

Komplexität von Grund auf

Alle Probleme, alle Reduktionen, alle Ideen – ausführlich
Analyse von Algorithmen und Komplexität – CAU Kiel, SS 2026

*Jedes Problem für sich. Jede Reduktion mit ihrer Idee.
Nichts abgekürzt, nichts zusammengelegt.*

Worum es geht. Dieses Heft enthält das komplette Fundament der Vorlesung. Sein Kernstück ist ein Katalog, der jedes Problem des Kurses in vier Schritten behandelt:

1. **Was gefragt ist** – die Definition, mit einem kleinen Beispiel;
2. **warum das Problem in NP liegt** – als ausgeschriebene Erklärung: welches Zertifikat, wie geprüft wird, warum das in beide Richtungen stimmt, wie lange es dauert;
3. **wo die Schwere herkommt** – die Reduktion, die das Problem hart macht, samt der Idee ihrer Konstruktion;
4. **wohin die Schwere weiterfließt** – jede Reduktion, die vom Problem ausgeht, mit ihrer eigenen Idee.

Danach folgen die unteren Schranken (ETH) und die Approximationsalgorithmen – beides ebenfalls mit den Beweis-Ideen, Problem für Problem und Algorithmus für Algorithmus.

Zwei Grundsätze. Ähnliche Dinge werden hier trotzdem *getrennt* behandelt – jedes Problem hat seinen eigenen Abschnitt, jeder Nachweis wird eigenständig geführt. Und nichts wird mit „das sieht man leicht“ übergangen: Wo eine Idee gebraucht wird, steht sie auch da.

Die Kästen im Heft.

★ Intuition	der Gedanke hinter dem Formalen
☞ Analogie	ein Alltagsbild
■ Definition	die präzise Fassung
▲ Satz	ein Ergebnis samt erklärter Beweisidee
● Beispiel	konkret durchgerechnet
☞ Idee	die Konstruktion hinter einer Reduktion oder einem Beweis
☆ Aha	die Pointe
✗ Stolperstein	wo Fehler passieren
➤ Zusammenhang	die Verbindung zum Rest

Inhalt

Teil A – Die Grundlagen	5
1 Der Ausgangspunkt: ein widerspenstiges Problem	5
2 Die Klasse P: was „schnell“ heißen soll	6
3 Die Klasse NP: Lösungen prüfen statt finden	7
4 Reduktionen: das Vergleichswerkzeug	8
5 NP-Vollständigkeit: die härtesten Probleme in NP	9
Teil B – Der Katalog aller Probleme	11
6 SAT	11
7 3-SAT	13
8 k -Clique	15
9 Independent Set	17
10 Vertex Cover	18
11 Feedback Vertex Set	20
12 Δ -Cover	21
13 k -Color	23
14 Hamiltonkreis	24
15 TSP	26
16 3-dimensionales Matching	27
17 3-Exact Cover	28
18 SubSet Sum	29
19 Partition	32
20 Knapsack	33

21 $P C_{\max}$ (Scheduling)	34
22 Hitting Set	35
23 Set Cover	36
24 Dominating Set	38
25 Longest Path	39
26 Das Halteproblem	39
Teil C – Untere Schranken mit der ETH	40
27 Das Handwerkszeug	41
28 Die Schranken im Einzelnen	42
Teil D – Approximation	44
29 Was eine Güte-Garantie ist	44
30 TSP ohne Zusatzannahme: hoffnungslos	45
31 Δ TSP ₁ : Gerüst, Rundweg, Abkürzung	45
32 Christofides: das Matching statt der Verdopplung	47
33 Knapsack, Stufe 1: der pure Greedy	48
34 Knapsack, Stufe 2: Modified Greedy	49
35 Knapsack, Stufe 3: Sahni-Schema	49
36 Knapsack, Stufe 4: das FPTAS	50
37 Scheduling, Regel 1: List Scheduling	50
38 Scheduling, Regel 2: LPT	51
39 Parallel-Task-Scheduling: Jobs, die mehrere Maschinen brauchen	51
40 Strip Packing: der NFDH-Algorithmus	52
41 MAX-3-SAT: zwei Versuche reichen für Güte 2	54

42 Vertex Cover: die Beide-Endpunkte-Regel	55
43 Schlusswort	55

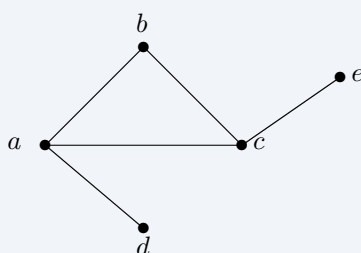
Teil A – Die Grundlagen

1 Der Ausgangspunkt: ein widerspenstiges Problem

Jede Theorie braucht einen Anlass. Unserer ist ein Problem, das sich mit einem Satz erklären lässt und trotzdem seit Jahrzehnten jedem schnellen Algorithmus widersteht.

Ein *Graph* besteht aus Punkten – den *Knoten* – und Verbindungslinien dazwischen, den *Kanten*. Eine *Clique* ist eine Gruppe von Knoten, in der jeder mit jedem verbunden ist. Die Frage „Gibt es im Graphen eine Clique aus k Knoten?“ ist das *Cliquenproblem*.

● Beispiel 1.1 Cliques in einem kleinen Graphen



Die Knoten a, b, c sind paarweise verbunden – eine Clique der Größe 3. Die Gruppe $\{a, c, d\}$ dagegen nicht: Zwischen c und d fehlt die Verbindung. Eine Clique aus vier Knoten gibt es in diesem Graphen nicht.

Wie findet ein Computer eine k -Clique? Der offensichtliche Weg: alle Knotengruppen der Größe k durchgehen und jede prüfen. Das funktioniert – aber die Anzahl der Gruppen explodiert. Bei n Knoten und $k = n/2$ sind es mindestens $2^{n/2}$ Stück. Jeder weitere Knoten *verdoppelt* den Aufwand ungefähr. Diese Wachstumsart heißt *exponentiell*, und sie ist durch keine Hardware der Welt einzuholen: Ein doppelt so schneller Rechner schafft genau *einen* Knoten mehr.

● Beispiel 1.2 Was exponentiell bedeutet

Knoten n	Gruppen (mind.)	Einordnung
20	rund tausend	sofort fertig
40	rund eine Million	unter einer Sekunde
100	rund 10^{15}	Wochen bis Monate
200	rund 10^{30}	länger als das Alter des Universums

Zwischen $n = 100$ und $n = 200$ liegt kein Faktor 2, sondern ein Faktor 10^{15} .

☆ Aha

Niemand hat je einen schnellen Algorithmus für die Clique gefunden – aber niemand hat bewiesen, dass es keinen gibt. Die Komplexitätstheorie macht aus dieser Verlegenheit ein Programm. Sie fragt nicht „wie knackt man dieses eine Problem?“, sondern „welche Probleme hängen so zusammen, dass sie alle gemeinsam stehen oder fallen?“ Auf diese Frage gibt es präzise Antworten, und sie füllen diesen Kurs.

2 Die Klasse P: was „schnell“ heißen soll

Bevor man über schwere Probleme redet, muss man festlegen, was „leicht“ ist. Zwei Vereinbarungen tragen alles Weitere.

Erstens: Laufzeit misst man in Abhängigkeit von der Eingabegröße n – zehn Zahlen sortieren ist etwas anderes als zehn Millionen. Und man nimmt den *schlimmsten Fall* über alle Eingaben einer Größe, denn eine Garantie, die nur „meistens“ hält, ist keine.

Zweitens: Als „schnell“ gilt eine Laufzeit, die höchstens wie eine feste Potenz von n wächst.

■ Definition 2.1 Die Klasse P

P ist die Menge aller Entscheidungsprobleme, die sich durch einen Algorithmus mit *polynomieller* Laufzeit lösen lassen – also $O(n^d)$ für eine Konstante d .

★ Intuition

Warum ausgerechnet „polynomiell“? Weil an dieser Grenze der qualitative Sprung passiert. Polynome wachsen zahm: n^2 bei doppelter Eingabe vervierfacht sich nur. Exponentialfunktionen wie 2^n *verdoppeln* sich bei jedem einzelnen Schritt von n . Außerdem ist die Grenze robust – sie bleibt dieselbe, egal ob man einen realen Rechner, eine Turingmaschine oder ein anderes vernünftiges Maschinenmodell zugrunde legt.

Ein *Entscheidungsproblem* verlangt als Antwort nur Ja oder Nein. Aus „finde die größte Clique“ wird „gibt es eine Clique mit mindestens k Knoten?“ – wer diese Frage für jedes k beantworten kann, kennt auch das Maximum. Formal ist ein Problem einfach die Menge seiner Ja-Eingaben. In P liegen zum Beispiel: Sortieren, kürzeste Wege, minimale Spannbäume, Matchings.

✗ Stolperstein

Vorsicht bei Zahlen in der Eingabe. Eine Zahl K belegt als Eingabe nur etwa $\log K$ Zeichen – ihre Ziffern. Ihr *Wert* ist demgegenüber exponentiell groß. Ein Verfahren, das K -mal iteriert, braucht also exponentiell viele Schritte *gemessen an der Eingabelänge*. Dieser unscheinbare Punkt erklärt später, warum Probleme über Zahlen (SubSet Sum, Partition, Knapsack) schwer sein können, obwohl man dort „nur addiert“.

3 Die Klasse NP: Lösungen prüfen statt finden

Die zweite Klasse entsteht aus einer Alltagsbeobachtung: Es ist etwas völlig anderes, eine Lösung zu *suchen*, als eine vorgelegte Lösung zu *kontrollieren*.

Analogie

Denk an ein Kreuzworträtsel. Es auszufüllen kann einen Abend dauern. Aber wenn dir jemand ein fertig ausgefülltes Gitter zeigt, siehst du in zwei Minuten, ob alles passt – du liest jede Zeile und vergleichst mit den Fragen. Suchen ist teuer, Kontrollieren ist billig. NP ist die Klasse der Probleme, bei denen wenigstens das *Kontrollieren* billig ist.

3.1 Zertifikate und Verifizierer

Die vorgelegte Lösung heißt in der Theorie *Zertifikat*; der Kontrolleur heißt *Verifizierer*.

■ Definition 3.1 Verifizierer und Zertifikat

Ein *Verifizierer* für ein Problem L ist ein Algorithmus A mit zwei Eingaben – der Instanz x und einem Vorschlag c –, für den gilt:

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

Ein solches c heißt *Zertifikat* für x .

Beide Richtungen dieser Gleichwertigkeit tragen Gewicht. Für eine Ja-Instanz *muss es* einen Vorschlag geben, den A gutheißt. Für eine Nein-Instanz darf A sich von *keinem* Vorschlag überzeugen lassen – sonst würde er falsche Ja-Antworten produzieren.

■ Definition 3.2 Die Klasse NP

NP enthält genau die Probleme mit einem *polynomiellen* Verifizierer: Für jede Ja-Instanz x gibt es ein Zertifikat, mit dem der Verifizierer in polynomieller Zeit – gemessen an $|x|$ – akzeptiert.

3.2 Dieselbe Klasse über das Raten

Es gibt eine zweite Beschreibung von NP, die in Beweisen oft bequemer ist. Man stellt sich eine Maschine vor, die an jeder Weggabelung *beide* Wege gleichzeitig weiterverfolgt – eine *nichtdeterministische* Maschine. Sie akzeptiert eine Eingabe, sobald auch nur *einer* ihrer vielen parallelen Rechenwege akzeptiert; bei einer Nein-Instanz scheitern alle Wege.

Der Zusammenhang zur ersten Sicht ist eng: Das Protokoll der Rate-Entscheidungen eines akzeptierenden Weges *ist* ein Zertifikat, und umgekehrt kann die Maschine ein Zertifikat einfach „erraten“ und danach deterministisch prüfen. „Rate die Lösung, prüfe sie“ – nach diesem Muster laufen alle NP-Nachweise über Maschinen. Beide Sichtweisen beschreiben exakt dieselbe Klasse.

3.3 Das Verhältnis von P und NP

▲ Satz 3.3 $P \subseteq NP$

Jedes Problem aus P liegt auch in NP.

Warum: Ein Algorithmus, der das Problem selbst löst, kann als Verifizierer arbeiten, der seinen Vorschlag gar nicht ansieht – er berechnet die Antwort einfach selbst. Schneller prüfen als lösen kann man immer.

Ob umgekehrt $NP \subseteq P$ gilt – ob also alles effizient Kontrollierbare auch effizient lösbar ist – ist die berühmteste offene Frage der Informatik: das P-gegen-NP-Problem, eines der sieben Millennium-Probleme. Die verbreitete Vermutung lautet $P \neq NP$. Der gesamte restliche Kurs ist eine Antwort auf die Frage, wie man *trotz* dieser Ungewissheit belastbare Aussagen über schwere Probleme trifft.

4 Reduktionen: das Vergleichswerkzeug

Wie vergleicht man zwei Probleme, für die man beide keinen schnellen Algorithmus kennt? Man zeigt, dass das eine im anderen *steckt*.

🔍 Analogie

Wer Temperaturen nur in Fahrenheit messen kann, beantwortet trotzdem jede Celsius-Frage: erst umrechnen, dann messen. Die Umrechnung ist billig; die eigentliche Arbeit macht das Fahrenheit-Thermometer. Wenn sich *jede* Frage vom Typ A billig in eine Frage vom Typ B umrechnen lässt, dann ist B mindestens so mächtig – und mindestens so schwer – wie A.

■ Definition 4.1 Polynomielle Reduktion $A \leq_p B$

Eine *Reduktion* von A auf B ist eine in Polynomialzeit berechenbare Übersetzung f , die jede A-Instanz w in eine B-Instanz $f(w)$ verwandelt und dabei die Antwort erhält:

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

Man schreibt $A \leq_p B$.

Die Erhaltung der Antwort umfasst beide Richtungen: Ja wird zu Ja *und* Nein wird zu Nein. Nur dann kann man einen B-Löser als A-Löser weiterverwenden.

★ Intuition

Lies $A \leq_p B$ als „B ist mindestens so schwer wie A“. Denn ein schneller Algorithmus für B ergäbe sofort einen für A (Übersetzung vorschalten). Die Härte wandert *in Pfeilrichtung*: Ist A hart, kann B nicht leicht sein.

✗ Stolperstein

Die Richtung entscheidet über alles. Willst du zeigen, dass ein neues Problem Y hart ist, dann übersetzt du ein *bekannt hartes* Problem X in Y : $X \leq_p Y$. Merksatz: *vom Bekannten zum Neuen*. Die umgekehrte Richtung $Y \leq_p X$ würde nur belegen, dass Y höchstens so schwer wie X ist – eine wertlose Aussage über Y . Dieser Richtungsfehler ist der häufigste Fehler in Klausuren.

▲ Satz 4.2 Reduktionen verketten sich

Aus $A \leq_p B$ und $B \leq_p C$ folgt $A \leq_p C$.

Warum: Die Hintereinanderausführung zweier polynomieller Übersetzungen ist wieder polynomiell – die erste Übersetzung liefert eine polynomiell große Zwischeninstanz, auf die die zweite polynomiell viel Zeit verwendet.

Diese Verkettung ist der Grund, warum wir gleich eine ganze *Kette* von Problemen aufbauen können: Die Härte des ersten Glieds erreicht über die Pfeile jedes spätere Glied.

5 NP-Vollständigkeit: die härtesten Probleme in NP

■ Definition 5.1 NP-schwer

Ein Problem H heißt *NP-schwer*, wenn sich *jedes* Problem aus NP auf H reduzieren lässt: $L \leq_p H$ für alle $L \in \text{NP}$.

■ Definition 5.2 NP-vollständig

Ein Problem heißt *NP-vollständig*, wenn es NP-schwer ist *und* selbst in NP liegt.

★ Intuition

NP-schwer ist die *Mindesthärte* – so schwer wie alles in NP zusammen. Die Mitgliedschaft in NP ist die *Höchsthärte* – nicht schwerer als prüfbar. NP-vollständige Probleme erfüllen beides zugleich: Sie markieren exakt die Spitze innerhalb von NP. Es gibt übrigens NP-schwere Probleme, die gar nicht in NP liegen – das Halteproblem am Ende des Katalogs ist so eines.

▲ Satz 5.3 Alles hängt an einem Faden

Sei H NP-vollständig. Dann gilt: $H \in P$ genau dann, wenn $P = NP$.

Warum: Hätte H einen polynomiellen Algorithmus, könnte jedes $L \in NP$ zuerst nach H übersetzt und dann dort gelöst werden – ganz NP fiel in P . Die Gegenrichtung ist unmittelbar, da $H \in NP$.

Ein einziges NP-vollständiges Problem mit schnellem Algorithmus würde also die gesamte Klasse kollabieren lassen. Solange das niemandem gelingt, ist „NP-vollständig“ das stärkste verfügbare Indiz für „praktisch nicht exakt lösbar“.

5.1 Cook–Levin: das erste harte Problem

Die Definition von NP-schwer verlangt Reduktionen von *allen* Problemen in NP – unendlich vielen. Für das allererste harte Problem muss man diese Hürde direkt nehmen. Cook und Levin haben es für SAT getan, das Erfüllbarkeitsproblem der Aussagenlogik.

▲ Satz 5.4 Satz von Cook und Levin

SAT ist NP-vollständig.

