

Warum manche Probleme schwer sind

Ein Anfänger-Guide zu Komplexitätstheorie und Approximation

Analyse von Algorithmen und Komplexität – CAU Kiel, Sommersemester 2026

*„Manche Probleme lösen wir in Sekunden.
Bei anderen läuft der Computer bis zum Hitzetod des Universums.
Dieser Guide erklärt dir, woran das liegt – und was man dann tut.“*

An wen sich dieser Guide richtet. An dich, wenn du gerade erst anfängst. Du brauchst keine Vorkenntnisse in Komplexitätstheorie. Was du brauchst: Neugier und die Bereitschaft, jedes Kapitel wirklich durchzuarbeiten – nicht zu überfliegen.

Wie du liest. Der Guide erzählt eine *Geschichte*, keine Liste. Jedes Kapitel beginnt mit einer Frage, die sich aus dem vorigen ergibt. Die farbigen Kästen haben feste Rollen:

★ Intuition	die Idee in einfachen Worten
🖼️ Analogie	ein Bild aus dem Alltag
■ Definition	die präzise Fassung
▲ Satz	ein Ergebnis mit kurzer Beweis <i>idee</i>
● Beispiel	durchgerechnet, mit echten Zahlen
☆ Aha	die Pointe, die alles zusammenzieht
✗ Stolperstein	der Fehler, den fast alle machen
➤ Zusammenhang	wie es mit dem Rest verbunden ist
✓ Selbst-Check	prüfe dich, bevor du weiterliest

Am Ende verstehst du *alle* Inhalte der Vorlesung und siehst, wie sie zusammenhängen. Der Schritt danach: Präsenz-, Haus- und Klausuraufgaben rechnen, um Routine zu bekommen. Dieser Guide gibt dir das *Warum*, die Aufgaben das *Können*.

Inhalt

1	Ein Problem, das sich wehrt	3
2	Was heißt eigentlich „schnell“?	5
3	Prüfen ist leichter als Lösen	7
4	„Mindestens so schwer wie“ – Reduktionen	10
5	Die härtesten Probleme	12
6	Der Problem-Zoo	14
7	Wie schwer <i>genau</i> ? Die ETH	18
8	Damit leben: gute Näherungen	20
9	Jenseits von NP: das Halteproblem	25
10	Das große Ganze	25

1 Ein Problem, das sich wehrt

★ In diesem Kapitel

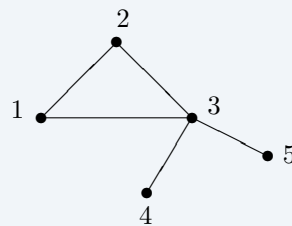
Wir treffen ein Problem, das einfach aussieht – und trotzdem jeden bekannten Computer in die Knie zwingt. Daran verstehst du, warum es eine ganze Theorie über „schwere“ Probleme gibt.

1.1 Freunde auf einer Party

Stell dir eine Party mit einigen Gästen vor. Manche kennen sich, manche nicht. Du fragst dich: *Gibt es eine Gruppe von k Gästen, die sich alle gegenseitig kennen?* Eine solche Gruppe, in der jeder jeden kennt, nennen wir eine *Clique*.

Das lässt sich als *Graph* zeichnen: Jeder Gast ist ein Punkt (ein *Knoten*), und eine Linie (eine *Kante*) zwischen zwei Punkten heißt „die beiden kennen sich“.

● Beispiel 1.1 Eine Clique mit dem Auge finden



Fünf Gäste. Die Gruppe $\{1, 2, 3\}$ ist eine Clique: 1–2, 1–3 und 2–3 sind alle da. Also kennen sich alle drei gegenseitig. Die Gruppe $\{2, 3, 4\}$ ist *keine* Clique – zwischen 2 und 4 fehlt die Kante. Es gibt hier keine Clique aus vier Gästen.

Bei fünf Gästen siehst du die Antwort sofort. Aber ein Computer „sieht“ nicht. Er muss rechnen. Und ab ein paar hundert Gästen wird selbst für den schnellsten Rechner der Welt genau das zum Problem.

1.2 Der naive Weg – und warum er explodiert

Wie würde ein Computer eine k -Clique suchen? Der naheliegende Weg: *probiere alle Gruppen aus k Gästen durch* und prüfe für jede, ob sich wirklich alle kennen.

Das klingt vernünftig. Rechnen wir nach, wie viele Gruppen das sind. Bei n Gästen und Gruppengröße $k = n/2$ gibt es „ n über $n/2$ “ viele Gruppen – und das sind mindestens $2^{n/2}$.

● Beispiel 1.2 Wie schnell $2^{n/2}$ wächst

Gäste n	Gruppen $\geq 2^{n/2}$	anschaulich
20	$\geq 1\,024$	ein Wimpernschlag
40	$\geq 1\,048\,576$	eine Sekunde
100	$\geq 2^{50} \approx 10^{15}$	über einen Monat bei $10^9/s$
200	$\geq 2^{100} \approx 10^{30}$	länger als das Alter des Universums

Jeder zusätzliche Gast *verdoppelt* ungefähr den Aufwand. Das ist der Unterschied zwischen „machbar“ und „unmöglich“.

★ Intuition

Wenn der Aufwand sich bei jedem weiteren Element *verdoppelt*, sprechen wir von *exponentiellem* Wachstum. Exponentielles Wachstum ist der natürliche Feind jeder Berechnung: Es macht schon mittelgroße Eingaben unlösbar. Kein schnellerer Rechner rettet dich – du müsstest ihn selbst verdoppeln, nur um *einen* Gast mehr zu schaffen.

▲ Satz 1.3 Der naive Clique-Algorithmus ist exponentiell

Für $k = |V|/2$ braucht der naive Algorithmus mindestens $2^{|V|/2}$ Schritte.

Beweisidee: Es gibt $\binom{n}{n/2} \geq 2^{n/2}$ Gruppen zu prüfen, und schon das Durchprobieren so vieler Gruppen kostet exponentiell viel Zeit.

1.3 Die Frage, um die es geht

Vielleicht ist der naive Weg einfach dumm. Vielleicht gibt es einen cleveren Trick, der die Clique *schnell* findet, ohne alle Gruppen durchzuprobieren.

Für das Cliquenproblem hat bis heute *niemand* so einen Trick gefunden. Aber – und das ist der Kern – niemand hat auch *bewiesen*, dass es keinen gibt. Diese Lücke zwischen „wir kennen keinen schnellen Weg“ und „es gibt keinen schnellen Weg“ ist der Ausgangspunkt der ganzen Vorlesung.

☆ Aha

Die Komplexitätstheorie beantwortet nicht die Frage „Wie löse ich dieses Problem schnell?“. Sie beantwortet die tiefere Frage: „Ist dieses Problem *überhaupt* schnell lösbar – oder gehört es zu einer Familie von Problemen, bei denen alle gemeinsam scheitern?“

✓ Selbst-Check

Bevor du weiterliest: Warum hilft ein doppelt so schneller Computer beim naiven Clique-Algorithmus fast nichts?

Antwort: Weil schon *ein* Gast mehr den Aufwand verdoppelt. Der schnellere Computer verschafft dir also gerade genug für einen einzigen zusätzlichen Gast – das Problem wächst schneller, als Hardware je aufholen kann.

