

Komplexität verstehen

Der ausführliche, grafische Guide zu
Analyse von Algorithmen und Komplexität
CAU Kiel – Sommersemester 2026 · Ausgabe 4

*Nicht auswendig lernen – verstehen.
Jeder Begriff mit Bild, Beispiel und einer Frage zum Selberdenken.*

Was dieser Guide ist. Ein Heft, das dir das *Fundament* des Kurses aufbaut – gründlich, in Ruhe, mit vielen Bildern. Kein Nachschlagewerk zum Überfliegen. Wir nehmen uns für jeden Gedanken den Platz, den er braucht. Wenn du es durcharbeitest, verstehst du den ganzen Stoff und siehst, wie alles zusammenhängt.

So arbeitest du damit. Lies langsam. Schau dir jedes Bild an und frag dich, was es zeigt, bevor du weiterliest. Am Ende jedes Kapitels stehen **Abruf-Fragen**: Beantworte sie *schriftlich aus dem Kopf*, bevor du im Lösungsteil (Anhang A) nachsiehst. Dieses aktive Erinnern ist der eigentliche Lerneffekt.

Die Kästen.

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

Inhalt

1	Ein Problem, das sich wehrt	3
2	Was heißt eigentlich „schnell“?	5
3	Prüfen ist leichter als Lösen	7
4	Reduktionen: Probleme miteinander vergleichen	10
5	Die härtesten Probleme: NP-vollständig	12
6	Der Problem-Zoo	15
7	Reduktionen im Detail	20
8	Wie schwer <i>genau</i> ? Die ETH	25
9	Damit leben: gute Näherungen	28
10	Jenseits von NP: das Halteproblem	35
11	Das große Ganze	35
A	Lösungen zu den Abruf-Fragen	36

1 Ein Problem, das sich wehrt

★ In diesem Kapitel

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

1.1 Ein Graph ist ein Netz aus Punkten und Linien

Bevor wir zum Problem kommen, das Vokabular. Ein *Graph* ist einfach ein Netz: eine Menge von Punkten, die *Knoten*, und Verbindungslinien zwischen manchen von ihnen, die *Kanten*. Mehr steckt zunächst nicht dahinter. Ein Graph kann ein Straßennetz sein, ein Freundeskreis, ein Stromnetz – die Mathematik dahinter ist dieselbe.

Wir stellen uns die Knoten als *Gäste auf einer Party* vor. Eine Kante zwischen zwei Gästen heißt: „die beiden kennen sich“.

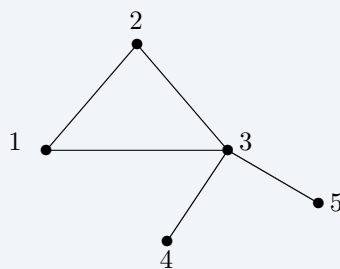
1.2 Was ist eine Clique?

■ Definition 1.1 Clique

Eine *Clique* ist eine Gruppe von Knoten, in der *jeder mit jedem* direkt verbunden ist. Auf der Party: eine Gruppe von Gästen, die sich *alle* gegenseitig kennen.

Das *Cliquenproblem* fragt: Gibt es in einem gegebenen Graphen eine Clique aus mindestens k Knoten? Sieh dir ein kleines Beispiel an und überzeuge dich, dass du „Clique“ verstanden hast.

● Beispiel 1.2 Eine Clique mit dem Auge finden



Die Gruppe $\{1, 2, 3\}$ bildet ein Dreieck: die Kanten 1–2, 1–3 und 2–3 sind alle da. Also kennen sich alle drei gegenseitig – eine Clique der Größe 3. Die Gruppe $\{2, 3, 4\}$ ist *keine* Clique: zwischen 2 und 4 gibt es keine Kante. Und eine Clique aus vier Gästen existiert hier gar nicht – probiere es aus.

Bei fünf Gästen erkennst du die Antwort auf einen Blick. Ein Computer „sieht“ den Graphen aber nicht als Bild. Er muss die Clique *ausrechnen*. Und genau das wird ab ein paar hundert Gästen zum Problem.

1.3 Der naheliegende Algorithmus – und warum er scheitert

Wie würde ein Computer eine k -Clique suchen, wenn er keinen cleveren Trick kennt? Er würde stur *jede mögliche Gruppe* aus k Gästen bilden und für jede prüfen, ob sich wirklich alle kennen. Das ist korrekt – es findet die Clique garantiert, falls es eine gibt. Aber es ist entsetzlich langsam. Der Grund liegt in der schieren *Anzahl* der Gruppen.

Nehmen wir n Gäste und suchen eine Clique der Größe $k = n/2$. Die Anzahl der Gruppen ist dann „ n über $n/2$ “, und das ist mindestens $2^{n/2}$. Diese Zahl wächst *explosionsartig*. Schau dir an, wie schnell:

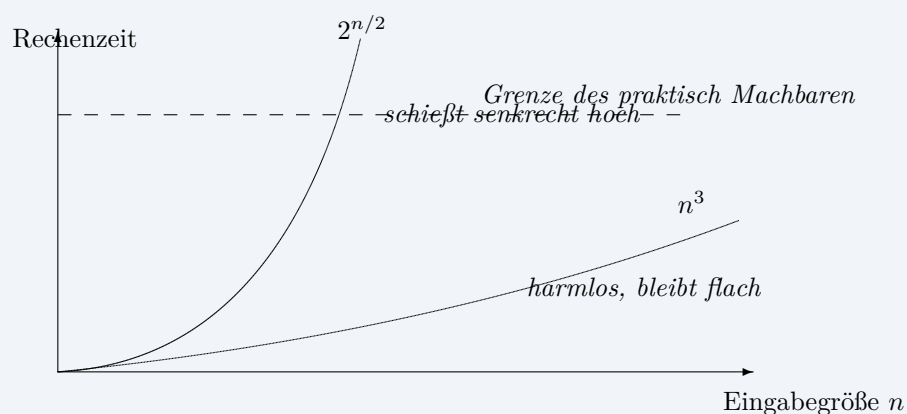
● Beispiel 1.3 $2^{n/2}$ in Zahlen – die Explosion

Gäste n	Gruppen (mind.)	Was das bedeutet
20	rund 1 000	sofort erledigt
40	rund 1 Million	noch ein Wimpernschlag
100	rund 10^{15}	über ein Monat bei einer Milliarde/s
200	rund 10^{30}	länger als das Universum existiert

Von 100 auf 200 Gäste wird die Aufgabe nicht doppelt so schwer, sondern um den Faktor 10^{15} schwerer.

Der Kern des Problems: Jeder *einzelne* zusätzliche Gast *verdoppelt* ungefähr die Arbeit. Diese Art von Wachstum heißt *exponentiell*, und sie ist der natürliche Feind jeder Berechnung. Das folgende Bild stellt sie einem gutmütigen *polynomiellen* Wachstum (n^3) gegenüber.

● Beispiel 1.4 Zwei Welten: n^3 gegen $2^{n/2}$



Die polynomielle Kurve bleibt lange in Bodennähe. Die exponentielle durchbricht schon bei kleiner Eingabe jede Zeitgrenze und ist danach nicht mehr einzufangen.

★ Intuition

Warum hilft schnellere Hardware nicht? Ein doppelt so schneller Computer schafft beim exponentiellen Algorithmus genau *einen* Gast mehr – weil ein Gast mehr die Arbeit verdoppelt und die schnellere Hardware genau diese eine Verdopplung ausgleicht. Willst du zehn Gäste mehr, brauchst du einen *tausendfach* schnelleren Rechner. Exponentielles Wachstum kann keine Hardware der Welt einholen.

▲ Satz 1.5 Der naive Clique-Algorithmus ist exponentiell

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

Woran das liegt: Es gibt $\binom{n}{n/2} \geq 2^{n/2}$ Gruppen der Größe $n/2$, und jede davon muss wenigstens angefasst werden. Schon das Aufzählen so vieler Gruppen kostet exponentiell viel Zeit.

