



Raymond Hemmecke

Vertretungsprofessor für Algorithmische Diskrete Mathematik

▶ Komplexitätstheorie

- ▶ Datenstrukturen und Kodierungsschemata
- ▶ asymptotische Notation, untere und obere Schranken
- ▶ Klassen \mathcal{P} , \mathcal{NP} , \mathcal{NP} -vollständig

▶ Algorithmen auf Graphen

- ▶ DFS-Algorithmus (aufspannende Bäume)
- ▶ Greedy-Algorithmus (minimale aufspannende Bäume)
- ▶ Dijkstra, Moore-Bellman, Yen-Variante (kürzeste Wege in Graphen)
- ▶ Ford-Fulkerson (maximale Flüsse in Netzwerken, Matching in bipartiten Graphen)

▶ Sortieren in Arrays

- ▶ Mergesort, Quicksort, Heapsort
- ▶ Divide-and-Conquer
- ▶ untere Komplexitätsschranken für das Sortieren



Grundlegende Begriffe der Graphentheorie

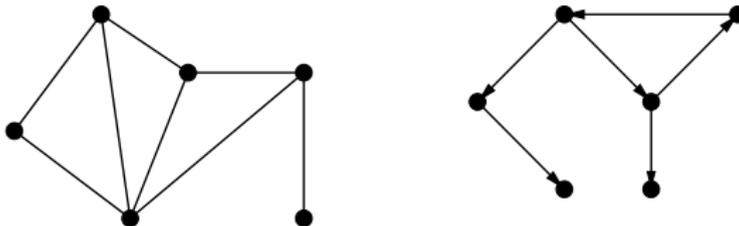
Was ist ein Graph?

Graph

- ▶ Ein ungerichteter **Graph** ist ein Paar $G = (V, E)$ disjunkter Mengen, wobei die Elemente von E ungeordnete Paare von Elementen aus V sind.
- ▶ Die Elemente aus V heißen **Knoten**, die Elemente aus E **Kanten**.

Digraph

- ▶ Ein **gerichteter Graph** ist ein Paar $G = (V, A)$ disjunkter Mengen, wobei die Elemente von A geordnete Paare von Elementen aus V sind.
- ▶ Die Elemente aus A heißen **Bögen** oder **gerichtete Kanten**.

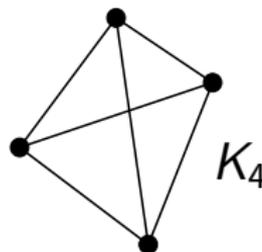
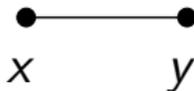
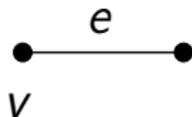


Inzidenz, Grad/Valenz

- ▶ Ein Knoten v heißt mit einer Kante e **inzident**, wenn $v \in e$ gilt.
- ▶ Der **Grad**/die **Valenz** eines Knoten v ist die Anzahl der mit v inzidenten Kanten.

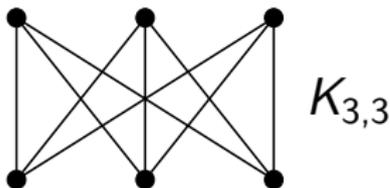
Adjazenz, vollständiger Graph

- ▶ Zwei Knoten x und y heißen **adjazent** oder **benachbart** in G , wenn $xy \in E(G)$ ist.
- ▶ Sind je zwei Knoten von G benachbart, so heißt G **vollständig**.



Definition

- ▶ Ein Graph heißt **bipartit**, wenn sich V disjunkt in zwei Teile V_1 und V_2 zerteilen lässt, so dass jede Kante in E einen Endknoten in V_1 und einen Endknoten in V_2 besitzt.



Fakt → Beweis als Übungsaufgabe

- ▶ Ein Graph ist genau dann bipartit, wenn er nur Kreise gerader Länge enthält.

Pfad, Weg, Kreis

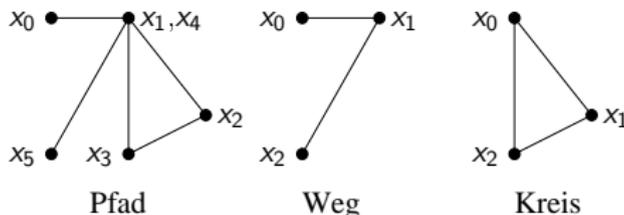
- ▶ Ein **Pfad** ist ein nichtleerer Graph $P = (V, E)$ der Form

$$V = \{x_0, x_1, \dots, x_k\}, E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}.$$

- ▶ Ein **Weg** ist ein Pfad, in dem alle x_i paarweise verschieden sind.
- ▶ Ein **Kreis** ist ein nichtleerer Graph $P = (V, E)$ der Form

$$V = \{x_0, x_1, \dots, x_k\}, E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k, x_kx_0\},$$

wobei die x_i paarweise verschieden sind.



