

Komplexität verstehen

Das grafische Fundament zu *Analyse von Algorithmen und Komplexität*

CAU Kiel – Sommersemester 2026 · Ausgabe 3

*Alles, was du brauchst, um den Kurs zu verstehen –
mit Bildern, Beispielen und Fragen zum Selbstprüfen.*

Wozu dieses Heft. Es baut dein *Fundament*: Begriffe, Regeln, Probleme und ihre Zusammenhänge. Danach kannst du den ganzen Stoff nachvollziehen – und bist bereit, in Präsenz-, Haus- und Klausuraufgaben selbst zu rechnen.

So arbeitest du damit (wichtig!). Am Ende jedes Kapitels stehen **Abruf-Fragen**. Beantworte sie *schriftlich aus dem Kopf, bevor* du im Lösungsanhang (Anhang A) nachsiehst. Dieses aktive Erinnern ist der eigentliche Lerneffekt – nur Lesen reicht nicht.

Die Kästen und ihre Rollen.

★ Intuition	die Idee in einfachen Worten
🖼️ Analogie	ein Bild aus dem Alltag
■ Definition	die präzise Fassung
▲ Satz	ein Ergebnis mit Beweis <i>idee</i>
● Beispiel	durchgerechnet, mit echten Zahlen
✓ Warum in NP?	Zertifikat und Prüfung des Problems
🔍 Idee	die Konstruktion hinter einer Reduktion/einem Beweis
☆ Aha	die Pointe
✗ Stolperstein	der häufige Fehler
➤ Zusammenhang	wie es ins Ganze passt
? Abruf	prüfe dich – Lösung nur im Anhang

Inhalt

1	Ein Problem, das sich wehrt	3
2	Was heißt „schnell“?	4
3	Prüfen ist leichter als Lösen	6
4	„Mindestens so schwer wie“ – Reduktionen	7
5	Die härtesten Probleme: NP-vollständig	9
6	Der Problem-Zoo	11
7	Wie schwer <i>genau</i> ? Die ETH	16
8	Damit leben: gute Näherungen	17
9	Jenseits von NP: das Halteproblem	21
10	Das große Ganze	21
A	Lösungen zu den Abruf-Fragen	22

1 Ein Problem, das sich wehrt

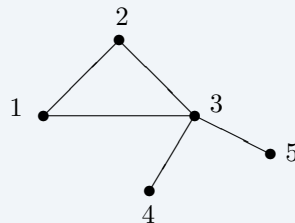
★ In diesem Kapitel

Wir treffen ein Problem, das harmlos aussieht und trotzdem jeden Computer überfordert. Daran verstehst du, warum es eine Theorie der „schweren“ Probleme gibt.

1.1 Cliques in einem Graphen

Ein *Graph* besteht aus *Knoten* (Punkte) und *Kanten* (Verbindungen). Eine *Clique* ist eine Gruppe von Knoten, in der *jeder mit jedem* verbunden ist. Das *Cliquenproblem* fragt: Gibt es eine Clique aus mindestens k Knoten?

● Beispiel 1.1 Clique mit dem Auge

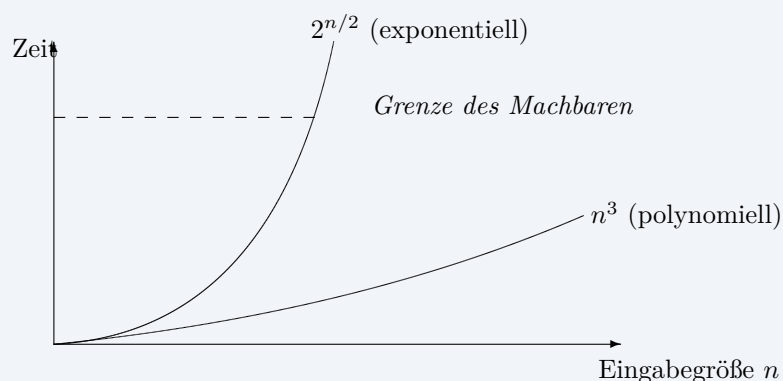


$\{1, 2, 3\}$ bildet ein Dreieck – eine Clique der Größe 3. $\{2, 3, 4\}$ ist keine: zwischen 2 und 4 fehlt die Kante. Eine 4er-Clique gibt es hier nicht.

1.2 Warum der naive Weg explodiert

Wie sucht ein Computer eine k -Clique? Naiv: *alle* Knotengruppen der Größe k durchprobieren. Bei n Knoten und $k = n/2$ sind das mindestens $2^{n/2}$ Gruppen.

● Beispiel 1.2 Der Graben zwischen n^3 und $2^{n/2}$



Die polynomielle Kurve bleibt lange flach. Die exponentielle schießt fast senkrecht nach oben und durchbricht früh jede Zeitgrenze.

● Beispiel 1.3 Dieselbe Kluft in Zahlen

n	n^3	$2^{n/2}$
20	8 000	rund 1 000
40	64 000	rund 1 Million
100	10^6	rund 10^{15} (über ein Monat bei 10^9 /s)
200	$8 \cdot 10^6$	rund 10^{30} (länger als das Universum alt ist)

★ Intuition

Exponentielles Wachstum verdoppelt den Aufwand bei jedem zusätzlichen Element. Ein doppelt so schneller Computer bringt dir dann genau *einen* Knoten mehr. Hardware kann exponentielles Wachstum niemals einholen. Genau das macht ein Problem „praktisch unlösbar“.

▲ Satz 1.4 Der naive Clique-Algorithmus ist exponentiell

Für $k = |V|/2$ braucht er mindestens $2^{|V|/2}$ Schritte.

Beweisidee: Es sind $\binom{n}{n/2} \geq 2^{n/2}$ Gruppen zu prüfen.

1.3 Die eigentliche Frage

Vielleicht ist der naive Weg nur dumm und es gibt einen cleveren Trick? Für die Clique kennt bis heute niemand einen – aber niemand hat auch bewiesen, dass es keinen gibt.

☆ Aha

Die Komplexitätstheorie fragt nicht „Wie löse ich das schnell?“, sondern „Ist das *überhaupt* schnell lösbar – oder gehört es zu einer ganzen Familie von Problemen, die gemeinsam scheitern?“ Diese zweite Frage können wir beantworten, obwohl die erste offen ist.

? Abruf – erst selbst beantworten

- A1.** Warum hilft ein 1000-mal schnellerer Computer beim naiven Clique-Algorithmus kaum?
A2. Was ist der Unterschied zwischen „wir kennen keinen schnellen Algorithmus“ und „es gibt keinen“ – und welchen der beiden Fälle hat die Wissenschaft für die Clique bewiesen?

2 Was heißt „schnell“?

★ In diesem Kapitel

Wir machen „effizient“ präzise und lernen die Klasse P kennen – die Probleme, die wir schnell lösen können.