1.4 Die Frage, die den ganzen Kurs trägt

Vielleicht ist der naive Algorithmus einfach nur ungeschickt. Vielleicht gibt es einen raffinierten Weg, die Clique *schnell* zu finden, ohne alle Gruppen durchzuprobieren?

Für das Cliquenproblem hat bis heute *niemand* so einen Weg gefunden. Und – das ist der eigentliche Punkt – niemand hat auch *bewiesen*, dass es keinen gibt. Wir stecken in einer Wissenslücke fest.

☆ Aha

Die Komplexitätstheorie stellt nicht die Frage „Wie löse ich dieses Problem schnell?“. Sie stellt die tiefere Frage: „Ist dieses Problem *überhaupt* schnell lösbar – oder gehört es zu einer großen Familie von Problemen, die alle gemeinsam schwer sind?“ Die erste Frage ist offen. Die zweite können wir beantworten. Genau darum geht es im ganzen Kurs.

? Abruf – erst selbst beantworten

A1. Erkläre in eigenen Worten, warum ein 1000-mal schnellerer Computer den naiven Clique-Algorithmus kaum brauchbarer macht.

A2. Was ist der Unterschied zwischen „wir kennen keinen schnellen Algorithmus“ und „es gibt keinen“? Welcher der beiden Fälle ist für die Clique bewiesen?

2 Was heißt eigentlich „schnell“?

★ In diesem Kapitel

Wir machen „schnell“ präzise. Das führt uns zur ersten großen Klasse: P, die Probleme, die wir effizient lösen können.

2.1 Geschwindigkeit misst man in der Eingabegröße

Es ergibt keinen Sinn zu sagen, ein Algorithmus brauche „drei Sekunden“. Drei Sekunden wofür – für zehn Zahlen oder für zehn Milliarden? Sinnvoll ist nur eine Aussage *relativ zur Größe der Eingabe*. Wir schreiben n für diese Größe und fragen: Wie viele Rechenschritte braucht der Algorithmus im schlimmsten Fall bei Eingabegröße n ?

Warum der *schlimmste* Fall? Weil wir eine *Garantie* wollen. Ein Algorithmus, der meistens schnell ist, aber bei ungünstigen Eingaben explodiert, gibt uns keine Sicherheit. Der schlimmste Fall über alle Eingaben der Größe n ist die ehrliche Messgröße.

2.2 Die Klasse P

■ Definition 2.1 Klasse P

P ist die Menge aller Entscheidungsprobleme, die ein Algorithmus in *polynomieller* Zeit löst – also mit einer Laufzeit von der Form $O(n^d)$, wobei d eine feste Zahl ist ($n, n^2, n^3, \dots$). Kurz gesagt: die *effizient lösbaren* Probleme.

Beispiele, für die wir schnelle, polynomielle Algorithmen kennen: Zahlen sortieren, den kürzesten Weg in einem Straßennetz finden, ein Stromnetz mit minimalen Kosten aufspannen (minimaler Spannbaum), Paare optimal zuordnen (Matching). All das liegt in P.

★ Intuition

„Polynomiell = effizient“ ist eine *Vereinbarung*, kein Naturgesetz. Ein Algorithmus mit Laufzeit n^{100} wäre praktisch nutzlos, zählt aber formal als polynomiell. Trotzdem funktioniert die Vereinbarung erstaunlich gut: Polynomielle Algorithmen aus der Praxis haben fast immer kleine Exponenten, und die Grenze „polynomiell gegen exponentiell“ bleibt dieselbe, egal welchen realistischen Rechnertyp man zugrunde legt. Diese Robustheit macht sie zur richtigen Trennlinie.

2.3 Warum wir alles als Ja/Nein-Frage schreiben

Ein Detail, das anfangs überrascht: Die Theorie behandelt fast nur *Entscheidungsprobleme* – Fragen mit Antwort „Ja“ oder „Nein“. Statt „Wie groß ist die größte Clique?“ fragt man „Gibt es eine Clique mit mindestens k Knoten?“.

Das wirkt wie ein Verlust an Information, ist aber keiner.

🗨 Analogie

Es ist wie beim Abwiegen ohne Anzeige, nur mit der Frage „schwerer als x ?“. Fragst du „schwerer als 1 kg? als 2 kg? als 3 kg?“ und so weiter, tastest du das genaue Gewicht ein. Die Ja/Nein-Frage für *jede* Schranke trägt also dieselbe Information wie die genaue Zahl. Deshalb ist die Ja/Nein-Version eines Problems „gleich schwer“ wie die Optimierungsversion – aber viel einfacher sauber zu vergleichen.

Formal kodiert man jede Eingabe als Zeichenkette (ein *Wort*) über einem Alphabet, und ein Problem wird zur *Menge aller Ja-Wörter*. Du musst diese Brille nicht ständig tragen. Merke dir nur den Kern: *Ein Problem ist die Menge seiner Ja-Eingaben*. Beim Cliquesproblem ist das die Menge aller Paare (G, k) , für die G tatsächlich eine k -Clique besitzt.

2.4 Ein Stolperstein, der später wichtig wird

✗ Stolperstein

Zahlen sind binär kodiert – und dadurch „riesig“ gegenüber ihrer Länge. Eine Zahl K steht als Eingabe nur mit ihren Ziffern da, also mit Länge etwa $\log K$. Ihr *Wert* K ist aber exponentiell größer als diese Länge. Ein Algorithmus, der „von 1 bis K zählt“, macht K Schritte – und das ist exponentiell in der Eingabelänge, nicht polynomiell. An diesem scheinbar harmlosen Punkt hängt später, warum SUBSET SUM schwer ist und was das Wort „pseudopolynomiell“ bedeutet. Merk ihn dir.

? Abruf – erst selbst beantworten

- A3.** Ist ein Algorithmus mit Laufzeit n^4 „effizient“ im Sinne der Theorie? Begründe.
A4. „Zähle von 1 bis zur eingegebenen Zahl K “ sieht kinderleicht aus. Warum ist dieser Algorithmus trotzdem *nicht* polynomiell?

3 Prüfen ist leichter als Lösen

★ In diesem Kapitel

Wir entdecken die zentrale Beobachtung der Theorie: Für viele schwere Probleme ist es hart, eine Lösung zu *finden*, aber leicht, eine vorgelegte Lösung zu *prüfen*. Daraus entsteht die Klasse NP – und ein Rezept, mit dem du für jedes Problem zeigst, dass es dazugehört.

3.1 Die Sudoku-Beobachtung

🗉 Analogie

Stell dir ein schweres Sudoku vor. Es zu *lösen* kann eine halbe Stunde dauern. Aber wenn dir jemand ein fertig ausgefülltes Gitter hinlegt und fragt „stimmt das?“, brauchst du eine Minute: Du prüfst jede Zeile, jede Spalte, jeden Block. *Lösen ist schwer, Prüfen ist leicht*. Diese Asymmetrie ist der ganze Kern von NP.

Genau so ist es bei der 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 im Handumdrehen: Du schaust für jedes Paar nach, ob die Kante da ist. Die vorgelegte Gruppe ist der Beweis – wir nennen sie ein *Zertifikat*.

3.2 Zertifikat und Verifizierer

■ Definition 3.1 Verifizierer und Zertifikat

Ein *Verifizierer* für ein Problem L ist ein Algorithmus A , der *zwei* Dinge bekommt: die eigentliche Frage (die Instanz x) und einen Lösungsvorschlag c . Er soll erfüllen:

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

Der Vorschlag c heißt *Zertifikat*.

Lies die beiden Richtungen einzeln, sie sind beide wichtig:

- Ist x eine *Ja*-Instanz, dann *gibt es* ein Zertifikat, das A überzeugt.
- Ist x eine *Nein*-Instanz, dann überzeugt A *kein* Zertifikat – egal, was man ihm vorlegt.