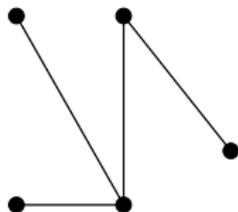
Zusammenhangskomponenten, Bäume und Wälder

Zusammenhängende Graphen

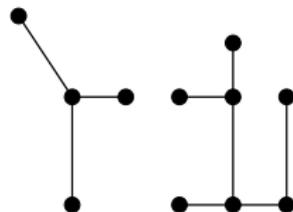
- ▶ Ein Graph G heißt **zusammenhängend**, falls es zu jedem Paar $v, w \in V$ einen Weg von v nach w in G gibt.
- ▶ Die zusammenhängenden Teile von G heißen **Zusammenhangskomponenten**.

Bäume und Wälder

- ▶ Ein **Baum** ist ein zusammenhängender Graph, der keinen Kreis besitzt.
- ▶ Ein **Wald** ist ein Graph, der keinen Kreis besitzt.



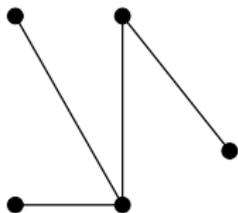
Baum



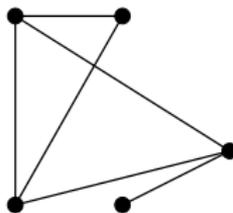
Wald, 2 Komponenten

Komplementärgraph

- ▶ Der zu G komplementäre Graph \bar{G} ist der Graph $\bar{G} = (V, \bar{E})$, mit $ij \in \bar{E} \Leftrightarrow ij \notin E$.



Graph



Komplement

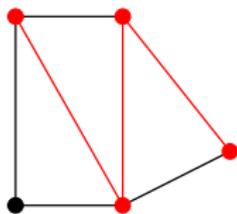
Fakt → Beweis als Übungsaufgabe

- ▶ Mindestens einer der Graphen G oder \bar{G} ist zusammenhängend.

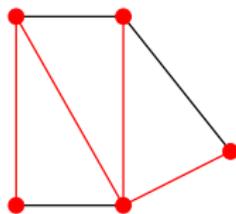
Untergraphen, aufspannende Untergraphen

Untergraph, aufspannende Untergraphen

- ▶ $G' = (V', E')$ heißt **Untergraph** von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$ gilt.
- ▶ $G' \subseteq G$ heißt **aufspannender Untergraph** von G , falls $V' = V$ gilt.



Graph mit Untergraph



aufspannender Baum

Finden eines aufspannenden Waldes \rightarrow DFS-Algorithmus

- ▶ Für gegebenes $G = (V, E)$ finde man einen aufspannenden Wald.

Probleme auf Graphen (1)

Finden eines Kreises/Hamiltonischen Kreises

- ▶ Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen Kreis enthält. Falls ja, gib einen solchen an.
- ▶ Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen hamiltonischen Kreis enthält. Falls ja, gib einen solchen an. (Ein **hamiltonischer Kreis** ist ein Kreis, der jeden Knoten genau einmal durchläuft.)

Finden eines aufspannenden Waldes → DFS-Algorithmus

- ▶ Für gegebenes $G = (V, E)$ finde man einen aufspannenden Wald.

Maximales Matching

- ▶ Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Kanten aus E , so dass je zwei Kanten nicht inzident sind.

Probleme auf Graphen (2)

Maximale stabile Mengen/Cliquen

- ▶ Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Knoten aus V , so dass je zwei Knoten nicht benachbart sind. → stabile Menge
- ▶ Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Knoten aus V , so dass je zwei Knoten benachbart sind. → Clique

Kürzeste Wege

- ▶ Für gegebenes $G = (V, E)$ und gegebenen Kantengewichten c_{ij} finde man einen kürzesten Weg zwischen zwei gegebenen Knoten v und w .

Färbungsproblem

- ▶ Finde für gegebenen Graphen $G = (V, E)$ die minimale Anzahl k von Farben, so dass sich die Knoten von G so mit k Farben färben lassen, dass benachbarte Knoten unterschiedlich gefärbt sind.

Asymptotische Notation

Obere Schranken: O -Notation

- ▶ Sei $g : \mathbb{N} \rightarrow \mathbb{R}$. Dann bezeichnet

$$O(g) := \{f : \mathbb{N} \rightarrow \mathbb{R} : \exists c > 0, n_0 \in \mathbb{N} \text{ mit } |f(n)| \leq c|g(n)| \forall n \geq n_0\}$$

die Menge aller Funktionen $f : \mathbb{N} \rightarrow \mathbb{R}$, für die zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt $|f(n)| \leq c|g(n)|$.

Bemerkungen

- ▶ Die asymptotische Notation vernachlässigt Konstanten und Terme niedrigerer Ordnung (wie z.B. die Terme $a_k n^k$ mit $k < m$ im Polynom).

Satz

- ▶ Für ein Polynom $f(n) = a_m n^m + \dots + a_1 n + a_0$ vom Grade m gilt $f \in O(n^m)$.