Die Idee: Sei L ein beliebiges Problem aus NP, akzeptiert von einer ratenden Maschine M in Polynomialzeit. Die gesamte Arbeit von M auf einer Eingabe u – Band, Kopf, Zustand, zu jedem Zeitpunkt – lässt sich durch endlich viele Wahrheitswerte beschreiben: „zum Zeitpunkt t steht die Maschine im Zustand q “, „zum Zeitpunkt t trägt Feld i das Zeichen a “. Aus diesen Aussagen baut man eine Formel, deren Klauseln erzwingen, dass der Start korrekt ist, jeder Schritt einer echten Maschinenregel folgt und das Ende akzeptierend ist. Eine erfüllende Belegung dieser Formel ist dann nichts anderes als ein *Protokoll eines akzeptierenden Rechenwegs*. Die Formel ist erfüllbar genau dann, wenn M die Eingabe akzeptiert – und sie ist in Polynomialzeit konstruierbar. Da L beliebig war, reduziert ganz NP auf SAT.

5.2 Der Dominoeffekt

📖 Idee: Das Vererbungsprinzip

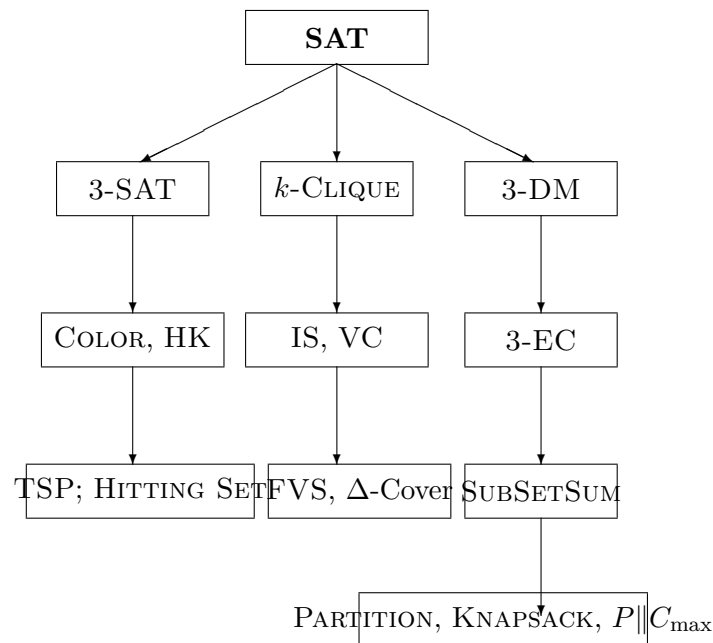
Sobald *ein* NP-vollständiges Problem existiert, wird der Nachweis für jedes weitere Problem Y dramatisch einfacher. Es genügen zwei Schritte:

1. Zeige $X \leq_p Y$ für irgendein *bereits* NP-vollständiges X .
2. Zeige $Y \in NP$ (Zertifikat und Verifizierer angeben).

Dann reduziert jedes $L \in NP$ über die Kette $L \leq_p X \leq_p Y$ auch auf Y – also ist Y NP-schwer, und mit $Y \in NP$ NP-vollständig.

Genau nach diesem Prinzip ist der folgende Katalog organisiert: SAT steht am Anfang, und jedes weitere Problem wird über eine Reduktion an ein schon behandeltes angeschlossen. Die

Landkarte zeigt, wer an wem hängt.



Teil B – Der Katalog aller Probleme

Jetzt zum Herzstück. Jedes Problem bekommt einen eigenen Abschnitt mit denselben vier Stationen: **Definition und Beispiel – warum in NP – woher die Härte – wohin die Härte weiterfließt**. Die Reihenfolge folgt der Landkarte, sodass jede eingehende Reduktion von einem Problem kommt, das du bereits kennst.

6 SAT

Am Anfang steht die Logik. SAT fragt, ob eine logische Formel überhaupt wahr werden *kann*.

■ Definition 6.1 SAT

Eine Formel in *konjunktiver Normalform* (KNF) ist ein UND über *Klauseln*; jede Klausel ist ein ODER über *Literale*, und ein Literal ist eine Variable x_j oder ihre Negation $\neg x_j$. Gefragt ist: Existiert eine Belegung der Variablen mit wahr und falsch, unter der *jede* Klausel wahr wird?

● Beispiel 6.2 Eine Formel entscheiden

Betrachte $\alpha = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. Wähle $x_2 = \text{wahr}$. Damit sind die ersten beiden Klauseln erfüllt. Die dritte verlangt nun $x_3 = \text{wahr}$ (denn $\neg x_2$ ist falsch). Mit $x_2 = x_3 = \text{wahr}$ und beliebigem x_1 sind alle drei Klauseln wahr – die Formel ist erfüllbar.

Warum SAT in NP liegt

Wir gehen den Nachweis in Ruhe durch.

Was das Zertifikat ist. Der natürliche Beleg für „erfüllbar“ ist eine erfüllende Belegung: für jede der v Variablen ein Wahrheitswert.

Wie geprüft wird. Der Verifizierer erhält die Formel und die Belegung. Er ersetzt jede Variable durch ihren Wert und geht dann Klausel für Klausel durch: Eine Klausel gilt als erfüllt, sobald mindestens eines ihrer Literale wahr ist. Findet er eine unerfüllte Klausel, lehnt er ab; andernfalls, wenn alle Klauseln bestanden haben, akzeptiert er.

Warum das in beide Richtungen stimmt. Wenn die Formel erfüllbar ist, existiert eine Belegung, die jede Klausel wahr macht – reicht man genau diese ein, akzeptiert der Verifizierer. Wenn die Formel dagegen unerfüllbar ist, lässt *jede* denkbare Belegung mindestens eine Klausel falsch, und der Verifizierer lehnt jeden Vorschlag ab. Er sagt also nie fälschlich Ja und nie fälschlich Nein.

Wie lange es dauert. Einsetzen und Auswerten berühren jedes Literal der Formel einmal – lineare Zeit. Der Verifizierer ist polynomiell, folglich $\text{SAT} \in \text{NP}$.

Woher die Härte kommt

Idee: Cook–Levin: Rechenwege werden Formeln

SAT bezieht seine Härte nicht aus einer Reduktion von einem anderen Katalogproblem – es ist der Ursprung. Der Satz von Cook und Levin (Teil A) zeigt direkt: Die Rechnung *jeder* ratenden Polynomialzeit-Maschine lässt sich als KNF-Formel kodieren, sodass erfüllende Belegungen exakt den akzeptierenden Rechenwegen entsprechen. Jedes Problem aus NP reduziert dadurch auf SAT.

Wohin die Härte weiterfließt

Vier Reduktionen gehen von SAT aus; jede wird beim jeweiligen Zielproblem vollständig ausgeführt, hier steht ihre Grundidee.

Idee: $\text{SAT} \leq_p \text{3-SAT}$: Ketten aus Hilfsvariablen

Lange Klauseln werden in Ketten kurzer Klauseln zerlegt, die über frische Hilfsvariablen aneinandergekoppelt sind. Die Kopplung ist so gebaut, dass die Kette genau dann komplett erfüllbar ist, wenn die Originalklausel ein wahres Literal enthält.

Idee: $\text{SAT} \leq_p k\text{-Clique}$: Literale als Knoten

Jedes Literalvorkommen wird ein Knoten; verbunden werden nur Knoten aus verschiedenen Klauseln, deren Literale einander nicht widersprechen. Eine Clique der Größe „Klauselzahl“ pickt dann aus jeder Klausel ein Literal, und alle gepickten vertragen sich – eine erfüllende Belegung.

Idee: $\text{SAT} \leq_p \text{3-DM}$: Zahnräder und Aufräumtripel

Pro Variable entsteht ein ringförmiges „Zahnrad“ aus Tripeln, das sich nur auf zwei Arten abdecken lässt – wahr oder falsch. Klausel-Tripel docken an wahre Literale an, und Aufräumtripel („Garbage Collection“) entsorgen die unbenutzten Literale, damit die Zählung exakt aufgeht.

Idee: $\text{SAT} \leq_p \text{Halteproblem}$: Suchen bis zum Erfolg

Zu jeder Formel lässt sich ein Programm bauen, das stur alle Belegungen durchprobiert und exakt dann stoppt, wenn es eine erfüllende findet. Ob dieses Programm jemals hält, ist damit dieselbe Frage wie die Erfüllbarkeit.

7 3-SAT

3-SAT ist SAT mit einer Formatvorschrift. Diese kleine Einschränkung macht es zum beliebtesten Startpunkt für weitere Reduktionen: Klauseln mit höchstens drei Literalen lassen sich viel leichter in Gadgets verbauen als beliebig lange.

■ Definition 7.1 3-SAT

Gegeben eine KNF-Formel, in der jede Klausel *höchstens drei* Literale enthält. Gefragt ist: Ist die Formel erfüllbar?

● Beispiel 7.2 Eine 3-SAT-Formel

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)$: Mit $x_1 = \text{wahr}$ und $x_2 = \text{wahr}$ sind beide Klauseln wahr, egal wie x_3 steht. Erfüllbar.

Warum 3-SAT in NP liegt

Auch wenn der Nachweis dem für SAT ähnelt, führen wir ihn hier für sich durch.

Was das Zertifikat ist. Eine Belegung der Variablen – ein Wahrheitswert pro Variable.

Wie geprüft wird. Der Verifizierer setzt die Belegung ein und kontrolliert die Klauseln nacheinander. Da jede Klausel höchstens drei Literale hat, ist der Test einer einzelnen Klausel eine Sache konstanter Zeit; insgesamt läuft er einmal über die Formel. Findet er eine falsche Klausel, lehnt er ab, sonst akzeptiert er.

Warum das in beide Richtungen stimmt. Für eine erfüllbare Formel führt die erfüllende Belegung zur Annahme. Für eine unerfüllbare bleibt unter jeder Belegung eine Klausel falsch, und jeder Vorschlag wird abgelehnt.

Wie lange es dauert. Ein Durchgang über die Formel, linear. Also $3\text{-SAT} \in \text{NP}$.

Woher die Härte kommt

Idee: $\text{SAT} \leq_p \text{3-SAT}$: die Kettenkonstruktion im Detail

Eine Klausel mit vielen Literalen, etwa $(y_1 \vee y_2 \vee y_3 \vee y_4 \vee y_5)$, wird durch eine Kette ersetzt:

$$(y_1 \vee y_2 \vee h_1) \wedge (\neg h_1 \vee y_3 \vee h_2) \wedge (\neg h_2 \vee y_4 \vee y_5),$$

mit neuen Hilfsvariablen h_1, h_2 , die nur in dieser Kette vorkommen. Kurze Klauseln bleiben unverändert.

Warum die Kette die Erfüllbarkeit bewahrt, sieht man an beiden Richtungen. Ist in der Originalklausel ein y_i wahr, dann setzt man alle Hilfsvariablen *vor* der Position von y_i auf wahr und alle danach auf falsch: Jedes Kettenglied vor y_i wird durch sein h -Literal wahr, das Glied mit y_i durch y_i selbst, und jedes Glied danach durch sein negiertes h -Literal. Ist dagegen *kein* y_i wahr, dann zwingt das erste Glied h_1 auf wahr, das zweite daraufhin h_2 auf wahr, und so weiter – bis das letzte Glied nur noch falsche Literale enthält. Die Kette ist dann unerfüllbar, genau wie die Originalklausel. Die Umformung vergrößert die Formel nur linear und ist damit polynomiell. Zusammen mit $3\text{-SAT} \in \text{NP}$ folgt: 3-SAT ist NP-vollständig.

Wohin die Härte weiterfließt

Idee: $3\text{-SAT} \leq_p k\text{-Color}$: Farben als Wahrheitswerte

Ein Gadget aus Variablen-, Klausel- und Hilfsknoten erzwingt, dass jede Variable eine von zwei Rollen-Farben („wahr“/„falsch“) trägt und jeder Klauselknoten nur dann färbbar ist, wenn seine Klausel ein wahres Literal hat. Details beim Färbungsproblem.

Idee: $3\text{-SAT}' \leq_p \text{Hamiltonkreis}$: Touren als Belegungen

Spezielle Bausteine übersetzen „Variable wahr oder falsch“ in „Baustein links- oder rechts herum durchlaufen“ und blockieren jede Tour, die eine Klausel komplett falsch ließe. Details beim Hamiltonkreis.

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

Die Literale bilden das Universum. Paar-Mengen $\{x_i, \bar{x}_i\}$ erzwingen eine Belegung, Klausel-Mengen verlangen ein getroffenes wahres Literal. Details beim Hitting Set.

Idee: $3\text{-SAT} \leq_p \text{SubSet Sum (streng)}$: Dezimalstellen

Eine besonders sparsame Kodierung über Ziffern zur Basis 10 erzeugt nur $O(m)$ Zahlen – wichtig für die unteren Schranken. Details beim SubSet Sum.

8 k -Clique

Das Problem aus der Einleitung bekommt jetzt seine vollständige Behandlung.

■ Definition 8.1 k -Clique

Gegeben ein ungerichteter Graph $G = (V, E)$ und eine Zahl k . Eine *Clique* ist eine Menge von Knoten, zwischen denen *sämtliche* Kanten vorhanden sind. Gefragt ist: Enthält G eine Clique mit mindestens k Knoten?

● Beispiel 8.2 Clique ja, Clique nein

Nimm die Knoten p, q, r, s mit den Kanten $\{p, q\}$, $\{q, r\}$, $\{p, r\}$ und $\{r, s\}$. Die Menge $\{p, q, r\}$ ist eine Clique der Größe 3 – alle drei möglichen Kanten sind da. Die Menge $\{q, r, s\}$ ist keine Clique, denn q und s sind nicht verbunden.

Warum k -Clique in NP liegt

Was das Zertifikat ist. Der Beleg für eine Ja-Antwort ist die Knotenmenge C selbst.

Wie geprüft wird. Der Verifizierer nimmt sich alle Paare von Knoten aus C vor. Für jedes Paar schlägt er nach, ob die Kante im Graphen existiert. Fehlt auch nur eine, lehnt er ab. Bestehen alle Paare den Test, zählt er noch $|C|$ und vergleicht mit k ; ist $|C| \geq k$, akzeptiert er.

Warum das in beide Richtungen stimmt. Besitzt der Graph eine Clique mit k Knoten, dann besteht genau diese Menge alle Paartests und die Größenprüfung – der Verifizierer akzeptiert sie. Besitzt er keine, dann hat jede eingereichte Menge mit $\geq k$ Knoten irgendwo ein unverbundenen Paar, und der Verifizierer weist sie zurück. Falsche Ja-Antworten sind ausgeschlossen.

Wie lange es dauert. Bei $|C| \leq |V|$ Knoten sind es höchstens $O(|V|^2)$ Paare, jeder Nachschlag in einer Adjazenzmatrix kostet konstante Zeit; die Größenprüfung ist billig. Insgesamt $O(|V|^2)$ – polynomiell, also k -CLIQUE $\in$ NP.

Woher die Härte kommt

☞ Idee: SAT $\leq_p k$ -Clique: ein Knoten je Literalvorkommen

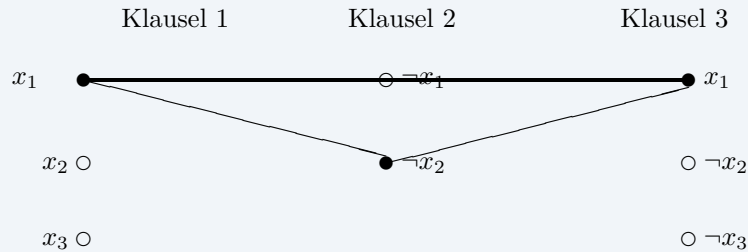
Gegeben eine KNF F mit Klauseln $F_1, \dots, F_m$. Der Graph entsteht so: Für das j -te Literal der Klausel i gibt es einen Knoten $[i, j]$. Eine Kante verbindet $[i, j]$ und $[i', j']$ genau dann, wenn erstens $i \neq i'$ (verschiedene Klauseln) und zweitens die beiden Literale *nicht* komplementär sind (nicht x und $\neg x$ derselben Variablen). Gesucht wird eine Clique der Größe $k = m$.

Weshalb die Übersetzung funktioniert: Innerhalb einer Klausel gibt es keine Kanten, also kann eine Clique aus jeder Klausel höchstens einen Knoten enthalten – eine m -Clique enthält demnach aus jeder Klausel *genau* einen. Ihre Literale vertragen sich paarweise (sonst fehlte eine Kante), man kann sie also alle zugleich wahr machen; das erfüllt jede Klausel. Umgekehrt: Erfüllt eine

Belegung die Formel, wählt man aus jeder Klausel ein wahres Literal – die zugehörigen m Knoten stammen aus verschiedenen Klauseln und widersprechen sich nicht, sind also paarweise verbunden: eine m -Clique. Der Graphaufbau berührt jedes Literalpaar einmal und ist polynomiell.

● Beispiel 8.3 Die Übersetzung an einer Formel

$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$ mit $m = 3$ Klauseln, gesucht also eine 3-Clique. Der Graph hat acht Knoten in drei Spalten. Fett hervorgehoben: die Clique aus x_1 (Klausel 1), $\neg x_2$ (Klausel 2) und x_1 (Klausel 3).



Die fette 3-Clique liefert die Belegung $x_1 = \text{wahr}$, $x_2 = \text{falsch}$ (x_3 frei) – und die erfüllt tatsächlich alle drei Klauseln. Dünne Kanten zwischen anderen verträglichen Paaren sind zur Übersicht weggelassen.

Wohin die Härte weiterfließt

🔍 Idee: $\text{Clique} \leq_p \text{Independent Set}$: Kanten umklappen

Man ersetzt den Graphen durch sein *Komplement* – vorhandene Kanten verschwinden, fehlende erscheinen – und behält k . Aus „alle verbunden“ wird „keiner verbunden“. Details beim Independent Set.