■ Definition 3.2 Die Klasse NP

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

★ Intuition

NP heißt „effizient *überprüfbar*“, nicht „effizient *lösbar*“. Das ist der ganze Witz. Eine Lösung zu finden mag exponentiell teuer sein – aber *hätte* man sie, wäre die Kontrolle billig. Das Zertifikat ist die kurze Antwort auf die Frage „und woher weiß ich, dass das stimmt?“.

3.3 Der NP-Dreischritt: so beweist man „ $\in$ NP“

Für *jedes* Problem läuft der Nachweis „liegt in NP“ nach demselben Muster ab. Dieses Rezept begegnet dir im ganzen Kurs, also lohnt es sich, es einmal in Ruhe anzusehen.

🔑 Idee: Der NP-Dreischritt

(1) **Zertifikat benennen.** Was ist die „Lösung“ des Problems? Eine Teilmenge, eine Belegung, eine Reihenfolge? Sag außerdem, dass sie polynomiell lang ist (sonst könnte man sie nicht schnell lesen).

(2) **Verifizierer beschreiben.** Ein Algorithmus, der *alle* geforderten Eigenschaften prüft. Dazu gehören immer drei Punkte:

- die Prüfschritte selbst,
- die Korrektheit in *beiden* Richtungen (Ja-Instanz $\Rightarrow$ ein Zertifikat wird akzeptiert; Nein-Instanz $\Rightarrow$ keines),
- die Laufzeit, konkret als $O(\cdot)$.

(3) **Alternativ: die Rate-Sicht.** Man darf NP auch über eine „ratende“ Maschine beschreiben, die die Lösung errät und dann prüft. Die Rateschritte *sind* das Zertifikat – es ist nur eine andere Sprache für dieselbe Sache.

Machen wir den Dreischritt einmal ganz konkret.

● Beispiel 3.3 Der NP-Dreischritt für Clique

Zertifikat: eine Knotenmenge C (etwa als Liste der gewählten Knoten). Sie ist höchstens so lang wie die Knotenzahl, also polynomiell.

Verifizierer: Prüfe zwei Dinge – erstens, ob wirklich *jedes* Paar aus C durch eine Kante verbunden ist (das kostet $O(|C|^2)$ Blicke in die Kantenliste); zweitens, ob $|C| \geq k$ ist ($O(|V|)$).

Korrektheit: Gibt es eine k -Clique, so ist genau sie ein Zertifikat, das akzeptiert wird. Gibt es keine, scheitert jeder Vorschlag an der Paar-Prüfung. Beide Richtungen stimmen.

Laufzeit: polynomiell. Also liegt CLIQUE in NP.

3.4 Die zweite Sicht: Raten

★ Intuition

Die dritte Zeile des Dreischritts verdient einen eigenen Gedanken. Stell dir einen Algorithmus vor, der an jeder Verzweigung nicht *eine* Möglichkeit wählt, sondern *alle gleichzeitig* verfolgt – wie ein Läufer, der sich an jeder Kreuzung in Kopien aufteilt und jede Kopie einen Weg gehen lässt. So ein Algorithmus heißt *nichtdeterministisch*. Er „akzeptiert“, wenn *irgendeine* Kopie ans Ziel kommt. Für eine Nein-Instanz kommt keine Kopie an. „Richtig raten“ und „ein Zertifikat geschenkt bekommen“ sind zwei Namen für genau dieselbe Fähigkeit.

3.5 P steckt in NP – und die Millionenfrage

▲ Satz 3.4 $P \subseteq NP$

Jedes effizient *lösbar*e Problem ist auch effizient *überprüfbar*.

Warum: Wenn du ein Problem selbst schnell lösen kannst, brauchst du das Zertifikat gar nicht. Du ignorierst es und rechnest die Antwort direkt aus. Der Löser ist damit ein besonders fauler Verifizierer.

Die spannende Frage ist die *Umkehrung*: Ist auch alles, was man schnell *prüfen* kann, schnell *lösbar*? Gilt also $P = NP$?

☆ Aha

Ob $P = NP$ gilt, weiß bis heute niemand. Es ist eines der sieben Millennium-Probleme, ausgelobt mit einer Million Dollar. Die meisten Fachleute vermuten $P \neq NP$ – dass Prüfen also *echt* leichter ist als Lösen. Beweisen konnte es keiner. Und der ganze Rest dieses Kurses – NP-Vollständigkeit, ETH, Approximation – ist nichts anderes als der kluge Umgang mit genau dieser Unwissenheit.

? Abruf – erst selbst beantworten

A5. Führe den NP-Dreischritt für VERTEX COVER durch: Was ist das Zertifikat, was prüft der Verifizierer, wie lange dauert das?

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

4 Reduktionen: Probleme miteinander vergleichen

★ In diesem Kapitel

Jetzt kommt das wichtigste Werkzeug des ganzen Kurses. Mit der *Reduktion* vergleichst du zwei Probleme nach Schwierigkeit – und zwar, ohne für auch nur eines von beiden einen Algorithmus zu kennen. Das klingt paradox. Wir bauen es langsam auf.

4.1 Die Grundidee: ein Problem durch ein anderes lösen

Angenommen, du hast ein fertiges Werkzeug für Problem B – eine Maschine, in die du eine B -Frage steckst und die Ja oder Nein ausspuckt. Jetzt stehst du vor Problem A , für das du *kein* Werkzeug hast. Kannst du A trotzdem lösen?

Ja – *falls* du jede A -Frage so *umformen* kannst, dass die B -Maschine sie beantwortet. Dann löst du A über den Umweg B : erst umformen, dann die B -Maschine fragen, deren Antwort ist auch die Antwort auf deine A -Frage.

Analogie

Du möchtest wissen, ob eine Strecke länger als eine Meile ist, hast aber nur ein Lineal mit Kilometer-Skala. Kein Problem: Du rechnest die Frage „länger als 1 Meile?“ um in „länger als 1,609 km?“ und misst dann mit dem km-Lineal. Die *Umrechnung* ist billig, das eigentliche *Messen* erledigt das fremde Werkzeug. Genau so arbeitet eine Reduktion: billiges Umformen plus fremder Löser.

4.2 Die präzise Definition

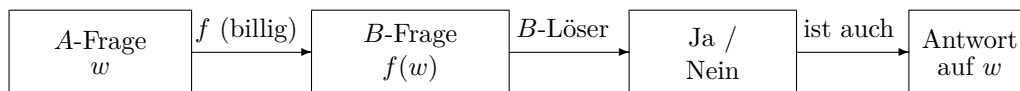
■ Definition 4.1 Polynomielle Reduktion $A \leq_p B$

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

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

Sie heißt *polynomiell*, wenn ein Algorithmus $f(w)$ in polynomieller Zeit berechnet. Wir schreiben dann $A \leq_p B$.

Die Bedingung $w \in A \iff f(w) \in B$ verlangt *beide* Richtungen: Aus einer Ja-Instanz von A wird eine Ja-Instanz von B , und aus einer Nein-Instanz von A wird eine Nein-Instanz von B . Nur so bleibt die Antwort erhalten. Das folgende Bild zeigt den Datenfluss, wenn man mit einer Reduktion das Problem A über einen B -Löser knackt.



★ Intuition

Was bedeutet $A \leq_p B$ inhaltlich? Genau dies: „ B ist *mindestens so schwer* wie A .“ Denn wer B schnell lösen kann, löst mit der billigen Umformung f auch A schnell. Anders gesagt: Die Schwierigkeit *fließt entlang des Pfeils*. Wenn A schwer ist, dann kann B nicht leicht sein – sonst wäre über den Umweg auch A leicht.

4.3 Die Richtung – lies das zweimal

Hier passiert der häufigste und teuerste Fehler im ganzen Kurs. Nimm dir Zeit.