2.1 Laufzeit misst man in der Eingabegröße

Ein Algorithmus ist schnell oder langsam stets *relativ zur Eingabegröße* n . Wir betrachten den *schlechtesten Fall*: die maximale Schrittzahl über alle Eingaben der Länge n .

■ Definition 2.1 Klasse P

P ist die Menge der Entscheidungsprobleme, die ein Algorithmus in *polynomieller* Zeit $O(n^d)$ (mit fester Zahl d) löst. Kurz: die *effizient lösbaren* Probleme.

★ Intuition

„Polynomiell = effizient“ ist eine Konvention, aber eine robuste: Praktische polynomielle Algorithmen haben kleine Exponenten (n , n^2 , n^3), und die Grenze bleibt stabil, egal welchen realistischen Rechner man annimmt. Der wahre Bruch verläuft zwischen polynomiell und exponentiell – den hast du in Kapitel 1 grafisch gesehen.

Beispiele in P: Sortieren, kürzeste Wege, minimale Spannbäume, Matching.

2.2 Probleme als Ja/Nein-Fragen

Die Theorie betrachtet *Entscheidungsprobleme* – Fragen mit Antwort Ja oder Nein. Statt „Wie groß ist die größte Clique?“ fragt man „Gibt es eine Clique mit $\geq k$ Knoten?“

🗨 Analogie

Wie „Ist das Paket schwerer als 5 kg?“ statt „Wie schwer genau?“. Beantworte die Ja/Nein-Frage für jede Schranke, kennst du am Ende auch den genauen Wert – du tastest ihn ein. Deshalb ist die Entscheidungsvariante „gleich schwer“ wie das Optimierungsproblem, aber leichter zu vergleichen.

Formal kodiert man jede Eingabe als *Wort* über einem Alphabet; ein Problem wird zur Menge seiner Ja-Wörter (einer *Sprache*). Merke dir schlicht: *ein Problem = die Menge seiner Ja-Eingaben*.

✗ Stolperstein

Zahlen sind binär kodiert. Eine Zahl K hat Eingabelänge nur $O(\log K)$, ihr Wert ist exponentiell größer. Ein Algorithmus, der bis K zählt, ist deshalb *nicht* polynomiell. Daran hängt später die Schwere von SUBSET SUM und der Begriff „pseudopolynomiell“.

? Abruf – erst selbst beantworten

A3. Ist ein Algorithmus mit Laufzeit n^4 „effizient“ im Sinne der Theorie? Begründe.

A4. Warum ist „zähle von 1 bis zur Eingabezahl K “ *kein* polynomieller Algorithmus, obwohl er so einfach aussieht?

3 Prüfen ist leichter als Lösen

★ In diesem Kapitel

Wir entdecken die zentrale Beobachtung der Theorie – Prüfen ist oft viel leichter als Lösen – und lernen daraus die Klasse NP sowie ein Rezept, mit dem du für *jedes* Problem zeigst, dass es in NP liegt.

3.1 Die Sudoku-Beobachtung

🔍 Analogie

Ein schweres Sudoku zu *lösen* dauert. Eine *fertige* Lösung zu *prüfen* geht in einer Minute: Zeilen, Spalten, Blöcke abhaken. Lösen ist schwer, Prüfen ist leicht. Diese Asymmetrie *ist* NP.

Die vorgelegte Lösung heißt *Zertifikat*, der Prüf-Algorithmus *Verifizierer*.

■ Definition 3.1 Verifizierer, Zertifikat, NP

Ein *Verifizierer* für ein Problem L bekommt die Instanz x und einen Vorschlag c und erfüllt

$$x \in L \iff \text{es gibt ein } c \text{ mit } A(x, c) = 1.$$

c heißt *Zertifikat*. $L \in \text{NP}$, wenn es einen Verifizierer gibt, der für jede Ja-Instanz ein *polynomiell langes* Zertifikat in *polynomieller* Zeit prüft.

★ Intuition

NP = effizient *überprüfbar*, nicht effizient *lösbar*. Das Zertifikat ist die abgekürzte Antwort auf „und woher weiß ich, dass das stimmt?“. Für Nein-Instanzen überzeugt *kein* Vorschlag den Verifizierer.

3.2 Der NP-Dreischritt – so zeigt man „ $\in \text{NP}$ “

Für *jedes* Problem läuft der NP-Nachweis nach demselben Muster. Diesen Dreischritt brauchst du im ganzen Kurs.

🔍 Idee: Nachweis „ $L \in \text{NP}$ “

- (1) **Zertifikat:** Was ist die „Lösung“? (Teilmenge, Belegung, Permutation.) Angeben, dass sie polynomiell lang ist.
- (2) **Verifizierer:** Ein Algorithmus, der *alle* geforderten Eigenschaften prüft. Zwei Richtungen der Korrektheit: Ja-Instanz $\Rightarrow$ es gibt ein akzeptiertes Zertifikat; Nein-Instanz $\Rightarrow$ keines wird akzeptiert. Laufzeit polynomiell.
- (3) **Alternativ als Raten (NDTM):** „Rate die Lösung, prüfe sie.“ Die Rateschritte *sind* das Zertifikat.

● Beispiel 3.2 Der Dreischritt für Clique

Zertifikat: eine Knotenmenge C (ein Bit pro Knoten, also polynomiell). **Verifizierer:** prüfe, ob alle Paare in C verbunden sind ($O(|C|^2)$) und ob $|C| \geq k$ ($O(|V|)$). **Korrektheit:** Gibt es eine k -Clique, wird sie akzeptiert; gibt es keine, scheitert jeder Vorschlag. Laufzeit polynomiell – also $\text{CLIQUE} \in \text{NP}$.

★ Intuition

Die zweite Sicht („Raten“) ist eine *nichtdeterministische* Maschine, die an jeder Verzweigung alle Wege gleichzeitig verfolgt. Sie akzeptiert, wenn *ein* Weg zum Ziel führt. Für eine Nein-Instanz führt *kein* Weg zum Ziel. „Zertifikat bekommen“ und „richtig raten“ sind zwei Namen für dieselbe Sache.

3.3 P steckt in NP – und die Millionenfrage

▲ Satz 3.3 $P \subseteq \text{NP}$

Jedes effizient lösbare Problem ist auch effizient überprüfbar.

Beweisidee: Wer selbst lösen kann, ignoriert das Zertifikat und rechnet die Antwort direkt aus.

Die offene Umkehrung lautet: Gilt $P = \text{NP}$? Ist alles, was man schnell *prüfen* kann, auch schnell *lösbar*?

☆ Aha

Ob $P = \text{NP}$, weiß niemand – eines der sieben Millennium-Probleme (1 Mio. Dollar). Die meisten glauben $P \neq \text{NP}$. *Alles Folgende – NP-Vollständigkeit, ETH, Approximation – ist der Umgang mit dieser Unwissenheit.*

? Abruf – erst selbst beantworten

A5. Nenne die drei Schritte des NP-Nachweises und wende sie auf VERTEX COVER an (Zertifikat? Prüfung? Laufzeit?).

