

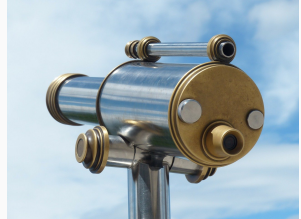
Algorithmen und Datenstrukturen

Kapitel 7: Komplexitätsklassen & NP-Vollständigkeit

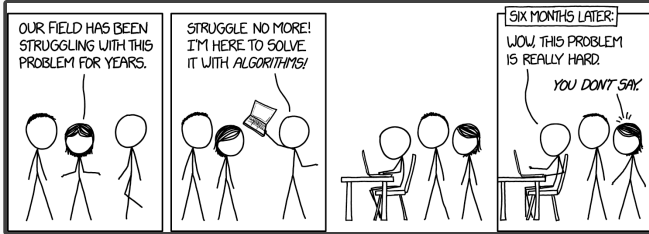
Prof. Dr. Peter Kling

Wintersemester 2020/21

- 1 Formalisierung von Problemen
- 2 Standard-Komplexitätsklassen
- 3 NP-vollständige Probleme
- 4 The End?



Effiziente Algorithmen vs Komplexität



XKCD Webcomic; klickt [hier](#) für das Original

Algorithmen sind die Geheimwaffe des Informatikers!

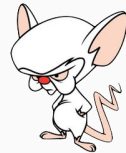
- haben gelernt, Probleme zu formalisieren...
- ...und für sie Algorithmen zu entwickeln & zu analysieren

FG1/ETI

A&D



Wer soll uns jetzt
noch stoppen?



Effiziente Algorithmen vs Komplexität



Algorithmen sind die Geheimwaffe des Informatikers!

- haben gelernt, Probleme zu formalisieren...
- ...und für sie Algorithmen zu entwickeln & zu analysieren



Wer soll uns jetzt noch stoppen?



- Wer die Referenz nicht kennt: **Pinky and the Brain** (definitiv Klausur-relevant!)

Wie zeigt man, dass **Problem X** komplex ist?

Variante 1

- unbedingter mathematischer Beweis der Komplexität
 - z. B.: Problem **X** **nicht** oder **nicht in Zeit $o(2^n)$** lösbar ist
- ↪ manchmal machbar, aber typischerweise extrem schwer

Variante 2

- bedingter mathematischer Beweis der Komplexität
 - angenommen Problem **Y** ist „bekanntermaßen“ schwer
 - zeige: **X** effizient lösbar \implies **Y** effizient lösbar
- ↪ oft machbar, gar nicht mal soooo schwer

Wie können wir so eine bedingte
Komplexität formalisieren?

└ Wie zeigt man, dass Problem X komplex ist?

Wie zeigt man, dass Problem X komplex ist?

Variante 1

- unbedingter mathematischer Beweis der Komplexität
- z.B.: Problem X nicht oder nicht in Zeit $o(2^n)$ lösbar ist
- manchmal machbar, aber typischerweise extrem schwer

Variante 2

- bedingter mathematischer Beweis der Komplexität
- angenommen Problem Y ist „bekanntermaßen“ schwer
- zeige: X effizient lösbar $\iff Y$ effizient lösbar
- oft machbar, gar nicht mal soooo schwer

Wie können wir so eine bedingte Komplexität formalisieren?

- „unbedingt“: keine Voraussetzungen außer typischer mathematische Axiome
- wir **reduzieren** Problem Y auf Problem X („Wenn wir X effizient lösen könnten, dann könnten wir auch Y effizient lösen.“)

Was sind **einfache Probleme**? Was sind **schwere Probleme**?

Einfach

- finde **kürzesten** Pfad in Graph
- finde **Euler**-Kreis
 - Kreis, der alle Kanten enthält
- **2**-Färbbarkeit von Graphen
 - adjazente Knoten benötigen unterschiedl. Farben

Schwer

- finde **längsten** Pfad in Graph
- finde **Hamilton**-Kreis
 - Kreis, der alle Knoten enthält
- **3**-Färbbarkeit von Graphen
 - adjazente Knoten benötigen unterschiedl. Farben
- **k**-Clique
 - vollständiger Teilgraph mit **k** Knoten
- **k**-Independent-Set
 - Teilgraph mit **k** Knoten ohne Kanten
- Partition
 - Teile eine Menge ganzer Zahlen...
 - ...in zwei Mengen gleicher Summe

Was sind einfache Probleme? Was sind schwere Probleme?