🔍 Idee: $\text{Clique} \leq_p \text{Vertex Cover}$: Komplement plus Restmenge

Ebenfalls über den Komplementgraphen, aber mit Schranke $n - k$: Das *Komplement der Cliquenmenge* wird dort zum Vertex Cover. Details beim Vertex Cover.

🔍 Idee: $\text{Clique} \leq_p \text{Varianten}$: Dummy- und Universalknoten

Für *CLIQUE-NOMEMBER* (Clique ohne einen bestimmten Knoten) fügt man einen *isolierten* Knoten hinzu – er kann ohnehin in keiner Clique liegen. Für *CLIQUE-UNIVERSAL* (Instanz muss einen mit allen verbundenen Knoten haben) fügt man einen *universellen* Knoten hinzu und erhöht k um eins – er vergrößert jede Clique um genau ein Mitglied.

9 Independent Set

Die Clique verlangt maximale Nähe – das Independent Set verlangt maximale Distanz.

■ Definition 9.1 Independent Set (IS)

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein *Independent Set* ist eine Knotenmenge, innerhalb derer *keine einzige* Kante verläuft. Gefragt ist: Enthält G ein Independent Set mit mindestens k Knoten?

● Beispiel 9.2 Unabhängige Knoten finden

Ein Pfad $p - q - r - s$ (drei Kanten). Die Menge $\{p, r\}$ ist unabhängig – p und r sind nicht direkt verbunden. Auch $\{p, s\}$ und $\{q, s\}$ sind unabhängig. $\{q, r\}$ ist es nicht, denn die Kante $\{q, r\}$ verläuft mitten durch die Menge.

Warum Independent Set in NP liegt

Was das Zertifikat ist. Die Knotenmenge I .

Wie geprüft wird. Der Verifizierer betrachtet jedes Knotenpaar aus I und schlägt nach, ob es im Graphen verbunden ist. Findet er auch nur eine Kante innerhalb von I , lehnt er ab. Andernfalls zählt er $|I|$; bei $|I| \geq k$ akzeptiert er.

Warum das in beide Richtungen stimmt. Existiert ein Independent Set der verlangten Größe, dann übersteht genau diese Menge alle Paartests – Annahme. Existiert keines, dann trägt jede eingereichte Menge dieser Größe mindestens eine innere Kante – Ablehnung. Der Verifizierer irrt sich in keiner Richtung.

Wie lange es dauert. Höchstens $O(|V|^2)$ Paare mit konstantem Nachschlag, dazu die Größenprüfung: insgesamt $O(|V|^2)$. Also $\text{IS} \in \text{NP}$.

Woher die Härte kommt

🔍 Idee: $\text{Clique} \leq_p \text{IS}$: der Komplementgraph

Aus der Clique-Instanz (G, k) wird die IS-Instanz $(\bar{G}, k)$, wobei $\bar{G}$ dieselben Knoten hat und *genau die* Kanten, die in G fehlen.

Die Übersetzung ist ein reiner Perspektivwechsel: Eine Menge C ist Clique in G genau dann, wenn zwischen ihren Knoten in G alle Kanten liegen – also in $\bar{G}$ *gar keine*. Das ist wortwörtlich die Definition eines Independent Set in $\bar{G}$. Ja-Instanzen entsprechen sich eins zu eins, ebenso Nein-Instanzen. Das Komplement zu bilden kostet $O(|V|^2)$ und ist polynomiell. Da CLIQUE NP-vollständig ist und $\text{IS} \in \text{NP}$, ist IS NP-vollständig.

Wohin die Härte weiterfließt

Im Kursmaterial tritt INDEPENDENT SET vor allem als Zwischenstation der Dualität auf: Eine Menge ist genau dann unabhängig, wenn ihr Komplement alle Kanten berührt – also ein Vertex Cover ist. Diese Beziehung nutzt die nächste Reduktion. Eine eigene ausgehende Reduktion auf ein weiteres Problem führt der Kurs nicht ein.

10 Vertex Cover

■ Definition 10.1 Vertex Cover (VC)

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein *Vertex Cover* ist eine Knotenmenge, die von *jeder* Kante mindestens einen Endpunkt enthält. Gefragt ist: Besitzt G ein Vertex Cover mit höchstens k Knoten?

🗨 Analogie

Stell dir die Kanten als Flure eines Museums vor und die Knoten als Ecken, an denen Wächter stehen können. Ein Vertex Cover ist eine Wächteraufstellung, bei der jeder Flur von mindestens einem Ende aus eingesehen wird – und gefragt ist, ob k Wächter reichen.

● Beispiel 10.2 Wächter platzieren

Wieder der Pfad $p - q - r - s$. Die Aufstellung $\{q, r\}$ deckt alle drei Flure: $\{p, q\}$ über q , $\{q, r\}$ über beide, $\{r, s\}$ über r . Mit nur einem Wächter geht es nicht – egal wo er steht, bleibt ein Flur an beiden Enden unbewacht.

Warum Vertex Cover in NP liegt

Was das Zertifikat ist. Die Wächtermenge C .

Wie geprüft wird. Der Verifizierer geht die Kantenliste durch. Bei jeder Kante $\{u, v\}$ fragt er: Liegt u in C oder v in C ? Sobald eine Kante beide Male Nein liefert, lehnt er ab. Am Ende zählt er $|C|$; bei $|C| \leq k$ akzeptiert er.

Warum das in beide Richtungen stimmt. Ein echtes Vertex Cover der Größe $\leq k$ besteht jeden Kantentest und die Zählung – es wird akzeptiert. Gibt es keines, dann lässt jede Menge mit $\leq k$ Knoten mindestens eine Kante unbedeckt, und der Test schlägt fehl. Beide Richtungen halten.

Wie lange es dauert. Ein Durchlauf über alle Kanten mit konstanten Nachschlägen: $O(|E|)$, dazu $O(|V|)$ fürs Zählen. Also VC $\in$ NP.

Woher die Härte kommt

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

Aus (G, k) mit n Knoten wird $(\bar{G}, n-k)$: derselbe Komplementgraph wie bei der IS-Reduktion, aber als Schranke die Restgröße $n-k$.

Der Weg führt über zwei Beobachtungen. Erstens: C ist Clique in G genau dann, wenn C Independent Set in $\bar{G}$ ist (Kanten umgeklappt). Zweitens: I ist Independent Set genau dann, wenn $V \setminus I$ Vertex Cover ist – denn „keine Kante verläuft innerhalb von I “ heißt „jede Kante hat ein Ende außerhalb von I “. Setzt man beides zusammen: G hat eine k -Clique genau dann, wenn $\bar{G}$ ein Vertex Cover der Größe $n-k$ hat. Beide Richtungen der Antwortgleichheit stecken in diesen beiden Äquivalenzen. Der Aufbau ist mit $O(|V|^2)$ polynomiell, also ist VC NP-vollständig.

☆ Aha

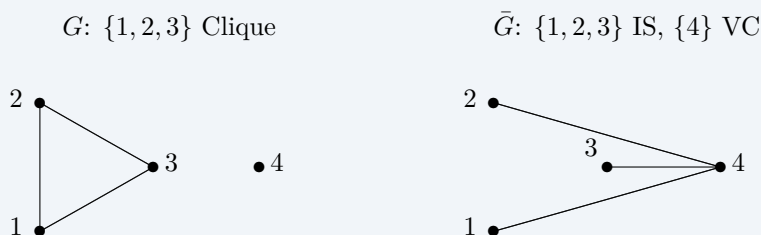
Damit schließt sich ein Dreieck aus drei Sichtweisen auf dieselbe Struktur:

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

Die drei Probleme bleiben eigenständig – aber wer eine der drei Übersetzungen verstanden hat, kann die anderen beiden selbst herleiten.

● Beispiel 10.3 Das Dreieck in einem Bild

Links G mit der Clique $\{1, 2, 3\}$; rechts das Komplement $\bar{G}$, in dem $\{1, 2, 3\}$ unabhängig ist und der übrig gebliebene Knoten 4 alle Kanten abdeckt.



Wohin die Härte weiterfließt

☞ Idee: $\text{VC} \leq_p \text{FVS}$: Kanten zu Mini-Kreisen

Jede ungerichtete Kante wird durch ein Paar entgegengesetzter Pfeile ersetzt – ein gerichteter 2-Kreis, den nur seine beiden Endknoten zerstören können. Details beim Feedback Vertex Set.

Idee: $VC \leq_p \Delta\text{-Cover}$: Kanten zu Dreiecken

Jede Kante bekommt einen neuen Spitzknoten und wird so zu einem Dreieck aufgepumpt; Kanten decken heißt dann Dreiecke treffen. Details beim $\Delta\text{-Cover}$.

11 Feedback Vertex Set

Zum ersten Mal wechseln wir zu *gerichteten* Graphen – die Kanten sind jetzt Pfeile.

■ Definition 11.1 Feedback Vertex Set (FVS)

Gegeben ein gerichteter Graph $G = (V, E)$ und eine Zahl k . Gesucht ist eine Menge X von höchstens k Knoten, deren Entfernung *alle gerichteten Kreise* zerstört – sodass der Restgraph kreisfrei ist. Gefragt: Existiert ein solches X ?

● Beispiel 11.2 Kreise brechen

Ein Graph enthalte die Pfeile $a \rightarrow b$, $b \rightarrow a$, $b \rightarrow c$ und $c \rightarrow b$. Es gibt zwei Kreise: $a \rightarrow b \rightarrow a$ und $b \rightarrow c \rightarrow b$. Beide laufen durch b – entfernt man b , ist der Rest kreisfrei. $X = \{b\}$ ist ein FVS der Größe 1.

Warum Feedback Vertex Set in NP liegt

Was das Zertifikat ist. Die zu entfernende Knotenmenge X .

Wie geprüft wird. Zwei Kontrollen. Erstens zählt der Verifizierer $|X|$ und vergleicht mit k . Zweitens muss er entscheiden, ob der Graph ohne X kreisfrei ist – dafür steht ihm ein klassischer Polynomialzeit-Algorithmus zur Verfügung: die *topologische Sortierung*. Sie ordnet die Knoten so an, dass alle Pfeile vorwärts zeigen, und gelingt *genau dann*, wenn kein Kreis existiert. Gelingt die Sortierung von $G \setminus X$ und stimmt die Größe, akzeptiert der Verifizierer.

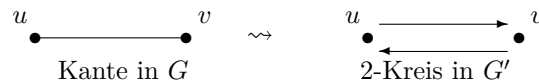
Warum das in beide Richtungen stimmt. Ist X ein echtes FVS der Größe $\leq k$, dann ist der Restgraph kreisfrei, die Sortierung klappt, und der Vorschlag wird angenommen. Gibt es kein solches FVS, dann hinterlässt jede Menge dieser Größe einen Kreis, die Sortierung scheitert, der Vorschlag wird abgelehnt.

Wie lange es dauert. Topologische Sortierung läuft in $O(|V| + |E|)$, die Zählung in $O(|V|)$. Also $FVS \in NP$.

Woher die Härte kommt

🔑 Idee: $VC \leq_p FVS$: antiparallele Bögen

Aus einer ungerichteten VC-Instanz (G, k) wird eine gerichtete FVS-Instanz (G', k) : Jede Kante $\{u, v\}$ von G wird durch die *beiden* Pfeile $u \rightarrow v$ und $v \rightarrow u$ ersetzt. Sonst ändert sich nichts, auch k bleibt.



Der Kern des Arguments: Jedes Pfeilpaar bildet einen Kreis der Länge zwei, und *andere* Kreise gibt es in G' nicht in neuer Qualität – jeder Kreis durchläuft solche Pfeilpaare. Einen 2-Kreis $u \rightarrow v \rightarrow u$ zerstört man nur, indem man u oder v entfernt. „Alle Kreise zerstören“ übersetzt sich damit in „von jeder Original-Kante einen Endpunkt entfernen“ – das ist wörtlich die Vertex-Cover-Bedingung. Konkret: Ist C ein Vertex Cover von G , dann bleibt nach Entfernen von C kein Pfeilpaar vollständig übrig, also kein Kreis – C ist ein FVS. Ist umgekehrt X ein FVS von G' , dann muss X jeden 2-Kreis anfassen, deckt also jede Kante von G ab – X ist ein Vertex Cover. Der Umbau kostet $O(|E|)$. FVS ist damit NP-vollständig.

Wohin die Härte weiterfließt

FEEDBACK VERTEX SET ist im Kurs ein Endpunkt – es dient nicht als Quelle einer weiteren Reduktion.

12 Δ -Cover

Noch einmal Vertex Cover als Quelle, diesmal mit einem anderen Gadget: Statt um Kanten geht es um Dreiecke.

■ Definition 12.1 Δ -Cover

Gegeben ein ungerichteter Graph $G = (V, E)$ und eine Zahl k . Ein Δ -Cover ist eine Knotenmenge, die jedes *Dreieck* des Graphen (jede 3-Clique) in mindestens einem Knoten trifft. Gefragt: Gibt es eines mit höchstens k Knoten?

● Beispiel 12.2 Dreiecke treffen

Zwei Dreiecke $\{a, b, c\}$ und $\{c, d, e\}$, die sich den Knoten c teilen. Die Menge $\{c\}$ trifft beide – ein Δ -Cover der Größe 1. Die Menge $\{a, d\}$ trifft ebenfalls beide, braucht aber zwei Knoten.

Warum Δ -Cover in NP liegt

Was das Zertifikat ist. Die Knotenmenge C_Δ .

Wie geprüft wird. Der Verifizierer zählt zunächst $|C_\Delta|$ und vergleicht mit k . Dann sucht er systematisch alle Dreiecke: Er läuft über sämtliche Knotentripel, prüft für jedes, ob alle drei Verbindungskanten existieren, und wenn ja, ob wenigstens einer der drei Knoten in C_Δ liegt. Ein ungetroffenes Dreieck führt zur Ablehnung; sonst wird akzeptiert.

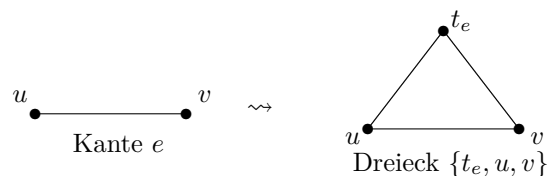
Warum das in beide Richtungen stimmt. Ein echtes Δ -Cover der verlangten Größe übersteht die Suche – kein Dreieck bleibt ungetroffen – und wird angenommen. Existiert keines, findet die Suche bei jedem Vorschlag dieser Größe ein verfehltes Dreieck und lehnt ab.

Wie lange es dauert. Es gibt $O(|V|^3)$ Tripel; Kanten- und Mitgliedschaftstests pro Tripel sind konstant bis linear. Insgesamt höchstens $O(|V|^4)$ – polynomiell, also Δ -COVER $\in$ NP.

Woher die Härte kommt

Idee: VC $\leq_p$ Δ -Cover: Kanten aufpumpen

Zur VC-Instanz (G, k) baut man G' : Für jede Kante $e = \{u, v\}$ kommt ein frischer Knoten t_e hinzu, verbunden mit u und mit v . Jede Kante wird so zur Spitze eines eigenen Dreiecks $\{t_e, u, v\}$. Die Schranke k bleibt.



Die Hinrichtung ist unkompliziert: Deckt C jede Kante von G , dann enthält C von jedem Gadget-Dreieck $\{t_e, u, v\}$ den Knoten u oder v – das Dreieck ist getroffen.

Stolperstein

Hier liegt eine Falle, die echte Punkte gekostet hat. G' enthält neben den Gadget-Dreiecken auch alle Dreiecke, die schon in G standen – denn die Originalkanten bleiben ja erhalten. Ein Beweis, der behauptet, jedes Dreieck von G' habe die Form $\{t_e, u, v\}$, ist falsch. Die geerbten Dreiecke muss man eigens behandeln: Ihre drei Seiten sind Originalkanten, jede davon wird vom Vertex Cover gedeckt, also liegt einer ihrer Eckknoten in C – auch sie sind getroffen. Erst mit diesem Fall ist die Hinrichtung vollständig.

Für die Rückrichtung nimmt man ein Δ -Cover C_Δ der Größe $\leq k$ und säubert es: Enthält es einen Gadget-Knoten t_e , ersetzt man ihn durch einen der beiden Endpunkte von e – das einzige Dreieck durch t_e ist sein eigenes, und das bleibt getroffen; die Menge wächst dabei nicht. Danach liegt C_Δ ganz in V , und weil es jedes Gadget-Dreieck $\{t_e, u, v\}$ treffen muss, enthält es u oder v – von jeder Kante e einen Endpunkt. Das ist ein Vertex Cover. Der Aufbau kostet $O(|E|)$; Δ -COVER ist NP-vollständig.

Wohin die Härte weiterfließt

Auch das Δ -Cover ist im Kurs ein Endpunkt ohne ausgehende Reduktion.

13 k -Color

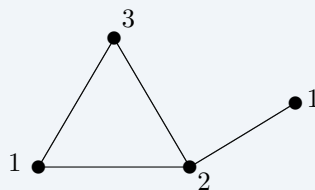
Zurück zur 3-SAT-Linie der Landkarte. Das Färbungsproblem ist ein Klassiker mit anschaulicher Deutung.

■ Definition 13.1 k -Color

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Eine gültige k -Färbung weist jedem Knoten eine von k Farben zu, wobei benachbarte Knoten nie dieselbe Farbe tragen dürfen. Gefragt: Lässt sich G so färben?

Die klassische Deutung: Länder einer Landkarte färben, sodass Nachbarländer verschieden aussehen. Schon mit drei Farben ist die Frage NP-vollständig.