A6. Warum ist „ L hat einen polynomiellen Verifizierer“ nicht dasselbe wie „ L ist in Polynomialzeit lösbar“?

4 „Mindestens so schwer wie“ – Reduktionen

★ In diesem Kapitel

Wir bauen das wichtigste Werkzeug der Theorie: die Reduktion. Mit ihr vergleichst du Probleme nach Schwierigkeit, ohne für eines einen Algorithmus zu kennen.

4.1 Ein Problem mit einem anderen lösen

Analogie

Du willst wissen, ob eine Strecke länger als 1 Meile ist, hast aber nur ein km-Lineal. Du rechnest „> 1 Meile“ um in „> 1,609 km“ und misst. Die *Umrechnung* ist billig, das *Messen* erledigt das fremde Werkzeug. Genau das ist eine Reduktion.

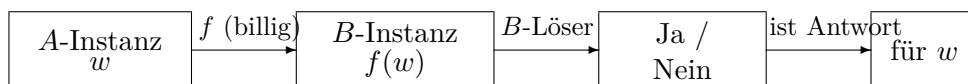
■ Definition 4.1 Polynomielle Reduktion $A \leq_p B$

Eine Funktion f , die jede A -Eingabe w in eine B -Eingabe $f(w)$ übersetzt, sodass

$$w \in A \iff f(w) \in B,$$

und die in polynomieller Zeit berechenbar ist. Schreibweise $A \leq_p B$.

Der Datenfluss einer Reduktion, wenn man A über einen B -Löser lösen will:



★ Intuition

$A \leq_p B$ heißt: „ B ist *mindestens so schwer* wie A .“ Denn ein schneller B -Löser ergibt (über f) auch einen schnellen A -Löser. Die Schwierigkeit fließt *entlang des Pfeils*: Ist A schwer, muss B es auch sein.

4.2 Die Richtung – der teuerste Fehler

✗ Stolperstein

Um zu zeigen, dass ein *neues* Problem Y schwer ist, reduzierst du ein *bekannt schweres* X auf Y : also $X \leq_p Y$ („bekannt $\leq_p$ neu“). Die Reduktion nimmt eine X -Instanz und baut eine Y -Instanz. **Nie umgekehrt.** $Y \leq_p X$ würde nur zeigen, dass Y *höchstens* so schwer wie X ist – das sagt über Y nichts.

Analogie

Willst du beweisen, dass ein neuer Boxer stark ist, schickst du den *amtierenden Champion* gegen ihn – nicht einen Anfänger. „Bekannt stark $\leq_p$ neu.“

▲ Satz 4.2 Transitivität

$A \leq_p B$ und $B \leq_p C \Rightarrow A \leq_p C$.

Beweisidee: Umformungen hintereinander schalten; die Zwischenausgabe ist polynomiell groß, also bleibt alles polynomiell.

Deshalb kann eine ganze *Kette* von Reduktionen die Schwere Glied für Glied weitertragen – das nutzen wir gleich.

? Abruf – erst selbst beantworten

A7. Du weißt: CLIQUE ist schwer. Du willst zeigen, dass VERTEX COVER schwer ist. In welche Richtung reduzierst du, und warum genau so?

A8. Was bedeutet $A \leq_p B$ für die relative Schwierigkeit von A und B – in einem Satz?

5 Die härtesten Probleme: NP-vollständig

★ In diesem Kapitel

Wir fassen die schwersten Probleme in NP mit einem Namen und sehen, warum *ein* schneller Algorithmus für eines von ihnen die ganze Landschaft zum Einsturz brächte.

■ Definition 5.1 NP-schwer, NP-vollständig

- L_0 ist *NP-schwer*: jedes $L \in \text{NP}$ lässt sich auf L_0 reduzieren ($L \leq_p L_0$).
- L_0 ist *NP-vollständig*: NP-schwer **und** selbst in NP.

★ Intuition

Zwei Schranken:

- *NP-schwer* = **untere** Schranke („mindestens so schwer wie alles in NP“).
- $\in \text{NP}$ = **obere** Schranke („nicht schwerer als NP“).

NP-vollständig = die schwersten Probleme *innerhalb* von NP.

✗ Stolperstein

NP-schwer heißt *nicht* „in NP“. Es gibt NP-schwere Probleme außerhalb von NP – das Halteproblem (Kapitel 9) ist nicht einmal lösbar und trotzdem NP-schwer.

▲ Satz 5.2 Ein schneller Algorithmus für eins löst alle

Ist L_0 NP-vollständig, dann: $P = NP \iff L_0 \in P$.

Beweisidee: Läge L_0 in P , könnte man jedes $L \in NP$ lösen: erst $L \leq_p L_0$ rechnen, dann den schnellen L_0 -Löser.

☆ Aha

Fände jemand für *ein einziges* NP-vollständiges Problem einen polynomiellen Algorithmus, wären *schlagartig alle* NP-Probleme effizient lösbar. Solange das niemandem gelingt, ist ein NP-Vollständigkeitsbeweis das stärkste Indiz für „vermutlich kein effizienter Algorithmus“.

5.1 Der Anker: Cook–Levin

▲ Satz 5.3 Cook (1971), Levin (1973)

SAT ist NP-vollständig – das *erste* solche Problem.

Beweisidee: Man nimmt ein *beliebiges* $L \in NP$ mit seiner ratenden Maschine und gießt deren Rechnung in eine logische Formel. Variablen beschreiben „Zustand zur Zeit t “, „Bandinhalt“ usw. Die Formel ist so gebaut, dass *erfüllende Belegungen genau akzeptierenden Rechenwegen entsprechen*. Also: L -Ja-Instanz $\iff$ Formel erfüllbar.

🔗 Idee: Vererbungskorollar – das Arbeitspferd

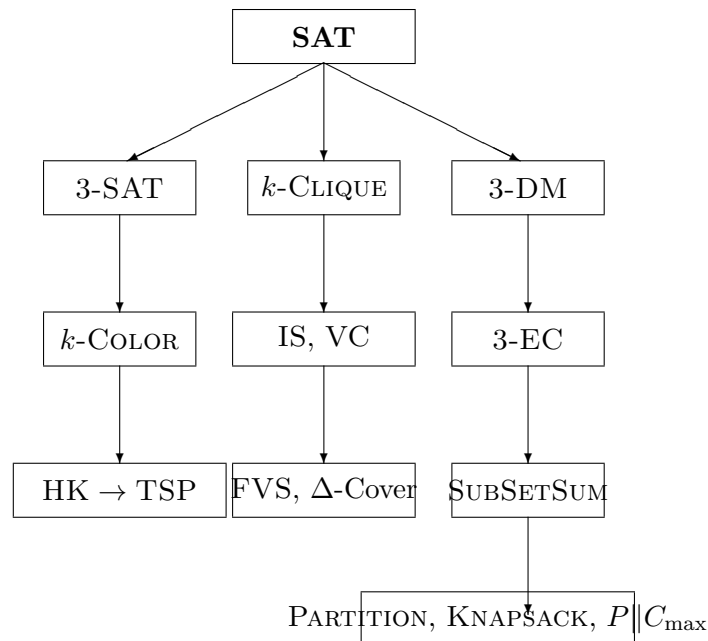
Um ein neues Problem Y als NP-vollständig zu zeigen, brauchst du nur zwei Dinge:

- (1) eine Reduktion $X \leq_p Y$ von *einem* bekannten NP-vollständigen X , und
- (2) den Nachweis $Y \in NP$ (NP-Dreischritt).