- LZ kürzester Pfad: $O(|E|)$
- LZ Euler-Kreis: $O(|E|)$
- LZ 2-Färbbarkeit: $O(|E|)$

Einfach

- finde **kürzesten** Pfad in Graph
- finde **Euler-Kreis**
 - Kreis, der alle Kanten enthält
- **2-Färbbarkeit** von Graphen
 - adjazente Knoten benötigen unterschiedl. Farben

Schwer

- finde **längsten** Pfad in Graph
- finde **Hamilton-Kreis**
 - Kreis, der alle Knoten enthält
- **3-Färbbarkeit** von Graphen
 - adjazente Knoten benötigen unterschiedl. Farben
- **k-Clique**
 - vollständiger Teilgraph mit k Knoten
- **k-Independent-Set**
 - Teilgraph mit k Knoten ohne Kanten
- **Partition**
 - Teile eine Menge ganzer Zahlen...
 - ...in zwei Mengen gleicher Summe

1) Formalisierung von Problemen

Optimierungsprobleme & Entscheidungsprobleme

Optimierungsproblem

Finde gültige Lösung mit **optimalem Wert**.

Entscheidungsproblem

Entscheide, ob eine gültige Lösung **existiert**.

SHORTESTPATH:

Kürzester Pfad von s nach t ?

k -PATH:

Pfad der Länge $\leq k$ von s nach t ?

Wir beschränken uns im Folgenden
auf Entscheidungsprobleme!

Warum?

Reduktion: SHORTESTPATH \rightarrow k -PATH

- Existiert Pfad der Länge $\leq n/2$?
 - ja \rightsquigarrow Existiert Pfad der Länge $\leq n/4$? ...
 - nein \rightsquigarrow Existiert Pfad der Länge $\leq 3n/4$? ...
- also: Binäre Suche!

Algorithmen und Datenstrukturen

Formalisierung von Problemen

Optimierungsprobleme & Entscheidungsprobleme

Optimierungsprobleme & Entscheidungsprobleme	
Optimierungsproblem Finde gültige Lösung mit optimalen Wert .	Entscheidungsproblem Entscheide, ob eine gültige Lösung existiert .
SHORTESTPATH: Kürzester Pfad von s nach t ?	k-PATH: Pfad der Länge $\leq k$ von s nach t ?
<div style="border: 1px solid orange; padding: 5px; text-align: center;"> Wir beschränken uns im Folgenden auf Entscheidungsprobleme! </div>	
Reduktion: SHORTESTPATH \rightarrow k-PATH <ul style="list-style-type: none"> Existiert Pfad der Länge $\leq n/3$? ja \rightarrow Existiert Pfad der Länge $\leq n/4$... nein \rightarrow Existiert Pfad der Länge $\leq 9n/47$... also: Binäre Suche! 	

- Optimierungsprobleme definieren immer eine zu optimierende Zielfunktion
- Optimierung kann entweder **Maximierung** oder **Minimierung** bedeuten
- Generell betrachtet man zu einem gegebenen Optimierungsproblem typischerweise ein entsprechendes **k-Threshold Problem**, welches nach der Existenz einer Lösung mit Kosten $\leq k$ (bei Minimierungsproblem) bzw. nach der Existenz einer Lösung mit Wert $\geq k$ (bei Maximierungsproblem) fragt.
- Aus einem Algorithmus für diese Entscheidungsvariante des Optimierungsproblems kann dann typischerweise das Optimierungsproblem mittels binärer Suche gelöst werden (auf Kosten eines zusätzlichen logarithmischen Faktors in der Laufzeit).