2 Was heißt eigentlich „schnell“?

★ In diesem Kapitel

Wir machen „schnell“ und „langsam“ präzise. Am Ende hast du die erste wichtige Klasse verstanden: P, die Probleme, die wir effizient lösen können.

2.1 Laufzeit hängt von der Eingabegröße ab

Ein Algorithmus ist nicht einfach „schnell“. Er ist schnell *relativ zur Größe seiner Eingabe*. Zehn Zahlen zu sortieren ist leicht; zehn Milliarden zu sortieren dauert. Wir messen deshalb die *Laufzeit als Funktion der Eingabelänge n* .

■ Definition 2.1 Laufzeit

$T_A(x)$ ist die Anzahl der Rechenschritte von Algorithmus A auf Eingabe x . Die *Worst-Case-Laufzeit* bei Eingabelänge n ist die schlimmste über alle Eingaben dieser Länge:

$$T_A(n) = \max\{T_A(x) \mid |x| = n\}.$$

Wir schauen bewusst auf den *schlechtesten* Fall. Ein Algorithmus, der „meistens“ schnell ist, aber gelegentlich explodiert, ist keine verlässliche Garantie.

2.2 Polynomiell gegen exponentiell – die entscheidende Grenze

Es gibt viele Wachstumsarten, aber die eine Grenze, auf die alles hinausläuft, ist: *polynomiell* gegen *exponentiell*.

- *Polynomiell* heißt: die Laufzeit ist $O(n^d)$ für eine feste Zahl d – also n , n^2 , n^3 , ...
- *Exponentiell* heißt: etwas wie 2^n , das sich bei jedem Schritt verdoppelt.

● Beispiel 2.2 Der Graben zwischen n^3 und 2^n

n	n^3 (polynomiell)	2^n (exponentiell)
10	1 000	1 024
20	8 000	rund 1 Million
50	125 000	rund 10^{15}
100	1 000 000	rund 10^{30}

Bei $n = 10$ sind beide harmlos. Bei $n = 100$ ist n^3 immer noch eine Millionstel Sekunde – 2^n dagegen jenseits von allem. *Diese Kluft ist der Grund, warum wir polynomiell mit „effizient“ gleichsetzen.*

★ Intuition

„Polynomiell = effizient“ ist eine *Konvention*, kein Naturgesetz. Ein n^{100} -Algorithmus wäre praktisch nutzlos. Aber die Konvention funktioniert erstaunlich gut: Polynomielle Algorithmen aus der Praxis haben fast immer kleine Exponenten (n, n^2, n^3), und die Grenze „polynomiell“ bleibt stabil, egal welchen realistischen Computer man zugrunde legt.

■ Definition 2.3 Die Klasse P

P ist die Menge aller Entscheidungsprobleme, die ein Algorithmus in *polynomieller* Zeit löst:

$$P = \{ L \mid \exists \text{ Algorithmus } A, T_A(n) = O(n^d), d = O(1), A \text{ löst } L \}.$$

In Worten: die Probleme, die wir *effizient* lösen können.

Beispiele für Probleme in P: Sortieren, kürzeste Wege im Graphen (Dijkstra), minimale Spannbäume, das Finden eines Matchings. Für all das kennen wir schnelle, polynomielle Algorithmen.

2.3 Warum wir Probleme als Ja/Nein-Fragen schreiben

Ein technischer, aber wichtiger Punkt. Die Theorie behandelt fast immer *Entscheidungsprobleme* – Fragen mit Antwort „Ja“ oder „Nein“. Statt „Wie groß ist die größte Clique?“ fragt man: „Gibt es eine Clique mit mindestens k Knoten?“

🗨️ Analogie

Das ist wie die Frage „Ist dieses Gewicht schwerer als 10 kg?“ statt „Wie schwer ist es genau?“. Wenn du die Ja/Nein-Frage für jede Schranke beantworten kannst, kennst du am Ende auch den genauen Wert – du tastest ihn ein. Deshalb ist die Entscheidungsvariante „gleich schwer“ wie das Optimierungsproblem, aber viel einfacher zu vergleichen und zu klassifizieren.

Um über „alle Probleme“ sauber reden zu können, kodiert man jede Eingabe als *Wort* über einem Alphabet, und ein Problem wird zur *Menge der Ja-Wörter*.

■ Definition 2.4 Alphabet, Wort, Sprache, Entscheidungsproblem

- Ein *Alphabet* Σ ist eine endliche Symbolmenge (z. B. $\{0, 1\}$). Σ^* sind alle endlichen Wörter darüber; $|x|$ ist die Länge von x .
- Eine *Sprache* ist eine Teilmenge $L \subseteq \Sigma^*$.
- Ein *Entscheidungsproblem* identifiziert man mit der Sprache L seiner Ja-Instanzen: A löst es, wenn $A(x) = 1$ für $x \in L$ und $A(x) = 0$ sonst.

Du musst diese Kodierbrille nicht ständig vor Augen haben. Merke dir nur: *Ein Problem = die Menge seiner Ja-Eingaben*. Beim Cliquesproblem ist das die Menge aller Paare (G, k) , für die G wirklich eine k -Clique besitzt.

✗ Stolperstein

Zahlen sind binär kodiert – und dadurch „riesig“. Eine Zahl K hat als Eingabe nur die Länge $O(\log K)$ (ihre Ziffern), ist als *Wert* aber exponentiell größer. Ein Algorithmus, der bis K zählt, ist also *nicht* polynomiell in der Eingabelänge. Genau daran hängt später, warum SUBSET SUM schwer ist und was „pseudopolynomiell“ bedeutet. Behalte es im Hinterkopf.

✓ Selbst-Check

Ist ein Algorithmus mit Laufzeit n^5 „effizient“ im Sinne der Theorie?

Antwort: Ja. n^5 ist polynomiell ($d = 5$ ist eine feste Zahl), also liegt das Problem in P. In der Praxis wäre n^5 vielleicht langsam – aber die Theorie zieht die Grenze bei polynomiell gegen exponentiell, und auf dieser Skala ist n^5 klar auf der guten Seite.

3 Prüfen ist leichter als Lösen

★ In diesem Kapitel

Wir entdecken die vielleicht wichtigste Beobachtung der ganzen Theorie: Für viele schwere Probleme ist es zwar hart, eine Lösung zu *finden* – aber kinderleicht, eine vorgelegte Lösung zu *prüfen*. Das führt uns zur Klasse NP.

3.1 Die Sudoku-Beobachtung

🧠 Analogie

Ein schweres Sudoku zu *lösen* kann dich eine halbe Stunde kosten. Aber wenn dir jemand ein ausgefülltes Gitter hinlegt und fragt „stimmt das?“, prüfst du es in einer Minute: jede Zeile, jede Spalte, jeder Block. *Lösen ist schwer, Prüfen ist leicht*. Genau diese Asymmetrie steckt hinter NP.

Übertragen auf die Clique: Eine Clique zu *finden* ist mühsam. Aber wenn dir jemand eine

Gruppe von Gästen nennt und behauptet „die bilden eine Clique“, prüfst du das schnell – du schaust einfach nach, ob zwischen jedem Paar eine Kante liegt.

Diese „vorgelegte Lösung“ bekommt einen Namen: *Zertifikat*.

3.2 Zertifikat und Verifizierer

■ Definition 3.1 Verifizierer und Zertifikat

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

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

Das c heißt *Zertifikat* (oder Zeuge). Der Verifizierer ist *polynomiell*, wenn er für jede Ja-Instanz ein Zertifikat gibt, das er in polynomieller Zeit prüft.

