

Das vollständige Fundament

Komplexitätstheorie und Approximation, ausführlich erklärt
Analyse von Algorithmen und Komplexität – CAU Kiel, SS 2026

*Das gesamte Wissen der Vorlesung – jedes Problem einzeln,
mit Idee statt Abkürzung, mit Bild wo es hilft.*

Was dieses Heft leistet. Es erklärt den kompletten Stoff des Kurses, ohne etwas zu überspringen. Der Kern ist ein *Katalog aller Probleme*. Zu jedem Problem findest du vier Dinge, jedes ausführlich:

1. die **Definition** – was gefragt ist, mit Beispiel;
2. **warum es in NP liegt** – die Idee des Zertifikats, Schritt für Schritt erklärt;
3. **woher seine Schwere kommt** – die Reduktion, die es schwer macht, mit ihrer Konstruktions-Idee;
4. **was daraus folgt** – die Reduktionen, die von ihm ausgehen, jede mit eigener Idee.

Danach folgen die untere-Schranken-Theorie (ETH) und die Approximation – auch dort steht bei jedem Ergebnis die *Idee* des Beweises.

Zwei Versprechen. Erstens: Kein Problem wird mit einem anderen zusammengelegt, auch wenn sie sich ähneln – jedes bekommt seinen eigenen, vollständigen Abschnitt. Zweitens: Nichts wird als „trivial“ oder „analog“ abgetan; jeder Nachweis wird wirklich ausgeführt.

Die farbigen Kästen.

★ Intuition	die Idee in einfachen Worten
☞ Analogie	ein Bild aus dem Alltag
■ Definition	die präzise Fassung
▲ Satz	ein Ergebnis mit erklärter Beweisidee
● Beispiel	durchgerechnet, mit echten Zahlen
☞ Idee	die Konstruktion hinter einer Reduktion oder einem Beweis
☆ Aha	die Pointe
✗ Stolperstein	der häufige Fehler
➤ Zusammenhang	wie es ins große Ganze passt

Inhalt

Teil A – Die Grundlagen	4
1 Ein Problem, das sich wehrt	4
2 Effizient lösbar: die Klasse P	4
3 Effizient überprüfbar: die Klasse NP	5
4 Reduktionen: Probleme vergleichen	7
5 NP-schwer, NP-vollständig, und der Anker	7
Teil B – Der Problem-Katalog	9
6 SAT	9
7 3-SAT	11
8 k -Clique	13
9 Independent Set	15
10 Vertex Cover	16
11 Feedback Vertex Set	18
12 Δ -Cover (Dreiecksüberdeckung)	19
13 k -Color (Färbung)	20
14 Hamiltonkreis	22
15 TSP (Traveling Salesman)	23
16 3-dimensionales Matching	24
17 3-Exact Cover	26
18 SubSet Sum	27
19 Partition	30
20 Knapsack (Rucksackproblem)	31

21 $P C_{\max}$ (Scheduling)	32
22 Hitting Set	33
23 Set Cover	34
24 Dominating Set	35
25 Longest Path	36
26 Das Halteproblem – schwer, aber nicht in NP	37
Teil C – Untere Schranken: die ETH	38
27 Die Bausteine der ETH	38
28 Untere Schranken für die einzelnen Probleme	40
Teil D – Approximation	41
29 Der Gütebegriff	41
30 TSP ist im Allgemeinen nicht approximierbar	42
31 ΔTSP_1 : der Algorithmus als Bilderfolge	43
32 Christofides: schlauer verdoppeln	44
33 Knapsack: Greedy	46
34 Knapsack: Modified Greedy	47
35 Knapsack: Sahni-Algorithmus	47
36 Knapsack: FPTAS	47
37 Scheduling: List Scheduling	48
38 Scheduling: LPT	49
39 MAX-3-SAT: Güte 2 mit zwei Belegungen	49
40 2-Approximation für Vertex Cover	50
41 Zum Schluss	50

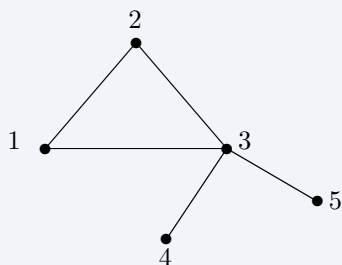
Teil A – Die Grundlagen

1 Ein Problem, das sich wehrt

Bevor wir die Theorie aufbauen, brauchen wir einen Grund für sie. Diesen Grund liefert ein einzelnes Problem, das harmlos aussieht und trotzdem jeden Computer überfordert: das Cliquesproblem.

Ein *Graph* ist ein Netz aus Punkten (*Knoten*) und Verbindungslinien (*Kanten*). Eine *Clique* ist eine Gruppe von Knoten, in der jeder mit jedem direkt verbunden ist. Das Cliquesproblem fragt: Gibt es in einem Graphen eine Clique aus mindestens k Knoten?

● Beispiel 1.1 Eine Clique erkennen



Die Gruppe $\{1, 2, 3\}$ bildet ein Dreieck – alle drei Kanten sind da, also eine Clique der Größe 3. Die Gruppe $\{2, 3, 4\}$ ist keine, weil zwischen 2 und 4 die Kante fehlt.

Der naheliegende Algorithmus probiert alle Gruppen aus k Knoten durch. Bei n Knoten und $k = n/2$ sind das mindestens $2^{n/2}$ Gruppen – eine Zahl, die sich mit jedem zusätzlichen Knoten ungefähr verdoppelt. Schon bei hundert Knoten ist das jenseits des Machbaren, und kein schnellerer Rechner holt exponentielles Wachstum je ein.

☆ Aha

Die Komplexitätstheorie fragt nicht „wie löse ich die Clique schnell?“, sondern „ist die Clique überhaupt schnell lösbar – oder gehört sie zu einer großen Familie von Problemen, die alle gemeinsam schwer sind?“ Genau diese zweite, tiefere Frage beantwortet der ganze Kurs. Wir bauen dafür jetzt die Werkzeuge.

2 Effizient lösbar: die Klasse P

Wir müssen zuerst „schnell“ präzise machen. Die Laufzeit eines Algorithmus misst man immer *relativ zur Größe seiner Eingabe*, geschrieben n , und man betrachtet den schlechtesten Fall – weil man eine Garantie will, keine Hoffnung.

■ Definition 2.1 Klasse P

P ist die Menge aller Entscheidungsprobleme, für die ein Algorithmus existiert, der sie in *polynomieller* Zeit löst – also mit Laufzeit $O(n^d)$ für eine feste Zahl d (etwa n , n^2 , n^3).

★ Intuition

Man setzt „polynomiell“ mit „effizient“ gleich. Das ist eine Konvention, aber eine gute: Der entscheidende Bruch verläuft zwischen polynomiell und exponentiell. Eine polynomielle Laufzeit wächst zahm, eine exponentielle explodiert. Und die Grenze bleibt dieselbe, egal welchen realistischen Rechnertyp man zugrunde legt.

Ein *Entscheidungsproblem* ist eine Frage mit Antwort Ja oder Nein. Statt „wie groß ist die größte Clique?“ fragt man „gibt es eine Clique mit $\geq k$ Knoten?“. Formal identifiziert man ein Problem mit der Menge seiner Ja-Eingaben. Beispiele in P: Sortieren, kürzeste Wege, minimale Spannbäume, Matching – für all das kennt man schnelle Algorithmen.

✗ Stolperstein

Zahlen sind binär kodiert. Eine Zahl K steht als Eingabe nur mit ihren Ziffern da, also mit Länge $O(\log K)$. Ihr *Wert* K ist exponentiell größer als diese Länge. Ein Algorithmus, der „bis K zählt“, macht K Schritte und ist damit *nicht* polynomiell in der Eingabelänge. An diesem Punkt hängt später die Schwere der Zahlprobleme.

3 Effizient überprüfbar: die Klasse NP

Jetzt kommt die zweite, wichtigere Klasse. Sie beruht auf einer Beobachtung, die den ganzen Kurs trägt: *Eine Lösung zu finden ist oft schwer – eine vorgelegte Lösung zu prüfen ist leicht.*

🗉 Analogie

Ein schweres Sudoku zu lösen dauert eine halbe Stunde. Ein fertig ausgefülltes Gitter zu prüfen dauert eine Minute: Zeilen, Spalten, Blöcke abhaken. Genau diese Asymmetrie zwischen Lösen und Prüfen ist der Kern von NP.

3.1 Die erste Sicht: Zertifikat und Verifizierer

Die „vorgelegte Lösung“ bekommt einen Namen.

■ Definition 3.1 Verifizierer, Zertifikat

Ein *Verifizierer* für ein Problem L ist ein Algorithmus A , der zwei Eingaben bekommt: die eigentliche Instanz x und einen Lösungsvorschlag c . Er erfüllt

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

Der Vorschlag c heißt *Zertifikat*.

Lies die beiden Richtungen einzeln, beide sind wichtig. Ist x eine Ja-Instanz, so *gibt es* ein Zertifikat, das A überzeugt. Ist x eine Nein-Instanz, so überzeugt A *kein* Zertifikat – man kann ihm vorlegen, was man will.

■ Definition 3.2 Klasse NP

NP ist die Menge aller Probleme, die einen Verifizierer besitzen, der ein *polynomiell langes* Zertifikat in *polynomieller* Zeit prüft.

★ Intuition

NP heißt „effizient *überprüfbar*“, nicht „effizient *lösbar*“. Das Zertifikat ist die kurze Antwort auf die Frage „und woher weiß ich, dass das stimmt?“. Wer die Lösung schon in der Hand hält, kann sie billig kontrollieren – sie zu *finden* ist die eigentliche Arbeit.

3.2 Die zweite Sicht: nichtdeterministisches Raten

Es gibt eine gleichwertige zweite Art, NP zu sehen. Stell dir einen Algorithmus vor, der an jeder Verzweigung nicht *eine* Möglichkeit wählt, sondern sich in Kopien aufteilt und *alle gleichzeitig* verfolgt. So ein Algorithmus heißt *nichtdeterministisch*. Er „akzeptiert“, wenn *irgendeine* Kopie ans Ziel kommt; für eine Nein-Instanz kommt keine an.

Sein Arbeitsmuster ist immer dasselbe: *rate die Lösung, dann prüfe sie*. Die Folge der Rate-Entscheidungen *ist* genau das Zertifikat. Deshalb beschreiben „richtig raten können“ und „ein Zertifikat geschenkt bekommen“ dieselbe Fähigkeit – und dieselbe Klasse NP.

Diese Rate-Sicht ist praktisch, wenn man einen NP-Nachweis in Worten führt: Man sagt, was die Maschine rät (das Zertifikat) und wie sie es danach deterministisch prüft.

3.3 P steckt in NP

▲ Satz 3.3 $P \subseteq NP$

Jedes effizient lösbare Problem ist auch effizient überprüfbar.

Warum: Wer ein Problem selbst schnell lösen kann, braucht das Zertifikat gar nicht. Er ignoriert es und rechnet die Antwort direkt aus. Der Löser ist damit ein besonders fauler Verifizierer.

Die große offene Frage ist die Umkehrung: Gilt $P = NP$? Ist alles, was man schnell prüfen kann, auch schnell lösbar? Niemand weiß es. Es ist eines der sieben Millennium-Probleme. Die meisten glauben $P \neq NP$. Der ganze Rest des Kurses ist der Umgang mit dieser Unwissenheit.

4 Reduktionen: Probleme vergleichen

Jetzt das wichtigste Werkzeug. Mit einer *Reduktion* vergleicht man zwei Probleme nach Schwierigkeit – ohne für eines von beiden einen Algorithmus zu kennen.

■ Definition 4.1 Polynomielle Reduktion $A \leq_p B$

Eine *Reduktion* von A auf B ist eine Funktion f , die jede A -Eingabe w in eine B -Eingabe $f(w)$ übersetzt, sodass die Antworten zusammenpassen:

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

Sie ist *polynomiell*, wenn f in polynomieller Zeit berechenbar ist. Man schreibt $A \leq_p B$.

★ Intuition

$A \leq_p B$ bedeutet: „ B ist *mindestens so schwer* wie A .“ Denn wer B schnell lösen kann, löst über die billige Umformung f auch A schnell: erst $f(w)$ berechnen, dann den B -Löser fragen, dessen Antwort ist auch die für w . Die Schwierigkeit fließt entlang des Pfeils.

✗ Stolperstein

Die Richtung ist der teuerste Fehler. Um zu zeigen, dass ein *neues* Problem Y schwer ist, reduziert man ein *bekannt schweres* X auf Y – also $X \leq_p Y$, „bekannt $\leq_p$ neu“. Die Reduktion nimmt eine X -Instanz und baut daraus eine Y -Instanz. Andersherum ($Y \leq_p X$) zeigt man nur, dass Y *höchstens* so schwer wie X ist – und das sagt über Y nichts.

▲ Satz 4.2 Transitivität

$A \leq_p B$ und $B \leq_p C$ ergeben zusammen $A \leq_p C$.

Warum: Man schaltet die beiden Umformungen hintereinander. Die Zwischenausgabe ist polynomiell groß, also bleibt die Verkettung polynomiell.

Deshalb kann eine ganze *Kette* von Reduktionen die Schwere Glied für Glied weitertragen. Genau diese Kette bauen wir im Problem-Katalog auf.

5 NP-schwer, NP-vollständig, und der Anker

Mit der Reduktion können wir „die schwersten Probleme in NP“ präzise fassen.

■ Definition 5.1 NP-schwer, NP-vollständig

Ein Problem L_0 heißt *NP-schwer*, wenn sich *jedes* $L \in \text{NP}$ auf L_0 reduzieren lässt ($L \leq_p L_0$). Es heißt *NP-vollständig*, wenn es NP-schwer ist **und** selbst in NP liegt.

★ Intuition

Zwei Schranken treffen sich. *NP-schwer* ist eine untere Schranke: mindestens so schwer wie alles in NP. *In NP* ist eine obere Schranke: nicht schwerer als NP. *NP-vollständig* heißt beides – die schwersten Probleme innerhalb von NP.

▲ Satz 5.2 Ein Algorithmus für eines löst alle

Ist L_0 NP-vollständig, dann gilt $P = \text{NP}$ genau dann, wenn $L_0 \in P$.

Warum: Läge L_0 in P, könnte man jedes $L \in \text{NP}$ lösen – erst die Reduktion $L \leq_p L_0$ rechnen, dann den schnellen L_0 -Löser. Damit läge ganz NP in P.

Damit die Reduktionskette starten kann, braucht man ein *erstes* NP-vollständiges Problem, das man von Hand als schwer nachweist. Das lieferten Cook und Levin.

▲ Satz 5.3 Cook (1971), Levin (1973)

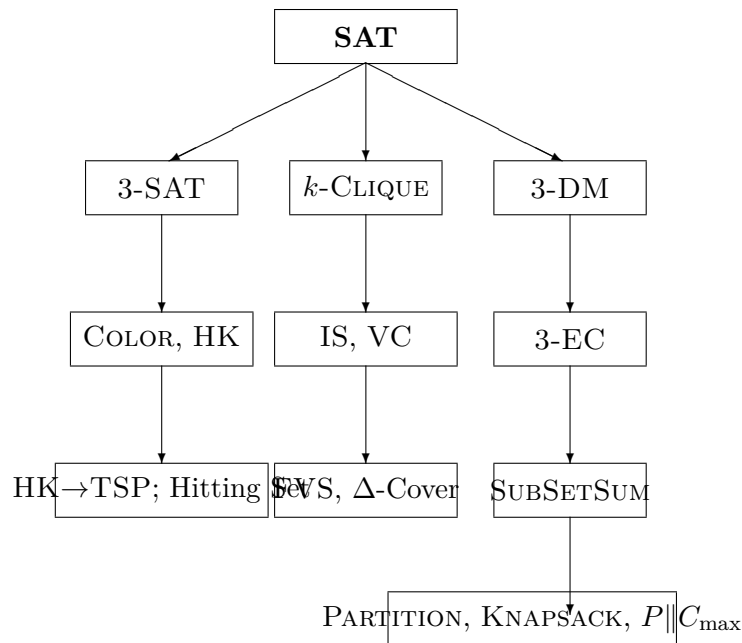
SAT ist NP-vollständig – das erste solche Problem.