Abstraktes Entscheidungsproblem $Q = (I, f)$

- Menge I von möglichen Instanzen
- Funktion $f: I \rightarrow \{0, 1\}$
- für $i \in I$: $f(i) = 1 \iff$ Instanz i hat gültige Lösung

Beispiel: k -PATH

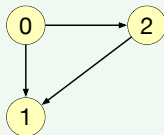
- Menge $I = \{ (G, s, t, k) \mid G = (V, E) \text{ ist unger. Graph} \wedge s, t \in V \wedge k \in \mathbb{N} \}$
- Funktionswert von f für Instanz $i = (G, s, t, k)$ ist
$$f(i) = \begin{cases} 1 & , \text{ falls in } G \text{ existiert Pfad } p \text{ von } s \text{ nach } t \text{ mit } |p| \leq k \\ 0 & , \text{ sonst} \end{cases}$$

...zur Kodierung von Problemen

- stellen Menge I als Menge aller möglichen Binärstrings $\{0,1\}^*$ dar
- damit wird f zu einer Funktion $f: \{0,1\}^* \rightarrow \{0,1\}$
- beachte: nicht alle Binärstrings müssen einer Instanz entsprechen!
 \rightsquigarrow für solche $s \in \{0,1\}^*$ setzen wir $f(s) = 0$

Beispiel: Mögliche Kodierung eines Graphen

- 00: binär 0
- 01: binär 1
- 10: nächstes Listenelement
- 11: nächste Liste
- Strings **ungerader** Länge entsprechen **keinem** Graphen!



11 0000 10 0001 10 0100 11 0001 11 0100 10 0001

Würden gerne die konkrete Kodierung ignorieren...

- ...dürfen wir aber nicht!

Beispiel: Algorithmus mit einziger Eingabe $k \in \mathbb{N}$

- die Laufzeit des Algorithmus sei $\Theta(k)$
- für unäre Kodierung: (z. B. $k = 5$ als 11111)
 - kodierte Eingabelänge $n = k$
 - also Laufzeit $\Theta(k) = \Theta(n)$ **linear** in n
- Für binäre Kodierung: (z. B. $k = 5$ als 101)
 - kodierte Eingabelänge $n = \lfloor \log k \rfloor + 1$
 - also Laufzeit $\Theta(k) = \Theta(2^n)$ **exponentiell** in n

ABER

Zwei Kodierungen heißen **polynomialzeit-äquivalent**, wenn sie sich in polynomieller Zeit ineinander umrechnen lassen.

Algorithmen und Datenstrukturen

Formalisierung von Problemen

Einfluss der Kodierung

- beachte: Polynome sind abgeschlossene unter **Verkettung**. Das heißt sind f und g Polynome, so ist auch $h: x \mapsto f(g(x))$ ein Polynom
- wir gehen im Folgenden immer von einer passenden, binären Standard-Kodierung aus
- insbesondere sei Kodierung einer natürlichen Zahl polynomialzeit-äquivalent zu binärer Kodierung...
- ...und Mengen/Listen z. B. polynomialzeit-äquivalent zu Kodierung wie Adjazenzlisten im Beispiel der vorherigen Folie

Würden gerne die konkrete Kodierung ignorieren...

- ...dürfen wir aber nicht!

Beispiel: Algorithmus mit einziger Eingabe $k \in \mathbb{N}$

- die Laufzeit des Algorithmus sei $\Theta(k)$
- für unäre Kodierung (z. B. $k = 5$ als 11111)
 - kodierte Eingabelänge $n = k$
 - also Laufzeit $\Theta(k) = \Theta(n)$ *linear* in n
- Für binäre Kodierung (z. B. $k = 5$ als 101)
 - kodierte Eingabelänge $n = \lceil \log k \rceil + 1$
 - also Laufzeit $\Theta(k) = \Theta(2^n)$ *exponentiell* in n

ABER

Zwei Kodierungen heißen **polynomialzeit-äquivalent**, wenn sie sich in polynomialer Zeit ineinander umrechnen lassen.

2) Standard-Komplexitätsklassen

Definition 7.1

Eine Funktion $f: \{0,1\}^* \rightarrow \{0,1\}^*$ heißt **polynomialzeit-berechenbar**, wenn es einen Algorithmus A und ein $c \in \mathbb{N}$ gibt, so dass A unter Eingabe $x \in \{0,1\}^*$ den Wert $f(x)$ in Zeit $O(|x|^c)$ berechnet.

Definition 7.2

Die **Komplexitätsklasse P** ist die Menge aller (konkret encodierten) Entscheidungsprobleme $f: \{0,1\}^* \rightarrow \{0,1\}$, so dass f polynomialzeit-berechenbar ist.