Lies die Äquivalenz genau: Für eine *Ja*-Instanz gibt es ein Zertifikat, das A überzeugt. Für eine *Nein*-Instanz gibt es *kein* solches Zertifikat – egal, was man A vorlegt, er sagt Nein.

■ Definition 3.2 Die Klasse NP

NP ist die Menge aller Probleme, die einen *polynomiellen Verifizierer* besitzen.

★ Intuition

NP ist „effizient überprüfbar“ – nicht „effizient lösbar“. Der Name ist Programm: Eine Lösung zu finden mag exponentiell teuer sein, aber *hätte* man sie, könnte man sie billig kontrollieren. Das Zertifikat ist die abgekürzte Antwort auf „und woher weiß ich, dass das stimmt?“.

● Beispiel 3.3 Zertifikate für vier Probleme

Problem	Zertifikat c	Prüfung in Polyzeit
k -CLIQUE	die Knotengruppe C	alle Paare verbunden? $ C \geq k$?
SAT	eine Belegung ψ	macht ψ die Formel wahr?
VERTEX COVER	die Knotenmenge C	jede Kante getroffen? $ C \leq k$?
SUBSET SUM	die Teilmenge S	ist $\sum_{j \in S} c_j = K$?

In jeder Zeile ist das Zertifikat kurz (polynomiell lang) und die Prüfung einfach. Genau das macht diese Probleme zu Mitgliedern von NP.

3.3 Die zweite Brille: Raten

Es gibt eine zweite, gleichwertige Art, NP zu sehen – über das *Raten*. Stell dir einen Algorithmus vor, der an jeder Verzweigung *gleichzeitig alle Möglichkeiten* ausprobieren darf. So ein Algorithmus heißt *nichtdeterministisch*.

★ Intuition

Der nichtdeterministische Algorithmus arbeitet nach dem Muster: *rate die Lösung, dann prüfe sie*. Er „rät“ in Polynomialzeit eine Lösungskandidatin und prüft sie deterministisch. Er akzeptiert, wenn *mindestens ein* Rateweg zu „Ja“ führt. Für eine Nein-Instanz führt *kein* Weg zum Ziel.

Formal beschreibt man das mit einer nichtdeterministischen Turingmaschine. Du musst die Maschine nicht auswendig können, aber einmal gesehen haben:

■ Definition 3.4 Nichtdeterministische Turingmaschine (NDTM), knapp

Eine NDTM darf zu jedem Zustand-Symbol-Paar *mehrere* (oder keine) Folgeschritte haben – sie verzweigt. Sie *akzeptiert* eine Eingabe, wenn *irgendein* Rechenweg in einem Endzustand hält, und zwar nach polynomiell vielen Schritten. Hat sie überall höchstens einen Folgeschritt, ist sie gewöhnlich (deterministisch) – das ist der Fall, der P definiert.

▲ Satz 3.5 Beide Sichten beschreiben dasselbe NP

Die Definition über Verifizierer und die über NDTMs ergeben genau dieselbe Klasse. *Beweisidee:* Die Folge der Rateentscheidungen einer NDTM *ist* ein Zertifikat. Umgekehrt rät eine NDTM zuerst das Zertifikat und simuliert dann den Verifizierer. „Raten“ und „ein Zertifikat bekommen“ sind zwei Namen für dasselbe.

3.4 P steckt in NP – und die Millionenfrage

▲ Satz 3.6 $P \subseteq NP$

Jedes effizient *lösbare* Problem ist auch effizient *überprüfbar*.

Beweisidee: Wenn du ein Problem selbst schnell lösen kannst, brauchst du gar kein Zertifikat – du ignorierst es und rechnest die Antwort direkt aus. Der Löser ist also ein (besonders fauler) Verifizierer.

Die große offene Frage ist die *Umkehrung*: Ist auch alles, was man schnell *prüfen* kann, schnell *lösbar*? Also gilt $P = NP$?

☆ Aha

Ob $P = NP$ gilt, weiß bis heute niemand. Es ist eines der sieben Millennium-Probleme, mit einer Million Dollar Preisgeld. Die meisten Forscher glauben $P \neq NP$ – also dass Prüfen *echt* leichter ist als Lösen. Beweisen konnte es keiner. *Alles Folgende in diesem Guide – NP-Vollständigkeit, ETH, Approximation – ist der geschickte Umgang mit genau dieser Unwissenheit.*

► Zusammenhang

Halte die drei Begriffe auseinander, sie kommen ständig wieder:

- P – effizient *lösbar* (finden ist leicht).
- NP – effizient *überprüfbar* (prüfen ist leicht).
- $P \subseteq NP$ ist bewiesen; ob Gleichheit gilt, ist offen.

Im nächsten Kapitel bauen wir das Werkzeug, mit dem man Probleme *innerhalb* von NP nach Schwierigkeit vergleicht – ohne $P = NP$ zu kennen.

✓ Selbst-Check

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

Antwort: Der Verifizierer bekommt das Zertifikat schon *geschenkt* und muss es nur prüfen. Der Löser hat niemanden, der ihm die Lösung vorlegt – er muss sie selbst finden, und das kann exponentiell viele Kandidaten bedeuten. Prüfen startet einen Schritt weiter als Lösen.

4 „Mindestens so schwer wie“ – Reduktionen

★ In diesem Kapitel

Wir bauen das wichtigste Werkzeug der ganzen Theorie: die *Reduktion*. Mit ihr vergleicht man Probleme nach Schwierigkeit – ohne für eines von beiden einen Algorithmus zu kennen. Das klingt wie Zauberei, ist aber eine schlichte Idee.

4.1 Ein Problem mit einem anderen lösen

Angenommen, du hast einen fertigen Löser für Problem B . Und du stehst vor Problem A , für das du keinen Löser hast. Wenn du jede A -Frage so *umformen* kannst, dass der B -Löser sie beantwortet, dann hast du A gelöst – geschenkt, über den Umweg B .

👉 Analogie

Du willst wissen, ob eine Strecke länger als eine Meile ist, hast aber nur ein Lineal in Kilometern. Kein Problem: Du rechnest „länger als 1 Meile?“ um in „länger als 1,609 km?“ und misst das. Die *Umrechnung* ist billig, das Messen erledigt das km-Lineal. Genau das ist eine Reduktion: *billiges Umformen* plus *fremder Löser*.

■ Definition 4.1 Polynomielle Reduktion $A \leq_p B$

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

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

Sie ist *polynomiell*, wenn f in polynomieller Zeit berechenbar ist (dann ist $f(w)$ auch nur polynomiell groß). Schreibweise: $A \leq_p B$.

Lies die Bedingung genau: f muss *Ja* auf *Ja* und *Nein* auf *Nein* abbilden. Eine Ja-Instanz von A wird zu einer Ja-Instanz von B – und eine Nein-Instanz von A zu einer Nein-Instanz von B . Beide Richtungen.

★ Intuition

$A \leq_p B$ heißt: „ B ist *mindestens so schwer* wie A .“ Denn wer B effizient lösen kann, löst damit auch A effizient (erst f rechnen, dann den B -Löser fragen). Die Schwierigkeit „fließt“ entlang des Pfeils: Ist A schwer, muss B es auch sein.