Du willst normalerweise zeigen, dass ein *neues* Problem Y *schwer* ist. Dazu nimmst du ein Problem X , von dem du *schon weißt*, dass es schwer ist, und reduzierst X auf Y – also $X \leq_p Y$.

✗ Stolperstein

Die Richtung ist **bekannt schwer** $\leq_p$ **neu**, niemals umgekehrt. Die Reduktion nimmt eine Instanz des *bekannten* Problems und baut daraus eine Instanz des *neuen*. Warum genau so? Weil $X \leq_p Y$ 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 höchstens so schwer wie das schwere X “ – und das sagt über Y überhaupt nichts.

🗨 Analogie

Du willst beweisen, dass ein neuer Boxer stark ist. Also lässt du den *amtierenden Champion* gegen ihn antreten. Besiegt der Neue den Champion, ist er stark. Du würdest nie den Neuen gegen einen *Anfänger* antreten lassen – ein Sieg da beweist nichts. „Champion (bekannt stark) gegen Neuen.“ Genau das ist $X_{\text{bekannt}} \leq_p Y_{\text{neu}}$.

4.4 Reduktionen lassen sich verketten

▲ Satz 4.2 Transitivität

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

Warum: Man schaltet die beiden Umformungen einfach hintereinander – erst $A \rightarrow B$, dann $B \rightarrow C$. Die Zwischenausgabe ist polynomiell groß, also bleibt auch die Gesamt-Umformung polynomiell. Zwei billige Schritte ergeben zusammen einen billigen Schritt.

Das ist mehr als eine technische Note: Es erlaubt uns, eine ganze *Kette* von Reduktionen zu bauen. Ist das erste Problem als schwer bekannt, trägt die Kette diese Schwere Glied für Glied bis ans Ende. Genau diese Kette bauen wir ab dem nächsten Kapitel auf.

? Abruf – erst selbst beantworten

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

A8. Formuliere in einem Satz, was $A \leq_p B$ über die relative Schwierigkeit von A und B aussagt.

5 Die härtesten Probleme: NP-vollständig

★ In diesem Kapitel

Mit der Reduktion in der Hand können wir jetzt sagen, was „die schwersten Probleme in NP“ genau sind – und warum ein einziger schneller Algorithmus für eines von ihnen die gesamte Landschaft zum Einsturz brächte.

5.1 Zwei Begriffe: NP-schwer und NP-vollständig

■ Definition 5.1 NP-schwer und NP-vollständig

- L_0 heißt *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 heißt *NP-vollständig*, wenn es NP-schwer ist **und** selbst in NP liegt.

★ Intuition

Denk an zwei Schranken, die von oben und unten zusammenkommen:

- *NP-schwer* ist eine **untere** Schranke – „mindestens so schwer wie alles in NP“. Alle NP-Probleme reduzieren auf L_0 , also ist L_0 mindestens so hart wie jedes von ihnen.
- $\in \text{NP}$ ist eine **obere** Schranke – „nicht schwerer als NP“.

NP-vollständig heißt: beide Schranken treffen sich. Das sind genau die schwersten Probleme *innerhalb* von NP.

✗ Stolperstein

NP-schwer heißt *nicht* automatisch „liegt in NP“. Es gibt Probleme, die NP-schwer sind, aber weit außerhalb von NP liegen – das Halteproblem (Kapitel 10) ist nicht einmal lösbar und trotzdem NP-schwer. „Schwer“ und „überhaupt in NP“ sind zwei verschiedene Achsen.

5.2 Warum ein Algorithmus für eines alle löst

▲ Satz 5.2 NP-vollständig verbindet P und NP

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

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

☆ Aha

Das ist die Sprengkraft der NP-Vollständigkeit. Fände jemand für *ein einziges* NP-vollständiges Problem einen effizienten Algorithmus, dann wären *schlagartig alle* Probleme in NP effizient lösbar – die Welt wäre eine andere. Genau deshalb gilt umgekehrt: Solange das niemandem gelingt, ist ein NP-Vollständigkeitsbeweis das stärkste bekannte Indiz dafür, dass ein Problem *keinen* effizienten Algorithmus hat.

5.3 Der Anker: der Satz von Cook und Levin

Damit die Reduktionskette überhaupt losgehen kann, braucht man ein *erstes* NP-vollständiges Problem – eines, das man von Hand als schwer nachweist, ohne sich auf ein anderes stützen zu können. Cook und Levin haben es geliefert. Es heißt SAT: Gegeben eine aussagenlogische Formel, gibt es eine Belegung ihrer Variablen, die sie wahr macht?

▲ Satz 5.3 Cook (1971) und Levin (1973)

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

Die Beweisidee in einem Bild: Man nimmt ein *beliebiges* Problem $L \in \text{NP}$ mit seiner ratenden Maschine und „gießt“ deren Rechnung in eine große logische Formel. Variablen der Formel beschreiben Aussagen wie „zur Zeit t ist die Maschine im Zustand q “ oder „zur Zeit t steht auf Bandfeld i das Symbol a “. Weitere Teilformeln erzwingen, dass ein gültiger Rechenablauf beschrieben wird. Das Entscheidende: Eine *erfüllende Belegung* der Formel entspricht *genau* einem akzeptierenden Rechenweg der Maschine. Also gilt: L -Ja-Instanz $\iff$ Formel erfüllbar. Da L beliebig war, ist SAT schwer für ganz NP.

Du musst diesen Beweis nicht im Detail beherrschen. Wichtig ist seine Rolle: SAT ist der *Anker*, an dem die ganze Kette hängt. Ab hier braucht man nie wieder „alle $L \in \text{NP}$ “ zu betrachten.

5.4 Das Arbeitspferd: das Vererbungskorollar

👉 Idee: Vererbungskorollar – so wächst die Liste

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

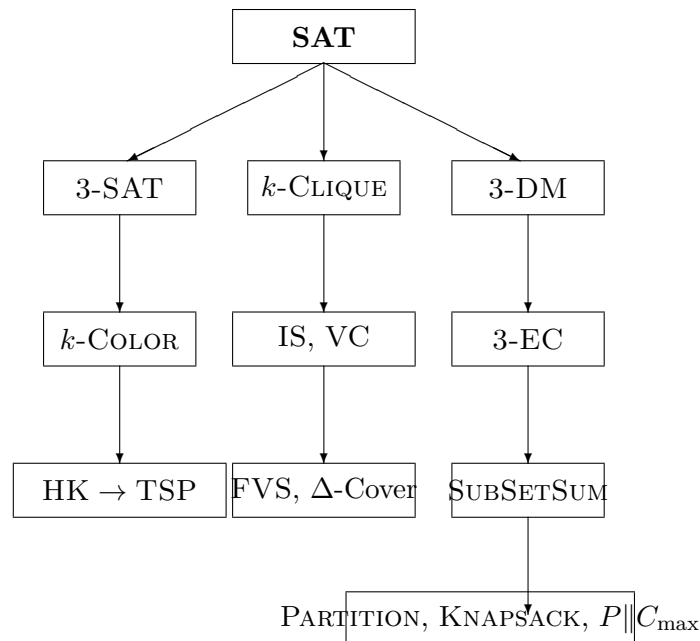
1. eine Reduktion $X \leq_p Y$ von *einem* bereits bekannten NP-vollständigen Problem X , und
2. den Nachweis $Y \in \text{NP}$ (den NP-Dreischritt aus Kapitel 3).

Dann ist Y automatisch NP-vollständig. Denn für jedes $L \in \text{NP}$ gilt $L \leq_p X \leq_p Y$ (Transitivität) – also ist Y NP-schwer, und mit $Y \in \text{NP}$ auch NP-vollständig.

Kein Wort mehr über „alle $L \in \text{NP}$ “. Eine Reduktion, ein NP-Nachweis – fertig. Genau so entsteht die folgende Landkarte.