Die nachfolgende Tabelle zeigt, wie sechs typischen Funktionen anwachsen, wobei die Konstante gleich 1 gesetzt wurde. Wie man feststellt, zeigen die Zeiten vom Typ $O(n)$ und $O(n \log n)$ ein wesentlich schwächeres Wachstum also die anderen.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Ω - und Θ -Notation

- ▶ Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Dann ist $f \in \Omega(g)$, wenn zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt:

$$|f(n)| \geq c|g(n)|.$$

- ▶ $f \in \Theta(g)$, wenn es positive Konstanten $c_1, c_2 \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$.
- ▶ Falls $f \in \Theta(g)$, dann ist g sowohl eine obere als auch eine untere Schranke für f .

Beispiel: Sequentielle Suche

- ▶ Sei $f(n)$ die Anzahl von Vergleichen bei der sequentiellen Suche nach einem bestimmten Wert in einem unsortierten Array mit n Komponenten. Dann ist $f \in O(n)$, da man ja mit n Vergleichen auskommt.
- ▶ Andererseits muss man auch jede Komponente überprüfen, denn ihr Wert könnte ja der gesuchte Wert sein. Also $f \in \Omega(n)$ und damit $f \in \Theta(n)$.

Matrixmultiplikation

- ▶ Bei der Matrixmultiplikation von $n \times n$ Matrizen ergibt sich die Berechnung eines Eintrags c_{ij} von $C = A \cdot B$ gemäß $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$. Sie erfordert also n Multiplikationen und $n - 1$ Additionen.
- ▶ Insgesamt sind für ganz C also n^2 Einträge c_{ij} zu berechnen, und somit $n^2(n + n - 1) = 2n^3 - n^2 = O(n^3)$ arithmetische Operationen insgesamt auszuführen.
- ▶ Da jeder Algorithmus für die Matrixmultiplikation n^2 Einträge berechnen muss, folgt andererseits, dass jeder Algorithmus zur Matrixmultiplikation von zwei $n \times n$ Matrizen $\Omega(n^2)$ Operationen benötigt.
- ▶ Es klafft zwischen $\Omega(n^2)$ und $O(n^3)$ also noch eine “Komplexitätslücke”. Die schnellsten bekannten Algorithmen zur Matrixmultiplikation kommen mit $O(n^{2,376})$ Operationen aus.



Komplexitätstheorie

Was ist ein Problem?

Definition

- ▶ Ein **Problem** ist eine allgemeine Fragestellung, bei der mehrere Parameter offen gelassen sind und für die eine Lösung oder Antwort gesucht wird.
- ▶ Ein Problem ist dadurch definiert, dass alle seine Parameter beschrieben werden und dass genau angegeben wird, welche Eigenschaften eine Antwort (Lösung) haben soll.
- ▶ Sind alle Parameter eines Problems mit expliziten Daten belegt, dann sprechen wir von einem Problembeispiel (**Instanz**).

Beispiel

- ▶ Finde einen kürzesten Hamiltonischen Kreis.
- ▶ Offene Parameter: Anzahl Städte, Entfernungen

Was ist ein Algorithmus?

- ▶ Ein **Algorithmus** ist eine Anleitung zur schrittweisen Lösung eines Problems. Wir sagen, ein Algorithmus A löst ein Problem Π , falls A für alle Problembeispiele $I \in \Pi$, eine Lösung in einer endlichen Anzahl an Schritten findet.
- ▶ Ein Schritt ist eine elementare Operation: Addieren, Subtrahieren, Vergleichen, Multiplikation, Division.
- ▶ Die Laufzeit eines Algorithmus ist die Anzahl der Schritte, die notwendig sind zur Lösung des Problembeispiels.

Forschungsschwerpunkte zu Algorithmen

- ▶ Entwurf von Algorithmen
- ▶ Berechenbarkeit: Was kann durch einen Algorithmus berechnet werden?
- ▶ Komplexität von Algorithmen
- ▶ Korrektheit von Algorithmen



Komplexität von Algorithmen

Was ist Effizienz?

- ▶ Komplexitätstheorie
- ▶ Speicher- und Laufzeitkomplexität

Trivial

- ▶ Laufzeit eines Algorithmus hängt ab von der “Größe” der Eingabedaten
- ▶ Laufzeitanalyse erfordert eine Beschreibung, wie Problembeispiele dargestellt werden (Kodierungsschema)
- ▶ Notwendigkeit exakter Definitionen
 - ▶ geeignetes Rechnermodell → **Turing-Maschine**

Ganze Zahlen

- ▶ Ganze Zahlen werden binär dargestellt, d.h. wir schreiben

$$n = \pm \sum_{i=0}^k x_i \cdot 2^i,$$

mit $x_i \in \{0, 1\}$ und $k = \lfloor \log_2(|n|) \rfloor$.

- ▶ D.h. die Kodierungslänge $\langle n \rangle$ einer ganzen Zahl n ist gegeben durch die Formel

$$\langle n \rangle = \lceil \log_2(|n| + 1) \rceil + 1 = \lfloor \log_2(|n|) \rfloor + 2,$$

wobei $+1$ wegen des Vorzeichens $+$ oder $-$.