Dann ist Y NP-vollständig – denn $L \leq_p X \leq_p Y$ für alle $L \in NP$.

5.2 Die Landkarte der Schwere

Alle Probleme des Kurses hängen an *einer* Kette ab SAT. Der Pfeil heißt „wird reduziert auf“ (Schwere fließt nach unten).



★ Intuition

Beim Lesen des nächsten Kapitels immer fragen: „Von welchem Nachbarn erbt dieses Problem seine Schwere?“ Die Probleme sind nicht einzeln schwer, sondern *gemeinsam* – verbunden durch diese Pfeile.

? Abruf – erst selbst beantworten

A9. Welche zwei Dinge musst du zeigen, um ein neues Problem als NP-vollständig zu beweisen? Welchen davon vergisst man leicht?

A10. Warum wäre $P = NP$, wenn jemand SUBSET SUM in Polynomialzeit löste?

6 Der Problem-Zoo

★ In diesem Kapitel

Wir gehen alle Probleme des Kurses durch. Zu jedem: was es fragt, *warum es in NP liegt* (Zertifikat), *woher* seine Schwere kommt und die *Idee* der Reduktion – oft als Bild.

So liest du jeden Eintrag: **Definition** (was gefragt ist) → **Warum in NP?** (Zertifikat) → **Idee** (woher die Schwere, wie die Reduktion baut). Alle diese Probleme sind NP-vollständig.

6.1 Logik: SAT und 3-SAT

■ Definition 6.1 SAT und 3-SAT

SAT: Formel in konjunktiver Normalform (UND von Klauseln, jede Klausel ein ODER von Literalen $x_j/\neg x_j$) – gibt es eine erfüllende Belegung?

3-SAT: dasselbe mit höchstens 3 Literalen pro Klausel.

✓ Warum in NP?

Zertifikat: die Belegung ψ (ein Bit pro Variable). **Prüfung:** Formel unter ψ auswerten – linear in der Formellänge.

☞ Idee: $\text{SAT} \leq_p \text{3-SAT}$ – lange Klauseln aufspalten

Ersetze eine lange Klausel per Hilfsvariablen durch eine *Kette* von 3er-Klauseln, z. B. $(y_1 \vee \dots \vee y_n) \rightsquigarrow (y_1 \vee y_2 \vee x_1) \wedge (\neg x_1 \vee y_3 \vee x_2) \wedge \dots$. Ist ein y_i wahr, lässt sich die x -Kette passend setzen; ist *keines* wahr, erzwingt die Kette einen Widerspruch. Erfüllbarkeit bleibt gleich.

6.2 Graphen: das Dualitätsdreieck

Drei Probleme, die im Grunde eines sind – aus drei Blickwinkeln.

■ Definition 6.2 Clique, Independent Set (IS), Vertex Cover (VC)

Gegeben G und k . **CLIQUE:** Gruppe, in der *alle* paarweise verbunden sind, Größe $\geq k$? **IS:** Gruppe, in der *keine* zwei verbunden sind, Größe $\geq k$? **VC:** Knotenmenge, die *jede* Kante berührt, Größe $\leq k$?

✓ Warum in NP?

Clique/IS: Zertifikat ist die Gruppe; prüfe alle Paare ($O(|V|^2)$) und die Größe. **VC:** Zertifikat ist die Knotenmenge; prüfe für jede Kante, ob ein Endpunkt drin liegt ($O(|E|)$), und die Größe.

☆ Aha

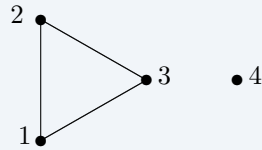
Ein Bild, drei Rollen. In G mit n Knoten:

$$C \text{ Clique in } G \iff C \text{ IS in } \bar{G} \iff V \setminus C \text{ VC in } \bar{G}.$$

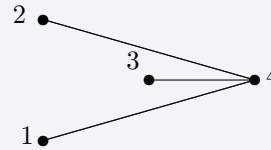
Links dieselbe Knotenmenge $\{1, 2, 3\}$ in G (Clique) und rechts im Komplement $\bar{G}$ (Independent Set); der Rest $\{4\}$ ist dort ein Vertex Cover.

● Beispiel 6.3 Das Dreieck sichtbar

G : $\{1, 2, 3\}$ ist Clique



$\bar{G}$: $\{1, 2, 3\}$ IS, $\{4\}$ VC



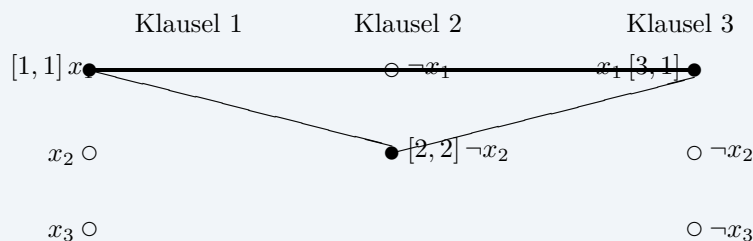
In $\bar{G}$ sind genau die früheren Nicht-Kanten vorhanden. $\{1, 2, 3\}$ hat dort keine innere Kante (IS), und Knoten 4 allein berührt alle Kanten (VC).

☞ Idee: $\text{SAT} \leq_p \text{Clique}$ – ein Knoten pro Literal

Baue für jedes Literalvorkommen einen Knoten $[i, j]$. Verbinde zwei Knoten genau dann, wenn sie aus *verschiedenen* Klauseln stammen und sich *nicht widersprechen*. Suche eine Clique der Größe $k = m$ (Anzahl Klauseln). Eine solche Clique wählt aus jeder Klausel ein widerspruchsfreies wahres Literal – also eine erfüllende Belegung.

● Beispiel 6.4 $\text{SAT} \leq_p \text{Clique}$ – die Lösungs-Clique

$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$, $m = 3$. Acht Knoten in drei Klausel-Spalten; fett die Clique $\{[1, 1], [2, 2], [3, 1]\}$ ($x_1, \neg x_2, x_1$).