● Beispiel 13.2 Drei Farben im Einsatz



Die Zahlen sind Farben. Das Dreieck links verbraucht zwingend drei verschiedene Farben. Der rechte Knoten hat nur einen Nachbarn (Farbe 2) und darf deshalb Farbe 1 wiederverwenden.

Warum k -Color in NP liegt

Was das Zertifikat ist. Die Farbzuzuweisung – pro Knoten eine Zahl zwischen 1 und k .

Wie geprüft wird. Der Verifizierer wandert über die Kantenliste. Für jede Kante vergleicht er die Farben ihrer beiden Endpunkte; sind sie irgendwo gleich, lehnt er ab. Übersteht die Färbung alle Kanten, akzeptiert er.

Warum das in beide Richtungen stimmt. Eine gültige Färbung passiert jeden Vergleich und wird angenommen. Ist der Graph nicht k -färbbar, enthält jede eingereichte Zuweisung mindestens eine gleichfarbige Kante – der Vergleich schlägt dort fehl, Ablehnung.

Wie lange es dauert. Ein Lauf über $|E|$ Kanten mit konstantem Vergleich: $O(|E|)$. Also k -COLOR $\in$ NP.

Woher die Härte kommt

🔑 Idee: 3-SAT $\leq_p$ k -Color: Färben heißt Belegen

Zur Formel mit n Variablen und m Klauseln baut man einen Graphen mit drei Sorten von Knoten: für jede Variable ein Paar $x_i, \bar{x}_i$ und ein Hilfsknoten v_i ; für jede Klausel ein Knoten F_j ; dazu ein zentraler Knoten z . Die Hilfsknoten $v_1, \dots, v_n$ und z sind untereinander komplett verbunden – sie bilden eine Clique und schlucken dadurch $n + 1$ paarweise verschiedene Farben. Jedes Paar $x_i, \bar{x}_i$ ist verbunden und zusätzlich so mit den Hilfsknoten verdrahtet, dass für die beiden nur noch zwei Farben in Frage kommen – eine spielt „wahr“, die andere „falsch“. Die Färbung der Literalknoten *ist* damit eine Belegung. Jeder Klauselknoten F_j schließlich ist genau mit den Literalknoten verbunden, die *nicht* in seiner Klausel vorkommen; dadurch findet er genau dann noch eine erlaubte Farbe, wenn eines seiner eigenen Literale die „wahr“-Farbe trägt. Insgesamt: Der Graph ist mit $n + 1$ Farben färbbar genau dann, wenn die Formel erfüllbar ist.

Wohin die Härte weiterfließt

Das Färbungsproblem gibt seine Härte im Kurs nicht weiter; es kehrt aber bei den unteren Schranken in Teil C zurück, wo seine besonders knappe $O(n + m)$ -Konstruktion den Ausschlag gibt.

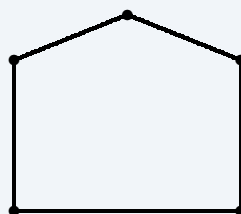
14 Hamiltonkreis

■ Definition 14.1 Hamiltonkreis (HK)

Gegeben ein Graph G . Gefragt ist, ob es eine geschlossene Tour gibt, die *jeden Knoten genau einmal* besucht. (Das Problem ist für gerichtete wie ungerichtete Graphen NP-vollständig.)

Nicht verwechseln mit dem Eulerkreis, der jede *Kante* genau einmal nutzt – der ist leicht zu finden. Beim Hamiltonkreis geht es um die *Knoten*, und das ändert alles.

● Beispiel 14.2 Eine Rundtour durch alle Knoten



Die dicke Route berührt jeden der fünf Knoten exakt einmal und kehrt zum Start zurück – ein Hamiltonkreis.

Warum Hamiltonkreis in NP liegt

Was das Zertifikat ist. Die Besuchsreihenfolge – eine Auflistung aller Knoten in Tourfolge.

Wie geprüft wird. Zwei Kontrollen. Der Verifizierer hakt erstens ab, dass in der Liste jeder Knoten genau einmal auftaucht (etwa mit einer Markierungstabelle). Zweitens läuft er die Liste entlang und schlägt für jedes Nachbarpaar – einschließlich des Schlusses von letzter zu erster Position – nach, ob die Kante existiert. Beide Kontrollen bestanden: Annahme; sonst Ablehnung.

Warum das in beide Richtungen stimmt. Ein echter Hamiltonkreis liefert als Liste genau ein Zertifikat, das beide Kontrollen besteht. Und jede Liste, die beide Kontrollen besteht, beschreibt ihrerseits eine geschlossene Tour über lauter echte Kanten durch alle Knoten – einen Hamiltonkreis. Es gibt keine Lücke zwischen „wird akzeptiert“ und „ist wirklich einer“.

Wie lange es dauert. Markieren in $O(|V|)$, Kantenchecks in $O(|V|)$ Nachschlägen: zusammen linear bis $O(|V| + |E|)$. Also $\text{HK} \in \text{NP}$.

Woher die Härte kommt

 **Idee: $3\text{-SAT}' \leq_p \text{HK}$: Bausteine mit zwei Durchgängen**

Für die Variante mit *genau* drei Literalen pro Klausel baut man den Graphen aus zwei Bausteintypen. Die *A-Bausteine* – leiterförmige Komponenten, eine je Literalvorkommen – lassen sich von einer Tour nur auf genau zwei Weisen durchqueren; die Wahl der Weise kodiert, ob die zugehörige Variable wahr oder falsch ist. Die *B-Bausteine* – einer je Klausel – sind so verdrahtet, dass eine Tour sie für jede *nichtleere* Teilmenge ihrer drei Anschlusskanten passieren kann, für die leere aber nicht. Die leere Teilmenge entspricht dem Fall „alle drei Literale falsch“. Eine Hamiltontour durch das Gesamtgebilde existiert deshalb genau dann, wenn man die Variablen so setzen kann, dass keine Klausel leer ausgeht – wenn die Formel erfüllbar ist.

Wohin die Härte weiterfließt

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

Man macht aus dem Graphen eine Distanztabelle: echte Kanten kosten 1, fehlende Verbindungen kosten $|V| + 1$, und als Budget setzt man $L = |V|$. Eine Rundreise im Budget kann sich keine einzige teure Nicht-Kante leisten – sie existiert also genau dann, wenn der Graph eine reine Kanten-Tour durch alle Knoten besitzt: einen Hamiltonkreis. Details beim TSP.

 **Idee: Kreis und Pfad ineinander überführen**

Vom *Pfad* zum *Kreis*: Ein zusätzlicher Knoten, der mit allen verbunden ist, schließt jeden Hamiltonpfad zu einem Kreis. Vom *Kreis* zum *Pfad*: Man verdoppelt einen Knoten v zu einem Zwilling v^* mit denselben Nachbarn und klemmt an v und v^* je einen neuen Endknoten vom Grad 1 – ein Hamiltonpfad zwischen den beiden Endknoten entspricht dann genau einem Hamiltonkreis durch v .

15 TSP

■ Definition 15.1 TSP (Entscheidungsvariante)

Gegeben n Städte, eine Distanz $d(i, j)$ für jedes Städtepaar und ein Budget L . Gefragt: Gibt es eine Rundreise, die jede Stadt genau einmal besucht und insgesamt höchstens L kostet?

Gegenüber dem Hamiltonkreis kommen Zahlen ins Spiel: Nicht *ob* eine Rundtour existiert, sondern ob eine *billige* existiert. In der Optimierungsfassung – kürzeste Rundreise – ist das TSP das Paradebeispiel des Approximationsteils.

● Beispiel 15.2 Ein Budget einhalten

Vier Städte an den Ecken eines Quadrats, Seitenlänge 1, Diagonalen 2. Die Tour einmal außen herum kostet 4. Bei Budget $L = 4$ lautet die Antwort Ja; bei $L = 3$ Nein, denn jede Rundreise über vier Städte braucht vier Teilstrecken der Länge ≥ 1 .

Warum TSP in NP liegt

Was das Zertifikat ist. Die Stadtreihenfolge der behaupteten Rundreise.

Wie geprüft wird. Der Verifizierer bestätigt zuerst, dass jede Stadt genau einmal in der Reihenfolge steht. Dann summiert er die Distanzen von Station zu Station, die Rückfahrt zum Start eingeschlossen, und vergleicht die Gesamtsumme mit L . Summe im Budget und Reihenfolge vollständig: Annahme.

Warum das in beide Richtungen stimmt. Existiert eine Rundreise im Budget, dann wird ihre Reihenfolge eingereicht und angenommen. Existiert keine, dann sprengt jede vollständige Reihenfolge das Budget – der Summenvergleich lehnt sie ab. Beide Richtungen greifen.

Wie lange es dauert. n Additionen und ein Vergleich: $O(n)$. Also $\text{TSP} \in \text{NP}$.

Woher die Härte kommt

🔑 Idee: $\text{HK} \leq_p \text{TSP}$: teuer machen, was verboten ist

Aus dem HK-Graphen G mit Knotenmenge V wird die TSP-Instanz: Städte sind die Knoten; $d(u, v) = 1$ für jede Kante von G , $d(u, v) = |V| + 1$ für jedes Nicht-Kanten-Paar; Budget $L = |V|$. Eine Rundreise besteht aus genau $|V|$ Teilstrecken. Bleibt sie im Budget $|V|$, muss jede Teilstrecke den Preis 1 haben – schon eine einzige Nicht-Kante spränge mit $|V| + 1$ allein über das Budget. Die Rundreise benutzt also nur echte Kanten und ist ein Hamiltonkreis. Hat G umgekehrt einen Hamiltonkreis, kostet dieser als Rundreise exakt $|V|$ und hält das Budget ein. Bemerkenswert: Das TSP bleibt sogar dann NP-vollständig, wenn die Distanzen symmetrisch sind und die Dreiecksungleichung erfüllen.

Wohin die Härte weiterfließt

Das TSP reicht seine Härte im Kurs nicht weiter. Es wird stattdessen zur Hauptbühne des Approximationsteils: Dort zeigen wir, dass es ohne Zusatzannahmen nicht einmal näherungsweise lösbar ist – und mit Dreiecksungleichung erstaunlich gut.

16 3-dimensionales Matching

Der dritte Ast der Landkarte führt weg von den Graphen, hinein in Mengen und schließlich Zahlen. Sein erstes Glied handelt von Dreier-Zuordnungen.

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

Gegeben drei gleich große, disjunkte Mengen U , V , W sowie eine Liste T von Tripeln der Form (u, v, w) mit $u \in U$, $v \in V$, $w \in W$. Gefragt: Kann man Tripel aus T so auswählen, dass jedes Element aller drei Mengen in *genau einem* gewählten Tripel vorkommt?

🗨 Analogie

Drei gleich große Gruppen – etwa Piloten, Flugzeuge und Zeitfenster – sollen zu kompletten Einsätzen kombiniert werden. Die Liste T enthält die *zulässigen* Kombinationen. Gesucht ist ein Einsatzplan, in dem jeder Pilot, jedes Flugzeug und jedes Zeitfenster genau einmal verplant ist.

● Beispiel 16.2 Ein gültiger Plan

$U = \{u_1, u_2\}$, $V = \{v_1, v_2\}$, $W = \{w_1, w_2\}$; T enthalte (u_1, v_1, w_2) , (u_2, v_2, w_1) und (u_1, v_2, w_1) . Die Auswahl der ersten beiden Tripel verplant jedes der sechs Elemente genau einmal – ein 3-dimensionales Matching. Nimmt man stattdessen das dritte Tripel, lässt sich kein zweites mehr ergänzen, ohne u_1 , v_2 oder w_1 doppelt zu verwenden.

Warum 3-DM in NP liegt

Was das Zertifikat ist. Die getroffene Tripelauswahl $M \subseteq T$.

Wie geprüft wird. Der Verifizierer legt für jedes Element aller drei Mengen einen Zähler an und geht die gewählten Tripel durch, wobei er für jede der drei Komponenten den Zähler erhöht. Am Ende kontrolliert er: Steht jeder Zähler auf genau 1? Zusätzlich prüft er $|M| = |U|$. Beides erfüllt: Annahme; irgendein Zähler auf 0 oder ≥ 2 : Ablehnung.

Warum das in beide Richtungen stimmt. Ein echtes Matching bringt konstruktionsgemäß jeden Zähler auf exakt 1 und wird angenommen. Gibt es keines, dann verfehlt jede Auswahl das Zählerbild – irgendwo entsteht eine Doppelung oder eine Lücke – und sie wird abgelehnt.

Wie lange es dauert. Einmal über die Auswahl laufen und Zähler pflegen: $O(|T|)$ plus $O(|U| + |V| + |W|)$ fürs Kontrollieren. Also $3\text{-DM} \in \text{NP}$.

Woher die Härte kommt

🔑 Idee: $\text{SAT} \leq_p \text{3-DM}$: Zahnräder, Andockstellen, Aufräumer

Drei Baustein-Sorten übersetzen eine Formel in Tripel. *Zahnräder*: Für jede Variable ordnet man Tripel ringförmig an; ein Matching kann den Ring nur in einer von zwei Verzahnungen abdecken – die eine heißt „wahr“, die andere „falsch“. Je nach Verzahnung bleiben genau die Literal-Elemente der einen oder der anderen Sorte unbedeckt zurück. *Andockstellen*: Für jede Klausel gibt es ein Elementpaar, dessen Tripel nur an ein *freigebliebenes* Literal der Klausel andocken können – das erzwingt, dass jede Klausel ein wahres Literal hat. *Aufräumer*: Zusätzliche Tripel („Garbage Collection“) sammeln alle danach noch unbedeckten Literal-Elemente ein, damit die Bilanz „jedes Element genau einmal“ exakt aufgeht. Ein Matching existiert genau dann, wenn die Formel erfüllbar ist.

Wohin die Härte weiterfließt

🔑 Idee: $\text{3-DM} \leq_p \text{3-EC}$: nur die Brille wechseln

Ein Tripel (u, v, w) ist nichts anderes als eine dreielementige Menge $\{u, v, w\}$ über dem Gesamtuniversum $U \cup V \cup W$. Mehr braucht die Reduktion nicht – ausgeführt wird sie beim 3-Exact Cover.

17 3-Exact Cover

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

Gegeben ein Universum U mit $3m$ Elementen und eine Familie dreielementiger Teilmengen $S_1, \dots, S_n \subseteq U$. Gefragt: Lassen sich Mengen so auswählen, dass jedes Element von U in *genau einer* gewählten Menge liegt – eine lückenlose Überdeckung ohne Überlappung?

● Beispiel 17.2 Exakt oder nicht

$U = \{1, \dots, 6\}$ mit $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4, 6\}$, $S_3 = \{1, 2, 3\}$. Die Wahl $\{S_1, S_2\}$ deckt jedes Element genau einmal – exakt. Die Wahl $\{S_2, S_3\}$ deckt die 2 doppelt und lässt die 5 aus – keine exakte Überdeckung.

Warum 3-EC in NP liegt

Was das Zertifikat ist. Die Auswahl der Mengen. Eine exakte Überdeckung von $3m$ Elementen durch Dreiermengen umfasst genau m Mengen.

Wie geprüft wird. Der Verifizierer führt eine Strichliste über die Elemente von U : Für jede gewählte Menge macht er bei ihren drei Elementen einen Strich. Danach kontrolliert er, dass jedes Element genau einen Strich trägt – keiner fehlt, keiner ist doppelt. Bestanden: Annahme;

sonst Ablehnung.

Warum das in beide Richtungen stimmt. Eine exakte Überdeckung erzeugt per Definition genau das Strichbild „überall eins“ und wird angenommen. Existiert keine, produziert jede Auswahl entweder eine Lücke oder eine Doppelung und fällt durch.

Wie lange es dauert. Strichliste führen und ablesen: linear in der Eingabe. Also $3\text{-EC} \in \text{NP}$.

Woher die Härte kommt

 **Idee: $3\text{-DM} \leq_p 3\text{-EC}$: dieselben Daten, neue Lesart**

Man nimmt die 3-DM-Instanz und liest sie um: Das Universum ist $U \cup V \cup W$ (Größe $3 \cdot |U|$), und jedes Tripel (u, v, w) wird zur Menge $\{u, v, w\}$. Es wird nichts hinzugefügt und nichts verändert.

Dass die Antwort erhalten bleibt, liegt daran, dass beide Bedingungen dasselbe sagen: „Jedes Element kommt in genau einem gewählten Tripel vor“ (3-DM) und „jedes Element liegt in genau einer gewählten Menge“ (3-EC) sind identische Forderungen, nur einmal über Tripel und einmal über Mengen formuliert. Eine Auswahl erfüllt die eine genau dann, wenn sie die andere erfüllt – in beide Richtungen. Die Umformung ist reine Notation und damit trivial polynomiell berechenbar in einem Durchlauf. Da 3-DM NP-vollständig ist und $3\text{-EC} \in \text{NP}$, ist 3-EC NP-vollständig.

Wohin die Härte weiterfließt

 **Idee: $3\text{-EC} \leq_p \text{SubSet Sum}$: Überdecken wird Addieren**

Jede Menge wird als Zahl in einem Stellensystem mit riesiger Basis kodiert; exakt überdecken heißt dann, die Zahl „lauter Einsen“ als Summe zu treffen. Vollständig ausgeführt beim SubSet Sum.

18 SubSet Sum

■ Definition 18.1 SubSet Sum

Gegeben ganze Zahlen $c_1, \dots, c_n$ und ein Ziel K . Gefragt: Existiert eine Auswahl dieser Zahlen, deren Summe *exakt* K ergibt?