Rationale Zahlen

- ▶ Sei $r \in \mathbb{Q}$. Dann existieren $p \in \mathbb{Z}$ und $q \in \mathbb{Z}$, teilerfremd, mit $r = \frac{p}{q}$.

$$\langle r \rangle = \langle p \rangle + \langle q \rangle$$

Vektoren

- Für $x = (x_1, \dots, x_n)^T \in \mathbb{Q}^n$ ist

$$\langle x \rangle = \sum_{i=1}^n \langle x_i \rangle.$$

Matrizen

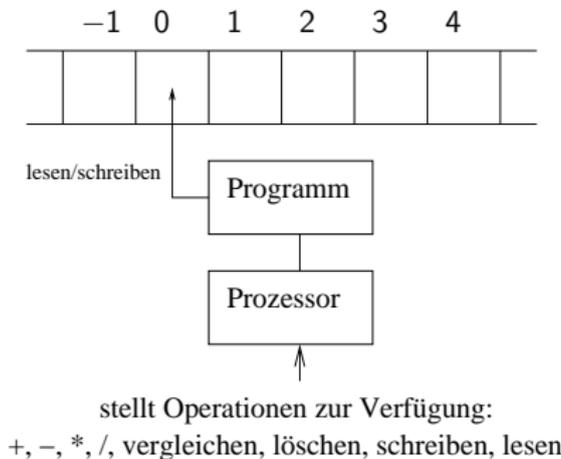
- Für $A \in \mathbb{Q}^{m \times n}$ ist

$$\langle A \rangle = \sum_{i=1}^m \sum_{j=1}^n \langle a_{ij} \rangle.$$

Nach Festlegung der Kodierungsvorschrift muss ein Rechnermodell entworfen werden, auf dem unsere Speicher- und Laufzeitberechnungen durchgeführt werden.

In der Komplexitätstheorie: **Turing-Maschine** (ein ganz normaler Computer)

Unendliches Band, auf dem Lese-/Schreiboperationen durchgeführt werden



Ablauf eines Algorithmus auf der Turing-Maschine



Algorithmus A soll Problembeispiel I des Problems Π lösen. Alle Problembeispiele liegen in kodierter Form vor.

Inputlänge

- ▶ Die Anzahl der Speicherplätze, die nötig sind, um I vollständig anzugeben, heißt **Inputlänge**, $\langle I \rangle$.

Der Algorithmus liest die Daten und beginnt dann, Operationen auszuführen, d.h. Zahlen zu berechnen, zu speichern, zu löschen.

Speicherbedarf

- ▶ Die Anzahl der Speicherplätze, die während der Ausführung des Algorithmus mindestens einmal benutzt werden, nennen wir **Speicherbedarf** von A zur Lösung von I .

Informell

- ▶ Die **Laufzeit** von A zur Lösung von I ist die Anzahl elementarer Operationen. Dazu zählen $+$, $-$, $*$, $/$, Vergleich, Löschen, Schreiben, Lesen von rationalen Zahlen.

Dies ist jedoch zu unpräzise!

- ▶ Zur Darstellung der entsprechenden Zahlen werden mehrere Bits benötigt.
- Für jede Operation müssen wir mehrere Bits zählen.

Korrekte Definition

- ▶ Die **Laufzeit** von A zur Lösung von I ist definiert durch die Anzahl elementarer Operationen, die A ausgeführt hat, um I zu lösen, multipliziert mit der größten Kodierungslänge der während der Berechnung aufgetretenen ganzen oder rationalen Zahl.

Laufzeitfunktion, Speicherplatzfunktion (worst-case)

Laufzeitfunktion, polynomielle Laufzeit

- ▶ Sei A ein Algorithmus zur Lösung eines Problems Π . Die Funktion $f_A : \mathbb{N} \rightarrow \mathbb{N}$,

$$f_A(n) = \max\{\text{Laufzeit von } A \text{ zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}$$

heißt **Laufzeitfunktion** von A .

- ▶ Der Algorithmus A hat eine **polynomielle Laufzeit** (kurz: A ist ein polynomialer Algorithmus), wenn es ein Polynom $p : \mathbb{N} \rightarrow \mathbb{N}$ gibt mit $f_A(n) \leq p(n)$ für alle $n \in \mathbb{N}$.

Speicherplatzfunktion, polynomieller Speicherbedarf

- ▶ Die **Speicherplatzfunktion** $s_A : \mathbb{N} \rightarrow \mathbb{N}$ von A ist definiert durch

$$s_A(n) = \max\{\text{Speicherbedarf von } A \text{ zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}.$$

- ▶ Der Algorithmus A hat polynomiellen Speicherbedarf, falls es ein Polynom $q : \mathbb{N} \rightarrow \mathbb{N}$ gibt mit $s_A(n) \leq q(n)$ für alle $n \in \mathbb{N}$.