Die drei fetten Kanten bilden das Dreieck (die 3-Clique). Sie liefert $x_1 = \text{wahr}$, $x_2 = \text{falsch}$ – und das erfüllt F .

☞ Idee: $\text{Clique} \leq_p \text{VC}$ und $\leq_p \text{IS}$

Reine Umformung (das Dualitätsdreieck): für IS gib $(\bar{G}, k)$ aus, für VC gib $(\bar{G}, n - k)$ aus. Kein Gadget nötig.

6.3 Färbung, Hamiltonkreis, TSP

■ Definition 6.5 k -Color, Hamiltonkreis (HK), TSP

k -COLOR: Knoten mit k Farben färben, sodass keine Kante gleiche Enden hat?

HK: Rundweg, der jeden Knoten genau einmal besucht?

TSP (Entscheidung): Rundreise durch alle Städte mit Länge $\leq L$?

✓ Warum in NP?

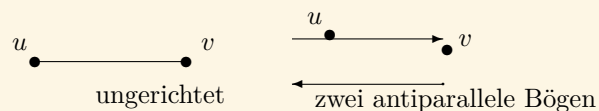
Color: die Färbung f ; prüfe jede Kante ($O(|E|)$). **HK:** die Knoten-Reihenfolge (Permutation); prüfe, ob jede Folgekante existiert und jeder Knoten einmal vorkommt. **TSP:** die Reihenfolge; addiere die Distanzen und vergleiche mit L .

🔍 Idee: Woher ihre Schwere kommt

k -COLOR erbt von 3-SAT (Gadget mit $n+1$ Farben, das jede Variable auf wahr/falsch zwingt). HK erbt von der 3-Literal-Variante 3-SAT' (A-Komponenten kodieren die Variablensetzung, B-Komponenten prüfen die Klauseln). TSP erbt von HK: echte Kanten Distanz 1, fehlende Kanten sehr groß, $L = |V|$ – eine kurze Tour *ist* ein Hamiltonkreis.

🔍 Idee: $VC \leq_p FVS$ – ungerichtet wird gerichtet

FEEDBACK VERTEX SET: entferne $\leq k$ Knoten, sodass ein *gerichteter* Graph kreisfrei wird. Reduktion: mache aus jeder ungerichteten Kante *zwei antiparallele Bögen* (einen 2-Kreis). Ein Knoten, der die Kante „überdeckt“, zerstört genau diesen Kreis.



6.4 Mengen und Zahlen

■ Definition 6.6 3-DM, 3-EC, SubSet Sum, Partition, Knapsack

3-DM: drei gleich große Mengen, Tripel; wähle Tripel, die jedes Element genau einmal treffen.

3-EC: 3-elementige Mengen; wähle einige, die das Universum exakt (disjunkt) überdecken.

SUBSET SUM: Zahlen und Ziel K ; Teilmenge mit Summe genau K ? PARTITION: Teilmenge mit genau der halben Gesamtsumme?

KNAPSACK: Auswahl mit Gewicht $\leq B$ und Profit $\geq P$?

✓ Warum in NP?

Für alle: das Zertifikat ist die *Auswahl* (Tripelmenge, Mengenauswahl, Teilmenge). Prüfung: Summen bzw. Überdeckungsbedingungen nachrechnen und Schranken vergleichen – alles in $O(n)$ bis $O(n^2)$.

🔍 Idee: 3-EC $\leq_p$ SubSet Sum – Mengen werden Ziffern

Schreibe jede 3-elementige Menge als Bitvektor über dem Universum und lies ihn als *Zahl* zur Basis $n+1$. Beispiel $U = \{u_1, \dots, u_6\}$:

Menge	Bitvektor	Zahl (Basis $n+1$)
$\{u_1, u_5, u_6\}$	100011	$(n+1)^0 + (n+1)^4 + (n+1)^5$
Ziel K	111111	$\sum_{i=0}^5 (n+1)^i$

Bei Basis $n+1$ und $\leq n$ Summanden gibt es *keinen Übertrag*. Die Zielsumme wird genau dann getroffen, wenn jede Stelle exakt einmal überdeckt ist – eine exakte Überdeckung.

🔍 Idee: SubSet Sum $\leq_p$ Partition – zwei Zusatzzahlen

Aus $(c_1, \dots, c_n, K)$ mit $N = \sum c_j + 1$ ergänze $c_{n+1} = N - K$ und $c_{n+2} = K + 1$. Gesamtsumme $2N$, Hälfte N . Die beiden Zusatzzahlen summieren zu $N + 1$, können also nie zusammen in eine Hälfte – eine Halbierung existiert genau dann, wenn eine Original-Teilmenge K trifft.

➤ Zusammenhang

KNAPSACK enthält SUBSET SUM als Spezialfall ($w_j = p_j$, $P = B$) – deshalb auch schwer. Und PARTITION reduziert weiter auf Scheduling (nächster Abschnitt).

6.5 Scheduling: $P \parallel C_{\max}$

■ Definition 6.7 $P \parallel C_{\max}$

n Jobs mit Laufzeiten p_j , m identische Maschinen. Verteile sie so, dass die höchste Maschinenlast (der *Makespan*) minimal wird.

✓ Warum in NP?

Zertifikat: die Zuordnung Job $\rightarrow$ Maschine. **Prüfung:** pro Maschine die Last summieren, Maximum bilden, mit der Schranke vergleichen – $O(n)$.

☞ Idee: Partition $\leq_p P2||C_{\max}$

Nimm die Partition-Zahlen als Jobs, zwei Maschinen. Beide Maschinen exakt gleich auszulasten heißt, die Zahlen zu *halbieren*. Also ist $P||C_{\max}$ schon für $m = 2$ NP-vollständig.

? Abruf – erst selbst beantworten

A11. Gib für HAMILTONKREIS das Zertifikat und die Prüfung an (NP-Dreischritt).

A12. Erkläre in eigenen Worten, warum bei $3\text{-EC} \leq_p \text{SUBSET SUM}$ die Basis $n + 1$ (und nicht z. B. Basis 2) gewählt wird.

A13. VERTEX COVER und INDEPENDENT SET hängen eng zusammen. Wie – und wie überführt man das eine ins andere?

7 Wie schwer *genau*? Die ETH

★ In diesem Kapitel

NP-Vollständigkeit sagt nur „vermutlich nicht polynomiell“. Die Exponentialzeit-Hypothese (ETH) sagt schärfer: *wie* exponentiell – und liefert konkrete untere Schranken.

