alle restlichen Graphkanten

Im folgenden Beispiel sind die Zeitintervalle und die Kantenklassifikation illustriert:



DFS−Suche, Zeitintervalle der Knoten und Kantenklassifikation

Gerichtete azyklische Graphen:
Gerichtete azyklische Graphen (dag's) sind gerichtete Graphen ohne gerichtete Kreise. Diese spielen in der Informatik an verschiedenster Stelle eine Rolle, zum Beispiel bei Vererbungshierarchien in Java. Oder man stelle sich eine Menge von Jobs vor die linear zu ordnen sind. Dabei gibt es Einschränkungen derart, dass ein Job a vor einem anderen Job b bearbeitet werden muß. Gesucht ist eine lineare Ordnung (eine *topologische Sortierung*), die alle diese Constraints berücksichtigt.
Definition: Eine topologische Sortierung eines gerichteten Graphen ist eine Nummerierung seiner Knoten derart, dass aus $(u, v) \in E$ folgt $u \leq v$ in der Nummerierung.

Offensichtlich muss der gerichtete Graph ein dag sein, um eine topologische Sortierung zuzulassen. Das es dann aber immer geht, überlegt man sich wie folgt.

Fakt: Ein gerichteter Graph ist ein dag genau dann, wenn ein DFS–Durchmustern keine Back–Kanten produziert.

Basierend auf dieser Einsicht kann man topologisches Sortieren wie folgt realisieren: Führe ein DFS für den dag durch; wenn immer ein Knoten schwarz wird, gib ihn aus.

Behauptung: Dies ergibt ein topologisch invers sortierte Knotenfolge.

Beweis: Zu zeigen, wenn $(u, v) \in E$, dann ist $f(v) \leq f(u)$. Wir betrachten den Moment, wenn die Kante (u, v) vom DFS untersucht wird. Zu diesem Zeitpunkt ist u grau. v kann nicht grau sein wegen obigen Fakts. Also bestehen nur

die beiden Möglichkeiten, v ist weiß oder schwarz. Aber in beiden Fällen ist offensichtlich $f(v)$ kleiner als $f(u)$, denn Knoten die unter u liegen, sind "eher" fertig als u.

2.2 Minimal aufspannende Bäume

Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph und w eine Gewichtsfunktion, die jeder Kante eine reelle Zahl zuordnet, dies könnte zum Beispiel deren Länge sein.
Wir wissen, G hat einen aufspannenden Baum. Sei dies $T = (V, E')$. Wir definieren

$$w(T) = \sum_{e \in E'} w(e)$$

.

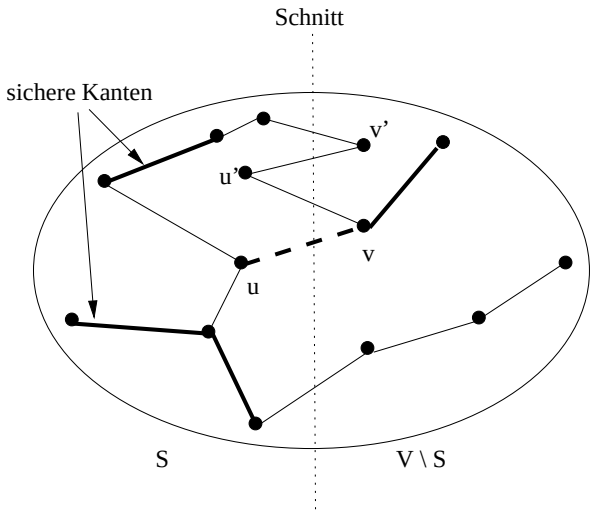
Aufgabe: Finde einen aufspannenden Baum mit minimalem Gesamtgewicht!

Offensichtlich hat jeder bewichtete zusammenhängende Graph einen minimal aufspannenden Baum (MST) und im Allgemeinen muss der auch nicht eindeutig sein. Wir werden zuerst einen generischen MST–Algorithmus kennenlernen und danach zwei konkrete Umsetzungen.

Definition: Ein *Schnitt* von G ist eine Zerlegung $(S, V \setminus S)$ seiner Knotenmenge. Eine Kantenmenge A *respektiert* einen Schnitt $(S, V \setminus S)$, falls keine Kante aus A einen Knoten aus S mit einem aus $V \setminus S$ verbindet.
Eine Kantenmenge A heißt *sicher*, wenn es einen MST T von G gibt, so dass A in der Kantenmenge von T enthalten ist.

Satz: (Generischer MST–Algorithmus)
Sei G ein gewichteter ungerichteter zusammenhängender Graph und sei A eine sichere Menge von Kanten, die einen Schnitt $(S, V \setminus S)$ respektiert. Wir betrachten eine leichteste Kante uv, mit $u \in S$, $v \in V \setminus S$. Dann ist $A \cup \{uv\}$ sicher.

Beweis: Sei T ein MST, der A enthält. Wir nehmen an, dass uv nicht zu T gehört, ansonsten ist nichts zu beweisen.
Wenn wir die Kante uv zu T hinzunehmen, entsteht genau ein Kreis C. Wir betrachten in C alle Kanten xy mit $x \in S, y \in V \setminus S$. Außer uv muss es wenigstens noch eine weitere solche Kante geben. Sei dies $u'v'$.
Streichen wir $u'v'$ aus $T \cup \{uv\}$, so entsteht ein aufspannender Baum T'. Da nach Annahme $w(uv) \leq w(u'v')$ und T ein MST ist, folgt sofort, T' ist MST und mithin ist $A \cup \{uv\}$ sicher.