Die Idee: Man nimmt ein *beliebiges* $L \in \text{NP}$ mit seiner ratenden Maschine und gießt deren gesamte Rechnung in eine große logische Formel. Variablen beschreiben Aussagen wie „zur Zeit t ist die Maschine im Zustand q “ oder „zur Zeit t steht auf Feld i das Symbol a “. Teilformeln erzwingen einen gültigen Ablauf. Das Entscheidende: Eine erfüllende Belegung entspricht *genau* einem akzeptierenden Rechenweg. Also ist die Formel erfüllbar genau dann, wenn L die Eingabe akzeptiert. Da L beliebig war, ist SAT schwer für ganz NP.

🔍 Idee: Das Vererbungskorollar – so wächst der Katalog

Um ein *neues* Problem Y als NP-vollständig nachzuweisen, genügen zwei Dinge: (1) eine Reduktion $X \leq_p Y$ von einem *bereits bekannten* NP-vollständigen X , und (2) der Nachweis $Y \in \text{NP}$. Denn dann gilt für jedes $L \in \text{NP}$ die Kette $L \leq_p X \leq_p Y$ – also ist Y NP-schwer, und mit $Y \in \text{NP}$ auch NP-vollständig.

Ab hier braucht man nie wieder „alle $L \in \text{NP}$ “ zu betrachten. Jedes neue Problem hängt sich über *eine* Reduktion an ein schon bekanntes. Der folgende Katalog geht genau so vor: Er beginnt bei SAT und hängt jedes Problem an ein früheres. Deshalb steht bei jedem Problem, *woher* seine Schwere kommt (die eingehende Reduktion) und *wohin* sie weiterfließt (die ausgehenden Reduktionen).



Teil B – Der Problem-Katalog

Jetzt kommt das Herzstück. Wir gehen alle Probleme des Kurses durch – jedes für sich, keines mit einem anderen zusammengelegt. Der Aufbau ist bei jedem gleich: erst die **Definition** mit Beispiel, dann **warum es in NP liegt**, dann **woher seine Schwere kommt** (die eingehende Reduktion) und schließlich **was daraus folgt** (die ausgehenden Reduktionen). Die Reihenfolge folgt der Reduktionskette: Jedes Problem wird an ein früheres gehängt, das du schon kennst.

6 SAT

SAT steht am Anfang von allem – es ist das erste NP-vollständige Problem und der Anker, an dem die ganze Kette hängt. Es geht um Formeln der Aussagenlogik in einer festen Form.

■ Definition 6.1 SAT

Eine *konjunktive Normalform* (KNF) ist ein UND von *Klauseln*, wobei jede Klausel ein ODER von *Literals* ist; ein Literal ist eine Variable x_j oder ihre Verneinung $\neg x_j$. SAT fragt: Gegeben eine KNF-Formel α , gibt es eine Belegung der Variablen mit wahr/falsch, die die ganze Formel wahr macht – also jede Klausel erfüllt?

● Beispiel 6.2 SAT von Hand

$\alpha = (x_1 \vee x_{10}) \wedge (x_1 \vee \neg x_{10}) \wedge (x_{10})$. Die dritte Klausel besteht nur aus x_{10} und zwingt daher $x_{10} = \text{wahr}$. Damit die zweite Klausel wahr wird – $\neg x_{10}$ ist jetzt falsch – muss $x_1 = \text{wahr}$ sein. Mit beiden auf wahr ist auch die erste Klausel erfüllt. Die Formel ist also erfüllbar, ein Zeuge ist $x_1 = x_{10} = \text{wahr}$.

Warum SAT in NP liegt

Wir führen den Nachweis vollständig, in vier Gedanken.

Das Zertifikat. Die „Lösung“ einer erfüllbaren Formel ist eine Belegung ψ – für jede Variable ein Wahrheitswert. Man kodiert sie mit einem Bit pro Variable. Hat die Formel v Variablen, so ist das Zertifikat v Bits lang, also höchstens so lang wie die Formel selbst und damit polynomiell.

Die Prüfung. Der Verifizierer bekommt die Formel α und die Belegung ψ . Er setzt ψ in α ein und wertet Klausel für Klausel aus: Eine Klausel ist wahr, sobald eines ihrer Literale wahr ist. Er akzeptiert genau dann, wenn *alle* Klauseln wahr sind.

Warum das korrekt ist – beide Richtungen. Ist α erfüllbar, so gibt es eine erfüllende Belegung; legt man genau sie als Zertifikat vor, macht sie jede Klausel wahr und der Verifizierer akzeptiert. Ist α dagegen unerfüllbar, so macht *keine* Belegung alle Klauseln wahr – egal, welches ψ man vorlegt, mindestens eine Klausel bleibt falsch und der Verifizierer lehnt ab. Beide Richtungen stimmen also.

Die Laufzeit. Das Einsetzen und Auswerten geht die Formel einmal durch, kostet also lineare Zeit in der Formellänge. Damit ist der Verifizierer polynomiell, und es folgt $\text{SAT} \in \text{NP}$.

Woher die Schwere kommt

📖 Idee: Cook–Levin: jede NDTM-Rechnung wird zu einer Formel

SAT ist nicht über eine Reduktion von einem *anderen* Problem schwer – es ist der Startpunkt. Cook und Levin zeigen direkt, dass sich *jedes* $L \in \text{NP}$ auf SAT reduziert. Zu einem L mit ratender Maschine M und Eingabe u baut man eine KNF α_u , deren Variablen die gesamte Rechnung von M beschreiben: „zur Zeit t ist der Zustand q “, „zur Zeit t steht auf Feld i das Symbol a “ und so weiter. Teilformeln erzwingen einen korrekten Anfang, korrekte Übergänge und einen akzeptierenden Schluss. Dann gilt: α_u ist erfüllbar genau dann, wenn M die Eingabe u akzeptiert. Also $u \in L \iff \alpha_u \in \text{SAT}$.

Was daraus folgt

Von SAT gehen mehrere Reduktionen aus – sie machen die nächsten Probleme schwer.

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

Eine lange Klausel wird mit einer neuen Hilfsvariablen in zwei kürzere zerlegt, etwa $(y_1 \vee y_2 \vee y_3 \vee y_4)$ in $(y_1 \vee y_2 \vee x) \wedge (\neg x \vee y_3 \vee y_4)$. Ist ursprünglich ein y_i wahr, lässt sich x passend setzen; ist keines wahr, entsteht in der Kette ein Widerspruch. So schrumpfen alle Klauseln auf höchstens drei Literale, ohne die Erfüllbarkeit zu ändern.

Idee: $\text{SAT} \leq_p k\text{-Clique}$: ein Knoten pro Literal

Man baut einen Graphen mit *einem Knoten pro Literalvorkommen* und verbindet zwei Knoten genau dann, wenn sie aus verschiedenen Klauseln stammen und sich nicht widersprechen. Gesucht wird eine Clique der Größe $k = m$ (Zahl der Klauseln). Eine solche Clique wählt aus jeder Klausel ein widerspruchsfreies, wahres Literal – also eine erfüllende Belegung. (Diese Reduktion zeichnen wir beim Cliquesproblem in Ruhe.)

Idee: $\text{SAT} \leq_p \text{3-dim. Matching}$: Zahnrad und Garbage Collection

Pro Variable baut man eine „Zahnrad“-Struktur aus Tripeln, die nur zwei konsistente Auswahlen zulässt – Variable wahr oder falsch. Klausel-Tripel prüfen, ob ein erfüllendes Literal gewählt wurde, und eine „Garbage-Collection“ aus weiteren Tripeln räumt die übrig gebliebenen Literale weg.

Idee: $\text{SAT} \leq_p \text{Halteproblem}$: alle Belegungen durchprobieren

Zu einer Formel φ baut man ein Programm, das der Reihe nach alle Belegungen durchprobiert und genau dann anhält, wenn es eine erfüllende findet. Dann hält das Programm genau dann, wenn φ erfüllbar ist. (Mehr dazu beim Halteproblem am Ende des Katalogs.)

7 3-SAT

3-SAT ist die „handliche“ Version von SAT. Kurze Klauseln lassen sich viel leichter in andere Probleme einbauen, deshalb starten fast alle späteren Reduktionen hier statt beim allgemeinen SAT.

■ Definition 7.1 3-SAT

Wie SAT, aber jede Klausel hat *höchstens drei* Literale. Frage: Ist die Formel erfüllbar?

● Beispiel 7.2 3-SAT von Hand

$\alpha = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$. Setzt man $x_2 = \text{wahr}$, so ist die zweite Klausel über x_2 erfüllt und die erste ebenfalls. Also erfüllbar, unabhängig von x_1 und x_3 .

Warum 3-SAT in NP liegt

Wir führen den Nachweis eigenständig und vollständig – auch wenn er dem von SAT ähnelt.

Das Zertifikat. Wieder eine Belegung ψ , ein Bit pro Variable, also polynomiell lang.

Die Prüfung. Der Verifizierer setzt ψ in die Formel ein und wertet jede Klausel aus. Weil jede Klausel jetzt höchstens drei Literale hat, kostet das Auswerten einer Klausel sogar nur konstante Zeit; er akzeptiert, wenn alle Klauseln wahr sind.

Warum das korrekt ist – beide Richtungen. Ist die Formel erfüllbar, wird die erfüllende Belegung akzeptiert. Ist sie unerfüllbar, bleibt bei jeder Belegung mindestens eine Klausel falsch, und der Verifizierer lehnt ab.

Die Laufzeit. Ein Durchlauf durch die Formel, also linear. Damit $3\text{-SAT} \in \text{NP}$.

Woher die Schwere kommt

🔍 Idee: $\text{SAT} \leq_p 3\text{-SAT}$: lange Klauseln zerlegen

Man ersetzt jede zu lange Klausel durch eine *Kette* kürzerer Klauseln, die durch neue Hilfsvariablen verbunden sind. Eine Klausel $(y_1 \vee y_2 \vee y_3 \vee y_4)$ wird etwa zu $(y_1 \vee y_2 \vee x) \wedge (\neg x \vee y_3 \vee y_4)$ mit einer neuen Variablen x . Bei längeren Klauseln setzt sich diese Kette fort: $(y_1 \vee y_2 \vee x_1) \wedge (\neg x_1 \vee y_3 \vee x_2) \wedge \dots$

Warum das die Erfüllbarkeit erhält – beide Richtungen. Ist die ursprüngliche Klausel erfüllt, so ist eines ihrer Literale y_i wahr; dann kann man die Hilfsvariablen der Kette so setzen, dass jede kurze Klausel wahr wird (vor y_i die x -Literale wahr, danach die $\neg x$ -Literale). Ist umgekehrt *kein* y_i wahr, so erzwingt die erste kurze Klausel, dass ihre Hilfsvariable wahr sein muss, diese erzwingt die nächste, und so fort – am Ende der Kette entsteht ein Widerspruch, die kurze Formel ist also nicht erfüllbar. Erfüllbar bleibt genau erfüllbar. Da jede Klausel nur konstant vergrößert wird, ist die Umformung polynomiell. Weil SAT NP-vollständig ist und $3\text{-SAT} \in \text{NP}$ gilt, ist auch 3-SAT NP-vollständig.

Was daraus folgt

🔍 Idee: $3\text{-SAT} \leq_p k\text{-Color}$: Wahrheit als Farbe

Man baut ein Gadget mit den Knoten $x_i, \bar{x}_i, v_i$ pro Variable, einem Klauselknoten pro Klausel und einem Zentrum z . Die v_i und z bilden eine Clique und erzwingen $n + 1$ Farben; dadurch bleibt jedem Paar $x_i, \bar{x}_i$ nur „wahr“ oder „falsch“. Die Klauselknoten sind so verdrahtet, dass sie sich genau dann korrekt färben lassen, wenn die Klausel erfüllt ist.

Idee: $3\text{-SAT}' \leq_p$ Hamiltonkreis: A- und B-Komponenten

Für die Variante mit genau drei Literalen pro Klausel baut man „A-Komponenten“, die die Variablensetzung kodieren (zwei Durchlaufrichtungen = wahr/falsch), und „B-Komponenten“ pro Klausel, die nur dann durchlaufbar sind, wenn mindestens ein Literal wahr ist. Ein Hamiltonkreis existiert genau dann, wenn eine erfüllende Belegung existiert.

Idee: $3\text{-SAT} \leq_p$ Hitting Set

Das Universum sind alle Literale $x_i, \bar{x}_i$. Pro Variable bildet man die Menge $\{x_i, \bar{x}_i\}$ (sie zwingt die Trefferwahl, genau ein Literal je Variable zu wählen), pro Klausel die Menge ihrer Literale (sie verlangt, dass ein erfüllendes Literal getroffen wird). Mit Schranke $k = n$ trifft eine Lösung genau dann alle Mengen, wenn die Formel erfüllbar ist.

8 k -Clique

Das Cliquenproblem aus der Einleitung, jetzt vollständig behandelt.

■ Definition 8.1 k -Clique

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Eine *Clique* ist eine Knotengruppe, in der alle paarweise durch eine Kante verbunden sind. Frage: Gibt es eine Clique der Größe $\geq k$?

● Beispiel 8.2 Eine Clique

Im Graphen mit $V = \{1, 2, 3, 4\}$ und Kanten $\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}$ ist $\{1, 2, 3\}$ eine Clique der Größe 3, weil die drei Kanten zwischen ihnen existieren. $\{1, 3, 4\}$ ist keine Clique, weil zwischen 1 und 4 die Kante fehlt.

Warum k -Clique in NP liegt

Das Zertifikat. Die Lösung ist die Knotengruppe $C \subseteq V$ selbst, etwa als Liste der gewählten Knoten. Sie hat höchstens $|V|$ Einträge, ist also polynomiell lang.

Die Prüfung. Der Verifizierer bekommt (G, k) und die Gruppe C . Er prüft zwei Dinge. Erstens, ob wirklich *jedes* Paar $\{u, v\}$ aus C durch eine Kante verbunden ist – dazu geht er alle Paare aus C durch und schlägt in der Kantenliste nach. Zweitens, ob $|C| \geq k$ gilt. Er akzeptiert, wenn beides zutrifft.

Warum das korrekt ist – beide Richtungen. Gibt es eine k -Clique, so ist genau sie ein Zertifikat, bei dem alle Paare verbunden sind und die Größe stimmt; der Verifizierer akzeptiert. Gibt es keine, so hat jede vorgelegte Gruppe der Größe $\geq k$ mindestens ein unverbundenes Paar, und der Verifizierer lehnt ab.

Die Laufzeit. Es gibt höchstens $\binom{|C|}{2} = O(|V|^2)$ Paare, jeder Kantennachschlag kostet $O(1)$;

die Größenprüfung kostet $O(|V|)$. Insgesamt $O(|V|^2)$, also polynomiell. Damit $k\text{-CLIQUE} \in \text{NP}$.

Woher die Schwere kommt

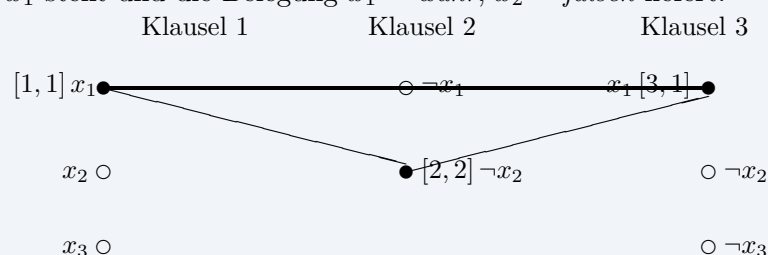
☞ Idee: $\text{SAT} \leq_p k\text{-CLIQUE}$: ein Knoten pro Literal

Zu einer KNF $F = F_1 \wedge \dots \wedge F_m$ baut man den Graphen so: Für jedes Literalvorkommen – das j -te Literal in Klausel i – setzt man einen Knoten $[i, j]$. Zwei Knoten werden verbunden, wenn sie aus *verschiedenen* Klauseln stammen *und* sich nicht widersprechen (nicht x gegen $\neg x$). Gesucht ist eine Clique der Größe $k = m$.