Bemerkungen

- ▶ Nehmen wir an, wir ermitteln die Rechenzeit $f(n)$ für einen bestimmten Algorithmus. Die Variable n kann z.B. die Anzahl der Ein- und Ausgabewerte sein, ihre Summe, oder auch die Größe eines dieser Werte. Da $f(n)$ maschinenabhängig ist, genügt eine a priori Analyse nicht. Jedoch kann man mit Hilfe einer a priori Analyse ein g bestimmen, so dass $f \in O(g)$.
- ▶ Wenn wir sagen, dass ein Algorithmus eine Rechenzeit $O(g)$ hat, dann meinen wir damit folgendes: Wenn der Algorithmus auf unterschiedlichen Computern mit den gleichen Datensätzen läuft, und diese Größe n haben, dann werden die resultierenden Laufzeiten immer kleiner sein als eine Konstante mal $|g(n)|$. Bei der Suche nach der Größenordnung von f werden wir darum bemüht sein, das kleinste g zu finden, so dass $f \in O(g)$ gilt.



Die Klassen \mathcal{P} , \mathcal{NP} , \mathcal{NP} -vollständig

Was ist ein Entscheidungsproblem?

- ▶ Problem, das nur zwei mögliche Antworten besitzt, “ja” oder “nein”.
- ▶ Beispiele: “Ist n eine Primzahl?” oder “Enthält G einen Kreis?”

Die Klasse \mathcal{P} : informelle Definition

- ▶ Klasse der Entscheidungsprobleme, für die ein polynomialer Lösungsalgorithmus existiert

Die Klasse \mathcal{P} : formale Definition

- ▶ Gegeben ein Kodierungsschema E und ein Rechnermodell M .
- ▶ Π sei ein Entscheidungsproblem, wobei jede Instanz aus Π durch Kodierungsschema E kodierbar sei.
- ▶ Π gehört zur Klasse \mathcal{P} (bzgl. E und M), wenn es einen auf M implementierbaren Algorithmus zur Lösung aller Problembeispiele aus Π gibt, dessen Laufzeitfunktion auf M polynomial ist.

Motivation

- ▶ Enthält G einen Kreis? \rightarrow "einfach" $\rightarrow \mathcal{P}$
- ▶ Enthält G einen hamiltonischen Kreis? \rightarrow "schwieriger" $\rightarrow \mathcal{NP}$

Die Klasse \mathcal{NP} : informelle Definition

- ▶ Entscheidungsproblem Π gehört zur Klasse \mathcal{NP} , wenn es folgende Eigenschaft hat:
 - ▶ Ist die Antwort "ja" für $I \in \Pi$, dann kann Korrektheit dieser Aussage mit Hilfe eines geeigneten Zusatzobjekts in polynomialer Laufzeit überprüft werden.

Beispiel: "Enthält G einen hamiltonischen Kreis?"

- ▶ Geeignetes Zusatzobjekt wäre ein hamiltonischer Kreis.
- ▶ Es muss nun lediglich geprüft werden, ob der angegebene Kreis tatsächlich ein hamiltonischer Kreis von G ist.

Die Klasse \mathcal{NP} (2)

Formale Definition

- ▶ Entscheidungsproblem Π gehört zu \mathcal{NP} , wenn es die folgende Eigenschaften hat:
 - (a) Für jedes Problembeispiel $I \in \Pi$, für das die Antwort “ja” lautet, gibt es mindestens ein Objekt Q , mit dessen Hilfe die Korrektheit der “ja”-Antwort überprüft werden kann.
 - (b) Es gibt einen Algorithmus, der Problembeispiele $I \in \Pi$ und Zusatzobjekte Q als Input akzeptiert und der in einer Laufzeit, die polynomial in $\langle I \rangle$ ist, überprüft, ob Q ein Objekt ist, aufgrund dessen Existenz eine “ja”-Antwort für I gegeben werden muss.

Bemerkungen

- ▶ Es wird keine Aussage über die Berechenbarkeit eines geeigneten Q gemacht. Q kann auch geraten werden.
- ▶ Da der Überprüfungsalgorithmus Q lesen muss, muss $\langle Q \rangle$ polynomial in $\langle I \rangle$ sein!
- ▶ Es wird keine Aussage zu möglicher “nein”-Antwort gemacht. \rightarrow coNP

Charakterisierung besonders schwieriger Probleme in \mathcal{NP}



Polynomiale Transformation von Problemen

- ▶ Gegeben seien zwei Entscheidungsprobleme Π und Π' .
- ▶ Eine polynomiale Transformation von Π in Π' ist ein polynomialer Algorithmus, der, gegeben ein (kodierte) Problembeispiel $I \in \Pi$, ein (kodierte) Problembeispiel $I' \in \Pi'$ produziert, so dass folgendes gilt:
 - ▶ Die Antwort auf I ist genau dann "ja", wenn die Antwort auf I' "ja" ist.

Die Klasse \mathcal{NP} -vollständig

- ▶ Ein Entscheidungsproblem Π heißt **\mathcal{NP} -vollständig**, falls $\Pi \in \mathcal{NP}$ und falls jedes andere Problem aus \mathcal{NP} polynomial in Π transformiert werden kann.