5.5 Die Landkarte der Schwere

Alle Probleme des Kurses hängen an *einer* Kette, die bei SAT beginnt. Jeder Pfeil ist eine Reduktion und heißt „wird reduziert auf“; die Schwere fließt von oben nach unten.



Im nächsten Kapitel gehen wir jedes dieser Probleme einzeln durch. Danach zeichnen wir die wichtigsten Pfeile – die Reduktionen – im Detail nach.

? Abruf – erst selbst beantworten

A9. Welche zwei Dinge musst du zeigen, um ein neues Problem als NP-vollständig zu beweisen? Welches der beiden wird am häufigsten vergessen?

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

6 Der Problem-Zoo

★ In diesem Kapitel

Wir lernen alle Probleme des Kurses kennen – jedes einzeln, mit Ruhe. Zu jedem gibt es die Definition (was gefragt ist) und die Begründung, *warum es in NP liegt* (das Zertifikat). Woher die Schwere kommt und wie die Reduktionen konkret arbeiten, folgt im nächsten Kapitel. Alle diese Probleme sind NP-vollständig.

6.1 SAT – das Ur-Problem der Logik

SAT steht am Anfang von allem. Es geht um aussagenlogische Formeln in einer festen Form, der *konjunktiven Normalform* (KNF): ein UND von *Klauseln*, wobei jede Klausel ein ODER von *Literalen* ist. Ein Literal ist eine Variable x_j oder ihre Verneinung $\neg x_j$.

■ Definition 6.1 SAT

Gegeben eine KNF-Formel $\alpha = C_1 \wedge \dots \wedge C_m$. Frage: Gibt es eine Belegung der Variablen mit wahr/falsch, die die ganze Formel wahr macht (also *jede* Klausel erfüllt)?

● Beispiel 6.2 SAT von Hand lösen

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

✓ Warum in NP?

Zertifikat: die Belegung ψ (ein Bit pro Variable). **Verifizierer:** setze ψ ein und werte jede Klausel aus; akzeptiere, wenn alle wahr sind. Das kostet lineare Zeit in der Formellänge. Also $\text{SAT} \in \text{NP}$.

6.2 3-SAT – SAT mit kurzen Klauseln

■ Definition 6.3 3-SAT

Wie SAT, aber jede Klausel hat *höchstens* 3 Literale.

Warum eine eigene Version? Weil kurze Klauseln viel leichter in andere Probleme „einzubauen“ sind. Fast alle späteren Reduktionen starten deshalb bei 3-SAT, nicht beim allgemeinen SAT. Der NP-Nachweis ist derselbe wie bei SAT (Belegung als Zertifikat).

6.3 Clique, Independent Set, Vertex Cover – ein Trio

Diese drei Graphprobleme hängen so eng zusammen, dass man sie am besten gemeinsam versteht. Zuerst jedes für sich.

■ Definition 6.4 Clique, Independent Set, Vertex Cover

Gegeben ein Graph G und eine Zahl k .

- **CLIQUE:** eine Knotengruppe, in der *alle* paarweise verbunden sind – gibt es eine der Größe $\geq k$?
- **INDEPENDENT SET (IS):** eine Knotengruppe, in der *keine* zwei verbunden sind – gibt es eine der Größe $\geq k$?
- **VERTEX COVER (VC):** eine Knotenmenge, die *jede* Kante berührt (mindestens ein Endpunkt liegt drin) – gibt es eine der Größe $\leq k$?

✓ Warum in NP?

Für alle drei ist das Zertifikat die *Knotenmenge* selbst. **Clique:** prüfe, ob alle Paare verbunden sind. **IS:** prüfe, ob kein Paar verbunden ist. **VC:** prüfe, ob jede Kante einen Endpunkt in der Menge hat. Dazu jeweils die Größe. Alles in $O(|V|^2)$ bzw. $O(|E|)$ – also liegen alle drei in NP.

Jetzt der Grund, warum man sie zusammen lernt. Es ist *ein und dasselbe Bild*, nur dreimal anders betrachtet.

☆ Aha

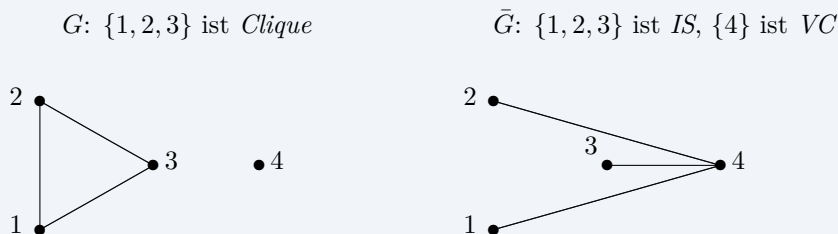
Das Dualitätsdreieck. Betrachte einen Graphen G und seinen *Komplementgraphen* $\bar{G}$ (dieselben Knoten, aber genau die Kanten, die G *nicht* hat). Dann gilt für jede Knotenmenge C :

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

Wer eines der drei versteht, versteht alle drei.

Das folgende Bild macht es greifbar: links ein Graph G , in dem $\{1, 2, 3\}$ eine Clique ist; rechts sein Komplement $\bar{G}$, in dem dieselbe Menge $\{1, 2, 3\}$ ein Independent Set ist – und der übrige Knoten $\{4\}$ ein Vertex Cover.

● Beispiel 6.5 Ein Bild, drei Rollen



In $\bar{G}$ sind genau die Nicht-Kanten von G eingezeichnet. $\{1, 2, 3\}$ hat dort keine innere Kante mehr (Independent Set). Und Knoten 4 allein berührt alle drei Kanten von $\bar{G}$ (Vertex Cover).

6.4 k -Color – die Landkarten-Färbung

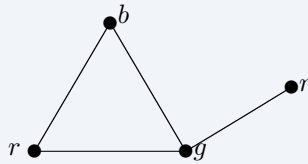
■ Definition 6.6 k -Color

Gegeben ein Graph G und eine Zahl k . Kann man jedem Knoten eine von k Farben geben, sodass *keine Kante* zwei gleichfarbige Enden verbindet?

Man denkt an eine Landkarte, deren Länder so gefärbt werden, dass keine zwei Nachbarn

dieselbe Farbe tragen. Schon $k = 3$ ist NP-vollständig.

● Beispiel 6.7 Eine gültige 3-Färbung



Drei Farben r, g, b . Jede Kante verbindet zwei *verschiedene* Farben – das Dreieck braucht alle drei, der rechte Knoten darf wieder r sein, weil er nur mit g benachbart ist.

✓ Warum in NP?

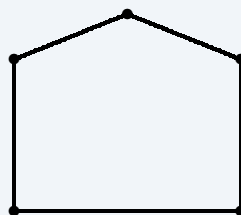
Zertifikat: die Färbung f (eine Farbe pro Knoten). **Verifizierer:** gehe alle Kanten durch und prüfe, ob ihre Enden verschiedene Farben haben – $O(|E|)$. Also k -COLOR $\in$ NP.

6.5 Hamiltonkreis – die Rundreise durch alle Knoten

■ Definition 6.8 Hamiltonkreis (HK)

Gegeben ein Graph G . Gibt es einen Rundweg, der *jeden* Knoten *genau einmal* besucht und wieder am Start endet?

● Beispiel 6.9 Ein Hamiltonkreis



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

✓ Warum in NP?

Zertifikat: die Besuchsreihenfolge (eine Permutation der Knoten). **Verifizierer:** prüfe, ob jeder Knoten genau einmal vorkommt und ob jede aufeinanderfolgende – und die schließende – Kante wirklich existiert; $O(|V| + |E|)$. Also HK $\in$ NP.

6.6 TSP – die kürzeste Rundreise

■ Definition 6.10 TSP (Entscheidungsvariante)

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

HK fragt nur „gibt es überhaupt eine Rundreise?“; TSP fügt Distanzen und eine Längengrenze hinzu. Als Optimierungsproblem („finde die kürzeste Rundreise“) ist TSP der Star des Approximationskapitels.