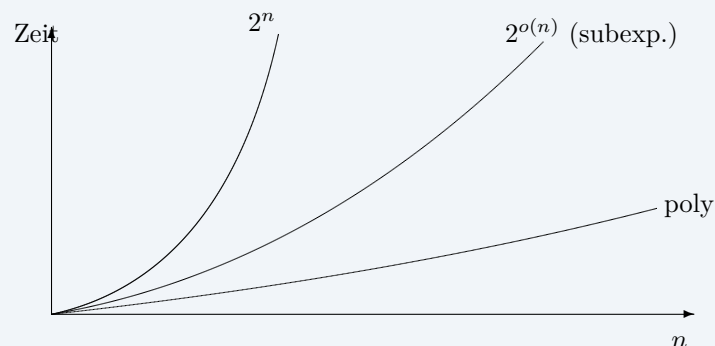
7.1 Die Lücke, die NP-Vollständigkeit lässt

NP-Vollständigkeit unterscheidet nicht zwischen 2^n und $2^{\sqrt{n}}$ – beide sind „nicht polynomiell“. Oft will man aber wissen: geht es wenigstens *subexponentiell*?

■ Definition 7.1 Klein-o und subexponentiell

$f(n) = o(g(n))$ heißt: f wächst echt langsamer, $f(n)/g(n) \rightarrow 0$. Achtung: δn ist *nicht* $o(n)$. Ein „ $2^{o(n)}$ -Algorithmus“ läuft subexponentiell (z. B. $2^{\sqrt{n}}$).

● Beispiel 7.2 Drei Wachstumsklassen im Bild



Die ETH behauptet: Für 3-SAT gibt es *keinen* Algorithmus der mittleren Sorte – es bleibt echt exponentiell.

■ Definition 7.3 Exponentialzeit-Hypothese (ETH)

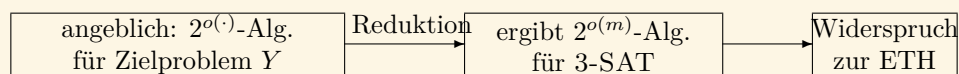
Es gibt ein $\delta > 0$, sodass 3-SAT mit n Variablen *nicht* in Zeit $2^{\delta n} \cdot \text{poly}$ lösbar ist. Kurz: kein $2^{o(n)}$ -Algorithmus für 3-SAT.

Das ist eine *Annahme*, stärker als $P \neq NP$, aber breit geglaubt.

7.2 Wie man ETH-Schranken beweist

🔗 Idee: Kontraposition entlang einer Reduktion

Man überträgt die Härte wie bei der NP-Vollständigkeit – nur zählt man die *Größe* der konstruierten Instanz mit:



▲ Satz 7.4 Sparsification-Lemma – warum es nötig ist

Unter der ETH gibt es auch keinen $2^{o(m)}$ -Algorithmus für 3-SAT ($m = \text{Klauselzahl}$).
Warum: Die ETH spricht über n (Variablen). Reduktionen erzeugen ihre Größe oft aus m (Klauseln). Da m groß gegen n sein kann, folgt eine m -Schranke *nicht* direkt aus der ETH – das Lemma schließt die Lücke.

✗ Stolperstein

Hängt der Parameter deiner konstruierten Instanz an der Klauselzahl m , musst du das Sparsification-Lemma zitieren. Das ist der am leichtesten vergessene Schritt in ETH-Beweisen.

Unter der ETH bekommt man so $2^{o(n)}$ -Schranken u. a. für 3-COLOR, CLIQUE, VC, IS, SUBSET SUM und PARTITION.

? Abruf – erst selbst beantworten

A14. Wenn die ETH falsch ist – folgt dann $P = NP$? Begründe.

A15. Wozu braucht man das Sparsification-Lemma, wo die ETH doch schon eine untere Schranke liefert?

8 Damit leben: gute Näherungen

★ In diesem Kapitel

Für schwere *Optimierungsprobleme* berechnet man in Polynomialzeit *beweisbar gute* Näherungen. Wir sehen den Gütebegriff und die wichtigsten Algorithmen – viele davon grafisch.

8.1 Was heißt „gute Näherung“?

■ Definition 8.1 Multiplikative Güte

Mit $\text{OPT}(I)$ = optimaler Wert. Algorithmus A hat Güte α , wenn für alle I : Minimierung: $A(I) \leq \alpha \cdot \text{OPT}(I)$; Maximierung: $\text{OPT}(I) \leq \alpha \cdot A(I)$. *Scharf*, wenn Instanzen den Faktor (fast) erreichen.

■ Definition 8.2 PTAS, EPTAS, FPTAS

Familie (A_ε) mit Güte $1 + \varepsilon$ für jedes ε : *PTAS* polynomiell in n je festem ε (ε darf im Exponenten stehen); *EPTAS* Laufzeit $f(1/\varepsilon) \cdot \text{poly}(n)$; *FPTAS* polynomiell in n und $1/\varepsilon$ (stärkste Form). *Pseudopolynomiell* = polynomiell in den Zahlenwerten (nicht in der Länge!).

8.2 TSP: der Algorithmus Schritt für Schritt

▲ **Satz 8.3** Allgemeines TSP ist gar nicht approximierbar

Für beliebige Distanzen: keine beschränkte Güte möglich (außer $P = NP$).

Beweisidee: Reduziere HK – echte Kanten Distanz 1, fehlende riesig. Ein Algorithmus mit Güte α müsste die billige Tour erkennen und löste damit HK.

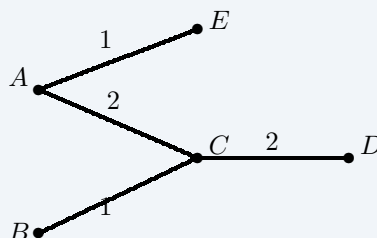
Ausweg: *metrisches* TSP mit Dreiecksungleichung $d(i, j) \leq d(i, k) + d(k, j)$. Dort funktioniert folgender Dreisprung.

👉 Idee: ΔTSP_1 (Güte 2): Spannbaum $\rightarrow$ verdoppeln $\rightarrow$ Eulerkreis $\rightarrow$ abkürzen

Der *minimale Spannbaum* ist billiger als die optimale Tour. Verdoppeln macht alle Grade gerade (Eulerkreis existiert). Abkürzen des Eulerkreises verlängert dank Dreiecksungleichung nicht.

- Beispiel 8.4 ΔTSP_1 durchgerechnet (Skript-Zahlen)

Städte A, B, C, D, E mit $d(A, E) = 1$, $d(B, C) = 1$, $d(A, C) = 2$, $d(C, D) = 2$ (übrige 2/3).
Links der **minimale Spannbaum** (fett, Gewicht 6):



Ablauf: MST-Gewicht 6 $\rightarrow$ verdoppeln (Gewicht 12) $\rightarrow$ Eulerkreis $[C, A, E, A, C, B, C, D, C]$ $\rightarrow$ abkürzen zu Tour $[C, A, E, B, D, C]$ mit Länge **10** $\leq 2 \cdot \text{OPT}$ (optimale Tour hat Länge 8).