Warum das stimmt: Eine m -Clique muss aus jeder Klausel genau einen Knoten nehmen – Knoten derselben Klausel sind nie verbunden, es passt also höchstens einer pro Klausel hinein, und bei m Knoten ist es genau einer. Weil verbundene Knoten sich nicht widersprechen, wählt die Clique aus jeder Klausel ein widerspruchsfreies Literal. Setzt man diese Literale auf wahr, erfüllt man jede Klausel. Umgekehrt liefert jede erfüllende Belegung eine solche Clique.

● Beispiel 8.3 $\text{SAT} \leq_p \text{CLIQUE}$ am Beispiel

$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$, also $m = 3$ und $k = 3$. Es entstehen acht Knoten in drei Spalten (eine je Klausel). Fett die Lösungs-Clique $\{[1, 1], [2, 2], [3, 1]\}$, die für $x_1, \neg x_2, x_1$ steht und die Belegung $x_1 = \text{wahr}$, $x_2 = \text{falsch}$ liefert.



Die drei fetten Kanten bilden das Dreieck (die 3-Clique). Die vielen dünnen Kanten zwischen anderen verträglichen Paaren sind der Übersicht halber weggelassen.

Was daraus folgt

☞ Idee: $\text{Clique} \leq_p \text{Vertex Cover}$: Komplement

Aus (G, k) mit $n = |V|$ macht man $(\bar{G}, n - k)$ – Komplementgraph und Schranke $n - k$. Eine Clique der Größe k in G ist ein Independent Set in $\bar{G}$, und dessen Rest $V \setminus C$ ein Vertex Cover der Größe $n - k$. (Ausführlich beim Vertex Cover.)

Idee: Clique $\leq_p$ Independent Set: Komplement

Aus (G, k) macht man $(\bar{G}, k)$. Eine Clique in G ist genau ein Independent Set im Komplementgraphen $\bar{G}$ – dort fehlen genau die Kanten, die G hat.

Idee: Clique $\leq_p$ Clique-Nomember: isolierter Dummy

CLIQUE-NOMEMBER fragt nach einer k -Clique, die einen bestimmten Knoten v *nicht* enthält. Man fügt einfach einen neuen, isolierten Knoten v hinzu – er kann in keiner Clique liegen, also ändert die Zusatzbedingung nichts an der Antwort.

Idee: Clique $\leq_p$ Clique-Universal: universeller Knoten

CLIQUE-UNIVERSAL verlangt, dass die Instanz einen *universellen* Knoten enthält (mit allen verbunden). Man fügt einen solchen Knoten u hinzu und erhöht die Schranke auf $k + 1$. Jede k -Clique in G wird mit u zu einer $(k + 1)$ -Clique, und umgekehrt.

9 Independent Set

INDEPENDENT SET ist das genaue Gegenstück zur Clique: Dort sind alle verbunden, hier keiner.

■ Definition 9.1 Independent Set (IS)

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein *Independent Set* ist eine Knotengruppe, in der *keine* zwei Knoten verbunden sind. Frage: Gibt es ein Independent Set der Größe $\geq k$?

● Beispiel 9.2 Ein Independent Set

Im Graphen mit $V = \{1, 2, 3, 4\}$ und Kanten $\{1, 2\}, \{2, 3\}, \{3, 4\}$ ist $\{1, 3\}$ ein Independent Set der Größe 2 – zwischen 1 und 3 gibt es keine Kante. Auch $\{1, 4\}$ ist eines. $\{1, 2\}$ ist keines, weil 1 und 2 verbunden sind.

Warum Independent Set in NP liegt

Das Zertifikat. Die Knotengruppe $I \subseteq V$, polynomiell lang.

Die Prüfung. Der Verifizierer geht alle Paare aus I durch und prüft, dass *keines* von ihnen durch eine Kante verbunden ist; außerdem prüft er $|I| \geq k$. Er akzeptiert, wenn beides zutrifft.

Warum das korrekt ist – beide Richtungen. Gibt es ein Independent Set der Größe $\geq k$, wird es akzeptiert. Gibt es keines, enthält jede vorgelegte Gruppe der Größe $\geq k$ ein verbundenes Paar, und der Verifizierer lehnt ab.

Die Laufzeit. Wieder $O(|V|^2)$ Paare mit je $O(1)$ -Kantennachschlag, plus $O(|V|)$ für die Größe. Insgesamt $O(|V|^2)$, also IS $\in$ NP.

Woher die Schwere kommt

☞ **Idee: Clique $\leq_p$ Independent Set: Komplementgraph**

Man bildet den Komplementgraphen $\bar{G}$ (dieselben Knoten, genau die fehlenden Kanten) und behält die Schranke k . Eine Knotengruppe C ist genau dann eine Clique in G , wenn zwischen ihren Knoten in $\bar{G}$ *keine* Kante liegt – also ein Independent Set in $\bar{G}$. So wird aus der k -Clique-Frage in G wörtlich die k -IS-Frage in $\bar{G}$.

Was daraus folgt

INDEPENDENT SET ist im Kurs vor allem das Bindeglied der Dualität zwischen Clique, IS und Vertex Cover: Eine Menge I ist genau dann ein Independent Set, wenn ihr Komplement $V \setminus I$ ein Vertex Cover ist. Diese Beziehung wird beim Vertex Cover benutzt. Eine eigenständige Reduktion von Independent Set auf ein weiteres Kursproblem wird im Material nicht geführt.

10 Vertex Cover

VERTEX COVER ist das dritte Problem des engen Verwandtenkreises – und der Startpunkt für mehrere weitere Reduktionen.

■ Definition 10.1 Vertex Cover (VC)

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein *Vertex Cover* ist eine Knotenmenge, die *jede* Kante berührt (von jeder Kante liegt mindestens ein Endpunkt in der Menge). Frage: Gibt es ein Vertex Cover der Größe $\leq k$?

● Beispiel 10.2 Ein Vertex Cover

Im Graphen mit $V = \{1, 2, 3, 4\}$ und Kanten $\{1, 2\}, \{2, 3\}, \{3, 4\}$ ist $\{2, 3\}$ ein Vertex Cover der Größe 2: Kante $\{1, 2\}$ berührt 2, $\{2, 3\}$ berührt beide, $\{3, 4\}$ berührt 3. Der einzelne Knoten $\{2\}$ genügt nicht – $\{3, 4\}$ bliebe unberührt.

Warum Vertex Cover in NP liegt

Das Zertifikat. Die Knotenmenge $C \subseteq V$, polynomiell lang.

Die Prüfung. Der Verifizierer geht *jede* Kante $\{u, v\} \in E$ durch und prüft, ob $u \in C$ oder $v \in C$ gilt – ob die Kante also berührt wird. Zusätzlich prüft er $|C| \leq k$. Er akzeptiert, wenn jede Kante berührt ist und die Größe stimmt.

Warum das korrekt ist – beide Richtungen. Gibt es ein Vertex Cover der Größe $\leq k$, wird es akzeptiert. Gibt es keines, bleibt bei jeder vorgelegten Menge dieser Größe mindestens eine Kante unberührt, und der Verifizierer lehnt ab.

Die Laufzeit. Ein Durchlauf durch alle Kanten, also $O(|E|)$, plus $O(|V|)$ für die Größe. Damit $VC \in NP$.

Woher die Schwere kommt

🔑 Idee: Clique $\leq_p$ Vertex Cover: Komplement und $n - k$

Aus der Clique-Instanz (G, k) mit $n = |V|$ bildet man den Komplementgraphen $\bar{G}$ und fragt nach einem Vertex Cover der Größe $\leq n - k$. Der Grund für das $n - k$ steckt in der Dualität: Eine Clique der Größe k in G ist ein Independent Set der Größe k in $\bar{G}$; das Komplement eines Independent Set ist ein Vertex Cover; also wird aus einem IS der Größe k ein VC der Größe $n - k$.

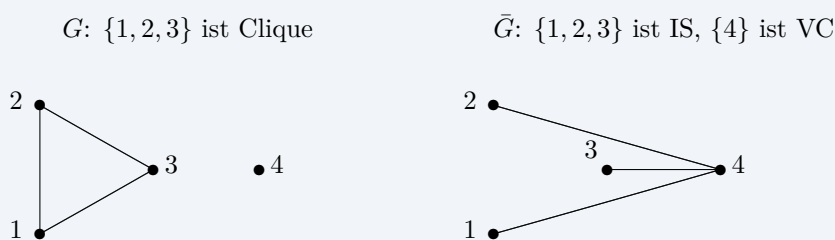
☆ Aha

Das **Dualitätsdreieck** fasst alle drei Probleme in einem Bild zusammen. Für einen Graphen G , seinen Komplementgraphen $\bar{G}$ und jede Knotenmenge C gilt:

$$C \text{ Clique in } G \iff C \text{ Independent Set in } \bar{G} \iff V \setminus C \text{ Vertex Cover in } \bar{G}.$$

● Beispiel 10.3 Ein Bild, drei Rollen

Links ein Graph G , in dem $\{1, 2, 3\}$ eine Clique ist; rechts sein Komplement $\bar{G}$, in dem dieselbe Menge ein Independent Set ist und der Knoten $\{4\}$ ein Vertex Cover.



Was daraus folgt

🔑 Idee: $VC \leq_p$ Feedback Vertex Set: antiparallele Bögen

Man macht aus jeder ungerichteten Kante $\{u, v\}$ zwei entgegengesetzte Pfeile $u \rightarrow v$ und $v \rightarrow u$ – einen gerichteten Kreis der Länge 2. Ein Knoten, der die Kante überdeckt, zerstört genau diesen Kreis. (Ausführlich beim FVS.)

☞ Idee: $VC \leq_p \Delta\text{-Cover}$: Kante wird Dreieck

Man hängt an jede Kante $\{u, v\}$ einen neuen Knoten v_e und bildet so das Dreieck $\{v_e, u, v\}$. Ein Knoten, der die Kante überdeckt, trifft auch dieses Dreieck. (Ausführlich beim $\Delta\text{-Cover}$.)

11 Feedback Vertex Set

Das erste Problem, das nicht mehr von der Clique-Familie kommt, sondern vom Vertex Cover. Es spielt auf *gerichteten* Graphen.

■ Definition 11.1 Feedback Vertex Set (FVS)

Gegeben ein *gerichteter* Graph $G = (V, E)$ und eine Zahl k . Frage: Gibt es eine Knotenmenge $X \subseteq V$ mit $|X| \leq k$, sodass $G \setminus X$ *kreisfrei* ist (keinen gerichteten Kreis mehr enthält)?

● Beispiel 11.2 Ein Feedback Vertex Set

Hat ein gerichteter Graph die Kreise $a \rightarrow b \rightarrow a$ und $b \rightarrow c \rightarrow b$, so zerstört das Entfernen von b beide Kreise auf einmal – $\{b\}$ ist ein Feedback Vertex Set der Größe 1.

Warum Feedback Vertex Set in NP liegt

Das Zertifikat. Die Knotenmenge $X \subseteq V$, ein Bit pro Knoten, also polynomiell lang.

Die Prüfung. Der Verifizierer prüft zuerst $|X| \leq k$. Dann muss er feststellen, ob $G \setminus X$ kreisfrei ist. Dafür benutzt er einen bekannten Algorithmus aus der Vorlesung: die *topologische Sortierung*. Sie findet genau dann eine Reihenfolge der Knoten, in der alle Kanten „nach vorne“ zeigen, wenn der Graph kreisfrei ist. Der Verifizierer akzeptiert, wenn die Größe stimmt und die Sortierung gelingt.

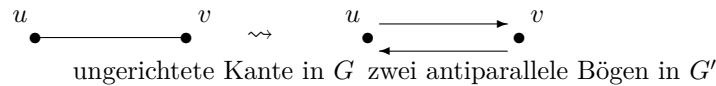
Warum das korrekt ist – beide Richtungen. Existiert ein FVS der Größe $\leq k$, so ist $G \setminus X$ kreisfrei, die topologische Sortierung gelingt, der Verifizierer akzeptiert. Existiert keines, so bleibt bei jeder Menge X dieser Größe ein Kreis übrig, die Sortierung scheitert, der Verifizierer lehnt ab.

Die Laufzeit. Die topologische Sortierung läuft in $O(|V| + |E|)$, die Größenprüfung in $O(|V|)$. Damit $FVS \in NP$.

Woher die Schwere kommt

☞ Idee: $VC \leq_p FVS$: jede Kante wird ein 2-Kreis

Aus der ungerichteten VC-Instanz (G, k) baut man einen gerichteten Graphen, indem man jede Kante $\{u, v\}$ durch *zwei antiparallele Bögen* $u \rightarrow v$ und $v \rightarrow u$ ersetzt. Die Schranke bleibt k .



Warum das stimmt: Jeder gerichtete Kreis in diesem Graphen benutzt Bögen, die aus Original-Kanten entstanden sind. Der kleinste solche Kreis ist ein 2-Kreis $u \rightarrow v \rightarrow u$, der genau aus einer Kante $\{u, v\}$ stammt. Um ihn zu zerstören, muss man u oder v entfernen – also die Kante „überdecken“. Ist C ein Vertex Cover, so berührt es jede Kante und zerstört damit jeden 2-Kreis; der Graph wird kreisfrei. Ist umgekehrt X ein FVS, so muss es jeden 2-Kreis treffen, also jede Kante überdecken – X ist ein Vertex Cover.

Was daraus folgt

Von FEEDBACK VERTEX SET geht im Kurs keine weitere Reduktion aus; es ist ein Endpunkt der Kette.

12 Δ -Cover (Dreiecksüberdeckung)

Auch dieses Problem erbt seine Schwere vom Vertex Cover, über ein anderes Gadget.

■ Definition 12.1 Δ -Cover

Gegeben ein ungerichteter Graph $G = (V, E)$ und eine Zahl k . Ein Δ -Cover ist eine Knotenmenge $C_\Delta \subseteq V$, die *jedes Dreieck* (jede 3-Clique) von G trifft – von jedem Dreieck liegt also mindestens ein Knoten in C_Δ . Frage: Gibt es ein Δ -Cover der Größe $\leq k$?

● Beispiel 12.2 Ein Δ -Cover

Besteht G aus einem einzigen Dreieck $\{a, b, c\}$, so genügt ein beliebiger seiner Knoten, etwa $\{a\}$, um das Dreieck zu treffen – ein Δ -Cover der Größe 1.

Warum Δ -Cover in NP liegt

Das Zertifikat. Die Knotenmenge $C_\Delta \subseteq V$, polynomiell lang.

Die Prüfung. Der Verifizierer prüft $|C_\Delta| \leq k$. Danach muss er sicherstellen, dass jedes Dreieck getroffen wird. Dazu geht er alle Knoten*tripel* durch – davon gibt es höchstens $\binom{|V|}{3} = O(|V|^3)$ – und für jedes Tripel prüft er, ob es ein Dreieck bildet (alle drei Kanten vorhanden) und, falls ja, ob mindestens einer seiner Knoten in C_Δ liegt. Er akzeptiert, wenn die Größe stimmt und kein Dreieck ungedeckt bleibt.

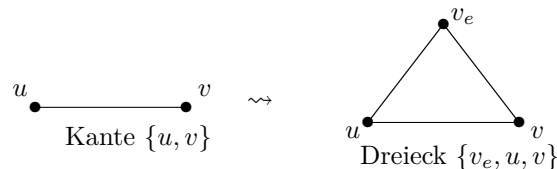
Warum das korrekt ist – beide Richtungen. Existiert ein Δ -Cover der Größe $\leq k$, wird es akzeptiert. Existiert keines, bleibt bei jeder Menge dieser Größe ein Dreieck ungetroffen, und der Verifizierer lehnt ab.

Die Laufzeit. Es gibt $O(|V|^3)$ Tripel, jede Dreiecks- und Trefferprüfung kostet $O(|V|)$; insgesamt $O(|V|^4)$, also polynomiell. Damit Δ -Cover $\in$ NP.