Bemerkung

- ▶ Falls ein \mathcal{NP} -vollständiges Entscheidungsproblem Π in polynomialer Zeit gelöst werden kann, so gilt das auch für jedes andere Problem aus \mathcal{NP} :

$$\Pi \text{ } \mathcal{NP} \text{-vollständig und } \Pi \in \mathcal{P} \Rightarrow \mathcal{P} = \mathcal{NP}.$$

- ▶ Diese Eigenschaft zeigt, dass (bzgl. polynomialer Lösbarkeit) kein Problem in \mathcal{NP} schwieriger ist als ein \mathcal{NP} -vollständiges.

Existieren \mathcal{NP} -vollständige Probleme? Ja.

- ▶ 3-SAT: $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots = \text{TRUE}$

Offene Fragen

$$\begin{aligned}\mathcal{P} &= \mathcal{NP} \cap \text{co}\mathcal{NP} \\ \mathcal{NP} &= \text{co}\mathcal{NP} \\ \mathcal{P} &\neq \mathcal{NP}\end{aligned}$$



Komplexität von Optimierungsproblemen

Transformation in Entscheidungsproblem

- ▶ Sei O ein Maximierungsproblem (Minimierungsproblem). Man legt zusätzlich zur Instanz I noch eine Schranke B fest und stellt die Frage:
 - ▶ Gibt es für I eine Lösung, deren Wert nicht kleiner (größer) als B ist?
- ▶ Damit ist ein Optimierungsproblem polynomial in ein Entscheidungsproblem transformierbar.

Die Klasse \mathcal{NP} -schwer: formale Definition

- ▶ Ein Optimierungsproblem O heißt **\mathcal{NP} -schwer**, falls es ein \mathcal{NP} -vollständiges Entscheidungsproblem Π gibt, so dass Π in polynomialer Zeit gelöst werden kann, wenn O in polynomialer Zeit gelöst werden kann.

Beispiele

- ▶ Rucksackproblem, Maschinenbelegung, stabile Menge maximaler Kardinalität, Clique maximaler Kardinalität, minimale Knotenüberdeckung

Traveling Salesman Problem

- ▶ Input: Instanz I ist charakterisiert durch eine Distanzmatrix $c_{ij} \in \mathbb{Z} \forall i, j \in V$, $|V| = n$.
- ▶ Problem: “Finde einen bzgl. c minimalen Hamiltonischen Kreis in K_n .”
- ▶ TSP-Entscheidungsproblem:
 - ▶ Input wie beim TSP, $B \in \mathbb{Z}$.
 - ▶ Problem: “Existiert eine TSP-Tour der Länge $\leq B$?”

Lemma

- ▶ Polynomialer Algorithmus zur Lösung des TSP-Entscheidungsproblems würde polynomialen Algorithmus zur Lösung des TSP implizieren.

Beweis

- ▶ Länge eines hamiltonischen Kreises in K_n ist ein ganzzahliger Wert im Intervall $[ns, nt]$, wobei $s = \min\{c_{ij} : i, j \in V, i \neq j\}$ und $t = \max\{c_{ij} : i, j \in V, i \neq j\}$.
- ▶ Rufe TSP-Entscheidungsproblem mit $B = \lfloor \frac{n(s+t)}{2} \rfloor$ auf, passe obere Schranke nt oder untere Schranke ns je nach Ergebnis an und iteriere.
- ▶ Insgesamt reichen demnach $\log(nt - ns)$ Iterationen aus. □



Datenstrukturen zur Kodierung von Graphen

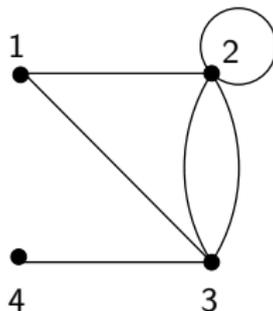
Definition

- ▶ Ist $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten, so sieht die Kantenliste wie folgt aus:

$$n, m, a_1, e_1, a_2, e_2, \dots, a_m, e_m,$$

wobei a_i, e_i die beiden Endknoten der Kante i sind.

- ▶ Die Reihenfolge des Aufführens der Endknoten von i bzw. den Kanten ist beliebig.
- ▶ Bei Schleifen wird der Endknoten zweimal aufgelistet.
- ▶ Bei der Bogenliste eines Digraphen verfahren wir genauso, müssen jedoch darauf achten, dass ein Bogen immer zunächst durch seinen Anfangs- und dann durch seinen Endknoten repräsentiert wird.



Kantenliste

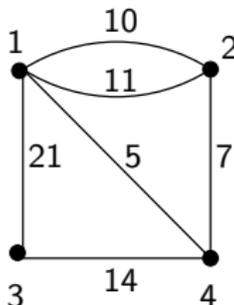
- ▶ Dieser Graph hat die folgende Kantenliste:

4, 6, 1, 2, 2, 3, 4, 3, 3, 2, 2, 2, 1, 3.

- ▶ Man beachte, dass hier die Reihenfolge der Knoten beliebig ist, da es sich um ungerichtete Kanten handelt.