- für solche Probleme können wir also **effizient** (in Polynomialzeit)...
- ...**selbst entscheiden**, ob eine Lösung existiert

Algorithmen und Datenstrukturen

Standard-Komplexitätsklassen

Komplexitätsklasse P

Definition 7.1

Eine Funktion $f: \{0,1\}^* \rightarrow \{0,1\}^*$ heißt **polynomialzeit-berechenbar**, wenn es einen Algorithmus A und ein $c \in \mathbb{N}$ gibt, so dass A unter Eingabe $x \in \{0,1\}^n$ den Wert $f(x)$ in Zeit $O(n^c)$ berechnet.

Definition 7.2

Die **Komplexitätsklasse P** ist die Menge aller (konkret encodierten) Entscheidungsprobleme $f: \{0,1\}^* \rightarrow \{0,1\}$, so dass f polynomialzeit-berechenbar ist.

- für solche Probleme können wir also **effizient** (in Polynomialzeit)...
- ...**selbst entscheiden**, ob eine Lösung existiert

- in Definition 7.1 bezeichnet $|x|$ die Länge des Bitstrings x
- einfacher ausgedrückt: f ist polynomialzeit-berechenbar, wenn es einen Algorithmus mit polynomieller Laufzeit gibt, der $f(x)$ für jedes x berechnet

Definition 7.3

Eine Funktion $f: \{0,1\}^* \rightarrow \{0,1\}^*$ heißt **polynomialzeit-verifizierbar**, wenn:

- Für jede **Eingabe** $x \in \{0,1\}^*$ existiert **Zertifikat** $y \in \{0,1\}^*$ mit $|y| = O(|x|^k)$.
- Es gibt einen Polynomialzeit-Algorithmus V mit $V(x, y) = 1 \iff f(x) = 1$.

Definition 7.4

Die **Komplexitätsklasse NP** ist die Menge aller (konkret encodierten) Entscheidungsprobleme $f: \{0,1\}^* \rightarrow \{0,1\}$, so dass f polynomialzeit-verifizierbar ist.

- für solche Probleme können wir also **effizient** (in Polynomialzeit)...
- ...ein Zertifikat („Beweis“) über die Existenz einer Lösung **überprüfen**

Algorithmen und Datenstrukturen

└ Standard-Komplexitätsklassen

└ Komplexitätsklasse NP

- V steht hier für Verifier

Definition 7.3

Eine Funktion $f: \{0,1\}^* \rightarrow \{0,1\}^*$ heißt **polynomialzeit-verifizierbar**, wenn:

- Für jede Eingabe $x \in \{0,1\}^*$ existiert Zertifikat $y \in \{0,1\}^*$ mit $|y| = O(|x|^k)$.
- Es gibt einen Polynomialzeit-Algorithmus V mit $V(x, y) = 1 \iff f(x) = 1$.

Definition 7A

Die Komplexitätsklasse **NP** ist die Menge aller (konkret encodierten) Entscheidungsprobleme $f: \{0,1\}^* \rightarrow \{0,1\}$, so dass f polynomialzeit-verifizierbar ist.

- für solche Probleme können wir also **effizient** (in Polynomialzeit)...
- ...ein Zertifikat („Beweis“) über die Existenz einer Lösung **überprüfen**

- betrachte Menge NP
- offensichtlich gilt $P \subseteq NP$
- gilt $NP \subseteq P$?
 - unklar, viele Forscher (nicht alle!) tendieren zu nein
 - dahinterstehende Frage:
„Ist das Finden einer Lösung schwerer als das Überprüfen einer Lösung?“

Was sind die schwierigsten
Probleme in NP?

Algorithmen und Datenstrukturen

Standard-Komplexitätsklassen

\mathcal{P} vs \mathcal{NP}

- betrachte Menge \mathcal{NP}
- offensichtlich gilt $\mathcal{P} \subseteq \mathcal{NP}$
- gilt $\mathcal{NP} \subseteq \mathcal{P}$?
 - unklar, viele Forscher (nicht alle) tendieren zu nein
 - dahinterstehende Frage
„Ist das Finden einer Lösung schwerer als das Überprüfen einer Lösung?“

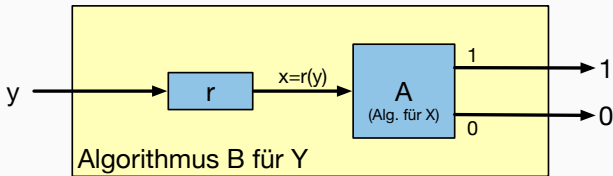
Was sind die schwierigsten Probleme in \mathcal{NP} ?