4.2 Die Richtung – der Fehler, der alle Punkte kostet

✗ Stolperstein

Die Reduktionsrichtung ist der häufigste und teuerste Fehler. Du willst zeigen, dass ein *neues* Problem Y schwer ist. Dann reduzierst du ein *bekannt schweres* Problem X auf Y , also $X \leq_p Y$ („bekannt $\leq_p$ neu“). Die Reduktion nimmt eine X -Instanz und baut daraus eine Y -Instanz. *Niemals* umgekehrt.

Warum diese Richtung? Weil du die Schwere von X auf Y *übertragen* willst. „ $X \leq_p Y$ “ bedeutet „ Y mindestens so schwer wie X “. Da X schon als schwer bekannt ist, erbt Y diese Schwere. Machst du es andersherum ($Y \leq_p X$), zeigst du nur „ Y ist *höchstens* so schwer wie das schwere X “ – das sagt über Y gar nichts aus.

🗨 Analogie

Du willst beweisen, dass ein neuer Gegner stark ist. Dann lässt du den *amtierenden Champion* gegen ihn antreten und zeigst: Wer den Neuen besiegt, besiegt auch den Champion. Du schickst nicht den Neuen gegen einen Anfänger. „Bekannt stark $\leq_p$ neu“.

4.3 Reduktionen lassen sich verketteten

▲ Satz 4.2 Transitivität

Gilt $A \leq_p B$ und $B \leq_p C$, dann auch $A \leq_p C$.

Beweisidee: Schalte die beiden Umformungen hintereinander: erst $A \rightarrow B$, dann $B \rightarrow C$. Die Ausgabe der ersten ist polynomiell groß, also bleibt auch die Verkettung polynomiell.

Das ist der Grund, warum wir gleich eine ganze *Kette* von Reduktionen bauen können: Einmal

X als schwer bekannt, überträgt sich die Schwere Glied für Glied auf alle folgenden Probleme.

✓ Selbst-Check

Du willst zeigen, dass VERTEX COVER schwer ist, und weißt, dass CLIQUE schwer ist. In welche Richtung reduzierst du?

Antwort: $\text{CLIQUE} \leq_p \text{VERTEX COVER}$ – vom bekannt schweren CLIQUE auf das neue VERTEX COVER. So erbt VERTEX COVER die Schwere. Andersherum hättest du nichts gezeigt.

5 Die härtesten Probleme

★ In diesem Kapitel

Jetzt fassen wir die schwersten Probleme in NP mit einem Namen: *NP-vollständig*. Wir sehen, warum es überhaupt ein „erstes“ solches Problem gibt und warum ein einziger schneller Algorithmus für eines von ihnen die gesamte Landschaft zum Einsturz brächte.

5.1 NP-schwer und NP-vollständig

Mit der Reduktion können wir „am schwersten in NP“ präzise machen: ein Problem, auf das sich *alle* Probleme aus NP reduzieren lassen.

■ Definition 5.1 NP-schwer, NP-vollständig

- L_0 ist *NP-schwer*, wenn sich *jedes* Problem $L \in \text{NP}$ auf L_0 reduzieren lässt: $L \leq_p L_0$ für alle $L \in \text{NP}$.
- L_0 ist *NP-vollständig*, wenn es NP-schwer ist **und** selbst in NP liegt.

★ Intuition

Trenne die zwei Hälften sauber:

- *NP-schwer* ist eine **untere** Schranke: „mindestens so schwer wie alles in NP“.
- $\in \text{NP}$ ist eine **obere** Schranke: „nicht schwerer als NP“.
- *NP-vollständig* = beides. Diese Probleme sind exakt die schwersten *innerhalb* von NP.

✗ Stolperstein

NP-schwer heißt *nicht* automatisch „in NP“. Es gibt Probleme, die NP-schwer sind, aber *außerhalb* von NP liegen – das Halteproblem (Kapitel 9) ist so eines. „Schwer“ und „überhaupt lösbar“ sind zwei verschiedene Achsen.

5.2 Warum ein einziger Algorithmus alles verändern würde

▲ Satz 5.2 NP-vollständig verbindet P und NP

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

Beweisidee: Läge L_0 in P , so könnte man *jedes* $L \in NP$ lösen: erst die Reduktion $L \leq_p L_0$ rechnen, dann den schnellen L_0 -Löser anwerfen. Also wäre ganz NP in P .

☆ Aha

Das ist die Sprengkraft der NP-Vollständigkeit. Fände jemand für *ein einziges* NP-vollständiges Problem einen polynomiellen Algorithmus, dann wären *schlagartig alle* NP-Probleme effizient lösbar – $P = NP$. Umgekehrt: Solange das niemandem gelingt, ist ein NP-Vollständigkeitsbeweis das stärkste bekannte Indiz „dieses Problem hat vermutlich keinen effizienten Algorithmus“.

5.3 Das erste NP-vollständige Problem: Cook–Levin

Damit die Reduktionskette starten kann, braucht man *ein* NP-vollständiges Problem „von Hand“ – ohne sich auf ein anderes stützen zu können. Das lieferten Cook und Levin. Das Problem heißt SAT: Gegeben eine aussagenlogische Formel, gibt es eine Belegung der Variablen, die sie wahr macht?

▲ Satz 5.3 Cook (1971), Levin (1973)

SAT ist NP-vollständig.

Beweisidee: Man nimmt ein *beliebiges* Problem $L \in NP$ mit seiner ratenden Maschine und „gießt“ deren Rechnung in eine Formel: Variablen beschreiben „welcher Zustand zur Zeit t “, „was steht auf dem Band“ usw. Die Formel ist so gebaut, dass ihre *erfüllenden Belegungen genau den akzeptierenden Rechenwegen entsprechen*. Also: L -Ja-Instanz $\iff$ Formel erfüllbar. Da L beliebig war, ist SAT schwer für ganz NP .

Du musst diesen Beweis nicht im Detail können. Wichtig ist die Rolle: SAT ist der *Anker*. Ab hier braucht man nie wieder „alle $L \in NP$ “ zu betrachten.

5.4 Der Motor: das Vererbungskorollar

▲ Satz 5.4 Vererbungskorollar

Ist L_0 bereits NP-vollständig, gilt $L_0 \leq_p L_1$, und liegt $L_1 \in NP$, dann ist auch L_1 NP-vollständig.

Beweisidee: Für jedes $L \in NP$ gilt $L \leq_p L_0 \leq_p L_1$ (Transitivität). Also ist L_1 NP-schwer; zusammen mit $L_1 \in NP$ ist es NP-vollständig.

☆ Aha

Das ist das Arbeitspferd. Um ein neues Problem als NP-vollständig zu entlarven, brauchst du nur zwei Dinge:

1. eine *Reduktion* von *einem* bekannten NP-vollständigen Problem auf das neue, und
2. den Nachweis, dass das neue Problem in NP liegt (Zertifikat angeben).

Kein Wort mehr über „alle $L \in \text{NP}$ “. So wächst die Liste der NP-vollständigen Probleme Schritt für Schritt – genau das machen wir im nächsten Kapitel.

✓ Selbst-Check

Du zeigst $\text{SAT} \leq_p Y$ und weist nach, dass $Y \in \text{NP}$. Was folgt – und welchen der beiden Teile darfst du auf keinen Fall vergessen?