● Beispiel 18.2 Ein Ziel treffen

Zahlen 5, 9, 11, 2 und Ziel $K = 16$. Die Auswahl $\{5, 9, 2\}$ summiert auf 16 – getroffen. $\{5, 11\}$ ergibt dagegen 16 ebenfalls! Beide sind gültige Zertifikate; eines genügt. Für $K = 8$ hingegen trifft keine Auswahl.

Warum SubSet Sum in NP liegt

Was das Zertifikat ist. Die Auswahl – pro Zahl ein Bit („dabei / nicht dabei“).

Wie geprüft wird. Der Verifizierer addiert die ausgewählten Zahlen und vergleicht das Ergebnis mit K . Bei Gleichheit akzeptiert er, sonst nicht.

Warum das in beide Richtungen stimmt. Gibt es eine treffende Auswahl, liefert genau sie die Summe K und wird angenommen. Gibt es keine, weicht jede eingereichte Auswahl von K ab und wird abgewiesen – der Vergleich lässt keinen Spielraum.

Wie lange es dauert. Höchstens n Additionen und ein Vergleich (auf langen Zahlen, deren Länge Teil der Eingabe ist): polynomiell. Also SUBSET SUM $\in$ NP. Die Schwere steckt nicht im Nachrechnen, sondern darin, unter 2^n Auswahlmöglichkeiten die richtige zu *finden*.

Woher die Härte kommt

 **Idee: 3-EC $\leq_p$ SubSet Sum: Stellenwertsystem ohne Übertrag**

Nummeriere die Universums-Elemente 1 bis $3m$ und gib jedem eine *Ziffernposition* in einem Stellensystem zur Basis $n + 1$ (n = Anzahl der Mengen). Jede Menge S_j wird zur Zahl mit Ziffer 1 an den Positionen ihrer drei Elemente:

$$c_j = \sum_{u_i \in S_j} (n + 1)^{i-1}.$$

Das Ziel K ist die Zahl mit Ziffer 1 an *allen* $3m$ Positionen.

Der entscheidende Kunstgriff ist die Basiswahl. Addiert man beliebig viele der c_j , dann sammeln sich an jeder Ziffernposition höchstens n Einsen – weniger als die Basis $n + 1$. Es entsteht also *nie ein Übertrag*, und die Addition wirkt an jeder Position unabhängig. „Summe = K “ zerfällt dadurch in $3m$ separate Bedingungen: *jede* Position wird von *genau einer* gewählten Menge bedient. Das ist Wort für Wort die exakte Überdeckung. Bei Basis 2 wäre das falsch – Überträge könnten Lücken und Doppelungen gegeneinander verrechnen.

● Beispiel 18.3 Die Kodierung nachgerechnet

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

Menge	Positionen	Wert zur Basis 5
S_1	1, 2, 3	$5^0 + 5^1 + 5^2 = 31$
S_2	4, 5, 6	$5^3 + 5^4 + 5^5 = 3875$
S_3	1, 2, 4	$5^0 + 5^1 + 5^3 = 131$
S_4	3, 5, 6	$5^2 + 5^4 + 5^5 = 3775$
Ziel K	1, ..., 6	$5^0 + \dots + 5^5 = 3906$

Probe: $31 + 3875 = 3906$ – und $\{S_1, S_2\}$ überdeckt U exakt. Genauso $131 + 3775 = 3906$ mit der exakten Überdeckung $\{S_3, S_4\}$. Auswählen, die nicht exakt überdecken, verfehlen die Zielzahl.

Neben dieser Kette gibt es eine zweite, gezielt sparsame Härte-Quelle, die direkt bei 3-SAT ansetzt. Sie wird in Teil C gebraucht.

☞ Idee: $3\text{-SAT} \leq_p \text{SubSet Sum (streng): Dezimalziffern}$

Man spendiert jeder Variablen und jeder Klausel eine eigene Dezimalstelle – die Zahlen haben $n + m$ Ziffern zur Basis 10. Für jede Variable x_i gibt es *zwei* Zahlen: eine für „wahr“, eine für „falsch“. Beide tragen an der Variablenstelle i eine 1 – das Ziel verlangt dort genau 1, erzwingt also die Wahl *einer* der beiden. An den Klauselstellen trägt die „wahr“-Zahl Einsen bei den Klauseln, in denen x_i positiv vorkommt, die „falsch“-Zahl bei denen mit $\neg x_i$. Das Ziel verlangt an jeder Klauselstelle eine 3; da eine erfüllte Klausel ein bis drei wahre Literale liefert, gleichen *zwei Ausgleichszahlen* pro Klausel (Wert 1 an genau dieser Stelle) die Differenz aus. Erfüllbarkeit und Zielsumme entsprechen sich exakt. Die Konstruktion erzeugt nur $2n + 2m$ Zahlen – diese Sparsamkeit macht sie für die unteren Schranken wertvoll.

Wohin die Härte weiterfließt

☞ Idee: $\text{SubSet Sum} \leq_p \text{Partition: das Gleichgewicht erzwingen}$

Zwei zusätzliche, geschickt dimensionierte Zahlen verwandeln „triff K “ in „teile alles in zwei gleiche Hälften“. Ausgeführt bei Partition.

☞ Idee: $\text{SubSet Sum} \leq_p \text{Knapsack: ein Spezialfall}$

Setzt man im Rucksack Gewicht = Profit und Kapazität = Zielprofit = K , kollabieren beide Rucksack-Bedingungen zu „Summe exakt K “. Ausgeführt beim Knapsack.

 Idee: SubSet Sum $\leq_p$ Kardinalitäts-Variante: Verschieben und Auffüllen

Verlangt die Variante zusätzlich „nimm genau n Zahlen“, erhöht man jede Originalzahl um 1 und legt n Füll-Einsen dazu; das Ziel wächst auf $K + n$. Die Füllzahlen gleichen die Anzahl aus, ohne die Substanz der Summe zu verändern.

19 Partition

■ **Definition 19.1 Partition**

Gegeben ganze Zahlen $c_1, \dots, c_n$. Gefragt: Kann man sie in zwei Gruppen mit *gleicher Summe* aufteilen – gibt es also eine Auswahl, deren Summe exakt die Hälfte der Gesamtsumme beträgt?

● **Beispiel 19.2 Ja-Instanz und Nein-Instanz**

Die Zahlen 1, 5, 6, 2 haben Gesamtsumme 14, Hälfte 7. Die Auswahl $\{1, 6\}$ trifft die 7, der Rest $\{5, 2\}$ ebenso – eine gültige Halbierung, Antwort Ja. Anders die Zahlen 4, 7, 3, 2: Gesamtsumme 16, Hälfte 8. Probiert man alle Teilsummen durch, erreicht man 6, 7 und 9, aber nie genau 8 – keine Halbierung möglich, Antwort Nein.

Warum Partition in NP liegt

Was das Zertifikat ist. Die Auswahl einer Seite der Aufteilung – pro Zahl ein Bit.

Wie geprüft wird. Der Verifizierer berechnet zweierlei: die Gesamtsumme N aller Zahlen und die Summe der ausgewählten. Er akzeptiert genau dann, wenn die Auswahl-Summe exakt $N/2$ beträgt.

Warum das in beide Richtungen stimmt. Lässt sich die Menge halbieren, dann erreicht die passende Auswahl exakt $N/2$ und wird angenommen. Lässt sie sich nicht halbieren, dann liegt jede Auswahl über oder unter $N/2$, und der Vergleich weist sie ab.

Wie lange es dauert. Zwei Summationen über n Zahlen: $O(n)$ Arithmetikschritte. Also $\text{PARTITION} \in \text{NP}$.

Woher die Härte kommt

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

Gegeben Zahlen $c_1, \dots, c_n$ mit Ziel K . Setze $N = (\sum_j c_j) + 1$ und lege *zwei neue Zahlen* dazu: $a = N - K$ und $b = K + 1$.

Die Rechnung dahinter: Die erweiterte Menge hat die Gesamtsumme $\sum_j c_j + (N - K) + (K + 1) = (N - 1) + N + 1 = 2N$; eine gültige Halbierung muss also je Seite exakt N erreichen. Nun das

zentrale Detail: $a + b = N + 1 > N$ – die beiden neuen Zahlen *zusammen* sprengen jede Hälfte. In jeder Halbierung liegen sie deshalb zwangsläufig auf *verschiedenen* Seiten. Betrachte die Seite mit $a = N - K$: Ihre übrigen Mitglieder sind Originalzahlen und müssen zusammen $N - (N - K) = K$ ergeben – eine Ziel-treffende Auswahl im Sinne von SubSet Sum. Umgekehrt: Trifft eine Original-Auswahl S das Ziel K , dann bildet $S \cup \{a\}$ eine Seite mit Summe $K + (N - K) = N$ – eine gültige Halbierung. Ja- und Nein-Antworten stimmen also überein. Zwei Zahlen anhängen kostet nichts; die Reduktion ist polynomiell, PARTITION NP-vollständig.

● Beispiel 19.3 Die Ausgleichsgewichte in Aktion

Zahlen 3, 5, 8, 4 (Summe 20), Ziel $K = 12$. Dann $N = 21$, und die neuen Zahlen sind $a = 9$ und $b = 13$. Erweiterte Menge: $\{3, 5, 8, 4, 9, 13\}$ mit Gesamtsumme 42, Halbierungswert 21. Die Original-Auswahl $\{8, 4\}$ trifft $K = 12$; zusammen mit $a = 9$ entsteht die Seite $\{8, 4, 9\}$ mit Summe 21 – die andere Seite $\{3, 5, 13\}$ hat ebenfalls 21. Es geht auf.

Wohin die Härte weiterfließt

📖 Idee: $\text{Partition} \leq_p P2\|C_{\max}$: Halbieren heißt Balancieren

Deutet man die Zahlen als Job-Laufzeiten auf zwei Maschinen, dann heißt „beide Maschinen gleichzeitig fertig“ genau „Zahlen halbiert“. Ausgeführt beim Scheduling.

20 Knapsack

■ Definition 20.1 Knapsack (Rucksackproblem)

Gegeben n Gegenstände – der i -te mit Gewicht w_i und Profit p_i – sowie eine Tragkraft B und ein Profitziel P . Gefragt: Gibt es eine Auswahl, deren Gesamtgewicht höchstens B und deren Gesamtprofit mindestens P ist?

● Beispiel 20.2 Packen mit zwei Bedingungen

Gegenstände (w, p) : $(2, 3)$, $(3, 4)$, $(4, 6)$; Tragkraft $B = 6$, Ziel $P = 9$. Die Auswahl $\{(2, 3), (4, 6)\}$ wiegt 6 und bringt Profit 9 – beide Bedingungen erfüllt, Antwort Ja. Die Auswahl aller drei wöge 9 und ist verboten.

Warum Knapsack in NP liegt

Was das Zertifikat ist. Die Packliste – pro Gegenstand ein Bit.

Wie geprüft wird. Der Verifizierer bildet zwei Summen über die ausgewählten Gegenstände: das Gesamtgewicht und den Gesamtprofit. Er akzeptiert genau dann, wenn das Gewicht $\leq B$ und der Profit $\geq P$ ist.

Warum das in beide Richtungen stimmt. Eine zulässige, profitable Auswahl besteht beide Vergleiche und wird angenommen. Existiert keine, dann reißt jede Auswahl mindestens eine der beiden Schranken, und einer der Vergleiche schlägt fehl.

Wie lange es dauert. Zwei Summen, zwei Vergleiche: $O(n)$. Also KNAPSACK $\in$ NP.

Woher die Härte kommt

Idee: SubSet Sum $\leq_p$ Knapsack: beide Schranken zusammenziehen

Aus der SubSet-Sum-Instanz $(c_1, \dots, c_n; K)$ macht man Gegenstände mit $w_i = p_i = c_i$ und setzt $B = P = K$. Die Rucksackfrage verlangt nun eine Auswahl mit Summe $\leq K$ (Gewicht) und Summe $\geq K$ (Profit) – da beide Summen identisch sind, geht das nur mit Summe *exakt* K . Damit sind die Ja-Instanzen beider Probleme dieselben, in beiden Richtungen. Die Umformung ist ein einfacher Durchlauf. KNAPSACK ist NP-vollständig.

Wohin die Härte weiterfließt

Keine weitere Reduktion geht vom Rucksack aus. Dafür ist er im Approximationsteil das Vorzeigebispiel für die ganze Palette: vom scheiternden Greedy bis zum FPTAS.

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

■ Definition 21.1 $P \parallel C_{\max}$

Gegeben n Jobs mit Laufzeiten $p_1, \dots, p_n$ und m baugleiche Maschinen. Jeder Job läuft komplett auf einer Maschine. Der *Makespan* eines Plans ist die Fertigstellungszeit der am längsten laufenden Maschine. Optimierungsziel: den Makespan minimieren; Entscheidungsfrage: Geht Makespan $\leq T$?

● Beispiel 21.2 Lasten verteilen

Jobs 5, 4, 3, 2 auf zwei Maschinen. Verteilt man $\{5, 2\}$ und $\{4, 3\}$, sind beide Maschinen nach 7 Zeiteinheiten fertig – Makespan 7. Schlechter wäre $\{5, 4\}$ gegen $\{3, 2\}$: Makespan 9. Unter 7 geht es nicht, denn die Gesamtarbeit 14 verteilt auf 2 Maschinen erzwingt mindestens 7.

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

Was das Zertifikat ist. Der Plan – für jeden Job die Nummer seiner Maschine.

Wie geprüft wird. Der Verifizierer summiert je Maschine die Laufzeiten der dort eingeplanten Jobs, bestimmt das Maximum dieser Summen und vergleicht es mit T . Liegt es darunter oder darauf, akzeptiert er.

Warum das in beide Richtungen stimmt. Ein Plan mit Makespan $\leq T$ besteht den Vergleich.

Existiert keiner, dann hat jeder eingereichte Plan eine Maschine über T , und das Maximum verrät sie.

Wie lange es dauert. Jeden Job einmal anfassen: $O(n)$. Also liegt die Entscheidungsvariante von $P||C_{\max}$ in NP.

Woher die Härte kommt

Idee: $\text{Partition} \leq_p P2||C_{\max}$: perfekte Balance

Die Partition-Zahlen werden Jobs, die Maschinenzahl ist 2, und als Makespan-Schranke wählt man $T = \frac{1}{2} \sum_j c_j$. Die Gesamtarbeit beider Maschinen ist fix $\sum_j c_j$; ein Makespan von höchstens T heißt also, dass *keine* Maschine über die Hälfte kommt – und weil beide zusammen das Ganze stemmen, tragen dann *beide exakt* die Hälfte. Ein solcher Plan ist wörtlich eine Halbierung der Zahlen, und jede Halbierung ist umgekehrt so ein Plan. Damit ist das Scheduling bereits mit *zwei* Maschinen NP-vollständig – mehr Maschinen machen es nicht leichter.

Wohin die Härte weiterfließt

Keine ausgehende Reduktion; im Approximationsteil liefert das Scheduling die Bühne für List Scheduling und LPT.

22 Hitting Set

■ Definition 22.1 Hitting Set

Gegeben ein Universum U , Mengen $F_1, \dots, F_r \subseteq U$ und eine Zahl k . Eine Menge $H \subseteq U$ „trifft“ F_i , wenn $H \cap F_i \neq \emptyset$. Gefragt: Gibt es ein H mit höchstens k Elementen, das *alle* F_i trifft?

Analogie

Die Mengen F_i sind Kundenwünsche – jeder Kunde nennt einige akzeptable Produkte. Gesucht ist ein Sortiment aus höchstens k Produkten, das *jeden* Kunden zufriedenstellt: Von jedem Wunschzettel muss mindestens ein Produkt im Sortiment sein.

● Beispiel 22.2 Ein kleines Sortiment

$U = \{1, 2, 3, 4, 5\}$; Wünsche $F_1 = \{1, 2\}$, $F_2 = \{2, 3\}$, $F_3 = \{4, 5\}$. Das Sortiment $\{2, 4\}$ trifft alle drei Wünsche – Größe 2. Mit einem einzigen Produkt geht es nicht, denn $F_1 \cup F_2$ und F_3 sind disjunkt.

Warum Hitting Set in NP liegt

Was das Zertifikat ist. Die Treffermenge H .

Wie geprüft wird. Der Verifizierer zählt $|H|$ gegen k . Danach nimmt er sich jede Menge F_i vor und sucht ein gemeinsames Element mit H . Wird er bei jeder fündig und stimmt die Größe, akzeptiert er; eine verfehlte Menge genügt zur Ablehnung.

Warum das in beide Richtungen stimmt. Ein echtes Hitting Set der Größe $\leq k$ übersteht alle Suchen. Existiert keines, lässt jede Menge H dieser Größe mindestens ein F_i unberührt, und die Suche dort scheitert.

Wie lange es dauert. Für jede der r Mengen ein Abgleich mit H : $O(r \cdot |U|)$. Also HITTING SET $\in$ NP.

Woher die Härte kommt

Idee: 3-SAT $\leq_p$ Hitting Set: Belegung als Auswahl

Das Universum besteht aus allen $2n$ Literalen. Es gibt zwei Sorten von Mengen: Für jede Variable die *Paarmenge* $\{x_i, \bar{x}_i\}$ und für jede Klausel die Menge ihrer (höchstens drei) Literale. Die Schranke ist $k = n$. Die n disjunkten Paarmengen zwingen jede Lösung, aus *jedem* Paar mindestens ein Literal zu nehmen – bei Budget n also *genau* eines: Das ist eine Belegung. Die Klauselmengen verlangen zusätzlich, dass diese Belegung in jeder Klausel ein Literal trifft – die Klausel also erfüllt. Ein Hitting Set der Größe n existiert demnach genau dann, wenn die Formel erfüllbar ist; beide Richtungen stecken in dieser Entsprechung. Der Aufbau ist ein Durchlauf über die Formel. HITTING SET ist NP-vollständig.