 Idee: ΔTSP_2 (Christofides, Güte 1,5): nur die schiefen Knoten reparieren

Statt *alle* Kanten zu verdoppeln, verbindet Christofides nur die Knoten mit *ungeradem* Grad im MST durch ein *minimales perfektes Matching*. Das ist billiger ($\text{Matching} \leq \text{OPT}/2$) und liefert Güte 1,5.

● **Beispiel 8.5 Christofides am selben Graphen**

Ungerade Grade im MST: $X = \{B, C, D, E\}$. Matching-Kosten: $d(B, C) = 1$, $d(D, E) = 2$, $d(B, E) = 2$, $d(C, D) = 2$, $d(B, D) = 3$, $d(C, E) = 3$. Minimales perfektes Matching: $\{B-C, D-E\}$, Gewicht 3. Ergebnis-Tour $[C, B, A, E, D, C]$ mit Länge **9** – besser als ΔTSP_1 .

 Stolperstein

Das Matching läuft nur über die *ungerad-gradigen* MST-Knoten, nicht über alle. Und ohne Dreiecksungleichung bricht das Abkürzen zusammen.

8.3 Knapsack: warum Gier scheitert – und wie man sie rettet

● **Beispiel 8.6 Greedy kann beliebig schlecht sein**

Zwei Gegenstände $(w, p) = (1, 1)$ und $(B, B - 1)$, Kapazität B . Greedy nimmt (nach Dichte) den kleinen – Gewinn 1. Optimal wäre der große – Gewinn $B - 1$. Das Verhältnis $B - 1$ wächst unbeschränkt.

 Idee: Modified Greedy (MGA): Güte 2

Rechne Greedy *und* „nur den profitabelsten Einzel-Gegenstand“, gib das Bessere aus. Greedy verliert nur am einen nicht mehr passenden „Split-Item“ – dessen Profit fängt das Einzel-Item ab. Näher ans Optimum kommen *Sahni* ($1 + 1/k$, ein PTAS) und das *FPTAS* (Profite herunterskalieren, exakt per DP; $1 + \varepsilon$ in $O(n^3/\varepsilon)$).

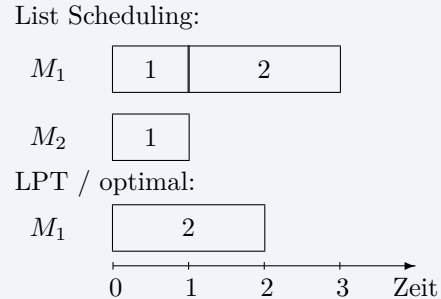
8.4 Scheduling: Regeln und Gantt-Bild

■ **Definition 8.7 List Scheduling und LPT**

List Scheduling: jeden Job auf die momentan *leerste* Maschine. *LPT*: zuerst *absteigend* sortieren, dann List Scheduling.

● Beispiel 8.8 Warum die Reihenfolge zählt (Gantt, $m = 2$)

Jobs der Größen 1, 1, 2. Oben List Scheduling in dieser Reihenfolge (Makespan 3), unten optimal / LPT (Makespan 2). Balkenlänge = Zeit.



List Scheduling stapelt die 2 auf eine schon belegte Maschine (Makespan 3). LPT legt die große 2 zuerst und erreicht Makespan 2.

▲ Satz 8.9 Güten

List Scheduling: $2 - \frac{1}{m}$; LPT: $\frac{4}{3} - \frac{1}{3m}$ – beide scharf.

Beweisidee: zwei Universalschranken – $\text{OPT} \geq \frac{1}{m} \sum p_i$ (Durchschnittslast) und $\text{OPT} \geq p_{\max}$ (größter Job). Der letzte Job auf der vollsten Maschine lag dort, als sie die leerste war.

8.5 MAX-3-SAT: Güte 2 mit zwei Belegungen

🔍 Idee: β_0 und β_1

Maximiere die Zahl erfüllter Klauseln. Werte nur „alles falsch“ (β_0) und „alles wahr“ (β_1) aus, nimm das Bessere. Jede Klausel hat ein positives *oder* negatives Literal, wird also von einem der beiden erfüllt. Damit erfüllen beide zusammen $\geq m$ Klauseln, die bessere $\geq m/2 \geq \text{OPT}/2$. Güte 2.

? Abruf – erst selbst beantworten

A16. Nenne die vier Schritte von ΔTSP_1 und sage bei jedem, warum er die Güte-2-Schranke stützt.

A17. Warum ist Christofides besser als ΔTSP_1 , obwohl beide mit demselben minimalen Spannbaum starten?

A18. Konstruiere (Idee) eine Instanz, bei der List Scheduling die Schranke $2 - \frac{1}{m}$ wirklich erreicht.

9 Jenseits von NP: das Halteproblem

★ In diesem Kapitel

Ein Blick über den Rand: ein Problem, das NP-schwer ist, aber *nicht* in NP liegt – weil es überhaupt nicht lösbar ist. Das zeigt, dass „schwer“ und „in NP“ wirklich zwei Achsen sind.

■ Definition 9.1 Halteproblem

Gegeben die Beschreibung eines Programms M und einer Eingabe w : hält M auf w irgendwann, oder läuft es ewig?

Das Halteproblem ist *unentscheidbar* – kein Algorithmus löst es, mit keiner Laufzeit. Trotzdem ist es NP-schwer.

☆ Aha

Man reduziert SAT darauf: Baue zu einer Formel φ ein Programm M_φ , das *alle* Belegungen durchprobiert und genau dann hält, wenn eine erfüllende existiert. Dann gilt φ erfüllbar $\iff M_\varphi$ hält. Der Clou: M_φ läuft womöglich ewig – aber die *Konstruktion* aus φ ist billig, und nur darauf kommt es bei einer Reduktion an. Also ist das Halteproblem mindestens so schwer wie SAT, liegt aber nicht in NP. **NP-schwer $\neq$ NP-vollständig.**

10 Das große Ganze

★ In diesem Kapitel

Wir ziehen die Fäden zusammen: eine Übersicht und ein Wahr/Falsch-Test zur Selbstkontrolle.

10.1 Die Reise auf einen Blick

Station	Kernidee
P	effizient <i>lösbar</i> (polynomiell)
NP	effizient <i>überprüfbar</i> (Zertifikat prüfen bzw. raten)
$P \subseteq NP$	Lösen ist mindestens so schwer wie Prüfen; Gleichheit offen
$A \leq_p B$	„ B mindestens so schwer wie A “; Richtung: bekannt $\leq_p$ neu
NP-vollständig	die schwersten in NP; SAT ist der Anker (Cook–Levin)
Vererbungschorollar	neues Problem schwer: eine Reduktion + Nachweis $\in$ NP
ETH	schärfer: konkrete $2^{o(\cdot)}$ -Schranken (Kontraposition, Sparsification)
Approximation	schwere Optimierung: beweisbar gute Näherung mit Gütegarantie

10.2 Wahr oder falsch? (Begründung zählt)

Aussage	W/F	Begründung
Löst eine ratende Maschine ein Problem in Polyzeit, dann auch eine gewöhnliche.	F*	Das wäre $P = NP$ – unbekannt.
$A \text{ NP-schwer} \Rightarrow A \in NP$.	F	Halteproblem: NP-schwer, nicht einmal lösbar.
$L \in NP$ und $L \leq_p 3\text{-SAT} \Rightarrow L \text{ NP-vollständig}$.	F	Falsche Richtung – <i>auf</i> 3-SAT reduzieren zeigt keine Schwere.
$3\text{-SAT} \leq_p L$ und $L \leq_p 3\text{-SAT} \Rightarrow L \text{ NP-vollständig}$.	W	Erstes gibt Schwere, zweites gibt $L \in NP$.
ETH falsch $\Rightarrow P = NP$.	F	$2^{o(n)}$ kann superpolynomiell bleiben.