Antwort: Es folgt: Y ist NP-vollständig. Vergiss nie den Teil $Y \in \text{NP}$. Ohne ihn hast du nur NP-Schwere gezeigt, nicht Vollständigkeit – Y könnte sonst noch schwerer als NP sein.

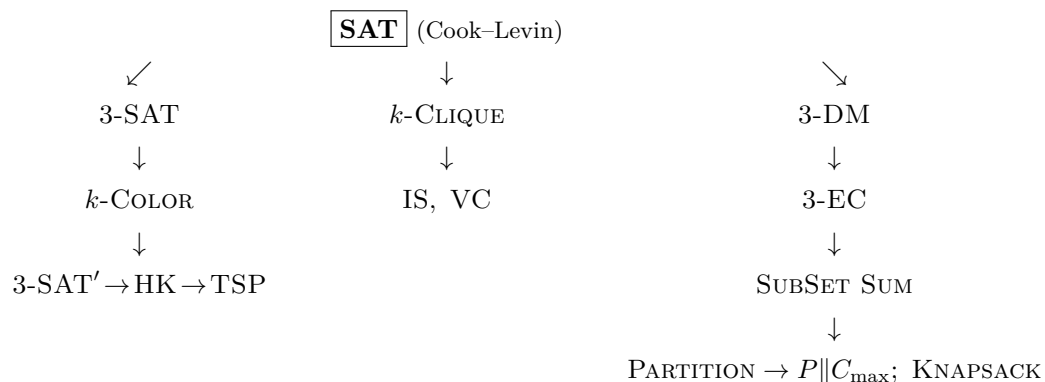
6 Der Problem-Zoo

★ In diesem Kapitel

Wir gehen die Probleme der Vorlesung durch – aber nicht als Liste zum Auswendiglernen, sondern als *Familie*, die über Reduktionen zusammenhängt. Zu jedem: worum es geht, warum es schwer ist, woher die Schwere kommt.

6.1 Die Landkarte: eine Kette ab SAT

Alle folgenden Probleme sind NP-vollständig. Der Beweis ist immer dasselbe Rezept (Vererbungs-korollar): reduziere von einem schon bekannten Problem und zeige $\in \text{NP}$. Der Pfeil $\leq_p$ heißt „wird reduziert auf“.



★ Intuition

Halte dir beim Lesen immer die Frage vor Augen: „Von welchem Nachbarn erbt dieses Problem seine Schwere?“ Das ist der rote Faden. Die Probleme sind nicht einzeln schwer – sie sind *gemeinsam* schwer, verbunden durch die Pfeile.

6.2 Die Logik-Probleme

■ Definition 6.1 SAT und 3-SAT

SAT: Gegeben eine Formel in *konjunktiver Normalform* (KNF) – also ein UND von Klauseln, jede Klausel ein ODER von Literalen (x_j oder $\neg x_j$). Frage: gibt es eine erfüllende Belegung?
3-SAT: dasselbe, aber jede Klausel hat höchstens 3 Literale.

● Beispiel 6.2 SAT von Hand

$\alpha = (x_1 \vee x_{10}) \wedge (x_1 \vee \neg x_{10}) \wedge (x_{10})$. Die dritte Klausel erzwingt $x_{10} = \text{wahr}$. Dann verlangt die zweite Klausel $x_1 = \text{wahr}$. Beide gesetzt, ist auch die erste erfüllt. Also erfüllbar mit $x_1 = x_{10} = \text{wahr}$.

SAT ist der Anker (Cook–Levin). 3-SAT ist die „handliche“ Version, von der aus fast alle weiteren Reduktionen starten – kurze Klauseln sind leichter zu verbauen.

➤ Zusammenhang

$\text{SAT} \leq_p \text{3-SAT}$: Man spaltet jede lange Klausel mit Hilfsvariablen in eine Kette von 3er-Klauseln auf, ohne die Erfüllbarkeit zu ändern. Dadurch bleibt 3-SAT genauso schwer wie SAT.

6.3 Die Graph-Probleme und ihr Dreieck

Drei Probleme, die im Grunde ein einziges sind – nur aus drei Blickwinkeln.

■ Definition 6.3 Clique, Independent Set, Vertex Cover

Gegeben ein Graph G und eine Zahl k .

- CLIQUE: eine Gruppe C , in der *alle* paarweise verbunden sind; gibt es eine mit $|C| \geq k$?
- INDEPENDENT SET (IS): eine Gruppe I , in der *keine* zwei verbunden sind; gibt es eine mit $|I| \geq k$?
- VERTEX COVER (VC): eine Knotenmenge C , die *jede* Kante berührt; gibt es eine mit $|C| \leq k$?

☆ Aha

Das Dualitätsdreieck. Diese drei sind dasselbe Problem in Verkleidung. In einem Graphen mit n Knoten gilt:

$$\begin{aligned} C \text{ ist Clique in } G &\iff C \text{ ist Independent Set im Komplementgraphen } \bar{G} \\ &\iff V \setminus C \text{ ist Vertex Cover in } \bar{G}. \end{aligned}$$

Merksatz: *Vertex Cover = alles außer einem Independent Set; Clique = Independent Set, wenn man alle Kanten umdreht.* Verstehst du eines, verstehst du alle drei – und die Reduktionen zwischen ihnen sind bloße Umformulierungen.

● Beispiel 6.4 Das Dreieck an einem Mini-Graphen

Nimm ein Dreieck plus einen isolierten vierten Knoten: $V = \{1, 2, 3, 4\}$, Kanten $\{1, 2\}, \{1, 3\}, \{2, 3\}$. $\{1, 2, 3\}$ ist eine *Clique* (alle verbunden). Im Komplement $\bar{G}$ (Kanten $\{1, 4\}, \{2, 4\}, \{3, 4\}$) ist $\{1, 2, 3\}$ ein *Independent Set* (keine zwei verbunden). Und $\{4\}$ ist dort ein *Vertex Cover* – der Knoten 4 berührt alle Kanten von $\bar{G}$. Ein Bild, drei Rollen.

■ Definition 6.5 Färbung und Hamiltonkreis

k -COLOR: Kann man die Knoten mit k Farben so färben, dass keine Kante zwei gleichfarbige Enden hat?

HAMILTONKREIS (HK): Gibt es einen Rundweg, der *jeden* Knoten genau einmal besucht?

Diese beiden sind „berühmte“ schwere Probleme. k -COLOR erbt seine Schwere von 3-SAT, HK von der 3-Literal-Variante 3-SAT'.

6.4 Eine Reduktion ganz durchgerechnet: $\text{SAT} \leq_p k\text{-Clique}$

Damit du *siehst*, wie eine Reduktion konkret arbeitet, hier eine komplett.

★ Intuition

Idee: Baue für jedes Literal-Vorkommen einen Knoten. Verbinde zwei Knoten genau dann, wenn sie *aus verschiedenen Klauseln* stammen und sich *nicht widersprechen*. Eine erfüllende Belegung wählt aus jeder Klausel ein wahres Literal – und diese Auswahl bildet dann eine Clique.

■ Definition 6.6 Die Konstruktion

Zur Formel $F = F_1 \wedge \dots \wedge F_m$ (Klausel F_i mit Literalen $y_{i1}, \dots$):

- *Knoten*: $[i, j]$ für das j -te Literal in Klausel i .
- *Kanten*: $\{[i, j], [i', j']\}$ genau dann, wenn $i \neq i'$ (verschiedene Klauseln) und die beiden Literale sich nicht widersprechen (nicht x und $\neg x$).
- *Gesuchte Cliquengröße*: $k = m$ (Anzahl der Klauseln).