- intuitiv: Wenn ich etwas lösen kann, muss ich die Lösung auch überprüfen können.
- formal: Wähle Zertifikat beliebig (z.B. Nullstring) und $V(x, y) = A(x)$, wobei A der Polynomialzeit-Algorithmus ist der f berechnet.
- Wikipedia ist wie (fast) immer ein guter Ausgangspunkt, um weiterführende Literatur zu finden!

Wie zeigt man, dass X mindestens so schwer wie Y ist?

Polynomialzeit-Reduktion

- $f_X, f_Y: \{0,1\}^* \rightarrow \{0,1\}$ zu Entscheidungsproblemen X und Y
- suche eine Reduktionsfunktion $r: \{0,1\}^* \rightarrow \{0,1\}^*$ so dass:
 - r ist polynomialzeit-berechenbar
 - $f_Y(y) = 1 \iff f_X(r(y)) = 1$
- gibt es solch ein r , so heißt Y polynomialzeit-reduzierbar auf X ($Y \leq_p X$)
- Intuition:
 - r erlaubt es Eingaben von Y mit Algorithmus für X zu lösen



$$\rightsquigarrow X \in P \implies Y \in P$$

Definition 7.5

Ein Entscheidungsproblem X heißt **NP-vollständig**, falls:

- (a) $X \in \text{NP}$ und
- (b) für alle $Y \in \text{NP}$ gilt: $Y \leq_p X$.

Gilt die zweite Bedingung (aber nicht notwendigerweise die erste), so heißt X **NP-schwer**.

- bezeichnen Menge solcher Entscheidungsprobleme als **NPC**

Theorem 7.1

Falls X NP-vollständig ist und $X \in P$, dann gilt $P = \text{NP}$.

Beweis.

- falls $A \in P$, kann jedes $Y \in P$ über Reduktionsfunktion $r...$
- ...in Polynomialzeit gelöst werden (vorherige Folie)



Algorithmen und Datenstrukturen

Standard-Komplexitätsklassen

NP-Vollständigkeit

- NPC: steht für NP-complete

Definition 2.5

Ein Entscheidungsproblem X heißt **NP-vollständig**, falls:

- (a) $X \in \text{NP}$ und
- (b) für alle $Y \in \text{NP}$ gilt: $Y \leq_p X$.

Gilt die zweite Bedingung (aber nicht notwendigerweise die erste), so heißt X **NP-schwer**.

- bezeichnen Menge solcher Entscheidungsprobleme als **NPC**

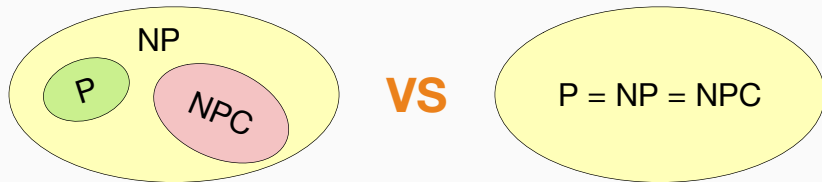
Theorem 2.1

Falls X NP-vollständig ist und $X \in P$, dann gilt $P = \text{NP}$.

Beweis.

- falls $A \in P$, kann jedes $Y \in P$ über Reduktionsfunktion r_{xy}
- ...in Polynomialzeit gelöst werden (vorherige Folie) □

Die Frage der Komplexitätstheorie



3) NP-vollständige Probleme

Wie zeigt man, dass **Problem X in NPC** liegt?

Referenzproblem + Reduktion

1. suche „passendes“ Problem **Y**, das **erwiesenermaßen** in NPC liegt
2. beweise, dass **X mindestens so schwer** wie **Y** ist

Zum 1. Schritt:

- erfordert Erfahrung, Übung, Suche, gute Quellen
- aber: Welches war das **erste** NP-vollständige Problem?

Satisfiability-Problem (SAT)

- Eingabe: logischer Ausdruck in konjunktiver Normalform
- Ausgabe: **1** \iff existiert erfüllende Belegung
- Beispiel: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee \neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$
- Satz von Cook und Levin: **SAT** \in NPC

Algorithmen und Datenstrukturen

NP-vollständige Probleme

Wie zeigt man, dass **Problem X** in **NPC** liegt?