Woher die Schwere kommt

☞ Idee: $\text{VC} \leq_p \Delta\text{-Cover}$: jede Kante wird ein Dreieck

Aus der VC-Instanz (G, k) baut man G' : Für jede Kante $e = \{u, v\}$ fügt man einen neuen Knoten v_e hinzu und die Kanten $\{v_e, u\}$ und $\{v_e, v\}$. So wird jede Kante zum Dreieck $\{v_e, u, v\}$ aufgeblasen. Die Schranke bleibt k .



Warum das stimmt: Ist C ein Vertex Cover, so liegt von jeder Kante $\{u, v\}$ ein Endpunkt in C – damit ist auch das zugehörige Gadget-Dreieck getroffen. Umgekehrt kann man aus einem Δ -Cover ein Vertex Cover machen.

✗ Stolperstein

Ein echter Punktabzug lauert hier. In der Rückrichtung darf man *nicht* behaupten, jedes Dreieck von G' habe die Gadget-Form $\{v_e, u, v\}$. Denn G' enthält alle Kanten von G weiter – also *erbt* G' auch jedes Dreieck, das G selbst schon hatte (drei paarweise verbundene Original-Knoten). Diese geerbten Dreiecke muss der Beweis mitbehandeln: Auch sie werden von einem Vertex Cover getroffen, weil ihre Kanten überdeckt sind. Wer diesen Fall vergisst, verliert Punkte.

Was daraus folgt

Von $\Delta\text{-COVER}$ geht im Kurs keine weitere Reduktion aus.

13 k -Color (Färbung)

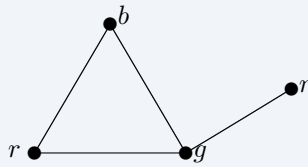
Ein Problem, das nicht von der Clique-Familie, sondern direkt von 3-SAT kommt.

■ Definition 13.1 k -Color

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Eine k -Färbung gibt jedem Knoten eine von k Farben, sodass keine Kante zwei gleichfarbige Enden verbindet. Frage: Ist G mit k Farben färbbar?

Man denkt an eine Landkarte, deren Länder so gefärbt werden, dass keine zwei Nachbarn dieselbe Farbe tragen. Schon $k = 3$ ist NP-vollständig.

● Beispiel 13.2 Eine gültige 3-Färbung



Drei Farben r, g, b . Das Dreieck braucht alle drei; der rechte Knoten darf wieder r sein, weil er nur mit g benachbart ist.

Warum k -Color in NP liegt

Das Zertifikat. Die Färbung selbst – eine Farbe pro Knoten, also eine Zahl aus $\{1, \dots, k\}$ je Knoten. Das ist polynomiell lang.

Die Prüfung. Der Verifizierer geht jede Kante $\{u, v\} \in E$ durch und prüft, ob ihre beiden Enden verschiedene Farben haben. Er akzeptiert, wenn keine Kante gleichfarbig ist.

Warum das korrekt ist – beide Richtungen. Ist G färbbar, wird eine gültige Färbung akzeptiert. Ist G nicht färbbar, hat jede vorgelegte Färbung eine gleichfarbige Kante, und der Verifizierer lehnt ab.

Die Laufzeit. Ein Durchlauf durch alle Kanten, also $O(|E|)$. Damit k -COLOR $\in$ NP.

Woher die Schwere kommt

☞ Idee: 3-SAT $\leq_p$ k -Color: Farben als Wahrheitswerte

Man baut pro Variable die Knoten $x_i, \bar{x}_i, v_i$, einen Knoten F_j pro Klausel und ein Zentrum z . Die Knoten $v_1, \dots, v_n, z$ bilden eine Clique und erzwingen genau $n + 1$ Farben; man kann z die Farbe „neutral“ geben und jedem v_i eine eigene. Für jedes Paar $x_i, \bar{x}_i$ (verbunden) bleibt dann nur die Wahl zwischen „wahr“ und „falsch“ – das kodiert die Belegung. Jeder Klauselknoten F_j ist mit genau den Literalen verbunden, die *nicht* in seiner Klausel stehen; er lässt sich genau dann korrekt färben, wenn ein Literal seiner Klausel wahr ist. Der Graph ist also mit $n + 1$ Farben färbbar genau dann, wenn die Formel erfüllbar ist.

Was daraus folgt

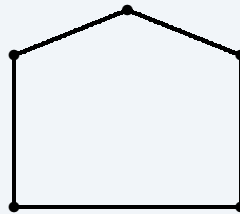
Von k -COLOR geht im Kurs keine weitere Härte-Reduktion aus; es taucht später nur bei den ETH-Schranken wieder auf.

14 Hamiltonkreis

■ Definition 14.1 Hamiltonkreis (HK)

Gegeben ein Graph $G = (V, E)$. Frage: Gibt es einen Rundweg, der *jeden* Knoten *genau einmal* besucht und am Start wieder ankommt? (Die Aussage gilt für gerichtete wie ungerichtete Graphen.)

● Beispiel 14.2 Ein Hamiltonkreis



Der fette Weg besucht alle fünf Knoten genau einmal und schließt sich.

Warum Hamiltonkreis in NP liegt

Das Zertifikat. Die Besuchsreihenfolge – eine Permutation der Knoten, die den behaupteten Rundweg angibt. Sie ist genau $|V|$ Einträge lang, also polynomiell.

Die Prüfung. Der Verifizierer prüft zweierlei: erstens, dass jeder Knoten in der Reihenfolge genau einmal vorkommt; zweitens, dass zwischen je zwei aufeinanderfolgenden Knoten – und zwischen dem letzten und dem ersten – wirklich eine Kante existiert. Er akzeptiert, wenn beides zutrifft.

Warum das korrekt ist – beide Richtungen. Gibt es einen Hamiltonkreis, so ist seine Reihenfolge ein akzeptiertes Zertifikat. Wird umgekehrt eine Reihenfolge akzeptiert, so besucht sie jeden Knoten einmal und benutzt nur echte Kanten – sie ist also ein Hamiltonkreis.

Die Laufzeit. Die Verschiedenheit prüft man in $O(|V|)$ (etwa durch Abhaken), die Kantenbedingung in $O(|V|)$ Nachschlägen. Insgesamt $O(|V| + |E|)$. Damit $\text{HK} \in \text{NP}$.

Woher die Schwere kommt

Idee: $3\text{-SAT}' \leq_p \text{HK}$: A- und B-Komponenten

Man baut für die Variante mit genau drei Literalen pro Klausel zwei Sorten von Gadgets. Die „A-Komponenten“ (Leitergraphen) haben genau zwei mögliche Durchlaufrichtungen und kodieren so die Setzung einer Variablen auf wahr oder falsch. Die „B-Komponenten“ – eine pro Klausel – sind so gebaut, dass sie sich für *jede nichtleere* Auswahl ihrer Kanten durchlaufen lassen, aber *nicht* für die leere. Die leere Auswahl entspricht „alle Literale dieser Klausel falsch“. Ein Hamiltonkreis existiert also genau dann, wenn jede Klausel mindestens ein wahres Literal hat – wenn die Formel erfüllbar ist.

Was daraus folgt

Idee: $\text{HK} \leq_p \text{TSP}$: Kanten als Distanzen

Man macht aus dem Graphen eine TSP-Instanz: Jede echte Kante bekommt Distanz 1, jede fehlende Kante die große Distanz $|V| + 1$, und die Schranke ist $L = |V|$. Eine Rundreise der Länge $\leq |V|$ kann nur echte Kanten benutzen – sie ist also ein Hamiltonkreis. (Ausführlich beim TSP.)

Idee: $\text{HamiltonianCycle} \leq_p \text{HamiltonianPath}$: Knoten aufspalten

Um aus der Kreis- eine Pfad-Frage zu machen, verdoppelt man einen Knoten v zu v und v^* (mit gleicher Nachbarschaft) und hängt zwei neue Grad-1-Knoten s (an v) und t (an v^*) als Pfadenden an. Ein Hamiltonpfad von s nach t entspricht genau einem Hamiltonkreis durch v . (Die Gegenrichtung $\text{PATH} \leq_p \text{CYCLE}$ schließt den Pfad über einen universellen Knoten zum Kreis.)

15 TSP (Traveling Salesman)

■ Definition 15.1 TSP, Entscheidungsvariante

Gegeben Städte mit paarweisen Distanzen $d(i, j)$ und eine Schranke L . Frage: Gibt es eine Rundreise durch alle Städte (jede genau einmal) mit Gesamtlänge $\leq L$?

Der Unterschied zum Hamiltonkreis: Dort fragt man nur, *ob* es eine Rundreise gibt; hier gibt es Distanzen und eine Längengrenze. Als Optimierungsproblem – „finde die kürzeste Rundreise“ – ist TSP der Star des Approximationskapitels.

● Beispiel 15.2 Eine kurze Rundreise

Vier Städte auf einem Quadrat, jede Seite Distanz 1, die Diagonalen Distanz 2. Die Rundreise entlang der vier Seiten hat Länge 4 und ist optimal; sie unterschreitet jede Schranke $L \geq 4$.

Warum TSP in NP liegt

Das Zertifikat. Die Reihenfolge der Städte – eine Permutation, die die Rundreise angibt. Polynomiell lang.

Die Prüfung. Der Verifizierer addiert die Distanzen entlang der Rundreise (inklusive der Rückkehr zum Start) und prüft, ob die Summe $\leq L$ ist; außerdem prüft er, dass jede Stadt genau einmal vorkommt. Er akzeptiert bei beidem.

Warum das korrekt ist – beide Richtungen. Gibt es eine Rundreise der Länge $\leq L$, wird ihre Reihenfolge akzeptiert. Gibt es keine, überschreitet jede vorgelegte Reihenfolge die Schranke, und der Verifizierer lehnt ab.

Die Laufzeit. Das Aufsummieren der n Distanzen kostet $O(n)$. Damit liegt die Entscheidungsvariante von TSP in NP.

Woher die Schwere kommt

🔍 Idee: $HK \leq_p$ TSP: Kanten Distanz 1, Nicht-Kanten teuer

Aus einem Graphen G für den Hamiltonkreis baut man die Städte = Knoten und setzt $d(u, v) = 1$, wenn $\{u, v\}$ eine Kante von G ist, sonst $d(u, v) = |V| + 1$. Die Schranke ist $L = |V|$. Eine Rundreise der Länge $\leq |V|$ besucht $|V|$ Städte und darf keine teure Nicht-Kante benutzen – sie besteht also nur aus echten Kanten und ist ein Hamiltonkreis. Umgekehrt hat jeder Hamiltonkreis genau die Länge $|V|$. TSP bleibt sogar dann NP-vollständig, wenn man symmetrische Distanzen mit Dreiecksungleichung verlangt.

Was daraus folgt

Von TSP geht keine Härte-Reduktion aus. Stattdessen ist es das zentrale Optimierungsproblem für die Approximation (Teil D): ΔTSP_1 und Christofides.

16 3-dimensionales Matching

Ab hier verlassen wir die Graphen und arbeiten mit Mengen und Zahlen. Den Anfang macht ein Problem, das direkt von SAT kommt.

■ Definition 16.1 3-dimensionales Matching (3-DM)

Gegeben drei gleich große Mengen U, V, W und eine Liste T von *Tripeln*, wobei jedes Tripel je ein Element aus U , aus V und aus W enthält. Frage: Gibt es eine Auswahl von Tripeln, die jedes Element aus U, V und W *genau einmal* trifft?

Man denkt an eine perfekte Zuordnung zu dritt: Jede Person aus U soll mit genau einer Aufgabe aus V und genau einem Termin aus W zusammengebracht werden, und die erlaubten Kombinationen stehen in T .

● Beispiel 16.2 Ein 3-dimensionales Matching

$U = \{a_1, a_2\}$, $V = \{b_1, b_2\}$, $W = \{c_1, c_2\}$ und die Tripel (a_1, b_1, c_1) , (a_2, b_2, c_2) , (a_1, b_2, c_2) . Die Auswahl der ersten beiden Tripel trifft jedes der sechs Elemente genau einmal – ein gültiges Matching.

Warum 3-DM in NP liegt

Das Zertifikat. Die ausgewählte Menge $M \subseteq T$ von Tripeln. Höchstens $|U|$ Tripel, also polynomiell lang.

Die Prüfung. Der Verifizierer prüft, dass M genau $|U|$ Tripel enthält und dass in M keine zwei Tripel in irgendeiner ihrer drei Komponenten übereinstimmen – dass also jedes Element aus U, V, W höchstens (und damit genau) einmal vorkommt. Dazu zählt er für jedes Element seine Vorkommen.

Warum das korrekt ist – beide Richtungen. Existiert ein Matching, wird es akzeptiert. Existiert keines, kommt bei jeder Auswahl ein Element doppelt oder gar nicht vor, und der Verifizierer lehnt ab.

Die Laufzeit. Das Zählen der Vorkommen über alle Tripel läuft in $O(|T|)$. Damit $3\text{-DM} \in \text{NP}$.

Woher die Schwere kommt

🔑 Idee: $\text{SAT} \leq_p 3\text{-DM}$: Zahnrad, Klauseln, Garbage Collection

Pro Variable baut man eine „Zahnrad“-Struktur aus Tripeln, die kreisförmig angeordnet sind. Ein perfektes Matching kann dieses Zahnrad nur auf zwei Arten abdecken – die eine steht für „Variable wahr“, die andere für „falsch“. Pro Klausel gibt es ein Knotenpaar v_j, w_j , dessen Tripel nur dann matchbar sind, wenn ein erfüllendes Literal der Klausel „frei“ geblieben ist. Weil dabei nicht alle Literale verbraucht werden, räumt eine „Garbage Collection“ aus zusätzlichen Tripeln die übrig gebliebenen Literale weg, damit am Ende doch jedes Element genau einmal getroffen ist.

Was daraus folgt

🔑 Idee: $3\text{-DM} \leq_p 3\text{-Exact Cover}$: Tripel als 3er-Mengen

Man muss gar nichts umbauen: Fasst man jedes Tripel (u, v, w) als die dreielementige Menge $\{u, v, w\}$ über dem Universum $U \cup V \cup W$ auf, so wird aus „jedes Element genau einmal treffen“ wörtlich „das Universum exakt überdecken“. Jede 3-DM-Instanz ist also direkt eine 3-EC-Instanz.

17 3-Exact Cover

■ Definition 17.1 3-Exact Cover (3-EC)

Gegeben ein Universum U mit $|U| = 3m$ und eine Familie $S_1, \dots, S_n$ von *dreielementigen* Teilmengen von U . Frage: Gibt es eine Auswahl von Mengen, die U *exakt* überdeckt – disjunkt und lückenlos, sodass jedes Element in genau einer gewählten Menge liegt?

● Beispiel 17.2 Eine exakte Überdeckung

$U = \{1, 2, 3, 4, 5, 6\}$ und die Mengen $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{1, 2, 4\}$. Die Auswahl $\{S_1, S_2\}$ überdeckt jedes Element genau einmal. Die Auswahl $\{S_1, S_3\}$ dagegen nicht: 1 und 2 lägen doppelt, 5 und 6 gar nicht.

Warum 3-EC in NP liegt

Das Zertifikat. Die ausgewählte Teilfamilie von Mengen. Da eine exakte Überdeckung genau m Mengen braucht (jede deckt drei der $3m$ Elemente), ist sie polynomiell lang.

Die Prüfung. Der Verifizierer markiert für jede gewählte Menge ihre drei Elemente und kontrolliert, dass am Ende jedes Element aus U *genau einmal* markiert wurde – keines doppelt, keines gar nicht.

Warum das korrekt ist – beide Richtungen. Existiert eine exakte Überdeckung, wird sie akzeptiert. Existiert keine, bleibt bei jeder Auswahl ein Element unbedeckt oder wird doppelt bedeckt, und der Verifizierer lehnt ab.

Die Laufzeit. Das Markieren und Kontrollieren läuft in $O(|U| + \text{Anzahl gewählter Mengen})$, also polynomiell. Damit $3\text{-EC} \in \text{NP}$.