✓ Warum in NP?

Zertifikat: die Reihenfolge der Städte. **Verifizierer:** addiere die Distanzen entlang der Rundreise und vergleiche mit L ; prüfe, dass jede Stadt genau einmal vorkommt. Linear. Also $\text{TSP} \in \text{NP}$.

6.7 Die Mengen- und Zahlprobleme

Die letzte Familie verlässt die Graphen und arbeitet mit Mengen und Zahlen. Wir stellen sie kurz nacheinander vor; ihr NP-Nachweis ist überall gleich einfach.

■ Definition 6.11 3-DM, 3-EC, SubSet Sum, Partition, Knapsack

- 3-DIM. MATCHING (3-DM): drei gleich große Mengen und eine Liste von Tripeln (je ein Element aus jeder Menge). Wähle Tripel, die jedes Element *genau einmal* treffen.
- 3-EXACT COVER (3-EC): eine Familie 3-elementiger Teilmengen eines Universums. Wähle einige, die das Universum *exakt* überdecken (disjunkt, lückenlos).
- SUBSET SUM: ganze Zahlen $c_1, \dots, c_n$ und ein Ziel K . Gibt es eine Teilmenge mit Summe *genau* K ?
- PARTITION: gibt es eine Teilmenge, die *genau die Hälfte* der Gesamtsumme trägt?
- KNAPSACK (RUCKSACK): Gegenstände mit Gewicht und Profit, Kapazität B , Zielprofit P . Gibt es eine Auswahl mit Gewicht $\leq B$ und Profit $\geq P$?

● Beispiel 6.12 SubSet Sum ausprobiert

Zahlen 3, 7, 8, 4 und Ziel $K = 15$. Probieren: $3 + 4 + 8 = 15$ – ja, die Teilmenge $\{3, 4, 8\}$ trifft das Ziel. Das Zertifikat ist genau diese Teilmenge.

✓ Warum in NP?

Für alle diese Probleme ist das Zertifikat die getroffene *Auswahl* (Tripelmenge, Mengenauswahl, Teilmenge). Der Verifizierer rechnet die geforderte Summe bzw. Überdeckung nach und vergleicht mit den Schranken – alles in $O(n)$ bis $O(n^2)$. Also liegen alle in NP.

6.8 Scheduling: $P||C_{\max}$

■ Definition 6.13 $P||C_{\max}$ (Scheduling auf identischen Maschinen)

Gegeben n Jobs mit Laufzeiten $p_1, \dots, p_n$ und m gleiche Maschinen. Verteile die Jobs auf die Maschinen so, dass die *höchste* Maschinenlast – der *Makespan* $C_{\max}$ – möglichst klein wird.

✓ Warum in NP?

Zertifikat: die Zuordnung „welcher Job auf welche Maschine“. **Verifizierer:** summiere pro Maschine die Laufzeiten, nimm das Maximum, vergleiche mit der Schranke – $O(n)$. Also liegt (die Entscheidungsvariante von) $P||C_{\max}$ in NP.

? Abruf – erst selbst beantworten

A11. Führe den NP-Dreischritt für HAMILTONKREIS aus: Zertifikat, Prüfung, Laufzeit.

A12. Erkläre das Dualitätsdreieck: Wie hängen CLIQUE, IS und VC zusammen, und über welchen Graphen?

7 Reduktionen im Detail

★ In diesem Kapitel

Jetzt zeichnen wir die Pfeile der Landkarte nach. Für jede Reduktion sehen wir: das Ziel, die Konstruktion Schritt für Schritt, ein Bild oder Zahlenbeispiel und – besonders wichtig – *beide* Richtungen der Korrektheit. Nimm dir für jede Reduktion Zeit; hier entsteht das eigentliche Verständnis.

Jede Reduktion beweist „ $X \leq_p Y$ “, also „ Y ist mindestens so schwer wie das bekannte X “. Der Aufbau ist immer gleich: Wir bauen aus einer X -Instanz eine Y -Instanz und zeigen, dass die Ja/Nein-Antwort dieselbe bleibt. „Beide Richtungen“ heißt: Ja-Instanz von X wird zu Ja-Instanz von Y ($\Rightarrow$), und umgekehrt ($\Leftarrow$).

7.1 $\text{SAT} \leq_p \text{Clique}$: aus Logik wird ein Graph

Ziel. Zeige, dass CLIQUE mindestens so schwer wie SAT ist, indem du aus einer Formel einen Graphen baust.

Die Konstruktion. Sei $F = F_1 \wedge \dots \wedge F_m$ eine KNF-Formel.

- *Ein Knoten pro Literalvorkommen.* Für das j -te Literal in Klausel i setze einen Knoten $[i, j]$.
- *Kanten nur zwischen „verträglichen“ Literalen.* Verbinde $[i, j]$ und $[i', j']$ genau dann, wenn sie in *verschiedenen* Klauseln liegen ($i \neq i'$) und sich nicht widersprechen (nicht x gegen $\neg x$).
- *Gesuchte Cliquengröße $k = m$* (die Zahl der Klauseln).

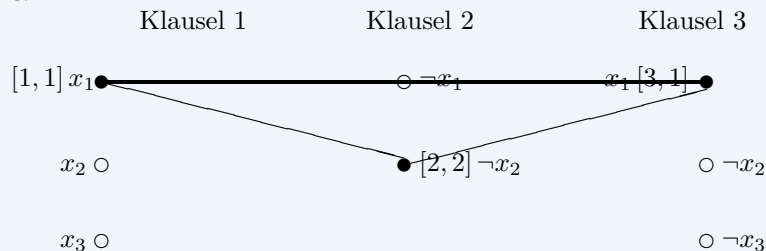
Diese Konstruktion ist offensichtlich in Polynomialzeit machbar.

★ Intuition

Warum $k = m$? Eine m -Clique muss aus *jeder* Klausel genau einen Knoten nehmen – denn Knoten derselben Klausel sind nie verbunden, es passt also höchstens einer pro Klausel hinein. Und weil sich verbundene Knoten nicht widersprechen, wählt die Clique aus jeder Klausel ein Literal, ohne sich zu widersprechen. Das ist *genau* eine erfüllende Belegung.

● Beispiel 7.1 Die Konstruktion an einem Beispiel

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



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

Korrektheit, beide Richtungen.

- $\Rightarrow$ Ist F erfüllbar, wähle in jeder Klausel ein wahres Literal. Die zugehörigen Knoten liegen in verschiedenen Klauseln und widersprechen sich nicht (alle sind ja wahr) – sie bilden eine m -Clique.
- $\Leftarrow$ Hat der Graph eine m -Clique, so enthält sie aus jeder Klausel genau einen Knoten, und keine zwei widersprechen sich. Setze die zugehörigen Literale auf wahr – das ist widerspruchsfrei und erfüllt jede Klausel, also F .

7.2 Clique $\leq_p$ VC: das Komplement genügt

Ziel. VERTEX COVER ist mindestens so schwer wie CLIQUE. Und diesmal brauchen wir gar kein Gadget – das Dualitätsdreieck erledigt alles.

Die Konstruktion. Aus der CLIQUE-Instanz (G, k) mit n Knoten mache die VC-Instanz $(\bar{G}, n - k)$: nimm den *Komplementgraphen* und ersetze die Schranke k durch $n - k$. In $O(n^2)$ berechenbar.

★ Intuition

Woher kommt das $n - k$? Aus dem Dualitätsdreieck. Eine Clique der Größe k in G ist ein Independent Set der Größe k in $\bar{G}$. Und das Gegenstück eines Independent Set ist ein Vertex Cover: Ist C ein Independent Set, so berührt $V \setminus C$ jede Kante. Aus einem IS der Größe k wird also ein VC der Größe $n - k$.

Korrektheit.