Wie zeigt man, dass **Problem X** in **NPC** liegt?

Referenzproblem • Reduktion

1. Suche „passendes“ Problem Y , das **erwünschtenmaßen** in NPC liegt
2. beweise, dass X **mindestens so schwer** wie Y ist.

Zum 1. Schritt:

- erfordert Erfahrung, Übung, Suche, gute Quellen
- aber: Welches war das **erste** NP-vollständige Problem?

Satisfiability-Problem (SAT)

- Eingabe: logischer Ausdruck in konjunktiver Normalform
- Ausgabe: $i \iff$ existiert erfüllende Belegung

- Beispiel: $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$
- Satz von Cook und Levin: SAT \in NPC

bei der Beispiel SAT-Formel wäre die Ausgabe 1, da $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$ eine erfüllende Belegung ist

- gute Quellen für NP-vollständige Probleme: [Complexity Zoo](#), Buch [Computers and Intractability: A Guide to the Theory of NP-Completeness](#) von Michael R. Garey und David S. Johnson, [Wikipedia](#), [Karp's 21 Problems](#)
- weitere Informationen zum Satz von Cook und Levin gibt es zum Beispiel auf [Wikipedia](#)

Warum ist SAT ein gutes „erstes“ NPC-Problem?

Zu zeigen:

1. $SAT \in NP$

- Zertifikat: erfüllende Belegung
- Überprüfung: durch Einsetzen der Belegung

2. für alle $Y \in NP$ gilt: $Y \leq_p SAT$

- Beweis recht technisch
- Grundidee: Algorithmen laufen auf einer RAM...
...eine RAM kann als **logischer Schaltkreis** beschrieben werden!
↪ simuliere Algorithmus auf RAM durch Boole'sche Formel

└ Warum ist SAT ein gutes „erstes“ NPC-Problem?

Zu zeigen:

1. $SAT \in NP$
 - Zerifikat: erfüllende Belegung
 - Überprüfung: durch Einsetzen der Belegung
2. für alle $Y \in NP$ gilt: $Y \leq_p SAT$
 - Beweis recht technisch
 - Grundidee: Algorithmen laufen auf einer RAM...
 - ...wobei RAM kann als logischer Schaltkreis beschrieben werden!
 - simuliere Algorithmus auf RAM durch Bool'sche Formel

- weitere Details zur Simulation einer RAM findet man z. B. im Cormen (3rd) Ch. 34.3

Wie sieht eine konkrete Reduktion aus?

CLIQUE-Problem

- Gegeben: $\langle G, k \rangle$ (unger. Graph $G = (V, E)$, Zahl $k \in \mathbb{N}$)
- Ausgabe: Enthält G eine k -Clique?

Theorem 7.2

Das CLIQUE-Problem ist NP-vollständig.

Beweis Teil 1/3: zeige $\text{CLIQUE} \in \text{NP}$

- Zertifikat: $V' \subseteq V$
 - hat polynomielle Größe, da $|V'| \leq |V|$
- Verifizierer:
 - Überprüfe ob $|V'| = k$ und...
 - ...ob für alle $u, v \in V'$ mit $u \neq v$ gilt, dass $\{u, v\} \in E$

Reduktion auf das CLIQUE-Problem

- zeigen nun, dass CLIQUE NP-schwer ist
- wir benutzen nicht direkt SAT, sondern 3-SAT
- bekannt: $SAT \leq_p 3\text{-SAT}$

Beweis Teil 2/3: zeige $3\text{-SAT} \leq_p \text{CLIQUE}$

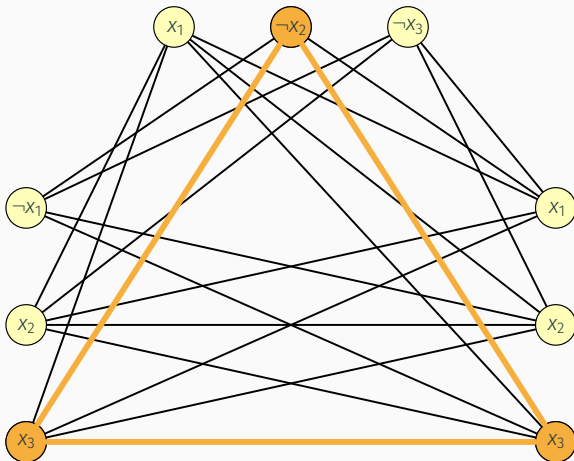
(Konstruktion der Reduktionsfunktion)