Woher die Schwere kommt

☞ Idee: $3\text{-DM} \leq_p 3\text{-EC}$: dieselbe Instanz, anders gelesen

Man muss an der Instanz gar nichts umbauen – man liest sie nur anders. Ein Tripel (u, v, w) aus der 3-DM-Instanz fasst man als die dreielementige Menge $\{u, v, w\}$ auf, und als Universum nimmt man $U \cup V \cup W$.

Warum das genau passt: Die 3-DM-Forderung „jedes Element aus U , V und W wird von den gewählten Tripeln genau einmal getroffen“ ist wörtlich die 3-EC-Forderung „jedes Element des Universums liegt in genau einer gewählten Menge“. Eine Auswahl von Tripeln ist also genau dann ein 3-dimensionales Matching, wenn dieselbe Auswahl, als Mengenfamilie gelesen, eine exakte Überdeckung ist. Die Umformung kostet keine Rechenzeit über das Umschreiben hinaus, ist also polynomiell. Da 3-DM NP-vollständig und $3\text{-EC} \in \text{NP}$ ist, ist auch 3-EC NP-vollständig.

Was daraus folgt

☞ Idee: $3\text{-EC} \leq_p \text{SubSet Sum}$: Mengen werden Ziffern

Man schreibt jede Menge als Bitvektor über dem Universum und liest ihn als Zahl zur Basis $n + 1$. Das Ziel ist die Zahl „überall 1“. Weil bei so großer Basis nie ein Übertrag entsteht, trifft eine Auswahl die Zielsumme genau dann, wenn sie jede Stelle genau einmal überdeckt – eine exakte Überdeckung. (Ausführlich mit Zahlen beim SubSet Sum.)

18 SubSet Sum

■ Definition 18.1 SubSet Sum

Gegeben ganze Zahlen $c_1, \dots, c_n$ und ein Zielwert K . Frage: Gibt es eine Teilmenge dieser Zahlen mit Summe *genau* K ?

● Beispiel 18.2 SubSet Sum ausprobiert

Zahlen 3, 7, 8, 4 und Ziel $K = 15$. Es gilt $3 + 4 + 8 = 15$ – die Teilmenge $\{3, 4, 8\}$ trifft das Ziel genau. Die Teilmenge $\{7, 4\}$ ergäbe nur 11.

Warum SubSet Sum in NP liegt

Das Zertifikat. Die Teilmenge $S \subseteq \{1, \dots, n\}$, ein Bit pro Zahl, also polynomiell lang.

Die Prüfung. Der Verifizierer addiert die Zahlen c_j für $j \in S$ und vergleicht die Summe mit K . Er akzeptiert bei Gleichheit.

Warum das korrekt ist – beide Richtungen. Existiert eine Teilmenge mit Summe K , wird sie akzeptiert. Existiert keine, trifft keine Teilmenge den Zielwert, und der Verifizierer lehnt ab.

Die Laufzeit. Es sind höchstens n Additionen, also $O(n)$ Arithmetikschritte. Damit $\text{SUBSET SUM} \in \text{NP}$. (Beachte: „ $O(n)$ Additionen“ zählt Rechenschritte auf großen Zahlen; die Schwere des Problems steckt nicht in der Prüfung, sondern im *Finden* der richtigen Teilmenge unter 2^n Möglichkeiten.)

Woher die Schwere kommt

☞ **Idee: $3\text{-EC} \leq_p \text{SubSet Sum}$: Ziffern zur Basis $n + 1$**

Man schreibt jede Menge S_j als Bitvektor über dem Universum (eine 1 an jeder Stelle, die S_j enthält) und liest diesen Bitvektor als Zahl zur *Basis* $n + 1$, wobei n die Anzahl der Mengen ist. Das Ziel K ist die Zahl zum Bitvektor „überall 1“.

Warum die krumme Basis $n + 1$? Damit beim Addieren *kein Übertrag* entsteht. Es gibt nur n Mengen, an jeder Stelle also höchstens n Einsen – das bleibt unter $n + 1$, keine Stelle läuft über. Ohne Übertrag zählt jede Stelle für sich, und „Summe trifft das Ziel“ bedeutet genau „jede Stelle wird genau einmal überdeckt“.

● Beispiel 18.3 Die Ziffern-Reduktion mit Zahlen

$U = \{1, \dots, 6\}$, Mengen $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{1, 2, 4\}$, $S_4 = \{3, 5, 6\}$; also $n = 4$, Basis 5.

Menge	Bitvektor	Zahl zur Basis 5
$S_1 = \{1, 2, 3\}$	111000	$5^0 + 5^1 + 5^2 = 31$
$S_2 = \{4, 5, 6\}$	000111	$5^3 + 5^4 + 5^5 = 3875$
$S_3 = \{1, 2, 4\}$	110100	$5^0 + 5^1 + 5^3 = 131$
$S_4 = \{3, 5, 6\}$	001011	$5^2 + 5^4 + 5^5 = 3775$
Ziel K	111111	$5^0 + \dots + 5^5 = 3906$

Es gilt $S_1 + S_2 = 31 + 3875 = 3906 = K$, und tatsächlich ist $\{S_1, S_2\}$ eine exakte Überdeckung. Ebenso $S_3 + S_4 = 3906$ mit exakter Überdeckung $\{S_3, S_4\}$.

Es gibt noch eine *zweite*, direktere Reduktion auf SUBSET SUM – diesmal von 3-SAT aus. Sie ist für die Praxis unwichtiger, aber für die unteren Schranken in Teil C zentral, weil sie besonders *wenige* Zahlen erzeugt.

Idee: 3-SAT $\leq_p$ SubSet Sum: Ziffern zur Basis 10

Man kodiert jede Zahl als Dezimalzahl mit $n + m$ Ziffern – eine Stelle pro Variable und eine pro Klausel. Der Aufbau hat drei Bausteine:

- *Zwei Items pro Variable.* Für jede Variable x_i gibt es zwei Zahlen: eine steht für „ x_i wahr“, die andere für „ x_i falsch“. Beide haben an der Variablen-Stelle i eine 1 (genau eine der beiden wird gewählt) und an den Klausel-Stellen eine 1, wo das jeweilige Literal vorkommt.
- *Der Zielwert* verlangt an jeder Variablen-Stelle genau eine 1 (also eine gültige Belegung) und an jeder Klausel-Stelle den Wert 3.
- *Dummy-Items pro Klausel.* Da eine erfüllte Klausel ein bis drei wahre Literale haben kann, an der Klausel-Stelle aber genau die 3 stehen muss, füllt man mit zwei Füll-Zahlen („Dummies“) pro Klausel den Rest auf 3 auf.

Der Trick der großen Ziffernabstände sorgt wieder dafür, dass es keine Überträge gibt, sodass jede Stelle für sich zählt. Entscheidend für Teil C: Diese Reduktion erzeugt nur $|A| = 2n + 2m = O(m)$ Zahlen. Deshalb liefert sie sogar eine $2^{o(n)}$ -untere Schranke für SUBSET SUM und PARTITION – viel schärfer als die Kette über 3-EC.

Was daraus folgt

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

Man ergänzt die Zahlen um zwei geschickt gewählte Werte, sodass die „halbe Gesamtsumme“-Frage von PARTITION genau die Zielsummen-Frage von SubSet Sum wird. (Ausführlich bei Partition.)

Idee: SubSet Sum $\leq_p$ Knapsack: Spezialfall

Setzt man beim Rucksack für jeden Gegenstand Gewicht gleich Profit ($w_j = p_j = c_j$) und Kapazität gleich Zielprofit ($B = P = K$), so fragt der Rucksack genau, ob eine Teilmenge die Summe K trifft. SubSet Sum ist also ein Spezialfall von Knapsack.

Idee: SubSet Sum $\leq_p$ SubSet Sum Cardinality: Padding und Shift

Die Variante verlangt zusätzlich, dass die Teilmenge genau n Zahlen enthält. Man erhöht jede Originalzahl um 1 („Shift“) und fügt n Einsen als „Padding“ hinzu; das Ziel wird $K + n$. Die Einsen füllen die geforderte Anzahl auf, ohne die eigentliche Summe zu stören.

19 Partition

■ Definition 19.1 Partition

Gegeben ganze Zahlen $c_1, \dots, c_n$. Frage: Gibt es eine Teilmenge, deren Summe *genau die Hälfte* der Gesamtsumme beträgt – die die Zahlen also in zwei gleich schwere Hälften teilt?

● Beispiel 19.2 Eine Partition

Zahlen 1, 5, 6, 2 mit Gesamtsumme 14, Hälfte also 7. Die Teilmenge $\{1, 6\}$ hat Summe 7, der Rest $\{5, 2\}$ ebenfalls – eine gültige Zweiteilung.

Warum Partition in NP liegt

Das Zertifikat. Die Teilmenge S , ein Bit pro Zahl, polynomiell lang.

Die Prüfung. Der Verifizierer berechnet die Gesamtsumme $N = \sum_j c_j$ und die Teilsumme $\sum_{j \in S} c_j$ und prüft, ob die Teilsumme gleich $N/2$ ist. Er akzeptiert bei Gleichheit.

Warum das korrekt ist – beide Richtungen. Existiert eine hälftige Teilmenge, wird sie akzeptiert. Existiert keine, trifft keine Teilmenge genau die Hälfte, und der Verifizierer lehnt ab.

Die Laufzeit. Zwei Summen über höchstens n Zahlen, also $O(n)$. Damit PARTITION $\in$ NP.

Woher die Schwere kommt

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

Aus den Zahlen $c_1, \dots, c_n$ mit Ziel K und $N = (\sum_j c_j) + 1$ bildet man eine neue Zahlenmenge: alle c_j *plus* zwei Zusatzzahlen $c_{n+1} = N - K$ und $c_{n+2} = K + 1$.

Warum das stimmt: Die Gesamtsumme der neuen Menge ist $\sum c_j + (N - K) + (K + 1) = 2N$, ihre Hälfte also N . Die beiden Zusatzzahlen summieren sich zu $(N - K) + (K + 1) = N + 1$ – das ist schon *mehr* als eine Hälfte, also können sie nie beide in derselben Hälfte liegen; sie werden immer getrennt. Trifft nun eine Teilmenge der Originalzahlen den Wert K , so ergänzt man sie um $c_{n+1} = N - K$ und erhält die Summe N – eine Halbierung. Umgekehrt enthält jede Halbierung genau eine Zusatzzahl; zieht man sie ab, bleibt eine Original-Teilmenge mit Summe K .

● Beispiel 19.3 Die Zusatzzahlen mit Zahlen

Zahlen 3, 5, 8, 4 (Summe 20), Ziel $K = 12$. Dann $N = 21$, die Zusatzzahlen sind $N - K = 9$ und $K + 1 = 13$. Neue Menge $\{3, 5, 8, 4, 9, 13\}$, Gesamtsumme 42, Hälfte 21. Die Original-Teilmenge $\{8, 4\}$ trifft $K = 12$; ergänzt um 9 ergibt $\{8, 4, 9\}$ die Summe 21 – eine gültige Halbierung.

Was daraus folgt

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

Man nimmt die Partition-Zahlen als Job-Laufzeiten und stellt zwei Maschinen bereit. Beide Maschinen exakt gleich auszulasten heißt, die Zahlen in zwei gleich schwere Hälften zu teilen. (Ausführlich beim Scheduling.)

20 Knapsack (Rucksackproblem)

■ Definition 20.1 Knapsack (Rucksack)

Gegeben n Gegenstände, jeder mit Gewicht w_i und Profit p_i , eine Kapazität B und ein Zielprofit P . Frage: Gibt es eine Auswahl von Gegenständen mit Gesamtgewicht $\leq B$ und Gesamtprofit $\geq P$?

● Beispiel 20.2 Eine Rucksack-Füllung

Drei Gegenstände $(w, p) = (3, 4), (4, 5), (5, 6)$, Kapazität $B = 7$, Zielprofit $P = 9$. Die ersten beiden zusammen haben Gewicht $3 + 4 = 7 \leq B$ und Profit $4 + 5 = 9 \geq P$ – sie erfüllen beide Schranken.

Warum Knapsack in NP liegt

Das Zertifikat. Die Auswahl $S \subseteq \{1, \dots, n\}$, ein Bit pro Gegenstand.

Die Prüfung. Der Verifizierer berechnet $\sum_{i \in S} w_i$ und $\sum_{i \in S} p_i$ und prüft die beiden Schranken $\sum_{i \in S} w_i \leq B$ und $\sum_{i \in S} p_i \geq P$. Er akzeptiert, wenn beide erfüllt sind.

Warum das korrekt ist – beide Richtungen. Existiert eine zulässige Auswahl mit genügend Profit, wird sie akzeptiert. Existiert keine, verletzt jede Auswahl eine der beiden Schranken, und der Verifizierer lehnt ab.

Die Laufzeit. Zwei Summen und zwei Vergleiche über n Gegenstände, also $O(n)$. Damit $\text{KNAPSACK} \in \text{NP}$.

Woher die Schwere kommt

☞ Idee: SubSet Sum $\leq_p$ Knapsack: Gewicht gleich Profit

Aus einer SubSet-Sum-Instanz $(c_1, \dots, c_n, K)$ macht man eine Knapsack-Instanz, indem man für jeden Gegenstand $w_i = p_i = c_i$ setzt und $B = P = K$ wählt. Dann verlangt der Rucksack eine Auswahl mit Gewicht $\leq K$ und Profit $\geq K$ – bei $w_i = p_i$ heißt das Summe genau K . SubSet Sum ist also der Spezialfall $w_i = p_i$ des Rucksacks.

Was daraus folgt

Von KNAPSACK geht keine Härte-Reduktion aus. Als Optimierungsproblem ist es in Teil D wichtig (Greedy, Modified Greedy, Sahni, FPTAS).

21 $P||C_{\max}$ (Scheduling)

■ Definition 21.1 $P||C_{\max}$ (Scheduling auf identischen Maschinen)

Gegeben n Jobs mit Laufzeiten $p_1, \dots, p_n$ und m gleiche Maschinen. Verteile die Jobs so auf die Maschinen, dass die *höchste* Maschinenlast – der *Makespan* $C_{\max}$ – möglichst klein wird. (Entscheidungsvariante: Gibt es eine Verteilung mit Makespan $\leq T$?)

● Beispiel 21.2 Ein Schedule

Drei Jobs der Größen 2, 3, 3 auf zwei Maschinen. Legt man den 2er und einen 3er auf M_1 (Last 5) und den anderen 3er auf M_2 (Last 3), so ist der Makespan 5. Besser: die beiden 3er auf getrennte Maschinen und den 2er dazu – Makespan 5 bleibt, denn eine Maschine trägt mindestens $8/2 = 4$ und der 2er passt nicht ohne einen 3er. Optimal ist hier Makespan 5.

Warum $P||C_{\max}$ in NP liegt

Das Zertifikat. Die Zuordnung „welcher Job läuft auf welcher Maschine“ – also eine Aufteilung der Jobs in m Gruppen.

Die Prüfung. Der Verifizierer summiert für jede Maschine die Laufzeiten der ihr zugeordneten Jobs, nimmt das Maximum über alle Maschinen (den Makespan) und prüft, ob es $\leq T$ ist. Er akzeptiert bei Erfolg.

Warum das korrekt ist – beide Richtungen. Existiert eine Verteilung mit Makespan $\leq T$, wird sie akzeptiert. Existiert keine, überschreitet jede Verteilung die Schranke, und der Verifizierer lehnt ab.

Die Laufzeit. Das Aufsummieren geht jeden Job einmal an, also $O(n)$. Damit liegt die Entscheidungsvariante in NP.

Woher die Schwere kommt

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

Man nimmt die Partition-Zahlen $c_1, \dots, c_n$ als Job-Laufzeiten und zwei Maschinen. Beide Maschinen tragen zusammen die Gesamtlast $N = \sum c_j$. Ein Makespan von genau $N/2$ ist nur möglich, wenn beide Maschinen exakt gleich ausgelastet sind – also wenn sich die Zahlen in zwei gleich schwere Hälften teilen lassen. Damit ist $P||C_{\max}$ schon für $m = 2$ Maschinen NP-vollständig.