Der MST T und die Kante uv definieren Kreis!

Der Satz kann algorithmisch umgesetzt werden. Man startet mit A leer, sucht sich einen Schnitt, der von A respektiert wird (dies sind am Anfang alle), nimmt die leichteste Kante über den Schnitt hinzu usw.

Die beiden MST–Algorithmen von Prim und Kruskal sind konkrete Umsetzungen davon.

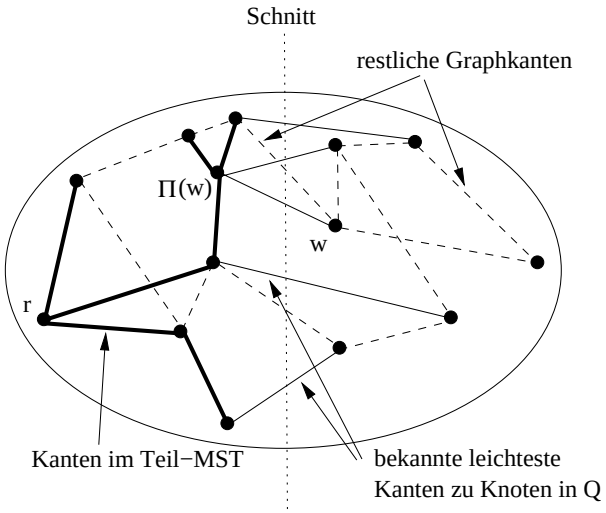
2.2.1 Der MST–Algorithmus von Prim

Dieser ist konzeptionell sehr einfach. Wir lassen den MST–Baum von einem beliebigen Startknoten r aus 'wachsen'. Die Frage ist, um welche Kante die Teillösung in einem Schritt erweitert wird. Wir schauen uns den Schnitt an, der die Teilösung vom Rest trennt. In einer Prioritätsschlange verwalten wir alle noch nicht erreichten Knoten w zusammen mit einem Schlüssel, der angibt, was im Moment das Gewicht einer leichtesten Kante ist, mittels derer w von einem der bereits in den Teilbaum aufgenommenen Knoten aus erreichbar ist.
Diese Schlüssel benötigen ggf. entsprechende Updates. Die π–Zeiger speichern wieder den aktuell gefundenen Teil–MST mit Wurzel r sowie die bekannten kürzesten Kanten zu Knoten in Q.

MST-Prim(G, w, r)

```
01 Q ← V(G)
02 for jedes u ∈ Q
03       key[u] ← ∞
04 key[r] ← 0
05 π[r] ← NIL
06 while Q ≠ ∅
07       do u ←Extract-Min(Q)
08           for jedes v ∈ Adj[u]
09               do if v ∈ Q und w(u,v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u,v)
```

Die Komplexität des Algorithmus hängt ab von der konkreten Realisierung des abstrakten Datentyps Prioritätsschlange ab, wir werden einige kennen lernen. Benutzt man für die Implementierung der Prioritätswarteschlange einen binären Heap , so ist die Komplexität $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$.



2.2.2 Der MST–Algorithmus von Kruskal

Zuerst werden die Kanten nach aufsteigenden Gewichten sortiert. Danach wird in dieser Reihenfolge, mit der leichtesten Kante beginnend, getestet, ob eine Kante, einen Kreis im bisher konstruierten Wald schließt (falls ja wird die Kante verworfen). Falls nein wird die sichere Menge um diese Kante erweitert. Anders gesagt, man prüft, ob es für die Kanten jeweils einen Schnitt gibt, für den die Kante die leichteste ist. Wenn die Kante einen Kreis schließt, so gibt es einen solchen Schnitt nicht, denn die Zusammenhangskomponenten der sicheren Kanten liegen auf einer Seite des Schnittes (respektieren ihn).

Das Interessanteste dabei ist die verwendete Datenstruktur, eine sogenannte Union-Find–Struktur zur Verwaltung von Partitionen einer Menge, hier der Knotenmenge V.

Die von der Datenstruktur unterstützten Operationen sind:

- MakeSet(v): Aus dem Element $v \in V$ wird eine Menge gemacht.
- Union(u,v): Die beiden Mengen, zu denen u bzw. v gehören, werden vereinigt.
- FindSet(v): Bestimme die Menge, zu der v gehört.

Dazu stelle man sich vor, dass jede Menge einen Repräsentanten hat, auf den alle Elemente der Menge zeigen. Bei der Union–Operation müssen also Zeiger umgehangen werden. Eine einfache Realisierung besteht darin, bei Union zweier Mengen, die Zeiger der kleineren Menge auf den Repräsentanten der größeren umzuleiten.
Man überlegt sich, dass damit insgesamt nur höchstens $|V| \log |V|$ mal Zeiger umgeleitet werden (s. Übungszettel 4).
Hier ist der Pseudocode für Kruskals Algorithmus.

MST-Kruskal(G, w)

```
01 A ← ∅
02 for jede Ecke v ∈ V(G)
03     do Make-Set(v)
04 Sortiere die Kanten aus E in nichtfallender Reihenfolge
   entsprechend ihres Gewichts
05 for jede Kante (u,v) in sortierter Reihenfolge
06     do if Find-Set(u) ≠ Find-Set(v)
07           then A ← A∪ {(u,v)}
08                Union(u,v)
09 return A
```