- $\Rightarrow$ Hat G eine Clique C mit $|C| \geq k$, so ist C ein Independent Set in $\bar{G}$, und $V \setminus C$ ein Vertex Cover in $\bar{G}$ mit $|V \setminus C| \leq n - k$.
- $\Leftarrow$ Hat $\bar{G}$ ein Vertex Cover D mit $|D| \leq n - k$, so ist $V \setminus D$ ein Independent Set in $\bar{G}$ mit $\geq k$ Knoten – also eine Clique der Größe $\geq k$ in G .

7.3 VC $\leq_p$ FVS: ungerichtet wird gerichtet

FEEDBACK VERTEX SET (FVS) fragt: Kann man aus einem *gerichteten* Graphen $\leq k$ Knoten entfernen, sodass *kein gerichteter Kreis* mehr übrig bleibt?

Die Konstruktion. Aus der VC-Instanz (G, k) mache jeden ungerichteten Kante $\{u, v\}$ zu zwei *antiparallelen Bögen* $u \rightarrow v$ und $v \rightarrow u$ – also einem kleinen Kreis der Länge 2. Schranke bleibt k .

● Beispiel 7.2 Eine Kante wird zu einem 2-Kreis



★ Intuition

Der 2-Kreis $u \rightarrow v \rightarrow u$ existiert genau wegen dieser einen Kante. Um ihn zu zerstören, muss man u oder v entfernen – also die Kante „überdecken“. Kreise kaputtmachen (FVS) und Kanten überdecken (VC) wird so dasselbe.

Korrektheit.

- $\Rightarrow$ Ist C ein Vertex Cover mit $|C| \leq k$, so berührt C jede Kante – also enthält C von jedem

2-Kreis einen Knoten. Entfernt man C , verschwindet jeder 2-Kreis. Da alle Kreise im konstruierten Graphen aus solchen Kanten bestehen, ist er danach kreisfrei.

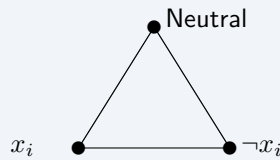
$\Leftarrow$ Ist X ein FVS mit $|X| \leq k$, so muss X jeden 2-Kreis treffen – also jede ursprüngliche Kante überdecken. Damit ist X ein Vertex Cover.

7.4 3-SAT $\leq_p$ 3-Color: Wahrheit als Farbe

Diese Reduktion ist raffinierter, deshalb nur die tragende Idee – die reicht fürs Verständnis.

Die Idee. Man benutzt drei Farben und tauft sie *Wahr*, *Falsch* und *Neutral*. Für jede Variable x_i baut man ein kleines *Gadget* aus den Knoten x_i und $\neg x_i$, die miteinander und mit einem gemeinsamen *Neutral*-Knoten verbunden sind. Dieses Dreieck erzwingt, dass x_i und $\neg x_i$ die beiden übrigen Farben bekommen – eine wird *Wahr*, die andere *Falsch*. So kodiert die Färbung eine Belegung.

● Beispiel 7.3 Das Variablen-Gadget



Das Dreieck braucht alle drei Farben. Der *Neutral*-Knoten nimmt die neutrale Farbe – also müssen x_i und $\neg x_i$ *Wahr* und *Falsch* tragen, in irgendeiner Reihenfolge. Genau das ist „die Variable ist wahr oder falsch, aber nicht beides“.

Für jede *Klausel* baut man ein weiteres Gadget, das genau dann korrekt gefärbt werden kann, wenn mindestens eines ihrer Literale die Farbe *Wahr* trägt. Zusammengesetzt gilt: Die Formel ist erfüllbar $\iff$ der Graph ist 3-färbbar.

7.5 3-EC $\leq_p$ SubSet Sum: Mengen werden Zahlen

Das ist die eleganteste Reduktion des Kurses. Sie verwandelt eine Mengen-Überdeckung in eine reine Zahlen-Addition.

Die Idee. Schreibe jede 3-elementige Menge als *Bitvektor* über dem Universum (eine 1 an jeder Stelle, die die Menge enthält). Lies diesen Bitvektor dann als *Zahl zur Basis $n + 1$* , wobei n die Anzahl der Mengen ist. Das Ziel K ist die Zahl zum Bitvektor „überall 1“.

★ Intuition

Warum die krumme Basis $n + 1$ statt einfach Basis 2? Damit beim Addieren *kein Übertrag* entsteht. Es gibt nur n Mengen, also an jeder Stelle höchstens n Einsen – das bleibt unter $n + 1$, die Stelle „läuft nie über“. Ohne Übertrag zählt jede Stelle für sich, und „Summe trifft genau das Ziel“ bedeutet „jede Stelle genau einmal überdeckt“ – also eine exakte Überdeckung. Bei Basis 2 gäbe es Überträge, und falsche Überdeckungen könnten die Zielsumme zufällig treffen.

● Beispiel 7.4 Komplett durchgerechnet

Universum $U = \{1, 2, 3, 4, 5, 6\}$ (also $3m$ mit $m = 2$). Mengen ($n = 4$): $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{1, 2, 4\}$, $S_4 = \{3, 5, 6\}$. Basis $n + 1 = 5$; Stelle für Element i ist 5^{i-1} .

Menge	Bitvektor ($u_1 \dots u_6$)	Zahl zur Basis 5
$S_1 = \{1, 2, 3\}$	111000	$5^0 + 5^1 + 5^2 = 31$
$S_2 = \{4, 5, 6\}$	000111	$5^3 + 5^4 + 5^5 = 3875$
$S_3 = \{1, 2, 4\}$	110100	$5^0 + 5^1 + 5^3 = 131$
$S_4 = \{3, 5, 6\}$	001011	$5^2 + 5^4 + 5^5 = 3775$
Ziel K	111111	$5^0 + \dots + 5^5 = 3906$

Rechne nach: $S_1 + S_2 = 31 + 3875 = 3906 = K$. Und tatsächlich ist $\{S_1, S_2\}$ eine exakte Überdeckung von U . (Ebenso $S_3 + S_4 = 131 + 3775 = 3906$, und $\{S_3, S_4\}$ überdeckt U ebenfalls exakt.) Jede Auswahl, deren Zahlen sich zu 3906 addieren, trifft jede Stelle genau einmal – eine exakte Überdeckung.

7.6 SubSet Sum $\leq_p$ Partition: zwei Zusatzzahlen

Ziel. PARTITION (Teilmenge mit halber Gesamtsumme) ist mindestens so schwer wie SUBSET SUM.

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

★ Intuition

Die Gesamtsumme der neuen Menge ist $\sum c_j + (N - K) + (K + 1) = \sum c_j + N + 1 = 2N$. Die Hälfte ist also N . Der Clou: Die beiden Zusatzzahlen summieren sich zu $(N - K) + (K + 1) = N + 1$ – das ist schon *mehr* als eine Hälfte. Sie können also niemals *beide* in derselben Hälfte liegen; sie werden immer getrennt.

● Beispiel 7.5 Mit echten Zahlen

Zahlen 3, 5, 8, 4 (Summe 20), Ziel $K = 12$. Dann $N = 21$, die Zusatzzahlen sind $N - K = 9$ und $K + 1 = 13$. Neue Menge: $\{3, 5, 8, 4, 9, 13\}$ mit Gesamtsumme 42, Hälfte 21.

Die Original-Teilmenge $\{8, 4\}$ trifft $K = 12$. Ergänzt um die Zusatzzahl 9 ergibt $\{8, 4, 9\}$ die Summe 21 – eine gültige Halbierung. Umgekehrt: Jede Halbierung enthält genau eine der beiden Zusatzzahlen; zieht man sie ab, bleibt eine Original-Teilmenge, die K trifft.

Korrektheit.

$\Rightarrow$ Trifft eine Teilmenge S der Originalzahlen den Wert K , so hat $S \cup \{N - K\}$ die Summe $K + (N - K) = N$ – eine Halbierung.