Was daraus folgt

Von $P||C_{\max}$ geht keine Härte-Reduktion aus. Als Optimierungsproblem ist es in Teil D wichtig (List Scheduling, LPT).

22 Hitting Set

■ Definition 22.1 Hitting Set

Gegeben ein Universum U , eine Familie von Mengen $F_1, \dots, F_r \subseteq U$ und eine Zahl k . Ein *Hitting Set* ist eine Menge $H \subseteq U$, die *jede* der Mengen trifft ($H \cap F_i \neq \emptyset$ für alle i). Frage: Gibt es ein Hitting Set der Größe $\leq k$?

● Beispiel 22.2 Ein Hitting Set

$U = \{1, 2, 3, 4\}$, Mengen $F_1 = \{1, 2\}$, $F_2 = \{2, 3\}$, $F_3 = \{3, 4\}$. Das Element $\{2\}$ trifft F_1 und F_2 , aber nicht F_3 . Die Menge $\{2, 3\}$ trifft alle drei – ein Hitting Set der Größe 2.

Warum Hitting Set in NP liegt

Das Zertifikat. Die Menge $H \subseteq U$, ein Bit pro Element, also polynomiell lang.

Die Prüfung. Der Verifizierer prüft $|H| \leq k$ und geht dann jede der Mengen F_i durch, um sicherzustellen, dass H mindestens ein Element von F_i enthält. Er akzeptiert, wenn die Größe stimmt und jede Menge getroffen ist.

Warum das korrekt ist – beide Richtungen. Existiert ein Hitting Set der Größe $\leq k$, wird es akzeptiert. Existiert keines, bleibt bei jeder Menge dieser Größe ein F_i ungetroffen, und der Verifizierer lehnt ab.

Die Laufzeit. Für jede der r Mengen ein Test gegen H , insgesamt $O(r \cdot |U|)$, also polynomiell. Damit HITTING SET $\in$ NP.

Woher die Schwere kommt

🔑 Idee: 3-SAT $\leq_p$ Hitting Set

Das Universum sind alle Literale $x_1, \bar{x}_1, \dots, x_n, \bar{x}_n$. Für jede Variable i bildet man die Menge $F_i = \{x_i, \bar{x}_i\}$ – sie zwingt das Hitting Set, aus jedem solchen Paar (mindestens) ein Literal zu wählen, was einer Belegung entspricht. Für jede Klausel j bildet man die Menge F_{n+j} ihrer Literale – sie verlangt, dass ein erfüllendes Literal getroffen wird. Mit Schranke $k = n$ existiert ein Hitting Set genau dann, wenn die Formel erfüllbar ist.

Was daraus folgt

Idee: Hitting Set $\leq_p$ Set Cover: Rollen tauschen

Hitting Set und Set Cover sind dual zueinander: Vertauscht man die Rollen von „Elementen“ und „Mengen“, wird aus dem einen das andere. Aus einem Hitting Set (wähle Elemente, die alle Mengen treffen) wird ein Set Cover (wähle Mengen, die alle Elemente überdecken). (Ausführlich beim Set Cover.)

23 Set Cover

■ Definition 23.1 Set Cover

Gegeben ein Universum U , eine Familie von Mengen $S_1, \dots, S_r \subseteq U$ und eine Zahl k . Frage: Gibt es eine Auswahl von höchstens k Mengen, deren Vereinigung ganz U ist (die also jedes Element überdeckt)?

● Beispiel 23.2 Ein Set Cover

$U = \{1, 2, 3, 4\}$, Mengen $S_1 = \{1, 2\}$, $S_2 = \{2, 3\}$, $S_3 = \{3, 4\}$. Die Auswahl $\{S_1, S_3\}$ überdeckt 1, 2 und 3, 4 – also ganz U , ein Set Cover der Größe 2.

Warum Set Cover in NP liegt

Das Zertifikat. Die Auswahl der Mengen (welche der S_i genommen werden).

Die Prüfung. Der Verifizierer prüft, dass höchstens k Mengen gewählt sind, und markiert alle Elemente der gewählten Mengen; er kontrolliert, dass am Ende jedes Element aus U markiert ist.

Warum das korrekt ist – beide Richtungen. Existiert ein Set Cover der Größe $\leq k$, wird es akzeptiert. Existiert keines, bleibt bei jeder Auswahl dieser Größe ein Element unüberdeckt, und der Verifizierer lehnt ab.

Die Laufzeit. Das Markieren läuft über alle Elemente der gewählten Mengen, also $O(r \cdot |U|)$, polynomiell. Damit SET COVER $\in$ NP.

Woher die Schwere kommt

Idee: Hitting Set $\leq_p$ Set Cover: die duale Sicht

Man baut den bipartiten Inzidenzgraphen zwischen Elementen und Mengen und vertauscht die beiden Seiten: Aus einem Element wird eine Menge, aus einer Menge ein Element. Konkret wird aus „Element u liegt in Menge F_i “ die Beziehung „neue Menge u enthält neues Element i “. Ein Hitting Set der Größe k auf der einen Seite entspricht dann genau einem Set Cover der Größe k auf der anderen. Die Reduktion ist *involutorisch*: zweimal angewandt landet man wieder bei der Ausgangsinstanz.

Was daraus folgt

Idee: Set Cover $\leq_p$ Hitting Set: die Rollen zurücktauschen

Man macht denselben Tausch noch einmal, jetzt in die andere Richtung. Aus einer Set-Cover-Instanz mit Universum U und Mengen $S_1, \dots, S_r$ baut man eine Hitting-Set-Instanz, deren Universum die *Mengen-Indizes* $\{1, \dots, r\}$ sind. Für jedes Element $u \in U$ bildet man dort die Menge aller Indizes i , für die $u \in S_i$ gilt.

Warum das passt: Eine Auswahl von Set-Cover-Mengen überdeckt genau dann jedes Element u , wenn sie für jedes u mindestens eine der Mengen enthält, die u abdecken – also mindestens einen Index der neuen Menge zu u trifft. Das ist wörtlich die Hitting-Set-Bedingung, und die Schranke k bleibt gleich. Weil diese Reduktion *involutorisch* ist – zweimal angewandt landet man bei der Ausgangsinstanz –, sind Set Cover und Hitting Set exakt gleich schwer. Im Kurs dient das vor allem den ETH-Schranken für beide Probleme.

24 Dominating Set

■ Definition 24.1 Dominating Set

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein *Dominating Set* ist eine Knotenmenge $D \subseteq V$, sodass jeder Knoten entweder selbst in D liegt oder einen Nachbarn in D hat. Frage: Gibt es ein Dominating Set der Größe $\leq k$?

● Beispiel 24.2 Ein Dominating Set

In einem Stern mit einem Mittelknoten und vier Blättern genügt der Mittelknoten: $\{m\}$ ist ein Dominating Set, denn jedes Blatt hat m als Nachbarn.

Warum Dominating Set in NP liegt

Das Zertifikat. Die Knotenmenge $D \subseteq V$, ein Bit pro Knoten, Länge $O(|V|)$.

Die Prüfung. Der Verifizierer prüft zuerst $|D| \leq k$. Dann geht er jeden Knoten $v \in V$ durch und prüft die *Dominanzbedingung*: ob $v \in D$ liegt oder ob v einen Nachbarn in D hat – dazu läuft er die Zeile von v in der Adjazenzmatrix durch. Er akzeptiert, wenn die Größe stimmt und jeder Knoten dominiert wird.

Warum das korrekt ist – beide Richtungen. Existiert ein Dominating Set der Größe $\leq k$, wird es akzeptiert. Existiert keines, bleibt bei jeder Menge dieser Größe ein Knoten undominiert, und der Verifizierer lehnt ab.

Die Laufzeit. Die Größenprüfung kostet $O(|V|)$; die Dominanzprüfung geht für jeden der $|V|$ Knoten eine Matrixzeile durch, also $O(|V|^2)$. Insgesamt $O(|V|^2)$. Damit DOMINATING SET $\in$ NP.

Woher die Schwere kommt

► Zusammenhang

Im Material dieses Kurses wird für DOMINATING SET nur die *NP-Zugehörigkeit* behandelt – es war eine Klausuraufgabe genau zu diesem Nachweis. Eine *Härte-Reduktion* wird in den Vorlesungs- und Übungsquellen nicht geführt. DOMINATING SET ist zwar bekanntlich NP-vollständig (Standardreduktion aus Vertex Cover), aber diese Reduktion gehört nicht zum gesicherten Kursstoff – deshalb wird sie hier nicht als solche dargestellt.

25 Longest Path

■ Definition 25.1 Longest Path

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Frage: Gibt es einen *einfachen* Pfad (kein Knoten doppelt) der Länge $\geq k$?

● Beispiel 25.2 Ein langer Pfad

In einem Weg-Graphen $1 - 2 - 3 - 4 - 5$ ist die ganze Kette ein einfacher Pfad der Länge 4 (vier Kanten). Für $k = 4$ ist die Antwort also Ja.

Warum Longest Path in NP liegt

Das Zertifikat. Eine Folge von $k + 1$ paarweise verschiedenen Knoten $(v_0, v_1, \dots, v_k)$ – der behauptete Pfad. Jeder Knoten braucht $O(\log |V|)$ Bits, die Folge also $O(k \log |V|)$ Bits, polynomiell.

Die Prüfung. Der Verifizierer prüft erstens, dass alle v_i paarweise verschieden sind, und zweitens, dass zwischen je zwei aufeinanderfolgenden Knoten eine Kante liegt. Er akzeptiert bei beidem.

Warum das korrekt ist – beide Richtungen. Hat G einen einfachen Pfad der Länge $\geq k$, so bilden seine ersten $k + 1$ Knoten ein akzeptiertes Zertifikat. Wird umgekehrt ein Zertifikat

akzeptiert, so ist es eine Folge verschiedener Knoten mit lauter echten Kanten – ein einfacher Pfad der Länge k .

Die Laufzeit. Die Verschiedenheit kostet $O(k^2)$ Vergleiche, die Kantenbedingung $O(k)$ Nachschläge; insgesamt $O(k^2) \subseteq O(|V|^2)$. Damit LONGEST PATH $\in$ NP.

Woher die Schwere kommt

► Zusammenhang

Wie bei DOMINATING SET behandelt der Kurs für LONGEST PATH nur die *NP-Zugehörigkeit* (Klausuraufgabe). Eine Härte-Reduktion steht nicht im Kursmaterial – obwohl LONGEST PATH bekanntlich NP-vollständig ist (es verallgemeinert den Hamiltonpfad). Wir halten uns hier an das, was der Kurs gesichert liefert.

26 Das Halteproblem – schwer, aber nicht in NP

Zum Abschluss des Katalogs ein Problem, das die Grenzen aufzeigt: Es ist NP-schwer, liegt aber *nicht* in NP – weil es überhaupt nicht lösbar ist.

■ Definition 26.1 Halteproblem

Gegeben die Beschreibung eines Programms M und einer Eingabe w . Frage: Hält M auf w irgendwann an, oder läuft es für immer?

Das Halteproblem ist *unentscheidbar* – kein Algorithmus löst es, mit keiner noch so großen Laufzeit. Das ist ein tiefes, bewiesenes Resultat (Turing). Deshalb kann es auch keinen Verifizierer geben, und das Halteproblem liegt *nicht* in NP.

Woher die Schwere kommt

🔍 Idee: $\text{SAT} \leq_p \text{Halteproblem}$

Zu einer Formel φ baut man ein Programm M_φ , das die Eingabe ignoriert, *alle* Belegungen der Reihe nach durchprobiert und genau dann anhält, wenn es eine erfüllende findet – sonst läuft es ewig weiter. Dann gilt: M_φ hält genau dann, wenn φ erfüllbar ist. Also $\varphi \in \text{SAT} \iff \langle M_\varphi, \varepsilon \rangle \in \text{HALTEPROBLEM}$.

☆ Aha

Der Clou liegt im Zeitverhalten. M_φ läuft im Nein-Fall *ewig* – aber die *Konstruktion* von M_φ aus φ ist billig (polynomiell), und nur das zählt bei einer Reduktion. Also ist das Halteproblem mindestens so schwer wie SAT, liegt aber nicht in NP. Das zeigt: **NP-schwer ist nicht dasselbe wie NP-vollständig**. NP-vollständig verlangt zusätzlich die Zugehörigkeit zu NP – die dem Halteproblem fehlt.

Teil C – Untere Schranken: die ETH

NP-Vollständigkeit sagt nur „vermutlich nicht polynomiell“. Sie sagt nicht, *wie* exponentiell ein Problem ist. Die Exponentialzeit-Hypothese (ETH) ist eine schärfere Annahme, aus der man konkrete *untere* Laufzeitschranken herleitet. In diesem Teil steht bei jedem Schritt die *Idee*.

27 Die Bausteine der ETH

Die o-Notation

■ Definition 27.1 Klein-o

$f(x) = o(g(x))$ bedeutet: f wächst *echt langsamer* als g , formal $f(x)/g(x) \rightarrow 0$. Für jedes feste $\delta > 0$ gilt $\delta n \neq o(n)$ – δn wächst nicht langsamer als n , nur mit anderem Vorfaktor.

Deshalb sind „kein $2^{\delta n}$ für ein festes δ “ und „kein $2^{o(n)}$ “ austauschbare Schreibweisen. Ein „ $2^{o(n)}$ -Algorithmus“ hätte einen Exponenten, der langsamer als linear wächst – er liefе *subexponentiell*, etwa mit $2^{\sqrt{n}}$.

Die Hypothese

■ Definition 27.2 Exponentialzeit-Hypothese (ETH)

Es gibt eine Konstante $\delta > 0$, sodass 3-SAT mit n Variablen *nicht* in Zeit $2^{\delta n} \cdot (n + m)^{O(1)}$ lösbar ist. Kurz: Es gibt keinen $2^{o(n)}$ -Algorithmus für 3-SAT.

Das ist eine *Annahme*, stärker als $P \neq NP$, aber breit geglaubt: Alle bekannten 3-SAT-Algorithmen brauchen c^n (das kleinste bekannte c liegt knapp über 1,3).

Das Sparsification-Lemma – und warum n gegen m

▲ Satz 27.3 Sparsification-Lemma

Unter der ETH gibt es auch keinen $2^{o(m)}$ -Algorithmus für 3-SAT, wobei m die *Klauselzahl* ist.

★ Intuition

Warum braucht man dieses Lemma überhaupt? Die ETH ist über die *Variablenzahl* n formuliert. Viele Reduktionen erzeugen ihre Zielgröße aber (auch) aus der *Klauselzahl* m . Und m kann viel größer als n sein. Deshalb folgt „kein $2^{o(m)}$ “ *nicht* automatisch aus „kein $2^{o(n)}$ “. Das Sparsification-Lemma überträgt die Härte von n auf m . Die Brücke zwischen beiden ist $n \leq 3m$ (jede Variable kommt in einer Klausel vor, jede Klausel hat ≤ 3 Literale).

Das Beweisschema: Kontraposition

💡 Idee: So beweist man eine untere Schranke

Man nimmt an, das Zielproblem hätte einen zu schnellen Algorithmus, und leitet daraus einen zu schnellen Algorithmus für 3-SAT ab – ein Widerspruch. Der Textbaustein, in *jedem* Teilpunkt zu wiederholen:

Angenommen, es gäbe einen Algorithmus, der [Zielproblem] in $2^{o(p)} \cdot |I|^{O(1)}$ löst. Weil der Parameter p mit der Reduktionsgröße $O(f(m))$ zusammenhängt, ergäbe die Kombination aus Reduktion und Algorithmus einen 3-SAT-Algorithmus in $2^{o(m)} \cdot |I|^{O(1)}$. Nach dem Sparsification-Lemma geht das nur, wenn die ETH nicht gilt – Widerspruch.

✗ Stolperstein

Entscheidend ist die Frage: *Hängt der Parameter an n oder an m ?* Hängt er nur an der Variablenzahl n , argumentiert man *direkt* mit der ETH und braucht das Lemma nicht. Hängt er (auch) an der Klauselzahl m , *muss* man das Sparsification-Lemma zitieren. Es immer oder nie zu nennen, kostet Punkte.