Wohin die Härte weiterfließt

Idee: Hitting Set $\leq_p$ Set Cover: Seiten wechseln

Elemente und Mengen tauschen die Plätze – was vorher traf, überdeckt nachher. Ausgeführt beim Set Cover.

23 Set Cover

■ Definition 23.1 Set Cover

Gegeben ein Universum U , Mengen $S_1, \dots, S_r \subseteq U$ und eine Zahl k . Gefragt: Gibt es höchstens k dieser Mengen, deren Vereinigung ganz U ergibt?

● Beispiel 23.2 Alles abdecken

$U = \{1, 2, 3, 4, 5\}$; $S_1 = \{1, 2, 3\}$, $S_2 = \{3, 4\}$, $S_3 = \{4, 5\}$. Die Wahl $\{S_1, S_3\}$ vereinigt sich zu $\{1, 2, 3, 4, 5\} = U$ – ein Set Cover der Größe 2.

Warum Set Cover in NP liegt

Was das Zertifikat ist. Die Liste der gewählten Mengen-Indizes.

Wie geprüft wird. Der Verifizierer zählt die gewählten Mengen gegen k . Dann markiert er alle Elemente, die in irgendeiner gewählten Menge vorkommen, und kontrolliert, ob ganz U markiert ist. Vollständig markiert und Anzahl im Rahmen: Annahme.

Warum das in beide Richtungen stimmt. Ein echtes Set Cover markiert alles und wird angenommen. Gibt es keines mit $\leq k$ Mengen, bleibt bei jeder solchen Wahl ein Element unmarkiert – Ablehnung.

Wie lange es dauert. Markieren über alle gewählten Mengen: $O(r \cdot |U|)$. Also SET COVER $\in$ NP.

Woher die Härte kommt

Idee: Hitting Set $\leq_p$ Set Cover: der Seitenwechsel

Aus der Hitting-Set-Instanz (Universum U , Mengen $F_1, \dots, F_r$, Schranke k) wird eine Set-Cover-Instanz mit *vertauschten Rollen*: Das neue Universum sind die Mengen-Nummern $\{1, \dots, r\}$; für jedes alte Element $u \in U$ entsteht die neue Menge $D_u = \{i \mid u \in F_i\}$ – die Nummern aller alten Mengen, die u enthalten. Die Schranke bleibt k .

Die Entsprechung: Wählt man im Original die Treffermenge $H = \{u_1, \dots\}$, dann überdecken die neuen Mengen $D_{u_1}, \dots$ genau die Nummern aller F_i , die von H getroffen werden. „ H trifft alle F_i “ übersetzt sich also in „die gewählten D_u überdecken alle Nummern“ – ein Set Cover derselben Größe. Und rückwärts genauso: Ein Set Cover aus D -Mengen benennt eine gleich große Trefferauswahl. Beide Richtungen sind ein reiner Blickwechsel. Bemerkenswert: Wendet man den Seitenwechsel zweimal an, steht die ursprüngliche Instanz wieder da – die Übersetzung ist ihre eigene Umkehrung. Deshalb sind Hitting Set und Set Cover exakt gleich schwer. SET COVER ist NP-vollständig.

Wohin die Härte weiterfließt

Idee: Set Cover $\leq_p$ Hitting Set: derselbe Wechsel zurück

Weil der Seitenwechsel seine eigene Umkehrung ist, funktioniert er wörtlich auch von Set Cover nach Hitting Set: Neue Elemente sind die Set-Cover-Mengenummern, neue Mengen entstehen pro Element u als Nummernliste seiner Überdecker, k bleibt. Die Antwort bleibt erhalten – mit derselben Begründung wie eben, nur mit vertauschten Namen. Im Kurs wird dieses Paar vor allem bei den unteren Schranken in Teil C gebraucht.

24 Dominating Set

■ Definition 24.1 Dominating Set

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Eine Menge $D \subseteq V$ *dominiert* den Graphen, wenn jeder Knoten entweder selbst zu D gehört oder mit einem Knoten aus D benachbart ist. Gefragt: Gibt es ein solches D mit höchstens k Knoten?

☞ Analogie

Denk an Funkmasten in Dörfern: Ein Mast versorgt sein eigenes Dorf und alle direkt verbundenen Nachbardörfer. Ein Dominating Set ist ein Mastenplan, bei dem jedes Dorf Empfang hat.

● Beispiel 24.2 Ein Mast genügt

Ein sternförmiges Netz: Zentrum z , außen a, b, c, d , jede Außenstelle nur mit z verbunden. Der Plan $D = \{z\}$ dominiert alles – z versorgt sich selbst und alle vier Nachbarn. Größe 1.

Warum Dominating Set in NP liegt

Was das Zertifikat ist. Die Auswahl D – ein Bit pro Knoten.

Wie geprüft wird. Zwei Schritte. Erstens zählt der Verifizierer $|D|$ gegen k . Zweitens prüft er die Versorgung: Für jeden Knoten v des Graphen fragt er, ob $v \in D$ gilt – und falls nicht, geht er die Nachbarn von v durch (eine Zeile der Adjazenzmatrix) und sucht dort ein Mitglied von D . Bleibt ein Knoten unversorgt, lehnt er ab; sonst akzeptiert er.

Warum das in beide Richtungen stimmt. Ein dominierendes D der Größe $\leq k$ lässt keinen Knoten unversorgt und besteht beide Schritte. Existiert keines, hat jede Auswahl dieser Größe irgendwo einen unversorgten Knoten, und die Versorgungsprüfung findet ihn.

Wie lange es dauert. Pro Knoten eine Matrixzeile durchsehen: $O(|V|^2)$ insgesamt, dazu $O(|V|)$ fürs Zählen. Also DOMINATING SET $\in$ NP.

Woher die Härte kommt

► Zusammenhang

Für DOMINATING SET enthält das Kursmaterial *nur* den NP-Zugehörigkeits-Nachweis – er war eine Klausuraufgabe. Eine Härte-Reduktion wird im Kurs nicht geführt. Das Problem ist in der Literatur als NP-vollständig bekannt, aber der zugehörige Beweis gehört nicht zum gesicherten Stoff dieses Kurses, und deshalb steht er auch nicht in diesem Heft.

25 Longest Path

■ Definition 25.1 Longest Path

Gegeben ein Graph $G = (V, E)$ und eine Zahl k . Ein Pfad heißt *einfach*, wenn er keinen Knoten wiederholt; seine Länge ist die Zahl seiner Kanten. Gefragt: Enthält G einen einfachen Pfad der Länge mindestens k ?

● Beispiel 25.2 Pfadlängen

Im Kreis aus fünf Knoten $1 - 2 - 3 - 4 - 5 - 1$ ist $1, 2, 3, 4, 5$ ein einfacher Pfad der Länge 4. Länger geht es nicht – ein sechster Schritt müsste einen besuchten Knoten wiederholen.

Warum Longest Path in NP liegt

Was das Zertifikat ist. Die Knotenfolge $(v_0, v_1, \dots, v_k)$ des behaupteten Pfades.

Wie geprüft wird. Der Verifizierer kontrolliert zweierlei: dass in der Folge kein Knoten doppelt vorkommt (paarweise Vergleiche oder eine Markierungstabelle), und dass jedes aufeinanderfolgende Paar (v_i, v_{i+1}) durch eine Kante verbunden ist (ein Nachschlag pro Schritt). Beides bestanden: Annahme.

Warum das in beide Richtungen stimmt. Hat der Graph einen einfachen Pfad der Länge $\geq k$, dann bilden dessen erste $k + 1$ Knoten ein Zertifikat, das beide Kontrollen besteht. Und jede Folge, die beide Kontrollen besteht, ist selbst ein einfacher Pfad der Länge k – der Graph hat dann einen. Kein Vorschlag kann den Verifizierer täuschen.

Wie lange es dauert. Verschiedenheit in $O(k^2)$, Kanten in $O(k)$: zusammen $O(k^2) \subseteq O(|V|^2)$. Also LONGEST PATH $\in$ NP.

Woher die Härte kommt

► Zusammenhang

Auch beim LONGEST PATH behandelt der Kurs ausschließlich die NP-Zugehörigkeit – ebenfalls als Klausuraufgabe. Eine Härte-Reduktion (etwa über den Hamiltonpfad, den Longest Path verallgemeinert) kommt im Kursmaterial nicht vor und wird hier deshalb nicht als Kursstoff ausgegeben.

26 Das Halteproblem

Der Katalog schließt mit einem Ausreißer – einem Problem, das NP-schwer ist, ohne in NP zu liegen. Es markiert die Außengrenze der bisherigen Begriffe.

■ Definition 26.1 Halteproblem

Gegeben der Quelltext eines Programms M und eine Eingabe w . Gefragt: Stoppt M , wenn man es auf w laufen lässt – oder läuft es endlos?

Turing hat bewiesen, dass *kein* Algorithmus diese Frage für alle Programme beantwortet – das Halteproblem ist *unentscheidbar*. Daraus folgt sofort: Es liegt nicht in NP. Denn jedes NP-Problem ist entscheidbar – zur Not probiert man alle Zertifikate durch, die der Verifizierer innerhalb seiner Laufzeitschranke überhaupt lesen kann; das dauert exponentiell lange, terminiert aber.

Woher die Härte kommt

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

Zu einer Formel φ konstruiert man ein Programm M_φ nach festem Muster: Es zählt alle Belegungen der Variablen von φ der Reihe nach auf, testet jede, und *stoppt*, sobald eine die Formel erfüllt – findet es keine, läuft die Schleife für immer weiter. Als Reduktionsausgabe dient das Paar $\langle M_\varphi, \text{leere Eingabe} \rangle$.

Die Antwort-Erhaltung: Ist φ erfüllbar, erreicht die Schleife irgendwann die erste erfüllende Belegung und M_φ hält – Ja bleibt Ja. Ist φ unerfüllbar, findet die Schleife nie einen Grund zu stoppen und läuft ewig – Nein bleibt Nein. Entscheidend ist die Trennung zweier Zeitbegriffe: M_φ selbst darf beliebig lange laufen, denn *ausgeführt* wird es in der Reduktion nie. Polynomiell sein muss nur das *Aufschreiben* seines Quelltexts aus φ – und der ist kaum mehr als die Formel plus eine Schleifenschablone. Damit reduziert SAT auf das Halteproblem, und das Halteproblem ist NP-schwer.

☆ Aha

Hier trennen sich die Begriffe sichtbar: Das Halteproblem ist NP-*schwer*, aber mangels Entscheidbarkeit nicht in NP – also nicht NP-*vollständig*. Die beiden Hälften der Vollständigkeit sind wirklich unabhängige Bedingungen, und dieses Problem erfüllt nur eine davon.

Wohin die Härte weiterfließt

Nirgendwohin – als unentscheidbares Problem ist das Halteproblem im Kurs die Endstation.

Teil C – Untere Schranken mit der ETH

„NP-vollständig“ bedeutet nur: vermutlich kein Polynomialzeit-Algorithmus. Offen bleibt, *wie langsam* es wirklich sein muss – 2^n ? $2^{\sqrt{n}}$? $n^{\log n}$? Die Exponentialzeit-Hypothese verschärft die Annahme so weit, dass sich konkrete Untergrenzen ausrechnen lassen.

27 Das Handwerkszeug

Echt langsamer wachsen: klein-o

■ Definition 27.1 Klein-o-Notation

$f(n) = o(g(n))$ bedeutet, dass der Quotient $f(n)/g(n)$ gegen null geht – f wird von g auf lange Sicht beliebig weit abgehängt. Wichtig: Ein konstanter Faktor rettet nichts, δn ist *nicht* $o(n)$.

Ein Algorithmus mit Laufzeit $2^{o(n)}$ hat einen Exponenten, der langsamer als jede Gerade wächst – man nennt das *subexponentiell*. $2^{\sqrt{n}}$ ist so ein Fall: enorm schneller als 2^n , aber immer noch weit von polynomiell entfernt.

Die Hypothese selbst

■ Definition 27.2 Exponentialzeit-Hypothese (ETH)

Es existiert eine Konstante $\delta > 0$, sodass kein Algorithmus 3-SAT mit n Variablen in Zeit $2^{\delta n} \cdot \text{poly}$ löst. Gleichwertig formuliert: 3-SAT hat keinen $2^{o(n)}$ -Algorithmus.

Bewiesen ist die ETH nicht – sie ist eine Arbeitshypothese, stärker als $P \neq NP$. Sie passt aber zum Forschungsstand: Trotz Jahrzehnten der Suche liegen die besten 3-SAT-Löser bei c^n mit c knapp über 1,3.

Von Variablen zu Klauseln: das Sparsification-Lemma

▲ Satz 27.3 Sparsification-Lemma

Unter der ETH hat 3-SAT auch keinen Algorithmus mit Laufzeit $2^{o(m)} \cdot \text{poly}$, wobei m die *Klauselzahl* bezeichnet.

★ Intuition

Wozu ein eigenes Lemma? Die ETH spricht von der *Variablenzahl* n . Reduktionen bemessen ihre Ausgabegröße aber meist an der *Klauselzahl* m – und m kann kubisch größer sein als n . Aus „kein $2^{o(n)}$ “ folgt darum noch lange nicht „kein $2^{o(m)}$ “. Das Lemma schließt diese Lücke und macht die Klauselzahl zur gleichwertigen Härte-Währung. Als Rückweg merkt man sich $n \leq 3m$: Mehr Variablen als Literale kann es nicht geben.

Das Argumentationsmuster

Idee: Kontraposition mit Größenrechnung

Jede ETH-Schranke folgt demselben Dreischritt. *Schritt 1:* Drücke die Zielgröße der Reduktion (Knotenzahl, Mengenzahl, Parameter k , ...) durch n und m aus. *Schritt 2:* Unterstelle einen zu schnellen Algorithmus für das Zielproblem und rechne nach, welche 3-SAT-Laufzeit die Kette „Reduktion + Algorithmus“ ergäbe. *Schritt 3:* Stelle den Widerspruch fest – zur ETH direkt, wenn die Größe nur an n hängt; zum Sparsification-Lemma, wenn sie an m hängt.

Stolperstein

Ob man das Sparsification-Lemma zitieren muss, entscheidet sich allein daran, *welche* Größe in Schritt 1 auftaucht. Nur n : die ETH genügt. Auch m (oder $n + m$, was $O(m)$ ist): ohne das Lemma bricht der Schluss zusammen. Wer es pauschal immer oder nie erwähnt, verliert Punkte.

Eine nützliche Ergänzung ist die *Wurzel-Regel*: Wächst die Zielgröße polynomiell, etwa $p = O(m^2)$, dann verbietet die Härte nur noch $2^{o(\sqrt{p})}$ – allgemein $2^{o(\sqrt[p]{p})}$ bei $p = O(m^c)$. Denn erst $\sqrt[p]{p}$ ist wieder von der Ordnung m .

28 Die Schranken im Einzelnen

Jetzt das Muster in Anwendung – für jedes Problem gesondert.

Untere Schranke für Clique

Die Reduktion von 3-SAT auf k -CLIQUE baut pro Literalvorkommen einen Knoten: Bei höchstens drei Literalen je Klausel sind das $|V| = O(m)$ Knoten, dazu $|E| = O(m^2)$ Kanten und die Zielgröße $k = m$. Angenommen, CLIQUE hätte einen $2^{o(|V|)}$ -Algorithmus. Wegen $|V| = O(m)$ wäre die Kette aus Reduktion und Algorithmus ein 3-SAT-Verfahren mit Laufzeit $2^{o(m)} \cdot \text{poly}$ – das verbietet das Sparsification-Lemma. Für die Kantenzahl greift die Wurzel-Regel: Aus $|E| = O(m^2)$ folgt, dass kein $2^{o(\sqrt{|E|})}$ -Algorithmus existiert. Und da $k = m$ direkt die Klauselzahl ist, scheidet auch ein $2^{o(k)}$ -Algorithmus aus.

Untere Schranke für Independent Set

Die Reduktion von CLIQUE auf INDEPENDENT SET ersetzt den Graphen durch sein Komplement und behält k . Dabei bleibt die Knotenzahl identisch, die Kantenzahl bleibt durch $O(|V|^2)$ beschränkt, und k ändert sich nicht. Ein $2^{o(|V|)}$ -Algorithmus für INDEPENDENT SET wäre darum – nach Vorschalten der Komplementbildung – ein $2^{o(|V|)}$ -Algorithmus für CLIQUE, den es nach dem vorigen Abschnitt nicht gibt. Mit derselben Übertragung scheiden $2^{o(\sqrt{|E|})}$ - und $2^{o(k)}$ -Algorithmen aus.

Untere Schranke für Vertex Cover

Die Reduktion von CLIQUE auf VERTEX COVER nutzt ebenfalls den Komplementgraphen, diesmal mit Schranke $k' = n - k$. Auch hier wächst nichts: Knotenzahl gleich, Kantenanzahl $O(|V|^2)$, und $k' \leq |V|$. Ein zu schneller VC-Algorithmus – sei es in $|V|$, in $\sqrt{|E|}$ oder in k' – ließe sich also in einen gleich schnellen Clique-Algorithmus verwandeln, im Widerspruch zur Clique-Schranke. VERTEX COVER hat somit dieselben drei Untergrenzen.

Untere Schranken für Hitting Set

Die Reduktion von 3-SAT liefert $|U| = 2n$ Elemente, $r = n + m$ Mengen und die Schranke $k = n$. Die drei Parameter verhalten sich unterschiedlich – genau deshalb ist dieses Problem das Lehrstück des Themas.