- Ziel:
 - gegeben Eingabe x für 3-SAT, Umwandlung in Eingabe $r(x)$ für CLIQUE
 - so, dass $x \in 3\text{-SAT} \iff r(x) \in \text{CLIQUE}$
- Eingabe von 3-SAT: $C_1 \wedge C_2 \wedge \dots \wedge C_n$
 - wobei $C_i = l_i^1 \vee l_i^2 \vee l_i^3$ (l_i^s ist s -tes Literal der i -ten Klausel)
- Funktion r konstruiert aus x die Eingabe $\langle G = (V, E), n \rangle$
 - Knoten V : einen Knoten v_i^s pro Literal
 - Kanten E : $\{v_i^s, v_j^t\} \in E$ falls:
 - $i \neq j$ (Kanten nur zwischen unterschiedlichen Klauseln)
 - $l_i^s \neq \neg l_j^t$ (zugehörige Literale sind kompatibel)

} polynomielle
Laufzeit

Illustration der Reduktionsfunktion

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Reduktion auf das CLIQUE-Problem

Beweis Teil 3/3: zeige $3\text{-SAT} \leq_p \text{CLIQUE}$

(Eigenschaft der Reduktionsfunktion)

Zu zeigen: $x \in 3\text{-SAT} \iff r(x) \in \text{CLIQUE}$

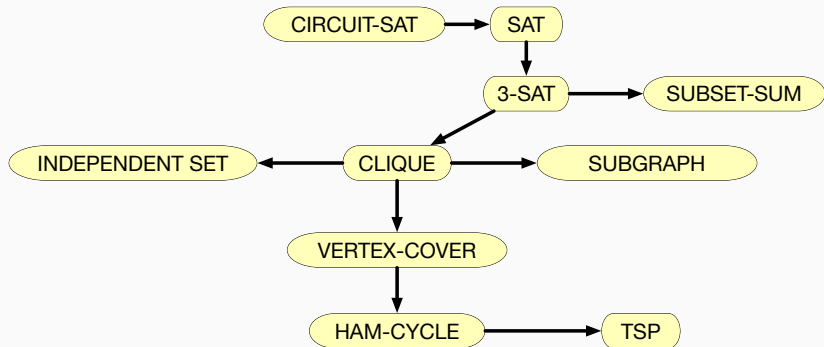
1. Richtung „ $x \in 3\text{-SAT} \implies r(x) \in \text{CLIQUE}$ “

- x erfüllbar \implies für alle i existiert s mit $I_i^s = 1$
- erstelle V' : sei $V'' = \{v_i^s \in V \mid I_i^s = 1\}$
 - $V' \subseteq V''$: wähle pro Klausel **genau ein** Literal
- $|V'|$ und V' ist eine Clique:
 - Kante zwischen je zwei $v_i^s, v_j^t \in V'$ (da $i \neq j$ und gültige Belegung)

2. Richtung „ $r(x) \in \text{CLIQUE} \implies x \in 3\text{-SAT}$ “

- betrachte k -Clique V' in $r(x)$ und L' zugehörige Menge an Literalen
- L' enthält je Klausel ein Literal
 - $|L'| = |V'| = k$ und Knoten innerhalb Klausel nicht adjazent
- können Belegung wählen, die alle Literale in L' erfüllt
 - Knoten zu inkonsistenten Literalen sind nicht adjazent
 - wähle $x_i = 1$ falls $x_i \in L'$ und $x_i = 0$ falls $\neg x_i \in L'$
- erhalten erfüllende Belegung für x

Eine Übersicht an Standard-Reduktionen



4) The End?

Natürlich nicht!

Algorithmik

- logische Fortsetzung von AD
- amortisierte Analyse, Fibonacci-Heaps, Netzwerkfluss, ...
- ...Matching, Matrixoperationen, (integrale) lineare Programmierung...
- ...Schnittprobleme, konvexe Hüllen, Voronoi-Diagramme, ...

Methods of Algorithm Design

- Spezialvorlesung zu Algorithmen
- Schwerpunkt: **Uncertainty**
- Algorithmenentwurf für Probleme mit unvollständiger Eingabe
- Listen-Management, Caching, k -Server, Scheduling, ...