* nicht als wahr beweisbar; äquivalent zu $P = NP$.

☆ Aha

Du hast das *Warum* beisammen: Was Probleme schwer macht, wie Schwere von einem zum anderen fließt, wo die Grenzen liegen, was man bei schweren Optimierungsproblemen tut. Das ist das Fundament.

Der nächste Schritt (Schritt 3 und 4 deines Lernmodells). Nimm die Präsenz-, Haus- und Klausuraufgaben. Führe dort selbst aus, was hier als Idee stand: Reduktionen *vollständig* beweisen (beide Richtungen!), Güte-Schranken herleiten, Worst-Case-Instanzen bauen. Dieses Heft gab dir das Verständnis – die Aufgaben geben dir das Können.

A Lösungen zu den Abruf-Fragen

Erst selbst beantwortet? Dann hier vergleichen.

A1. Ein konstanter Faktor ($1000 \approx 2^{10}$) verschiebt die exponentielle Kurve nur um eine *additive* Konstante: du schaffst rund 20 Knoten mehr – einmalig. Das Problem wächst weiter exponentiell, die Hardware nur einmal.

A2. „Kennen keinen“ = niemand hat einen Algorithmus gefunden (Wissenslücke). „Gibt keinen“ = bewiesene Nichtexistenz. Für die Clique gilt nur Ersteres – ein Nichtexistenzbeweis fehlt (P-vs-NP offen).

A3. Ja. n^4 ist polynomiell (fester Exponent $d = 4$), also in P und im Sinne der Theorie effizient.

A4. Bis K zu zählen kostet K Schritte. K ist aber exponentiell in seiner Eingabelänge $O(\log K)$. Die Laufzeit ist also exponentiell in der Eingabegröße.

A5. Zertifikat: eine Knotenmenge C . **Verifizierer:** prüfe, ob jede Kante einen Endpunkt in C hat ($O(|E|)$) und ob $|C| \leq k$; akzeptiere genau dann. **Korrektheit:** gibt es ein VC der Größe $\leq k$, wird es akzeptiert, sonst keines. **Laufzeit** polynomiell. (NDTM: C raten, dann prüfen.)

A6. Der Verifizierer bekommt das Zertifikat *geschenkt* und prüft nur. Der Löser muss die Lösung selbst *finden* – und dafür können exponentiell viele Kandidaten in Frage kommen. Prüfen startet einen Schritt weiter als Lösen.

A7. $\text{CLIQUE} \leq_p \text{VERTEX COVER}$ – vom bekannt schweren CLIQUE auf das neue VC. So erbt VC die Schwere. Die Gegenrichtung würde über VC nichts aussagen.

A8. $A \leq_p B$ bedeutet: B ist *mindestens so schwer* wie A .

A9. (1) eine Reduktion von einem bekannten NP-vollständigen Problem auf Y ; (2) der Nachweis $Y \in \text{NP}$. Vergessen wird leicht (2) – ohne ihn hat man nur NP-*Schwere*, nicht Vollständigkeit. (Beim Reduktionsteil vergisst man gern die Rückrichtung der Korrektheit.)

A10. SUBSET SUM ist NP-vollständig. Läge *ein* NP-vollständiges Problem in P, ließe sich jedes $L \in \text{NP}$ via $L \leq_p \text{SUBSET SUM}$ effizient lösen – also $\text{P} = \text{NP}$.

A11. Zertifikat: eine Reihenfolge (Permutation) der Knoten. **Verifizierer:** prüfe, ob jeder Knoten genau einmal vorkommt und jede aufeinanderfolgende (und die schließende) Kante existiert – $O(|V| + |E|)$. **NDTM:** Reihenfolge raten, dann prüfen.

A12. Bei Basis $n + 1$ und höchstens n Summanden gibt es *keinen Übertrag*; jede Ziffer zählt für sich. Bei Basis 2 könnten Überträge falsche Teilsummen zufällig aufs Ziel bringen. Die große Basis macht „Summe trifft Ziel“ äquivalent zu „jede Stelle genau einmal überdeckt“.

A13. C ist Vertex Cover $\iff V \setminus C$ ist Independent Set (im selben Graphen). Überführung: aus der VC-Instanz (G, k) mach die IS-Instanz $(G, n - k)$.

A14. Nein. „ETH falsch“ heißt nur: 3-SAT geht in $2^{o(n)}$ – das kann noch superpolynomiell sein (z. B. $2^{\sqrt{n}}$). Umgekehrt gilt: $\text{P} = \text{NP}$ würde die ETH widerlegen.

A15. Die ETH liefert eine Schranke in der Variablenzahl n . Viele Reduktionen erzeugen ihre Instanzgröße aus der Klauselzahl m , und m kann $\gg n$ sein. Eine $2^{o(m)}$ -Schranke folgt daher *nicht* direkt aus der ETH – das Sparsification-Lemma liefert sie.

A16. (1) Minimaler Spannbaum T : $w(T) \leq \text{OPT}$. (2) Kanten verdoppeln: Gesamtlänge $2w(T) \leq 2\text{OPT}$. (3) Eulerkreis: nutzt die geraden Grade, ändert die Länge nicht. (4) Abkürzen: verlängert wegen der Dreiecksungleichung nicht. Also $d(R) \leq 2\text{OPT}$.

A17. ΔTSP_1 verdoppelt *alle* MST-Kanten – das kostet zusätzlich rund $w(T) \approx \text{OPT}$. Christofides fügt stattdessen nur ein minimales perfektes Matching auf den ungerad-gradigen Knoten hinzu (Kosten $\leq \text{OPT}/2$). Deshalb $1,5 \text{OPT}$ statt 2OPT .

A18. $m(m - 1)$ Jobs der Größe 1, danach *ein* Job der Größe m – in dieser Reihenfolge. List Scheduling verteilt die Einsen gleichmäßig (je $m - 1$) und legt den großen Job obendrauf: Makespan $2m - 1$. Optimal ist m (großer Job allein). Verhältnis $\frac{2m-1}{m} = 2 - \frac{1}{m}$.