Erstens, die Universumsgröße. Ein Algorithmus in $2^{o(|U|)}$ wäre wegen $|U| = 2n$ einer in $2^{o(n)}$. Vorgeschaltet an die Reduktion entstünde ein 3-SAT-Löser in $2^{o(n)} \cdot \text{poly}$ – das widerspricht unmittelbar der ETH. Das Sparsification-Lemma wird hier nicht angerührt, weil m in der Rechnung nirgends vorkommt.

Zweitens, die Mengenzahl. Ein Algorithmus in $2^{o(r)}$ trifft auf $r = n + m \leq 3m + m = 4m$, ist also einer in $2^{o(m)}$. Die Kette ergäbe einen 3-SAT-Löser in $2^{o(m)} \cdot \text{poly}$ – das verbietet erst das Sparsification-Lemma, denn die ETH allein sagt über m nichts. Hier ist das Zitat des Lemmas Pflicht.

Drittens, die Schranke k . Ein Algorithmus in $2^{o(k)}$ ist wegen $k = n$ einer in $2^{o(n)}$ – derselbe unmittelbare ETH-Widerspruch wie beim ersten Parameter, wieder ohne Lemma.

Untere Schranke für SubSet Sum und Partition

Hier zählt sich die strenge Dezimal-Reduktion aus dem Katalog aus: Sie erzeugt aus einer 3-SAT-Formel nur $2n + 2m$ Zahlen, und mit $n \leq 3m$ sind das höchstens $8m = O(m)$ Stück. Hätte SUBSET SUM einen Algorithmus in $2^{o(\#Zahlen)}$, dann lief die Kette in $2^{o(m)} \cdot \text{poly}$ für 3-SAT – vom Sparsification-Lemma ausgeschlossen. SUBSET SUM hat also keinen subexponentiellen Algorithmus in der Anzahl seiner Zahlen. Die Reduktion auf PARTITION fügt lediglich zwei weitere Zahlen hinzu und verschiebt die Größenordnung nicht – dieselbe Schranke gilt daher auch für PARTITION.

Untere Schranke für 3-Color

Für das Färbungsproblem existiert eine eigens sparsame Konstruktion, die aus einer 3-SAT-Formel einen Graphen mit nur $O(n + m)$ Knoten macht. Wegen $n \leq 3m$ ist das $O(m)$. Ein $2^{o(|V|)}$ -Algorithmus für 3-COLOR ergäbe darum einen $2^{o(m)}$ -3-SAT-Löser – Widerspruch zum Sparsification-Lemma. Für die Kantenanzahl ($O(m^2)$) schließt die Wurzel-Regel zusätzlich jeden $2^{o(\sqrt{|E|})}$ -Algorithmus aus.

Untere Schranke für Set Cover

Der Seitenwechsel zwischen HITTING SET und SET COVER vertauscht die Rollen der beiden Größen: Aus dem Universum wird die Mengenliste und umgekehrt. Die Hitting-Set-Schranken

wandern dadurch mit vertauschten Parametern hinüber: SET COVER hat weder in der Universumsgröße noch in der Mengenzahl einen subexponentiellen Algorithmus – bei der m -abhängigen Größe wieder mit dem Sparsification-Lemma als Schlussstein.

Teil D – Approximation

Wo exakte Lösungen (vermutlich) exponentiell teuer sind, verschiebt man den Anspruch: schnell rechnen, dafür nur *nahe* ans Optimum – aber mit beweisbarer Garantie, wie nahe. Für jeden Algorithmus dieses Teils steht hier seine Garantie samt der Idee, warum sie hält.

29 Was eine Güte-Garantie ist

■ Definition 29.1 Multiplikative Güte

Bezeichne $\text{OPT}(I)$ den optimalen Wert der Instanz I . Ein Algorithmus A *garantiert die Güte* α , wenn für jede Instanz gilt: beim Minimieren $A(I) \leq \alpha \cdot \text{OPT}(I)$, beim Maximieren $\text{OPT}(I) \leq \alpha \cdot A(I)$. *Scharf* heißt die Garantie, wenn es Instanzen gibt, die den Faktor beliebig genau ausreizen.

★ Intuition

Eine Güte ist ein Versprechen über den *schlechtesten* Fall. Güte 2 beim Minimieren: Was auch immer kommt, meine Lösung kostet höchstens das Doppelte des Bestmöglichen. Je näher an 1, desto besser das Versprechen.

Bei manchen Problemen kann man die Genauigkeit sogar frei wählen – gegen entsprechende Rechenzeit. Solche Familien von Algorithmen tragen je nach Preis-Leistungs-Verhältnis einen von drei Namen; wir definieren sie nacheinander.

■ Definition 29.2 PTAS

Ein *polynomielles Approximationsschema* ist eine Familie $(A_\varepsilon)_{\varepsilon>0}$ mit Güte $1 + \varepsilon$, deren Laufzeit für jedes *festgehaltene* ε polynomiell in n ist.

Die Schwäche steckt im Kleingedruckten: ε darf im Exponenten von n auftauchen. Eine Laufzeit wie $n^{1/\varepsilon}$ ist für jedes feste ε polynomiell – aber schon bei $\varepsilon = 0,1$ steht dort n^{10} .

■ Definition 29.3 EPTAS

Ein *effizientes* PTAS hat Laufzeiten der Form $f(1/\varepsilon) \cdot \text{poly}(n)$: Die Abhängigkeit von ε ist aus dem n -Exponenten herausgezogen und in einen eigenen Faktor verbannt.

Der Faktor $f(1/\varepsilon)$ darf groß sein – aber die Skalierung in der Eingabegröße bleibt ein festes Polynom, egal wie genau man es haben will.

■ Definition 29.4 FPTAS

Ein *voll polynomiell* PTAS läuft polynomiell in n und in $1/\varepsilon$ zugleich – etwa in $O(n^3/\varepsilon)$. Mehr kann man von einem Approximationsschema nicht verlangen.

Dazu ein Nachbarbegriff, den man hier braucht: *pseudopolynomiell* nennt man Laufzeiten, die polynomiell in den *Werten* der Eingabezahlen sind. Wegen der Binärkodierung (Teil A) sind solche Werte exponentiell in der Eingabelänge – pseudopolynomiell ist also gerade *nicht* polynomiell im strengen Sinn.

30 TSP ohne Zusatzannahme: hoffnungslos

▲ Satz 30.1 Keine Güte fürs allgemeine TSP

Gäbe es für das TSP mit beliebigen Distanzen einen Polynomialzeit-Algorithmus mit irgendeiner festen Güte α , dann wäre $P = NP$.

Die Idee: Man präpariert die HK-Reduktion mit einem Abgrund. Echte Kanten kosten 1, fehlende aber nicht bloß $|V| + 1$, sondern einen gigantischen Betrag – so groß, dass selbst das α -Fache einer reinen Kanten-Tour billiger bleibt als eine einzige Nicht-Kante. Ein α -approximativer Algorithmus müsste dann bei Graphen *mit* Hamiltonkreis eine Tour unterhalb des Abgrunds abliefern und bei Graphen *ohne* zwangsläufig darüber liegen – an seiner Ausgabe ließe sich HK exakt ablesen. Ein NP-vollständiges Problem wäre in Polynomialzeit gelöst.

Rettung bringt die *Dreiecksungleichung* $d(i, j) \leq d(i, k) + d(k, j)$ – Umwege lohnen nie. Reale Distanzen erfüllen sie ohnehin. Unter dieser Annahme („metrisches TSP“) existieren gleich zwei gute Näherungsverfahren.

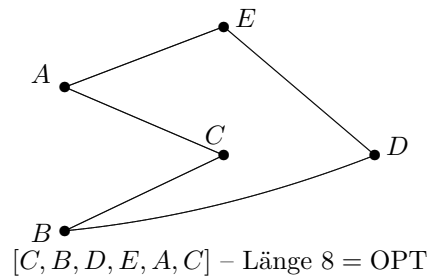
31 ΔTSP_1 : Gerüst, Rundweg, Abkürzung

🗺 Idee: Der Bauplan

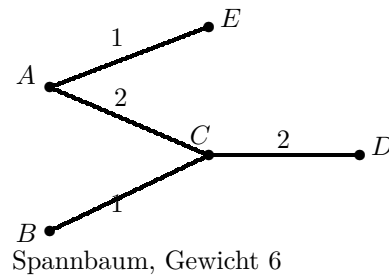
Drei Zutaten: Ein *minimaler Spannbaum* verbindet alle Städte zum kleinstmöglichen Preis. Das *Verdoppeln* seiner Kanten macht jeden Grad gerade, sodass ein *Eulerkreis* existiert – ein Rundweg über alle Kanten. *Abkürzen* verwandelt ihn in eine echte Tour, die jede Stadt nur einmal anläuft.

Wir verfolgen den Ablauf an einem festen Beispiel mit den Städten A, B, C, D, E und den Distanzen $d(A, E) = 1$, $d(B, C) = 1$, $d(A, C) = 2$, $d(C, D) = 2$ (alle übrigen 2 oder 3).

Vorab – die optimale Tour. Zum späteren Vergleich: Die beste Rundreise ist $[C, B, D, E, A, C]$ mit Länge 8. Der Algorithmus kennt sie nicht.



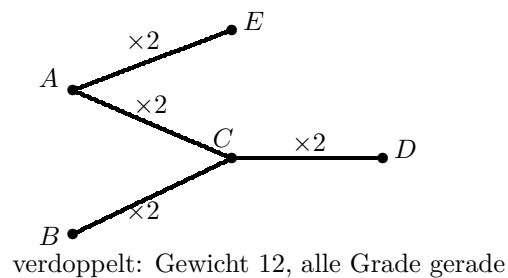
Erster Schritt – der Spannbaum. Der billigste Baum durch alle fünf Städte besteht aus $A-E$ (1), $B-C$ (1), $A-C$ (2) und $C-D$ (2); er wiegt 6.



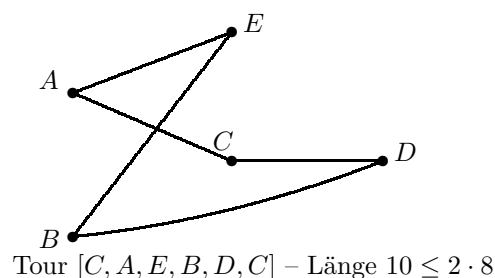
★ Intuition

Der Spannbaum ist der Anker der ganzen Analyse: Er kostet *höchstens* OPT. Denn streicht man aus der optimalen Tour irgendeine Kante, entsteht ein Gerüst, das alle Städte verbindet – und der *minimale* Spannbaum unterbietet jedes solche Gerüst.

Zweiter und dritter Schritt – verdoppeln, Eulerkreis. Jede Baumkante wird zweifach gelegt; nun hat jede Stadt geraden Grad, und ein Rundweg über sämtliche Kantenkopien existiert. Er wiegt $2 \cdot 6 = 12$, zum Beispiel $[C, A, E, A, C, B, C, D, C]$ – mit Wiederholungsbesuchen bei A und C.



Vierter Schritt – abkürzen. Beim Ablufen des Eulerkreises überspringt man jede schon besuchte Stadt und fliegt direkt zur nächsten neuen. Aus $[C, A, E, A, C, B, C, D, C]$ wird so die Tour $[C, A, E, B, D, C]$ der Länge 10.



▲ Satz 31.1 ΔTSP_1 garantiert Güte 2

Drei Ungleichungen tragen den Beweis. *Erstens* wiegt der Spannbaum höchstens OPT (Tour minus Kante ist ein Gerüst). *Zweitens* wiegt der Eulerkreis exakt das Doppelte des Baums, also höchstens 2OPT . *Drittens* macht Abkürzen nichts teurer – die Dreiecksungleichung garantiert, dass der Direktflug zur nächsten neuen Stadt nie länger ist als der Umweg über besuchte. Zusammen: Tourlänge $\leq 2\text{OPT}$. Im Beispiel: 10 bei Optimum 8, Garantie wäre 16.

★ Intuition

Die Garantie ist im Wesentlichen scharf: Es gibt sternförmige Instanzen mit Optimum n , auf denen das Verfahren $2n - 2$ produziert – für großes n beliebig nah am Faktor 2.

32 Christofides: das Matching statt der Verdopplung

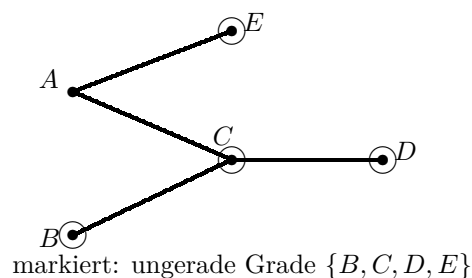
Das Verdoppeln *aller* Kanten ist Verschwendung – schuld an ungeraden Graden sind nur *manche* Knoten. Christofides repariert gezielt.

🔍 Idee: Gezielt reparieren

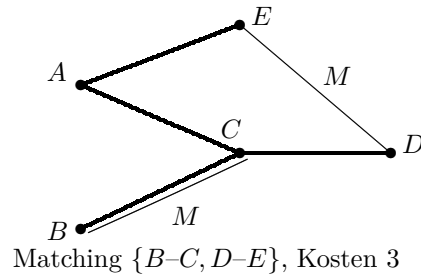
Nur die Knoten mit ungeradem Grad im Spannbaum stören den Eulerkreis. Ihre Anzahl ist immer gerade – man kann sie also paaren. Christofides verbindet sie durch das *billigste perfekte Matching* und erhält so gerade Grade zum halben Preis.

Erster Schritt – Spannbaum wie gehabt. Kanten $A-E$, $B-C$, $A-C$, $C-D$; Gewicht 6.

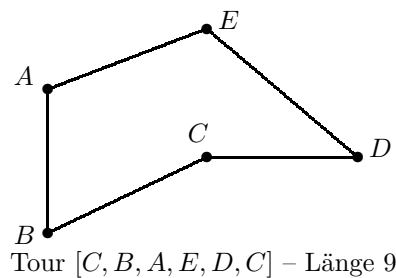
Zweiter Schritt – die Störer finden. Grade im Baum: A hat 2, alle übrigen 1 oder 3 – ungerade sind B, C, D, E (im Bild markiert).



Dritter Schritt – die Störer paaren. Unter den möglichen Paarungen von $\{B, C, D, E\}$ ist $\{B-C, D-E\}$ mit Kosten $1 + 2 = 3$ die billigste. Diese zwei Matchingkanten (dünn) kommen zum Baum – jetzt sind alle Grade gerade.



Vierter Schritt – Eulerkreis und Abkürzung. Aus Baum plus Matching entsteht ein Eulerkreis; abgekürzt ergibt er die Tour $[C, B, A, E, D, C]$ der Länge 9 – einen Schritt näher am Optimum 8 als die 10 von ΔTSP_1 .



▲ Satz 32.1 Christofides garantiert Güte 1,5

Die Tour kostet Baum plus Matching, also höchstens $OPT + d(M)$. Der Kern ist die Schranke $d(M) \leq OPT/2$: Läuft man die *optimale* Tour ab und notiert nur die ungerad-gradigen Knoten in Besuchsreihenfolge, so zerfallen deren Verbindungsstücke in *zwei* perfekte Paarungen – die „geraden“ und die „ungeraden“ Abschnitte im Wechsel. Dank Dreiecksungleichung kosten beide zusammen höchstens OPT ; die billigere also höchstens $OPT/2$. Das *minimale* Matching unterbietet auch diese. Ergebnis: $Tour \leq OPT + OPT/2 = 1,5 OPT$. Auch diese Schranke wird von Beispielfamilien asymptotisch erreicht.

✗ Stolperstein

Zwei beliebte Fehler: Das Matching wird über den *ungerad-gradigen Baumknoten* gebildet – nicht über allen Knoten. Und ohne Dreiecksungleichung platzt der Abkürzungsschritt, genau wie bei ΔTSP_1 .

33 Knapsack, Stufe 1: der pure Greedy

Beim Optimier-Rucksack will man mit Tragkraft B möglichst viel Profit einpacken. Der erste Versuch liegt nahe.

📖 Idee: Greedy nach Profitdichte

Bewerte jeden Gegenstand mit seiner *Dichte* p_i/w_i – Profit pro Kilo. Sortiere absteigend danach und packe der Reihe nach ein, was noch in den Rucksack passt.

▲ Satz 33.1 Greedy allein hat keine Güte-Garantie

Für jede noch so große Zahl α gibt es Instanzen, auf denen Greedy um mehr als den Faktor α am Optimum vorbeipackt.

Das Gegenbeispiel: Zwei Gegenstände bei Tragkraft B – ein Krümel mit Gewicht 1 und Profit 1 (Dichte 1) und ein Brocken mit Gewicht B und Profit $B - 1$ (Dichte knapp unter 1). Greedy greift zuerst zum dichteren Krümel; danach passt der Brocken nicht mehr. Ergebnis: Profit 1 statt $B - 1$. Mit wachsendem B wird das Verhältnis beliebig schlecht – der Fehler ist, dass Dichte nichts über *absolute* Profite sagt.

34 Knapsack, Stufe 2: Modified Greedy

🔍 Idee: Zwei Kandidaten statt einem

Berechne *beides*: die Greedy-Packung *und* den besten Einzelgegenstand. Gib das profitablere von beiden aus.