Warum es stimmt. Eine m -Clique muss aus jeder Klausel genau einen Knoten nehmen (Knoten derselben Klausel sind nie verbunden) und darf keine zwei widersprüchlichen Literale enthalten. Das ist *genau* eine Belegung, die in jeder Klausel ein Literal wahr macht – also eine erfüllende Belegung. Und umgekehrt.

● Beispiel 6.7 Mit echten Zahlen

$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$, also $m = 3$, gesucht $k = 3$. Es entstehen 8 Knoten ($3 + 2 + 3$). Kanten zwischen allen Paaren aus verschiedenen Klauseln *außer* den widersprüchlichen, z. B. bekommt $\{[1, 1], [2, 1]\}$ (x_1 gegen $\neg x_1$) *keine* Kante. Die Clique $\{[1, 1], [2, 2], [3, 1]\}$ steht für $x_1, \neg x_2, x_1$ und liefert die Belegung $x_1 = \text{wahr}$, $x_2 = \text{falsch}$ (und x_3 frei). Tatsächlich erfüllt das F .

6.5 Die Zahl- und Mengen-Probleme

■ Definition 6.8 3-DM, 3-EC, SubSet Sum, Partition, Knapsack

- 3-DIM. MATCHING (3-DM): drei gleich große Mengen und Tripel; wähle Tripel, die jedes Element genau einmal treffen.
- 3-EXACT COVER (3-EC): 3-elementige Mengen; wähle einige, die das Universum *exakt* (disjunkt) überdecken.
- SUBSET SUM: Zahlen $c_1, \dots, c_n$ und Ziel K ; gibt es eine Teilmenge mit Summe genau K ?
- PARTITION: gibt es eine Teilmenge, die genau die *Hälfte* der Gesamtsumme trägt?
- KNAPSACK: Gegenstände mit Gewicht und Profit, Kapazität B , Zielprofit P ; passt eine Auswahl mit Gewicht $\leq B$ und Profit $\geq P$?

➤ Zusammenhang

Diese fünf hängen in einer sauberen Kette: $\text{SAT} \leq_p \text{3-DM} \leq_p \text{3-EC} \leq_p \text{SUBSET SUM} \leq_p \text{PARTITION}$, und KNAPSACK enthält SUBSET SUM als Spezialfall ($w_j = p_j, P = B$). Schwere fließt von der Logik (SAT) über Mengen (3-EC) bis zu reinen Zahlen (SUBSET SUM).

6.6 Zwei kurze, elegante Reduktionen

● Beispiel 6.9 $3\text{-EC} \leq_p \text{SubSet Sum}$ – Mengen werden Ziffern

Kodiere jede 3-elementige Menge als Bitvektor über dem Universum und lies ihn als Zahl zur Basis $n + 1$. Beispiel: $U = \{u_1, \dots, u_6\}$, Menge $\{u_1, u_5, u_6\}$ wird zum Bitvektor 100011, also zur Zahl $(n+1)^0 + (n+1)^4 + (n+1)^5$. Zielwert K ist die Zahl zum Vektor 111111.

Der Trick: Bei Basis $n + 1$ und höchstens n Summanden gibt es *keinen Übertrag*. Eine Teilmenge trifft die Zielsumme genau dann, wenn jede Stelle exakt einmal getroffen wird – also eine exakte Überdeckung.

● Beispiel 6.10 $\text{SubSet Sum} \leq_p \text{Partition}$ – zwei Zusatzzahlen

Aus $(c_1, \dots, c_n, K)$ mit $N = \sum_j c_j + 1$ mache die Zahlenmenge, ergänzt um $c_{n+1} = N - K$ und $c_{n+2} = K + 1$. Die Gesamtsumme ist dann $2N$, die Hälfte N .

Der Trick: Die zwei Zusatzzahlen summieren sich zu $N + 1$ – sie können also nie in derselben Hälfte landen. Eine Halbierung existiert genau dann, wenn eine Original-Teilmenge den Wert K trifft.

■ Definition 6.11 Scheduling: $P \parallel C_{\max}$

n Jobs mit Laufzeiten $p_1, \dots, p_n$, m identische Maschinen. Verteile die Jobs so, dass die am stärksten belastete Maschine – der *Makespan* $C_{\max} = \max_i \sum_{J_j \text{ auf } M_i} p_j$ – minimal wird.

► Zusammenhang

$P \parallel C_{\max}$ ist schon für $m = 2$ Maschinen NP-vollständig – über $\text{PARTITION} \leq_p P2 \parallel C_{\max}$: Zwei Maschinen exakt gleich auszulasten, heißt die Jobs zu *halbieren*. Damit schließt sich der Kreis von der Logik bis zum Scheduling.

✓ Selbst-Check

Warum ist SUBSET SUM „schwer“, obwohl es doch nur um Addieren von Zahlen geht?

Antwort: Weil die Zahlen *binär* kodiert sind (Stolperstein aus Kapitel 2). Ihr Wert ist exponentiell in der Eingabelänge. Alle Teilsummen auszuprobieren bedeutet 2^n Teilmengen – das simple „Addieren“ versteckt eine exponentielle Suche.

7 Wie schwer *genau*? Die ETH

★ In diesem Kapitel

NP-Vollständigkeit sagt nur „vermutlich nicht polynomiell“. Aber *wie* exponentiell? Die Exponentialzeit-Hypothese (ETH) gibt eine schärfere Antwort – und erlaubt konkrete untere Laufzeitschranken.

7.1 Die Lücke, die NP-Vollständigkeit lässt

NP-Vollständigkeit ist ein grobes Sieb. Sie unterscheidet nicht zwischen einem Problem, das 2^n braucht, und einem, das nur $2^{\sqrt{n}}$ braucht – beide sind „nicht polynomiell“. Für viele Fragen will man aber genau wissen: *Geht es wenigstens subexponentiell?*

■ Definition 7.1 Klein-o und „subexponentiell“

$f(n) = o(g(n))$ heißt: f wächst *echt langsamer* als g , also $f(n)/g(n) \rightarrow 0$. Wichtig: δn ist *nicht* $o(n)$ für festes $\delta > 0$. Ein „ $2^{o(n)}$ -Algorithmus“ ist einer, dessen Exponent langsamer als linear wächst – er läuft *subexponentiell* (z. B. $2^{\sqrt{n}}$).

7.2 Die Hypothese und warum man sie braucht

■ Definition 7.2 Exponentialzeit-Hypothese (ETH)

Es gibt eine Konstante $\delta > 0$, sodass 3-SAT mit n Variablen *nicht* in Zeit $2^{\delta n} \cdot \text{poly}$ gelöst werden kann. Kurz: es gibt *keinen* $2^{o(n)}$ -Algorithmus für 3-SAT.

Das ist eine *Annahme*, stärker als $P \neq NP$, aber allgemein geglaubt (die besten bekannten 3-SAT-Algorithmen brauchen c^n). Aus ihr leitet man untere Schranken für viele Probleme ab.