Definition

- ▶ Haben die Kanten oder Bögen Gewichte, so repräsentiert man eine Kante (einen Bogen) entweder durch Anfangsknoten, Endknoten, Gewicht oder macht eine Kanten- bzw. Bogenliste wie oben und hängt an diese noch eine Liste mit den m Gewichten der Kanten $1, 2, \dots, m$ an.
- ▶ Der Speicheraufwand einer Kanten- bzw. Bogenliste beträgt $2(m + 1)$ Speicherzellen, eine Liste mit Gewichten erfordert $3m + 2$ Zellen.

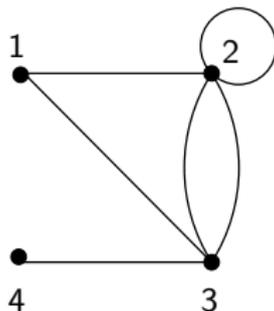


Kantenliste mit Gewichten

- ▶ Dieser gewichtete Graph ist in den beiden folgenden Kantenlisten mit Gewichten gespeichert:
 - ▶ 4,6,1,2,10,1,2,11,2,4,7,4,3,14,3,1,21,1,4,5
 - ▶ 4,6,1,2,1,2,2,4,3,4,1,4,1,3,11,10,7,14,5,21

Definition

- ▶ Sei $G = (V, E)$ ein ungerichteter Graph mit $V = \{1, \dots, n\}$, so ist die symmetrische Matrix $A \in \mathbb{R}^{n \times n}$ mit
 - ▶ $a_{ij} = a_{ji} =$ Anzahl der Kanten, die i mit j , $i \neq j$, verbinden,
 - ▶ $a_{ii} = 2 \cdot$ (Anzahl der Schleifen, die i enthalten), $i = 1, \dots, n$,die **Adjazenzmatrix** von G .
- ▶ Wegen der Symmetrie genügt es, die obere Dreiecksmatrix von A zu speichern.
- ▶ Hat G Kantengewichte und ist G einfach, so setzt man
 - ▶ $a_{ij} =$ Gewicht der Kante, wenn $ij \in E$,
 - ▶ $a_{ij} = 0, -\infty, +\infty$, anderenfalls.
- ▶ Der Speicheraufwand für Adjazenzmatrizen beträgt n^2 Zellen (bzw. $\frac{n(n-1)}{2}$ für obere Dreiecksmatrix).



Adjazenzmatrix

- Die Adjazenzmatrix dieses Graphen ist:

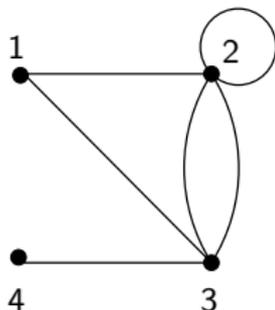
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Definition

- ▶ Die Adjazenzmatrix eines Digraphen $D = (V, A)$ mit $V = \{1, \dots, n\}$ ohne Schleifen ist definiert durch
 - ▶ $a_{ii} = 0, i = 1, \dots, n,$
 - ▶ $a_{ij} =$ Anzahl der Bögen in A mit Anfangsknoten i und Endknoten j .
- ▶ Bogengewichte einfacher Digraphen werden wie im ungerichteten Fall behandelt.
- ▶ Der Speicheraufwand für Adjazenzmatrizen beträgt n^2 Zellen (bzw. $\frac{n(n-1)}{2}$ für obere Dreiecksmatrix).

Definition

- Speichert man für einen Graphen $G = (V, E)$ die Anzahl von Knoten und für jeden Knoten seinen Grad und die Namen der Nachbarknoten, so nennt sich eine solche Datenstruktur **Adjazenzliste** von G .



Adjazenzliste

	Knotennummer	Grad	Nachbarknoten
4	1	2	2, 3
	2	4	1, 3, 3, 2
	3	4	1, 2, 2, 4
	4	1	3

- Die Adjazenzliste dieses Graphen ist:

4, 1, 2, 2, 3, 2, 4, 1, 3, 3, 2, 3, 4, 1, 2, 2, 4, 4, 1, 3.

Adjazenzlisten für Digraphen

- ▶ Bei Digraphen geht man analog vor: nur speichert man lediglich die Nachfolger eines Knotens (jeder Bogen kommt also nur einmal vor).
- ▶ Die Speicherung der Adjazenzliste eines Graphen erfordert $2n + 1 + 2m$ Speicherplätze, ihre Speicherung bei einem Digraphen erfordert $2n + 1 + m$ Speicherplätze.

Vor- und Nachteil von Adjazenzlisten

- ▶ kompakt, “gut für dünn besetzte Graphen”,
- ▶ aber schlecht: Suchen, ob Kante ij in G enthalten ist.