Die Wurzel-Regel

Wächst der Zielparameter *polynomiell* in m , etwa $p = O(m^c)$, so wird die Schranke zu $2^{o(\sqrt[p]{p})}$. Denn aus $q = o(\sqrt[p]{p})$ und $p = O(m^c)$ folgt $q = o(m)$. Beispiel: Erzeugt eine Reduktion $O(m^2)$ Kanten, so lautet die Schranke „kein $2^{o(\sqrt{|E|})}$ “.

28 Untere Schranken für die einzelnen Probleme

Wendet man das Schema auf die Reduktionen des Katalogs an, bekommt man für jedes Problem eine untere Schranke – mit dem Parameter, an dem sie hängt. Wir gehen die Probleme einzeln durch.

Clique

Die Reduktion $3\text{-SAT} \leq_p k\text{-CLIQUE}$ erzeugt einen Graphen mit $|V| = O(m)$ Knoten (ein Knoten pro Literalvorkommen), $|E| = O(m^2)$ Kanten und Cliquengröße $k = m$. Setzt man einen $2^{o(|V|)}$ -Algorithmus für CLIQUE an, so wäre er wegen $|V| = O(m)$ ein $2^{o(m)}$ -Algorithmus für 3-SAT – ausgeschlossen nach dem Sparsification-Lemma. Also gibt es keinen $2^{o(|V|)}$ -Algorithmus. Für die Kanten liefert die Wurzel-Regel wegen $|E| = O(m^2)$ die Schranke „kein $2^{o(\sqrt{|E|})}$ “. Und wegen $k = m$ gibt es keinen $2^{o(k)}$ -Algorithmus.

Independent Set

Die Reduktion $\text{CLIQUE} \leq_p \text{INDEPENDENT SET}$ bildet nur den Komplementgraphen und lässt k unverändert. Der Komplementgraph hat *dieselbe* Knotenzahl $|V|$ und dieselbe Cliquengröße k ; die Kantenanzahl bleibt in der Größenordnung $O(|V|^2)$. Weil sich keiner der drei Parameter vergrößert, überträgt sich jede Schranke von CLIQUE wörtlich: kein $2^{o(|V|)}$, kein $2^{o(\sqrt{|E|})}$, kein $2^{o(k)}$ für INDEPENDENT SET.

Vertex Cover

Auch VERTEX COVER bekommt man über den Komplementgraphen – hier mit Schranke $k' = n - k$. Die Knoten- und Kantenanzahl bleiben wie beim Independent Set unverändert, und $k' = n - k \leq n = O(|V|)$. Deshalb gelten für VERTEX COVER dieselben unteren Schranken: kein $2^{o(|V|)}$, kein $2^{o(\sqrt{|E|})}$, kein $2^{o(k')}$.

Hitting Set – der Fall, an dem man beide Situationen sieht

Die Reduktion $3\text{-SAT} \leq_p \text{HITTING SET}$ liefert die Parameter $|U| = 2n$ (das Universum sind die Literale), $r = n + m$ (die Zahl der Mengen) und $k = n$. An diesen drei Parametern sieht man nebeneinander, wann man das Sparsification-Lemma braucht und wann nicht.

Die Schranke in $|U| = 2n$. Nimmt man einen $2^{o(|U|)}$ -Algorithmus für HITTING SET an, so ist er wegen $|U| = 2n$ ein $2^{o(n)}$ -Algorithmus. Über die Reduktion ergäbe das einen $2^{o(n)}$ -Algorithmus für 3-SAT – das widerspricht *direkt* der ETH. Hier braucht man das Sparsification-Lemma nicht, weil $|U|$ nur an der Variablenzahl n hängt.

Die Schranke in $r = n + m$. Nimmt man einen $2^{o(r)}$ -Algorithmus an, so ist er wegen $r = n + m$ und $n \leq 3m$, also $r = O(m)$, ein $2^{o(m)}$ -Algorithmus für 3-SAT. Dass es keinen solchen gibt, sagt *nicht* die ETH selbst, sondern erst das Sparsification-Lemma – weil der Parameter an der Klauselzahl m hängt. Diesen Zusatzschritt muss man hier also zitieren.

Die Schranke in $k = n$. Nimmt man einen $2^{o(k)}$ -Algorithmus an, so ist er wegen $k = n$ ein $2^{o(n)}$ -Algorithmus und ergäbe wie im ersten Fall einen $2^{o(n)}$ -Algorithmus für 3-SAT – direkter Widerspruch zur ETH, ohne Sparsification-Lemma.

SubSet Sum und Partition

Die gewöhnliche Kette bis SUBSET SUM erzeugt sehr viele Zahlen und gibt nur eine schwache Wurzel-Schranke. Die *strenge* Reduktion $3\text{-SAT} \leq_p \text{SUBSET SUM}$ (Ziffern zur Basis 10, siehe beim SubSet Sum) erzeugt dagegen nur $|A| = 2n + 2m \leq 8m = O(m)$ Zahlen. Ein $2^{o(n)}$ -Algorithmus für SUBSET SUM – wobei n hier die *Anzahl der Zahlen* ist – ergäbe über diese wenigen Zahlen einen $2^{o(m)}$ -Algorithmus für 3-SAT, ausgeschlossen nach dem Sparsification-Lemma. Es gibt also keinen $2^{o(n)}$ -Algorithmus für SUBSET SUM. Über die Reduktion $\text{SUBSET SUM} \leq_p \text{PARTITION}$, die nur zwei Zahlen hinzufügt, überträgt sich dieselbe Schranke auf PARTITION.

3-Color

Für 3-COLOR benutzt man die eigene $O(n + m)$ -Gadget-Konstruktion aus dem Katalog: Sie baut aus einer 3-SAT-Formel einen Graphen mit $|V| = O(n + m) = O(m)$ Knoten. Ein $2^{o(|V|)}$ -Algorithmus für 3-COLOR wäre damit ein $2^{o(m)}$ -Algorithmus für 3-SAT – nach dem Sparsification-Lemma ausgeschlossen. Also gibt es keinen $2^{o(|V|)}$ -Algorithmus für 3-COLOR.

Set Cover

Für SET COVER nutzt man die Dualität zu HITTING SET: Der Rollentausch vertauscht Universum und Mengenzahl, macht also aus $|U|$ die Mengenzahl und aus der Mengenzahl das neue $|U|$. Damit übertragen sich die Hitting-Set-Schranken direkt: kein $2^{o(|U|)}$ und kein $2^{o(r)}$ für SET COVER, wieder mit dem Sparsification-Lemma bei der m -abhängigen Mengenzahl.

Teil D – Approximation

Wenn ein Optimierungsproblem (vermutlich) keine schnelle exakte Lösung hat, berechnet man in Polynomialzeit eine *beweisbar gute* Näherung. Bei jedem Algorithmus steht hier die Güte (exakt) und die *Idee* des Güte-Beweises – die zentrale Ungleichung.

29 Der Gütebegriff

■ Definition 29.1 Multiplikative Güte

Sei $\text{OPT}(I)$ der optimale Wert einer Instanz I . Ein Algorithmus A hat *Güte* α , wenn für *alle* Instanzen gilt: bei Minimierung $A(I) \leq \alpha \cdot \text{OPT}(I)$; bei Maximierung $\text{OPT}(I) \leq \alpha \cdot A(I)$. Die Güte heißt *scharf*, wenn Instanzen den Faktor beliebig genau erreichen.

★ Intuition

Güte α ist eine Garantie für den schlimmsten Fall. Güte 2 bei einem Minimierungsproblem heißt: Egal welche Eingabe, die Lösung ist nie mehr als doppelt so teuer wie die beste mögliche. Kleineres α ist besser; $\alpha = 1$ wäre exakt optimal.

Manche Probleme erlauben sogar eine *einstellbare* Genauigkeit – eine ganze Familie (A_ε) mit Güte $1 + \varepsilon$ für jedes ε . Je nach Laufzeit trägt so eine Familie einen eigenen Namen; wir betrachten die drei Stufen einzeln.

■ Definition 29.2 PTAS

Eine Familie (A_ε) mit Güte $1 + \varepsilon$ heißt *PTAS* (polynomielles Approximationsschema), wenn die Laufzeit für *jedes feste* ε polynomiell in n ist.

Der Haken: ε darf im Exponenten von n stecken, etwa $n^{1/\varepsilon}$. Für kleines ε wird das schnell unpraktisch.

■ Definition 29.3 EPTAS

Ein *EPTAS* ist ein PTAS, dessen Laufzeit die Form $f(1/\varepsilon) \cdot \text{poly}(n)$ hat – ε steht nicht mehr im Exponenten von n .

■ Definition 29.4 FPTAS

Ein *FPTAS* ist ein PTAS, dessen Laufzeit polynomiell in n und in $1/\varepsilon$ ist, etwa $O(n^3/\varepsilon)$. Das ist die stärkste Form.

Ein verwandter Begriff: *pseudopolynomiell* heißt eine Laufzeit, die polynomiell in den *Zahlenwerten* der Eingabe ist (etwa in der Kapazität). Wegen der binären Kodierung (Kapitel 2) ist das *nicht* dasselbe wie polynomiell in der Eingabelänge.

30 TSP ist im Allgemeinen nicht approximierbar

▲ Satz 30.1 Kein Näherungsalgorithmus für allgemeines TSP

Für beliebige Distanzen gibt es *keinen* Näherungsalgorithmus mit beschränkter Güte – außer $P = NP$.

Die Idee: Man reduziert HAMILTONKREIS. Echte Kanten bekommen Distanz 1, fehlende Kanten eine *riesige* Distanz. Gäbe es einen Algorithmus mit fester Güte α , müsste er die billige Hamilton-Tour (Länge $|V|$) von jeder teuren unterscheiden – und könnte damit HK exakt entscheiden.

Der Ausweg ist eine realistische Zusatzannahme: das *metrische* TSP mit symmetrischen Distanzen und der Dreiecksungleichung $d(i, j) \leq d(i, k) + d(k, j)$. Dort funktionieren die folgenden zwei Algorithmen.

31 ΔTSP_1 : der Algorithmus als Bilderfolge

Idee: Der Dreisprung

Man baut zuerst ein billiges Gerüst, das alle Städte verbindet (einen minimalen Spannbaum), macht daraus einen Rundweg über alle Kanten (einen Eulerkreis, indem man alle Grade gerade macht) und kürzt diesen zur Tour ab.

Wir gehen es an einem festen Beispiel durch: Städte A, B, C, D, E mit $d(A, E) = 1$, $d(B, C) = 1$, $d(A, C) = 2$, $d(C, D) = 2$ (übrige Distanzen 2 oder 3). Das Layout bleibt in allen Bildern gleich.

Bild 1 – die optimale Tour (zum Vergleich). So sieht die beste Rundreise aus; sie hat Länge 8. Der Algorithmus kennt sie nicht, wir zeigen sie nur zum Vergleich.

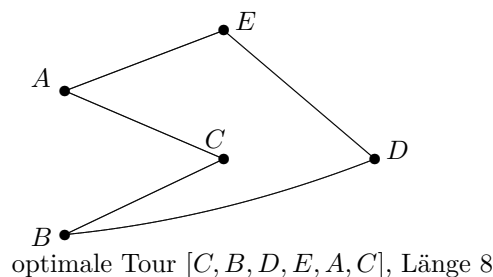
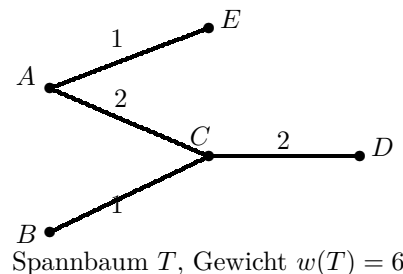


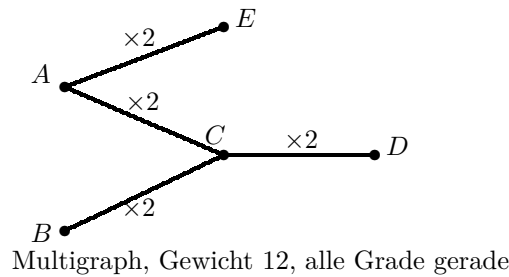
Bild 2 – Schritt 1: minimaler Spannbaum. Der Algorithmus verbindet alle Städte so billig wie möglich zu einem Baum: Kanten $A-E$ (1), $B-C$ (1), $A-C$ (2), $C-D$ (2) mit Gewicht 6.



★ Intuition

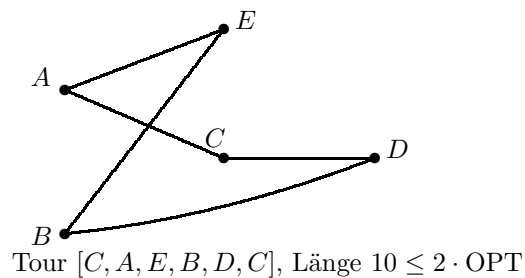
Warum ein Spannbaum? Weil er *garantiert billiger* ist als die optimale Tour: Nimm die optimale Tour und lösche eine Kante – übrig bleibt ein Weg durch alle Städte, also ein Spannbaum. Der *minimale* Spannbaum ist höchstens so teuer. Also $w(T) \leq \text{OPT}$. Das ist die erste der drei Ungleichungen des Güte-Beweises.

Bild 3 – Schritt 2 und 3: verdoppeln, dann Eulerkreis. Jede Baumkante wird verdoppelt (im Bild „ $\times 2$ “). Dadurch hat jeder Knoten geraden Grad – die Bedingung für einen Eulerkreis, der jede Kante genau einmal nutzt. Sein Gewicht ist $2 \cdot 6 = 12$.



Ein Eulerkreis ist z. B. $[C, A, E, A, C, B, C, D, C]$ mit Länge 12; manche Städte werden dabei mehrfach besucht.

Bild 4 – Schritt 4: abkürzen zur Tour. Eine echte Rundreise darf jede Stadt nur einmal besuchen. Man läuft den Eulerkreis ab und *überspringt* bereits besuchte Städte. Aus $[C, A, E, A, C, B, C, D, C]$ wird die Tour $[C, A, E, B, D, C]$.



👉 Idee: Die Güte 2 – drei Ungleichungen

ΔTSP_1 hat Güte 2, und der Beweis besteht aus drei Schritten: (1) $w(T) \leq \text{OPT}$ (Spannbaum billiger als Tour, s.o.); (2) der Eulerkreis auf den verdoppelten Kanten hat Länge $2w(T) \leq 2\text{OPT}$; (3) das Abkürzen verlängert wegen der Dreiecksungleichung nicht, also $d(R) \leq 2w(T) \leq 2\text{OPT}$. Im Beispiel: Länge 10, garantiert ≤ 16 , tatsächlich nahe am Optimum 8.

★ Intuition

Der schlimmste Fall wird von einer Stern-Konstruktion erreicht: n Städte, deren optimale Tour Länge n hat, während ΔTSP_1 auf Länge $2n - 2$ kommt. Das Verhältnis geht gegen 2 – die Schranke ist also (fast) scharf. Nötig sind Symmetrie und Dreiecksungleichung.

32 Christofides: schlauer verdoppeln

ΔTSP_1 verdoppelt *alle* Kanten – das kostet zusätzlich etwa $w(T) \approx \text{OPT}$, daher die Güte 2. Christofides repariert nur die *ungeraden* Knoten und kommt so auf Güte 1,5.

 Idee: Nur die schiefen Knoten reparieren

Ein Eulerkreis braucht nur, dass jeder Knoten geraden Grad hat. Im Spannbaum haben ohnehin nur manche Knoten ungeraden Grad. Statt alles zu verdoppeln, verbindet Christofides genau diese ungeraden Knoten durch ein *minimales perfektes Matching* – die billigste Art, sie paarweise zu verkuppeln.

Bild 1 – der minimale Spannbaum. Christofides startet mit demselben Spannbaum wie ΔTSP_1 : Kanten $A-E$ (1), $B-C$ (1), $A-C$ (2), $C-D$ (2), Gewicht 6.