★ Intuition

Das Beweismuster ist immer *Kontraposition über eine Reduktion*: „Gäbe es einen zu schnellen Algorithmus für mein Zielproblem, dann ergäbe die Reduktion einen zu schnellen Algorithmus für 3-SAT – Widerspruch zur ETH.“ Man überträgt also die Härte-Annahme entlang eines Pfeils, genau wie bei der NP-Vollständigkeit – nur zählt man diesmal die *Größe* der konstruierten Instanz mit.

▲ Satz 7.3 Sparsification-Lemma (und warum es nötig ist)

Unter der ETH gibt es sogar keine $2^{o(m)}$ -Lösung für 3-SAT, wobei m die *Klauselzahl* ist. *Warum man das braucht*: Die ETH spricht über die Variablenzahl n . Viele Reduktionen erzeugen ihre Instanzgröße aber aus der Klauselzahl m . Weil m gegenüber n groß sein kann, folgt eine m -Schranke *nicht* direkt aus der ETH. Das Sparsification-Lemma schließt genau diese Lücke.

☆ Aha

Faustregel für die Praxis: *Hängt der Parameter deiner konstruierten Instanz an der Klauselzahl m , zitiere das Sparsification-Lemma.* Das ist der Standardschritt in jedem ETH-Beweis – und der, den man am leichtesten vergisst.

Unter der ETH bekommt man so untere Schranken $2^{o(n)}$ z. B. für 3-COLOR, CLIQUE, VERTEX COVER, INDEPENDENT SET sowie SUBSET SUM und PARTITION.

✓ Selbst-Check

Wenn die ETH *falsch* ist – folgt dann $P = NP$?

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

8 Damit leben: gute Näherungen

★ In diesem Kapitel

Wenn ein Problem (vermutlich) keine schnelle *exakte* Lösung hat, geben wir uns mit einer *beweisbar guten* Näherung zufrieden. Wir lernen, was „beweisbar gut“ heißt, und sehen die wichtigsten Näherungsalgorithmen der Vorlesung in Aktion.

8.1 Was heißt „gute Näherung“?

■ Definition 8.1 Multiplikative Güte

Sei $\text{OPT}(I)$ der optimale Wert einer Instanz I . Ein Algorithmus A hat *Güte* α , wenn für *alle* Instanzen gilt:

- Minimierung (z. B. TSP, Scheduling): $A(I) \leq \alpha \cdot \text{OPT}(I)$ – „höchstens α -mal so groß wie das Optimum“.
- Maximierung (z. B. Knapsack): $\text{OPT}(I) \leq \alpha \cdot A(I)$ – „mindestens ein α -tel des Optimums“.

Die Güte ist *scharf*, wenn es Instanzen gibt, die den Faktor (fast) erreichen.

★ Intuition

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

■ Definition 8.2 PTAS, EPTAS, FPTAS – Näherung nach Wunsch

Manche Probleme erlauben eine *ganze Familie* (A_ε) mit Güte $1 + \varepsilon$ für *jedes* $\varepsilon > 0$ – man stellt die Genauigkeit selbst ein. Nach Laufzeit gestaffelt:

- *PTAS*: für jedes feste ε polynomiell in n (aber ε darf im Exponenten stecken, z. B. $n^{1/\varepsilon}$).
- *EPTAS*: Laufzeit $f(1/\varepsilon) \cdot \text{poly}(n) - \varepsilon$ raus aus dem Exponenten von n .
- *FPTAS*: polynomiell in n und $1/\varepsilon$ (z. B. $O(n^3/\varepsilon)$). Die stärkste Form.

Pseudopolynomiell nennt man eine Laufzeit, die polynomiell in den *Zahlenwerten* der Eingabe ist – erinnere den Stolperstein: das ist *nicht* polynomiell in der Eingabelänge.

8.2 TSP: von „unmöglich“ zu Güte 1,5

Das Traveling-Salesman-Problem (kürzeste Rundreise durch alle Städte) zeigt die ganze Spannweite der Approximation.

▲ Satz 8.3 Allgemeines TSP ist gar nicht approximierbar

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

Beweisidee: Reduziere HAMILTONKREIS. Echte Kanten bekommen Distanz 1, fehlende Kanten eine *riesige* Distanz. Ein Algorithmus mit beschränkter Güte müsste die billige (Hamilton-)Tour von der teuren unterscheiden – und könnte damit HK exakt lösen.

Der Ausweg ist eine sinnvolle Zusatzannahme: das *metrische* TSP. Distanzen sind symmetrisch und erfüllen die *Dreiecksungleichung* $d(i, j) \leq d(i, k) + d(k, j)$ – ein Umweg ist nie kürzer als der direkte Weg. Das ist realistisch (echte Entfernungen) und macht das Problem approximierbar.

■ Definition 8.4 Eulerkreis

Ein *Eulerkreis* in einem (Multi-)Graphen ist ein Rundweg, der jede *Kante* genau einmal benutzt. Er existiert genau dann, wenn der Graph zusammenhängend ist und *jeder Knoten* *geraden Grad* hat.

★ Intuition

Beide TSP-Algorithmen folgen demselben Dreisprung: (1) baue ein billiges Gerüst, das alle Städte verbindet (einen minimalen Spannbaum), (2) mache daraus einen Eulerkreis (alle Grade gerade), (3) kürze ihn zu einer Tour ab – was dank Dreiecksungleichung nie teurer wird.

■ Definition 8.5 ΔTSP_1 – MST-Verdopplung (Güte 2)

1. Minimalen Spannbaum T berechnen.
2. Jede Kante von T verdoppeln $\rightarrow$ alle Grade gerade.
3. Eulerkreis bestimmen.
4. Zur Tour abkürzen (schon besuchte Städte überspringen).

Warum Güte 2: $w(T) \leq \text{OPT}$ (eine optimale Tour minus eine Kante ist ein Spannbaum), Verdoppeln gibt $2w(T)$, Abkürzen verlängert nicht. Also $\leq 2 \text{OPT}$.

● Beispiel 8.6 ΔTSP_1 komplett durchgerechnet

Städte $\{A, B, C, D, E\}$ mit u. a. $d(A, E) = 1$, $d(B, C) = 1$, $d(A, C) = 2$, $d(C, D) = 2$ (die übrigen 2 oder 3). Optimale Tour: $[C, B, D, E, A, C]$ mit Länge **8**.

- *MST* T : Kanten $A-E$ (1), $B-C$ (1), $A-C$ (2), $C-D$ (2); Gewicht $w(T) = 6$.
- *Verdoppeln* $\rightarrow$ Multigraph mit Gewicht 12.
- *Eulerkreis* $[C, A, E, A, C, B, C, D, C]$, Länge 12.
- *Abkürzen* $\rightarrow$ Tour $[C, A, E, B, D, C]$ mit Länge **10** $\leq 2 \cdot 8$.

Christofides verbessert Schritt 2: Statt *alle* Kanten zu verdoppeln, repariert er nur die „schiefen“ Knoten – die mit ungeradem Grad.

■ Definition 8.7 ΔTSP_2 – Christofides (Güte 1,5)

1. Minimalen Spannbaum T berechnen.
2. $X :=$ Knoten mit *ungeradem* Grad in T (ihre Anzahl ist gerade).
3. Minimales perfektes Matching K auf X berechnen.
4. $T + K$ hat überall geraden Grad $\rightarrow$ Eulerkreis $\rightarrow$ abkürzen.

Warum Güte 1,5: Das Matching kostet höchstens $\text{OPT}/2$ (die optimale Tour zerfällt auf X in zwei Matchings), also $\leq w(T) + \text{OPT}/2 \leq 1,5 \text{OPT}$.