Finden von aufspannenden Wäldern

Depth-First Search Algorithmus

Input/Output

- ▶ **Input:** Einfacher Graph $G = (V, E)$ in Form einer Adjazenzliste, d.h. für jeden Knoten v ist eine Nachbarliste $N(v)$ gegeben.
- ▶ **Output:** Kantenmenge T (= aufspannender Wald) und Kantenmenge B (mit $B \cup T = E$ und $B \cap T = \emptyset$)

Idee und Ziel des DFS-Algorithmus

- ▶ Berechnet für einen Graphen G :
 - ▶ die Anzahl der Komponenten,
 - ▶ entscheidet, ob G Kreis enthält und
 - ▶ findet für jede Komponente einen aufspannenden Baum.
- ▶ Tiefensuche: “so schnell wie möglich einen Knoten verlassen”
- ▶ rekursives Hinzufügen von Kanten zum Wald T oder zu kreisschließenden Kanten B
- ▶ DFS-Algorithmus teilt die Kanten des Graphen in zwei Teilmengen:
 - ▶ Kante xy heißt Vorwärtskante, wenn wir im Verfahren von einem markierten Knoten x entlang xy zum Knoten y gehen und y markieren.
 - Kante des aufspannenden Waldes
 - ▶ Anderfalls heißt xy Rückwärtskante.
 - Kante würde Kreis schließen

Algorithmus

- ▶ Alle Knoten seien unmarkiert.
- ▶ Setze $T := \emptyset$, $B := \emptyset$.
- ▶ Für alle $v \in V$ führe aus
 - ▶ Ist v unmarkiert, dann **CALL SEARCH**(v).
 - ▶ (**Bemerkung:** Es existieren genau soviele Zusammenhangskomponenten in G wie hier **SEARCH**(v) aufgerufen wird.)
- ▶ Gib T und B aus.

Funktion **SEARCH**(v)

- ▶ Markiere v .
- ▶ Für alle Knoten $w \in N(v)$ führe aus:
 - ▶ Ist w markiert und $vw \notin T$, setze $B := B \cup \{vw\}$.
 - ▶ Ist w unmarkiert, setze $T := T \cup \{vw\}$ und **CALL SEARCH**(w).

Laufzeit

- ▶ Laufzeit des Verfahrens: $O(|V| + |E|)$, d.h. polynomiell

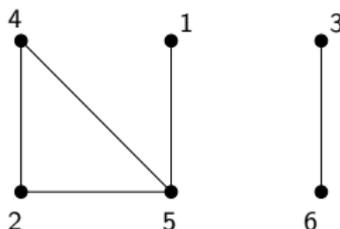
Korrektheit

- ▶ Die Menge aller Vorwärtskanten ist ein Wald von G , der in jeder Komponente einen aufspannenden Baum bildet.
 - Beweis als Übungsaufgabe
- ▶ G enthält einen Kreis $\Leftrightarrow B \neq \emptyset$.

Datenstruktur

- ▶ DFS-Algorithmus ist konzipiert für die Datenstruktur einer Adjazenzliste!

Beispiel für DFS-Algorithmus



Adjazenzliste (in Tabellenform)

Dieser Graph hat die folgende Adjazenzliste:

	Knotennummer	Grad	Nachbarknoten
6	1	1	5
	2	2	4, 5
	3	1	6
	4	2	2, 5
	5	3	1, 2, 4
	6	1	3

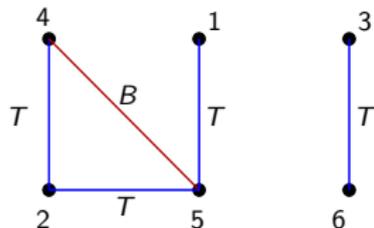
Beispiel für DFS-Algorithmus (2)

- ▶ $1 \rightarrow 5 \rightarrow 2 \rightarrow 4$
- ▶ $4 \rightarrow 2$ ($24 \in T$)
- ▶ $4 \rightarrow 5$
- ▶ Knoten 4 erledigt, zurück zu Knoten 2
- ▶ $2 \rightarrow 5$ ($52 \in T$)
- ▶ Knoten 2 erledigt, zurück zu Knoten 5
- ▶ $5 \rightarrow 1$ ($15 \in T$)
- ▶ Knoten 5 erledigt, zurück zu Knoten 1, Knoten 1 erledigt, Komponente erledigt
- ▶ $3 \rightarrow 6$
- ▶ Knoten 6 erledigt, zurück zu Knoten 3, Knoten 3 erledigt, Komponente erledigt

$$T = \{15, 52, 24\}$$

$$B = \{45\}$$

$$T = \{15, 52, 24, 36\}$$





Finden von gewichts-maximalen Wäldern

Input/Output

- ▶ **Input:** Graph $G = (V, E)$ mit Kantengewichten c_e , für alle $e \in E$.
- ▶ **Output:** Ein Wald $W \subseteq E$ mit maximalem Gewicht $c(W)$.

Algorithmus

1. (Sortieren)
 $c(e_1) \geq c(e_2) \geq \dots \geq c(e_K) > 0 \geq c(e_{K+1}) \geq \dots \geq c(e_{|E|})$.
2. (Initialisierung)
Setze $W := \emptyset$.
3. **Für** $i = 1, \dots, K$ führe durch:
 Falls $W \cup \{e_i\}$ keinen Kreis enthält,
 setze $W := W \cup \{e_i\}$.
4. (Ausgabe)
Gib W aus.