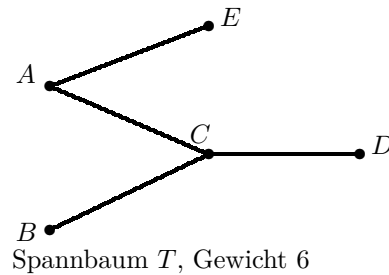


Bild 2 – die ungeraden Knoten. Man zählt die Grade im Baum: A hat Grad 2 (gerade), alle anderen ungerade. Die ungeraden Knoten $X = \{B, C, D, E\}$ sind mit einem Ring markiert.

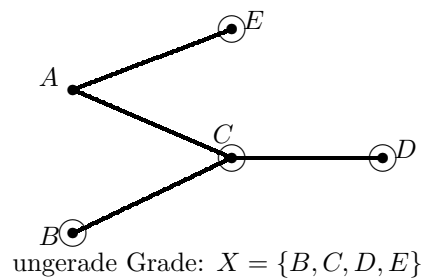


Bild 3 – das Matching. Die billigste Paarung der vier Knoten ist $\{B-C, D-E\}$ mit Kosten $1 + 2 = 3$. Diese zwei Matching-Kanten (dünn) kommen zum Baum dazu – jetzt haben alle Knoten geraden Grad.

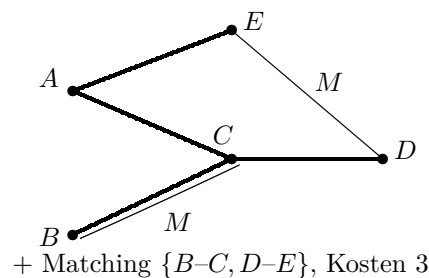
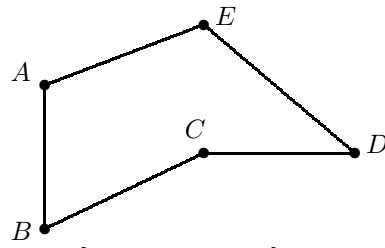


Bild 4 – Eulerkreis und abkürzen. Aus $T + M$ gewinnt man einen Eulerkreis und kürzt ihn ab. Ergebnis: Tour $[C, B, A, E, D, C]$ mit Länge 9 – besser als die 10 von ΔTSP_1 .



Tour $[C, B, A, E, D, C]$, Länge $9 \leq 1,5 \cdot \text{OPT}$

🔍 Idee: Die Güte $1,5$ – warum das Matching $\leq \text{OPT}/2$ kostet

Die Tour hat Länge $w(T) + d(M) \leq \text{OPT} + d(M)$. Der Kern ist die Abschätzung $d(M) \leq \text{OPT}/2$: Betrachtet man die ungeraden Knoten X in der Reihenfolge einer optimalen Tour, so bilden sie *zwei* sich abwechselnde Matchings M_1, M_2 . Wegen der Dreiecksungleichung ist $\text{OPT} \geq d(M_1) + d(M_2) \geq 2d(M)$, da M das *minimale* perfekte Matching ist. Also $d(M) \leq \text{OPT}/2$ und die Tour $\leq w(T) + \text{OPT}/2 \leq 1,5 \text{OPT}$.

✗ Stolperstein

Zwei Dinge werden oft falsch gemacht: Das Matching läuft *nur* über die ungerad-gradigen Knoten des Spannbaums, nicht über alle. Und ohne die Dreiecksungleichung bricht das Abkürzen zusammen.

33 Knapsack: Greedy

Beim Rucksack als Optimierungsproblem packt man Gegenstände mit maximalem Gesamtprofit ein, ohne die Kapazität B zu überschreiten. Wir behandeln vier Verfahren, jedes einzeln.

🔍 Idee: Greedy

Sortiere die Gegenstände nach „Profit pro Gewicht“ (Profitedichte p_i/w_i) und pack der Reihe nach ein, was noch passt.

▲ Satz 33.1 Greedy hat *keine* beschränkte Güte

Das Verhältnis OPT/GA kann beliebig groß werden.

Gegenbeispiel: Ein Gegenstand mit Gewicht 1 und Profit 1, ein anderer mit Gewicht B und Profit $B - 1$, Kapazität B . Greedy schaut auf die Dichte – der kleine hat Dichte 1, der große $(B - 1)/B < 1$ – und nimmt den kleinen, Profit 1. Optimal wäre der große mit Profit $B - 1$. Das Verhältnis $B - 1$ wächst mit B ins Unendliche.

34 Knapsack: Modified Greedy

Idee: Modified Greedy (MGA)

Rechne zwei Kandidaten aus – das Greedy-Ergebnis *und* „nimm nur den einen profitabelsten Gegenstand“ – und gib den besseren aus.

▲ Satz 34.1 MGA hat Güte 2

$\text{OPT} \leq 2 \text{MGA}$ für alle Instanzen.

Die zentrale Ungleichungskette: Man betrachtet die *fraktionale* Relaxierung, in der man Gegenstände auch anteilig einpacken darf; ihr Optimum OPT_f ist mindestens OPT . Die fraktionale Lösung packt die Gegenstände nach Dichte, bis das erste nicht mehr ganz passt – dieses „Split-Item“ $k + 1$ wird anteilig genommen. Damit gilt

$$\text{OPT} \leq \text{OPT}_f \leq \text{GA} + p_{k+1} \leq \text{GA} + p_{\max} \leq 2 \cdot \max\{\text{GA}, p_{\max}\} = 2 \text{MGA}.$$

Greedy verliert also seinen Rückstand nur an dem einen Split-Item, und dessen Profit fängt der zweite Kandidat (bestes Einzel-Item) ab.

35 Knapsack: Sahni-Algorithmus

Idee: Sahni (k -Enumeration)

Probiere *alle* Vorauswahlen von höchstens k Gegenständen durch. Fülle jede Vorauswahl mit Greedy auf und gib die beste Gesamtlösung aus.

▲ Satz 35.1 Sahni ist ein PTAS

Der Sahni-Algorithmus erreicht Güte $1 + 1/k$ bei Laufzeit $O(n^{k+1})$.

Je größer k , desto besser die Garantie ($k = 2$ gibt $3/2$, $k = 3$ gibt $4/3$), aber desto teurer die Laufzeit. Für festes k ist sie polynomiell – also genau ein PTAS. Modified Greedy ist der Spezialfall, in dem man das beste Einzel-Item vorwählt.

36 Knapsack: FPTAS

Idee: FPTAS durch Profit-Skalierung

Runde die Profite auf gröbere Werte ab (teile durch ein passendes K und runde), löse das gröbere Problem *exakt* mit dynamischer Programmierung, und gib diese Lösung aus.

▲ Satz 36.1 FPTAS für Knapsack

Das Verfahren erreicht Güte $1 + \varepsilon$ bei Laufzeit $O(n^3/\varepsilon)$.

Die zwei zentralen Ungleichungen: Mit skalierten Profiten $p'_i = \lfloor p_i/K \rfloor$ gilt (1) $\text{OPT} \leq K \cdot \text{OPT}(I_K) + Kn$ – durch das Abrunden verliert jedes Item höchstens K , bei $\leq n$ Items also $\leq Kn$ gesamt; und (2) $K \cdot \text{OPT}(I_K) \leq A_K(I)$ – die skalierte Lösung ist nicht schlechter. Zusammen $\text{OPT} \leq A_K + Kn$, und mit $\text{OPT} \geq p_{\max}$ und der Wahl von K folgt der relative Fehler $\leq \varepsilon$. Das exakte DP ist pseudopolynomiell ($O(n^2 p'_{\max})$); durch das Abrunden werden die Werte klein genug, dass die Laufzeit $O(n^3/\varepsilon)$ wird.

37 Scheduling: List Scheduling

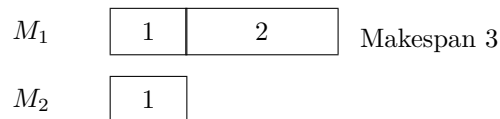
Zurück zu $P||C_{\max}$: n Jobs auf m Maschinen, minimiere die höchste Last.

🗨 Idee: List Scheduling

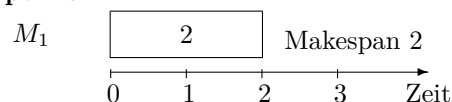
Gehe die Jobs in ihrer gegebenen Reihenfolge durch und lege jeden Job auf die Maschine, die im Moment die *kleinste* Last hat.

Das folgende Gantt-Bild zeigt an drei Jobs der Größen 1, 1, 2 auf zwei Maschinen, warum die Reihenfolge zählt. Oben List Scheduling (Makespan 3), unten die optimale Verteilung (Makespan 2). Die Balkenlänge ist die Zeit.

List Scheduling (Reihenfolge 1, 1, 2):



optimal:



▲ Satz 37.1 List Scheduling hat Güte $2 - \frac{1}{m}$ (scharf)

Die Idee: Betrachte die Maschine mit der höchsten Last L und den *letzten* Job J_k , der auf ihr landete. In dem Moment, als J_k zugewiesen wurde, war diese Maschine die *leerste* – also waren alle Maschinen mindestens so voll wie $L - p_k$. Daraus folgt $\sum p_i \geq m(L - p_k) + p_k$. Mit den beiden *Universalschranken* $\text{OPT} \geq \frac{1}{m} \sum p_i$ (die Last verteilt sich auf m Maschinen) und $\text{OPT} \geq p_k$ (der Job muss irgendwo laufen) ergibt sich $L = \text{LS}(I) \leq (2 - \frac{1}{m}) \text{OPT}$.

★ Intuition

Der schlimmste Fall: $m(m-1)$ Jobs der Größe 1, danach *ein* Job der Größe m . List Scheduling verteilt die Einsen gleichmäßig (je $m-1$) und setzt den großen Job obendrauf: Last $2m-1$. Optimal wäre m (großer Job allein). Das Verhältnis $\frac{2m-1}{m} = 2 - \frac{1}{m}$ erreicht die Schranke exakt.

38 Scheduling: LPT

📖 Idee: LPT (Longest Processing Time first)

Sortiere die Jobs zuerst *absteigend* nach Laufzeit – die längsten zuerst – und wende darauf List Scheduling an.

Der einzige Unterschied zu List Scheduling ist die Sortierung. Sie sorgt dafür, dass die großen Jobs früh verteilt werden, solange die Maschinen noch leer sind.

▲ Satz 38.1 LPT hat Güte $\frac{4}{3} - \frac{1}{3m}$ (scharf)

Die Idee (minimales Gegenbeispiel): Man betrachtet ein kleinstes Gegenbeispiel – eine Instanz mit möglichst wenigen Jobs, bei der LPT die Schranke verletzt. In so einer minimalen Instanz muss der *letzte* (also kleinste) Job J_n den Makespan bestimmen; täte er es nicht, könnte man ihn weglassen und hätte ein noch kleineres Gegenbeispiel.

Aus der Annahme, dass LPT die Schranke $\frac{4}{3} - \frac{1}{3m}$ verletzt, rechnet man dann $p_n > \text{OPT}/3$ heraus. Das ist eine starke Aussage: Wenn schon der kleinste Job größer als ein Drittel des Optimums ist, kann im *optimalen* Schedule keine Maschine mehr als zwei Jobs tragen (drei Jobs à $> \text{OPT}/3$ ergäben $> \text{OPT}$).

Nun formt man den optimalen Schedule schrittweise in den LPT-Schedule um. Jeder Schritt ist ein *Austausch* zweier Jobs zwischen zwei Maschinen – man tauscht so, dass die Reihenfolge Stück für Stück der von LPT gleicht. Man weist für jeden solchen Austausch nach, dass der Makespan dabei *nie steigt* (weil bei höchstens zwei Jobs pro Maschine ein Tausch die stärker belastete Maschine nur entlasten kann). Am Ende steht ein LPT-Schedule mit dem Makespan des Optimums, also $\text{LPT} = \text{OPT}$ – im Widerspruch dazu, dass die Instanz ein Gegenbeispiel war. Damit kann es kein Gegenbeispiel geben, und die Schranke gilt.

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

📖 Idee: Zwei komplementäre Belegungen

Beim Optimierungsproblem MAX-3-SAT will man möglichst viele Klauseln erfüllen. Der Algorithmus wertet die Formel nur *zweimal* aus – einmal mit „alle Variablen falsch“ (β_0), einmal mit „alle wahr“ (β_1) – und gibt die bessere zurück.

▲ Satz 39.1 Der Algorithmus hat Güte 2

$$v(A(\varphi)) \geq \frac{1}{2} v(\text{OPT}(\varphi)).$$

Die zentrale Beobachtung: Jede Klausel enthält mindestens ein Literal – ist es positiv, wird die Klausel von β_1 erfüllt; ist es negativ, von β_0 . Jede Klausel wird also von *mindestens einer* der beiden Belegungen erfüllt, folglich $v(\beta_0) + v(\beta_1) \geq m$. Die bessere von beiden erfüllt somit $\geq m/2$ Klauseln. Da eine optimale Belegung höchstens alle m Klauseln erfüllt ($\text{OPT} \leq m$), ist $v(A) \geq m/2 \geq \text{OPT}/2$.

★ Intuition

Der schlimmste Fall baut man aus komplementären Klauseln, etwa $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_0 \vee \neg x_2 \vee \neg x_3)$. Beide Klauseln sind gemeinsam erfüllbar ($\text{OPT} = 2$), aber β_1 erfüllt nur die erste und β_0 nur die zweite – der Algorithmus erreicht also nur $1 = \text{OPT}/2$.

40 2-Approximation für Vertex Cover

🔍 Idee: Unüberdeckte Kanten voll aufnehmen

Solange es eine noch unüberdeckte Kante gibt, nimm *beide* ihre Endpunkte ins Vertex Cover auf.

▲ Satz 40.1 Der Algorithmus hat Güte 2

$|C| \leq 2 |C^*|$, wobei C^* ein optimales Vertex Cover ist.

Die Idee (Optimum von unten über ein Matching): Sei A die Menge der Kanten, für die der Algorithmus beide Endpunkte genommen hat; dann ist $|C| = 2|A|$. Die Kanten in A sind paarweise *knotendisjunkt* – sobald eine Kante genommen wurde, waren ihre Endpunkte überdeckt, eine spätere Kante aus A kann sie also nicht berühren. A ist damit ein *Matching*. Jedes Vertex Cover muss für jede Matching-Kante einen *eigenen* Knoten stellen, also $|C^*| \geq |A|$. Zusammen: $|C| = 2|A| \leq 2|C^*|$.

★ Intuition

Das Muster „Optimum von unten durch ein Matching abschätzen“ ist eines der drei Standard-Argumente für Güte-Beweise. Die anderen beiden sind „Widerspruch und Zählen“ (wie bei MAX-3-SAT) und „die zwei Universalschranken“ (wie beim Scheduling).

41 Zum Schluss

Du hast jetzt das gesamte Fundament der Vorlesung beisammen – und zwar vollständig, nicht in Stichworten. Für jedes Problem des Katalogs weißt du, wie es definiert ist, warum es in

NP liegt, woher seine Schwere kommt und welche Reduktionen von ihm ausgehen. Du kennst die ETH und das Schema, mit dem man untere Schranken herleitet. Und du kennst jeden Approximationsalgorithmus mit seiner Güte und der Idee des Güte-Beweises.

☆ Aha

Der rote Faden war die *Reduktion*. Sie verbindet alle Probleme zu einer einzigen Familie: Von SAT aus fließt die Schwere über die ganze Landkarte, und dieselbe Idee – „bekannt $\leq_p$ neu“ – trägt in der ETH sogar konkrete untere Schranken. Wo exakte Lösungen zu teuer werden, springt die Approximation ein und rettet mit beweisbaren Garantien, was zu retten ist. Das ist die ganze Geschichte des Kurses.

Der nächste Schritt. Dieses Heft gibt dir das Verständnis. Um es in Können zu verwandeln, rechne die Präsenz-, Haus- und Klausuraufgaben und führe dort selbst aus, was hier als Idee steht: Reduktionen vollständig beweisen (beide Richtungen!), Güte-Schranken herleiten, Worst-Case-Instanzen bauen. Viel Erfolg.