● Beispiel 8.8 Christofides am selben Graphen

MST wie oben. Ungerade Grade: $X = \{B, C, D, E\}$. Matching-Kosten: $d(B, C) = 1$, $d(D, E) = 2$, $d(B, E) = 2$, $d(C, D) = 2$, $d(B, D) = 3$, $d(C, E) = 3$. Minimales perfektes Matching: $\{B-C, D-E\}$ mit Gewicht 3. Der Eulerkreis $[C, B, C, A, E, D, C]$ kürzt ab zu $[C, B, A, E, D, C]$ mit Länge **9** – besser als die 10 von ΔTSP_1 .

✗ Stolperstein

Zwei Details, die Christofides oft falsch gemacht wird: (1) Das Matching wird *nur* auf den ungerad-gradigen Knoten des MST gebildet, nicht auf allen. (2) Ohne die Dreiecksungleichung bricht das Abkürzen zusammen – nur sie garantiert, dass die abgekürzte Tour nicht länger wird.

8.3 Knapsack: warum Gier allein scheitert

Der Rucksack als Optimierungsproblem: pack Gegenstände mit maximalem Gesamtprofit ein, ohne die Kapazität B zu sprengen.

★ Intuition

Der naheliegende *Greedy*: nach „Profit pro Gewicht“ sortieren und der Reihe nach einpacken, was passt. Meistens gut – aber ohne jede Garantie.

● Beispiel 8.9 Greedy kann beliebig schlecht sein

Zwei Gegenstände: $(w_0, p_0) = (1, 1)$ und $(w_1, p_1) = (B, B - 1)$, Kapazität B . Greedy nimmt Gegenstand 0 (Dichte 1 schlägt $(B - 1)/B$), Gewinn nur 1. Optimal wäre Gegenstand 1 mit Gewinn $B - 1$. Das Verhältnis $\text{OPT}/\text{GA} = B - 1$ wächst unbeschränkt – *keine* konstante Güte.

☆ Aha

Die Reparatur ist verblüffend einfach. *Modified Greedy (MGA)*: rechne Greedy und „nimm nur den einen profitabelsten Gegenstand“, gib das Bessere aus. Schon das garantiert Güte 2. Der Grund: Greedy verliert nur am einen „Split-Item“, das nicht mehr passte – und dessen Profit fängt das Einzel-Item ab.

Will man noch näher ans Optimum, gibt es zwei Stufen: der *Sahni*-Algorithmus probiert alle kleinen Vorauswahlen (Güte $1 + 1/k$, ein PTAS), und das *FPTAS* skaliert die Profite herunter und löst exakt per dynamischer Programmierung – Güte $1 + \varepsilon$ in Zeit $O(n^3/\varepsilon)$.

8.4 Scheduling: einfache Regeln, scharfe Schranken

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

■ Definition 8.10 List Scheduling und LPT

List Scheduling: geh die Jobs der Reihe nach durch, leg jeden auf die *momentan am wenigsten belastete* Maschine.

LPT (Longest Processing Time): sortiere die Jobs zuerst *absteigend*, dann List Scheduling.

▲ Satz 8.11 Güten von List Scheduling und LPT

List Scheduling hat Güte $2 - \frac{1}{m}$, LPT die bessere Güte $\frac{4}{3} - \frac{1}{3m}$ – beide scharf.

Beweisidee (List Scheduling): Sei J_k der letzte Job auf der vollsten Maschine. Vor seiner Zuweisung war diese Maschine die leerste, also waren *alle* mindestens so voll. Mit den zwei Universalschranken $\text{OPT} \geq \frac{1}{m} \sum p_i$ (Durchschnittslast) und $\text{OPT} \geq p_{\max}$ (größter Job) folgt die Schranke.

★ Intuition

Diese *zwei Schranken* – Durchschnittslast und größter Job – sind das Universalwerkzeug *aller* Scheduling-Güte-Beweise. Sortieren (LPT) hilft, weil der störende „letzte große Job“ dann früh drankommt, wenn noch alles leer ist.

● Beispiel 8.12 Warum List Scheduling die Schranke wirklich erreicht

$m(m-1)$ Jobs der Größe 1, danach *ein* Job der Größe m – in dieser Reihenfolge. List Scheduling verteilt die Einsen gleichmäßig (je $m-1$) und setzt den großen obendrauf: Last $2m-1$. Optimal wäre: großer Job allein, Einsen auf die restlichen Maschinen – Last m . Verhältnis $\frac{2m-1}{m} = 2 - \frac{1}{m}$. LPT würde den großen Job zuerst legen und das vermeiden.

8.5 MAX-3-SAT: Güte 2 fast geschenkt

● Beispiel 8.13 Zwei Belegungen genügen

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

✓ Selbst-Check

Warum braucht Christofides zwingend die Dreiecksungleichung, ΔTSP_1 scheinbar auch, das allgemeine TSP aber „gibt es gar nicht“?

Antwort: Beide Näherungsalgorithmen *kürzen* einen Eulerkreis ab – das darf nur nicht verteuern, was allein die Dreiecksungleichung sichert. Ohne sie (allgemeines TSP) kann Abkürzen beliebig teuer werden, und tatsächlich ist dort *jede* beschränkte Güte unmöglich (außer $P = NP$).

9 Jenseits von NP: das Halteproblem

★ In diesem Kapitel

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

■ Definition 9.1 Halteproblem

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

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

☆ Aha

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

10 Das große Ganze

★ In diesem Kapitel

Wir ziehen die Fäden zusammen. Eine Landkarte, ein Selbsttest – und der Weg nach vorn.

10.1 Die Reise in einem Bild

Station	Kernidee
P	effizient <i>lösbar</i> (polynomiell)
NP	effizient <i>überprüfbar</i> (Zertifikat prüfen / raten)
$P \subseteq NP$	Lösen ist schwerer (oder gleich schwer) wie Prüfen; Gleichheit offen
Reduktion $A \leq_p B$	„ B mindestens so schwer wie A “; Richtung: bekannt $\leq_p$ neu
NP-vollständig	die schwersten in NP; SAT ist der Anker (Cook–Levin)
Vererbungs-korollar	neues Problem schwer zeigen: eine Reduktion $+ \in NP$
ETH	schräfer als NP-Vollständigkeit: konkrete $2^{o(\cdot)}$ -Schranken
Approximation	bei schweren Optimierungsproblemen: beweisbar gute Näherung

10.2 Verständnis-Check: wahr oder falsch?

Kannst du jede Aussage begründen? Die Begründung zählt, nicht das Kreuz.

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

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

10.3 Wie es weitergeht

☆ Aha

Du hast jetzt das *Warum* beisammen: Was Probleme schwer macht, wie man Schwere von einem Problem auf ein anderes überträgt, wo die Grenzen liegen und was man tut, wenn exakte Lösungen zu teuer sind. Das ist das Fundament.

Der nächste Schritt ist die *Praxis*. Nimm dir die Präsenz-, Haus- und Klausuraufgaben vor und führe dort selbst aus, was hier nur als Idee stand: Reduktionen vollständig beweisen (beide Richtungen!), Güte-Schranken sauber herleiten, Worst-Case-Instanzen konstruieren. Dieser Guide gibt dir das Verständnis – die Aufgaben geben dir das Können.

Viel Erfolg.