▲ Satz 34.1 Modified Greedy garantiert Güte 2

Der Schlüssel ist die *gebrochene* Variante des Rucksacks, bei der man Gegenstände auch anteilig einpacken darf. Ihr Optimum OPT_f ist mindestens so groß wie das echte OPT , und es hat eine einfache Gestalt: Greedy-Reihenfolge, bis das erste Stück – das *Split-Item* – nur noch teilweise hineinpasst. Daraus liest man ab:

$$\text{OPT} \leq \text{OPT}_f \leq \text{Greedy-Profit} + p_{\text{split}} \leq \text{Greedy-Profit} + p_{\text{max}} \leq 2 \cdot \max\{\text{Greedy}, p_{\text{max}}\}.$$

Der rechte Ausdruck ist genau das Doppelte der Modified-Greedy-Ausgabe. Der Algorithmus verliert also höchstens den Profit *eines* Gegenstands – und den sichert der zweite Kandidat ab.

35 Knapsack, Stufe 3: Sahni-Schema

🔍 Idee: Kleine Vorauswahl raten, Rest gierig

Für einen Parameter k : Probiere *jede* Teilmenge von höchstens k Gegenständen als festen Grundstock durch, fülle den Rest der Tragkraft mit Greedy und behalte die beste aller so entstandenen Packungen.

▲ Satz 35.1 Sahni ist ein PTAS

Die Güte beträgt $1 + 1/k$ bei Laufzeit $O(n^{k+1})$. Mit $k = 2$ erreicht man Faktor $3/2$, mit $k = 3$ Faktor $4/3$ – beliebig nah an 1, wenn man k wachsen lässt. Der Preis: k wandert in den Exponenten der Laufzeit. Genau das Profil eines PTAS. (Modified Greedy steckt als Spezialfall $k = 1$ mit drin.)

36 Knapsack, Stufe 4: das FPTAS

🔍 Idee: Profite vergrößern, dann exakt rechnen

Es gibt ein exaktes Rucksack-Verfahren per dynamischer Programmierung, dessen Laufzeit von der *Größe der Profitwerte* abhängt – pseudopolynomiell. Skaliert man alle Profite um einen Faktor K herunter (mit Abrunden), werden die Werte klein und das exakte Verfahren schnell. Die Abrundung kostet Genauigkeit – aber kontrollierbar wenig.

▲ Satz 36.1 Das Skalierungsverfahren ist ein FPTAS

Zwei Abschätzungen steuern den Fehler. Nach unten: Die auf den vergrößerten Profiten optimale Packung verliert gegenüber OPT höchstens K pro Gegenstand – durch das Abrunden – also insgesamt höchstens Kn . Nach oben: Ihre echte Profitsumme ist mindestens so groß wie ihr vergrößerter Wert mal K . Wählt man K proportional zu $\varepsilon p_{\max}/n$, dann ist der Verlust Kn höchstens ein ε -Anteil von $p_{\max} \leq \text{OPT}$ – Güte $1 + \varepsilon$. Die Laufzeit des exakten Verfahrens auf den kleinen Werten beträgt $O(n^3/\varepsilon)$: polynomiell in n und $1/\varepsilon$, also ein FPTAS.

37 Scheduling, Regel 1: List Scheduling

🔍 Idee: Immer zur leersten Maschine

Nimm die Jobs in gegebener Reihenfolge und lege jeden auf die Maschine, die gerade am wenigsten zu tun hat.

Das Gantt-Bild zeigt die Schwäche: drei Jobs 1, 1, 2 auf zwei Maschinen.

List Scheduling (Reihenfolge 1, 1, 2):

M_1

1	2
---	---

 Makespan 3

M_2

1

besser:

M_1

2

 Makespan 2

0 1 2 3 Zeit

Die beiden Einsen blockieren beide Maschinen, die späte Zwei stapelt sich obendrauf. Wer die Zwei zuerst platziert, erreicht Makespan 2.

▲ **Satz 37.1** List Scheduling garantiert $2 - \frac{1}{m}$ – scharf

Betrachte die am Ende vollste Maschine und den *letzten* Job J_k , der dort landete. Im Moment seiner Zuteilung war diese Maschine die *leerste* – alle m Maschinen trugen damals also mindestens „Endlast minus p_k “. Summiert ergibt das $\sum_i p_i \geq m(\text{LS} - p_k) + p_k$. Nun die beiden *Universalschranken* jedes Schedules: Das Optimum ist mindestens die Durchschnittslast $\frac{1}{m} \sum_i p_i$, und mindestens so groß wie jeder einzelne Job, insbesondere p_k . Beides eingesetzt liefert $\text{LS} \leq (2 - \frac{1}{m}) \text{OPT}$. Die Schranke wird erreicht: $m(m-1)$ Einer-Jobs, gefolgt von einem Job der Länge m , treiben List Scheduling auf $2m-1$ bei Optimum m .

38 Scheduling, Regel 2: LPT

☞ **Idee: Die Großen zuerst**

Sortiere die Jobs absteigend nach Laufzeit und wende dann List Scheduling an. Die dicken Brocken kommen so aufs leere Feld; die kleinen füllen am Ende nur noch Lücken.

▲ **Satz 38.1** LPT garantiert $\frac{4}{3} - \frac{1}{3m}$ – scharf

Der Beweis arbeitet mit einem *kleinsten Gegenbeispiel*: einer Instanz mit minimaler Jobzahl, die die Schranke verletzt. Dort muss der zuletzt platzierte – wegen der Sortierung kleinste – Job den Makespan setzen; andernfalls ließe er sich streichen und ein kleineres Gegenbeispiel bliebe übrig. Aus der verletzten Schranke errechnet man, dass dieser kleinste Job länger als $\text{OPT}/3$ dauert. Dann aber passt in den *optimalen* Plan höchstens ein Job-Duo pro Maschine – drei Jobs über $\text{OPT}/3$ sprengten sie. Pläne mit höchstens zwei Jobs pro Maschine lassen sich nun durch schrittweises *Tauschen* umsortieren: Je zwei Jobs wechseln so die Maschinen, dass die Anordnung der LPT-Reihenfolge angeglichen wird, wobei bei nur zwei Jobs pro Maschine kein Tausch die vollste Maschine voller macht. Am Ende steht der LPT-Plan mit Optimal-Makespan – das Gegenbeispiel widerspricht sich selbst, also existiert keines.

39 Parallel-Task-Scheduling: Jobs, die mehrere Maschinen brauchen

Eine Verallgemeinerung des Scheduling, die in neueren Klausuren auftaucht: Jobs können mehrere Maschinen *gleichzeitig* belegen.

■ Definition 39.1 Parallel-Task-Scheduling

Gegeben n Jobs und m Maschinen. Jeder Job j hat eine Ausführungszeit p_j und belegt während dieser Zeit *gleichzeitig* $q_j \in [m]$ Maschinen. Gesucht ist ein Schedule ohne Überlappungen auf derselben Maschine mit minimalem Makespan.

Auch hier arbeitet ListScheduling: Es nimmt die Jobs der Reihe nach und platziert jeden zum *frühestmöglichen* Zeitpunkt auf den q_j am wenigsten belasteten Maschinen.

● Beispiel 39.2 ListScheduling mit Maschinenbedarf

$m = 3$; Jobs als (p_j, q_j) : $(1, 1)$, $(1, 3)$, $(2, 2)$, $(3, 1)$. Die Maschinenlasten C entwickeln sich so: Job 1 startet bei 0 auf einer Maschine – $C = (1, 0, 0)$. Job 2 braucht alle drei Maschinen; die höchste Last ist 1, also läuft er in $[1, 2]$ – $C = (2, 2, 2)$. Job 3 belegt zwei Maschinen ab Zeit 2, läuft in $[2, 4]$ – $C = (4, 4, 2)$. Job 4 nimmt die leerste Maschine ab Zeit 2, läuft in $[2, 5]$ – $C = (5, 4, 4)$. Makespan: 5.

Für den Sonderfall $q_j = 1$ ist das genau das klassische ListScheduling mit seiner Güte $2 - \frac{1}{m}$. Interessant wird es, wenn alle Maschinenbedarfe in einem festen Band liegen.

▲ Satz 39.3 Güte im Band $\frac{m}{k+1} < q_j \leq \frac{m}{k}$

Liegen alle Bedarfe in diesem Band (für ein festes $k \geq 2$), so gilt $LS(I) \leq OPT(I) + (1 - \frac{1}{k})p_{\max} \leq (2 - \frac{1}{k})OPT(I)$.

Die Idee: Betrachte einen Job j , der den Makespan bestimmt, mit Startzeit s . Zu jedem Zeitpunkt vor s waren weniger als $q_j \leq \frac{m}{k}$ Maschinen frei – sonst hätte ListScheduling den Job früher platziert. Also waren mehr als $\frac{(k-1)m}{k}$ Maschinen belegt. Da jeder Job höchstens $\frac{m}{k}$ Maschinen belegt, liefen mindestens k Jobs; da jeder *mehr* als $\frac{m}{k+1}$ belegt, passen keine $k+1$ nebeneinander – es liefen also stets *genau* k Jobs. Das ergibt $k \cdot s \leq \sum_i p_i - p_j$. Im Optimum laufen ebenfalls höchstens k Jobs parallel, also $\sum_i p_i \leq k \cdot OPT$. Zusammen folgt $s \leq OPT - \frac{1}{k}p_j$ und damit $LS = s + p_j \leq OPT + (1 - \frac{1}{k})p_j$.

★ Intuition

Der wichtigste Spezialfall ist $k = 2$, also $\frac{m}{3} < q_j \leq \frac{m}{2}$: Vor dem Start des entscheidenden Jobs laufen immer genau zwei Jobs, und es folgt $LS \leq OPT + \frac{1}{2}p_{\max}$. Das Argument ist eine direkte Verallgemeinerung der beiden Universalschranken des klassischen ListScheduling – nur zählt man hier die *parallel laufenden Jobs* statt der Maschinen.

40 Strip Packing: der NFDH-Algorithmus

Ein Packungsproblem mit demselben Approximations-Geist: Rechtecke in einen Streifen packen.

■ Definition 40.1 Strip Packing

Gegeben eine Folge von Rechtecken $r_1, \dots, r_n$ mit Breiten $w(r_i) \leq 1$ und Höhen $h(r_i)$, absteigend nach Höhe sortiert ($h(r_i) \geq h(r_{i+1})$), sowie ein Streifen der Breite 1 mit unbeschränkter Höhe. Gesucht ist eine überlappungsfreie Packung mit minimaler Gesamthöhe.

☞ Idee: NFDH (Next Fit Decreasing Height)

NFDH packt *stufenweise*: Die Rechtecke werden der Reihe nach links-bündig auf die aktuelle Stufe gelegt. Passt eines nicht mehr in die Breite, wird eine neue Stufe eröffnet – auf Höhe des *ersten* Rechtecks der alten Stufe (das wegen der Sortierung das höchste ist).

● Beispiel 40.2 NFDH gegen die optimale Packung

Fünf Rechtecke der Höhe 1 mit Breiten $\frac{1}{2}, \frac{1}{4}, \frac{1}{2}, \frac{1}{4}, \frac{1}{2}$. NFDH links: Stufe 1 fasst $\frac{1}{2} + \frac{1}{4}$; das nächste $\frac{1}{2}$ passt nicht mehr – Stufe 2 mit $\frac{1}{2} + \frac{1}{4}$; das letzte $\frac{1}{2}$ eröffnet Stufe 3. Höhe 3. Optimal rechts: $\frac{1}{2} + \frac{1}{2}$ und $\frac{1}{2} + \frac{1}{4} + \frac{1}{4}$ – Höhe 2.

NFDH: Höhe 3

$\frac{1}{2}$	
$\frac{1}{2}$	$\frac{1}{4}$
$\frac{1}{2}$	$\frac{1}{4}$

Optimal: Höhe 2

$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$
$\frac{1}{2}$	$\frac{1}{2}$	

Die Rate beträgt hier $\frac{3}{2}$.

★ Intuition

Der schlimmste Fall treibt die Rate gegen 2: Nimm $2t$ Rechtecke der Höhe 1 mit Breiten abwechselnd 1 und ε . Bei NFDH füllt jedes Breite-1-Rechteck seine Stufe komplett; jedes ε -Rechteck eröffnet eine neue Stufe, auf die das nächste Breite-1-Rechteck nicht mehr passt – $2t$ Stufen. Optimal packt man alle ε -Rechtecke zusammen auf eine einzige Stufe: Höhe $t + 1$. Die Rate $\frac{2t}{t+1}$ geht gegen 2.

▲ Satz 40.3 NFDH garantiert $2 \cdot \text{OPT} + 1$ (für Höhen 1)

Seien $W_1, \dots, W_S$ die belegten Breiten der NFDH-Stufen; die Packungshöhe ist S .

Die Idee: Für benachbarte Stufen gilt $W_i + W_{i+1} > 1$ – denn das erste Rechteck der Stufe $i + 1$ passte nicht mehr auf Stufe i , und seine Breite steckt in W_{i+1} . Summiert man diese Ungleichungen über $i = 1, \dots, S - 1$, steht links höchstens $2 \sum_i W_i$ und rechts $S - 1$:

$$2 \sum_i W_i > S - 1.$$

Die Gesamtfläche der Rechtecke ist $\sum_i W_i$ (alle Höhen 1), und sie muss in jede Packung der Breite 1 hineinpassen: $\text{OPT} \geq \sum_i W_i$. Also $2 \text{OPT} > S - 1$, das heißt $\text{NFDH} = S \leq 2 \text{OPT} + 1$.

★ Intuition

Für gemischte Höhen $h(r_i) \leq 1$ bleibt die Schranke bestehen. Die Idee: Die Stufenhöhen $H_1 \geq H_2 \geq \dots$ sind die Höhen der jeweils ersten Rechtecke, und jedes Rechteck der Stufe i ist mindestens H_{i+1} hoch – die Fläche der Stufe i ist also mindestens $W_i \cdot H_{i+1}$. Zusammen mit $W_i + W_{i+1} > 1$ ergibt sich $\sum_{i \geq 2} H_i \leq 2 \cdot \text{Fläche} \leq 2 \text{OPT}$, und die erste Stufe steuert höchstens $H_1 \leq 1$ bei: wieder $\text{NFDH} \leq 2 \text{OPT} + 1$.

41 MAX-3-SAT: zwei Versuche reichen für Güte 2

🗨 Idee: Alles an, alles aus

Gesucht ist eine Belegung, die möglichst viele Klauseln erfüllt. Der Algorithmus testet nur zwei Kandidaten: die Belegung, die alle Variablen auf wahr setzt, und die, die alle auf falsch setzt. Die bessere gewinnt.

▲ Satz 41.1 Der Zwei-Versuche-Algorithmus hat Güte 2

Jede Klausel enthält mindestens ein Literal. Ein positives Literal wird von der Alles-wahr-Belegung erfüllt, ein negatives von der Alles-falsch-Belegung – folglich erfüllt *jede* Klausel mindestens einen der beiden Kandidaten. Zählt man zusammen, erfüllen beide Kandidaten gemeinsam mindestens m Klauseln; der bessere allein also mindestens $m/2$. Mehr als m kann auch das Optimum nicht erfüllen, also liefert der Algorithmus mindestens die Hälfte des Optimums. Scharf ist das auch: Ein Klauselpaar, von dem eines nur positive und eines nur negative Literale enthält, ist gemeinsam erfüllbar ($\text{OPT} = 2$), aber jeder der beiden Kandidaten schafft nur eine der Klauseln.

42 Vertex Cover: die Beide-Endpunkte-Regel

Idee: Kante schnappen, beide Enden nehmen

Solange noch eine unbedeckte Kante existiert: Nimm *beide* Endpunkte in die Lösung und streiche alle dadurch bedeckten Kanten. Wiederhole.

▲ Satz 42.1 Die Regel garantiert Güte 2

Nenne A die Menge der geschnappten Kanten; die Ausgabe hat $2|A|$ Knoten. Zwei geschnappte Kanten können keinen Endpunkt teilen – sonst wäre die später geschnappte beim Schnappen der früheren schon bedeckt gewesen. A ist also ein *Matching*: lauter knotenfremde Kanten. Jedes Vertex Cover – auch das optimale – muss jede dieser $|A|$ Kanten mit einem *jeweils eigenen* Knoten bedecken, braucht also mindestens $|A|$ Knoten. Die Ausgabe mit $2|A|$ Knoten ist demnach höchstens doppelt so groß wie das Optimum.

★ Intuition

Damit sind alle drei Grundmuster der Güte-Beweise versammelt: das Optimum von unten durch eine *Struktur* abschätzen (hier: das Matching), ein *Zählargument* über komplementäre Kandidaten (MAX-3-SAT), und die *Universalschranken* Durchschnittslast und Maximaljob (Scheduling). Fast jeder Güte-Beweis des Kurses ist eine Variation eines dieser drei.

43 Schlusswort

Damit liegt das ganze Fundament vor dir. Du kennst für jedes Problem des Kurses die Definition, den NP-Nachweis, die Herkunft seiner Härte und ihre Weitergabe. Du kannst untere Schranken aus der ETH herleiten und weißt bei jedem Parameter, ob das Sparsification-Lemma gebraucht wird. Und du kennst die Approximationsalgorithmen samt der Ideen hinter ihren Garantien.

☆ Aha

Wenn du ein einziges Bild mitnimmst, dann dieses: Alles hängt an der *Reduktion*. Sie trägt die Härte von SAT durch die gesamte Landkarte, sie macht aus der ETH konkrete Zahlen, und ihr Scheitern beim allgemeinen TSP zeigt, wo nicht einmal Näherung hilft. Wer Reduktionen lesen und bauen kann, hat den Kurs verstanden.

Und jetzt? Verständnis wird durch Anwendung zu Können. Nimm dir die Präsenz-, Haus- und Klausuraufgaben vor und führe selbst aus, was hier als Idee steht – Reduktionen mit beiden Richtungen, Schranken-Herleitungen, Worst-Case-Konstruktionen. Viel Erfolg dabei.