$\Leftarrow$ Jede Halbierung enthält genau eine Zusatzzahl. Enthält sie $N - K$, so tragen die Originalzahlen darin $N - (N - K) = K$ bei – eine Teilmenge mit Summe K .

? Abruf – erst selbst beantworten

A13. Erkläre, warum bei $3\text{-EC} \leq_p \text{SUBSET SUM}$ die Basis $n + 1$ (und nicht Basis 2) gewählt wird.

A14. Bei $\text{SUBSET SUM} \leq_p \text{PARTITION}$: Warum können die beiden Zusatzzahlen nie in derselben Hälfte landen?

A15. Rechne die $\Rightarrow$ -Richtung von $\text{CLIQUE} \leq_p \text{VC}$ an einem selbst gewählten kleinen Graphen nach.

8 Wie schwer *genau*? Die ETH

★ In diesem Kapitel

NP-Vollständigkeit sagt nur „vermutlich nicht polynomiell“. Sie sagt nicht, *wie* exponentiell. Die Exponentialzeit-Hypothese (ETH) ist eine schärfere Annahme, aus der man konkrete untere Laufzeitschranken herleitet. Wir sehen auch, wie so ein Schranken-Beweis wirklich aussieht.

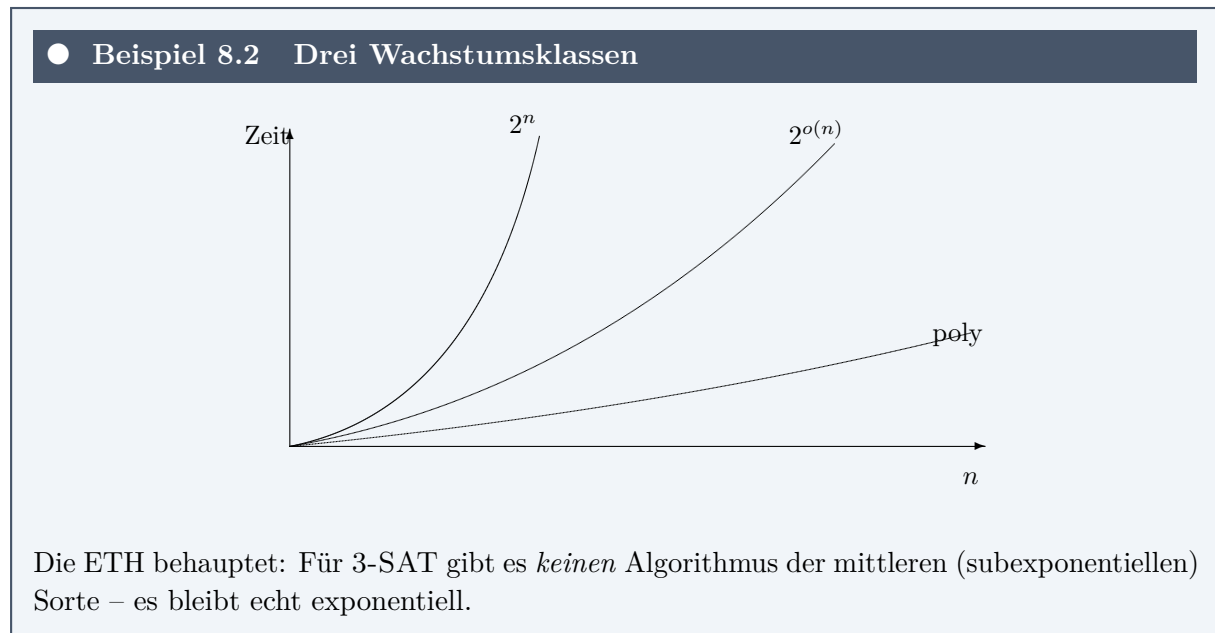
8.1 Die Lücke, die NP-Vollständigkeit offen lässt

NP-Vollständigkeit ist ein grobes Sieb. Sie wirft ein Problem, das 2^n braucht, in denselben Topf wie eines, das nur $2^{\sqrt{n}}$ braucht – beide heißen schlicht „nicht polynomiell“. Für viele Fragen will man aber genauer wissen: Geht es wenigstens *subexponentiell*, also spürbar besser als 2^n ?

■ Definition 8.1 Klein-o und „subexponentiell“

$f(n) = o(g(n))$ heißt: f wächst *echt langsamer* als g , formal $f(n)/g(n) \rightarrow 0$. Achtung: δn ist *nicht* $o(n)$ für ein festes $\delta > 0$. Ein „ $2^{o(n)}$ -Algorithmus“ hat einen Exponenten, der langsamer als linear wächst – er läuft *subexponentiell*, z. B. mit $2^{\sqrt{n}}$.

Das Bild zeigt die drei Welten nebeneinander: harmlos polynomiell, subexponentiell (die mittlere Kurve), und echt exponentiell.



8.2 Die Hypothese

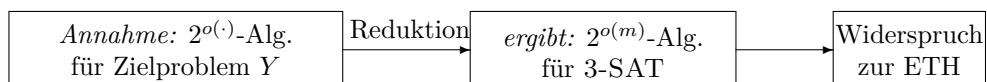
■ Definition 8.3 Exponentialzeit-Hypothese (ETH)

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

Das ist eine *Annahme* – stärker als $P \neq NP$, aber breit geglaubt, weil alle bekannten 3-SAT-Algorithmen c^n brauchen. Aus ihr leitet man untere Schranken für viele andere Probleme ab.

8.3 Wie ein ETH-Schranken-Beweis aussieht

Das Beweismuster ist immer *Kontraposition entlang einer Reduktion*: Man nimmt an, das Zielproblem hätte einen zu schnellen Algorithmus, und leitet daraus einen zu schnellen Algorithmus für 3-SAT ab – Widerspruch zur ETH.



Das Entscheidende – und der Grund, warum viele hier scheitern – ist die *Größenrechnung* im mittleren Pfeil. Machen wir sie einmal konkret.

Idee: Konkret: die Schranke für 3-Color

Es gibt eine Reduktion $3\text{-SAT} \leq_p 3\text{-COLOR}$, die aus einer Formel mit n Variablen und m Klauseln einen Graphen mit $O(n+m)$ *Knoten* baut. Nimm nun an, jemand könnte 3-COLOR in $2^{o(N)}$ lösen, wobei N die Knotenzahl ist. Setze die Reduktion davor:

$$\underbrace{2^{o(N)}}_{\text{Annahme}} \text{ mit } N = O(n+m) \implies 3\text{-SAT in } 2^{o(n+m)}.$$

Nun ist $n+m$ höchstens $O(m)$ (jede Variable kommt in einer Klausel vor, also $n \leq 3m$). Damit hätten wir einen $2^{o(m)}$ -Algorithmus für 3-SAT. Das ist verboten – *sofern* wir auch $2^{o(m)}$ ausschließen dürfen. Genau dafür brauchen wir den nächsten Satz.

8.4 Warum das Sparsification-Lemma nötig ist

▲ Satz 8.4 Sparsification-Lemma

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

★ Intuition

Beachte den feinen Unterschied: Die ETH selbst spricht über die *Variablenzahl* n . Die Größenrechnung oben landete aber bei einer Schranke in der *Klauselzahl* m . Und m kann viel größer als n sein (bis $\sim n^3$). Deshalb folgt „kein $2^{o(m)}$ “ *nicht* direkt aus „kein $2^{o(n)}$ “. Das Sparsification-Lemma füllt genau diese Lücke: Es überträgt die Härte von n auf m . Erst mit ihm schließt sich der Widerspruch im Bild oben.

✗ Stolperstein

Immer wenn die Größe deiner konstruierten Instanz an der *Klauselzahl* m hängt (und das ist fast immer so), musst du das Sparsification-Lemma zitieren. Es ist der am häufigsten vergessene Schritt in ETH-Beweisen.

Mit demselben Muster bekommt man unter der ETH untere Schranken $2^{o(n)}$ auch für CLIQUE, VERTEX COVER, INDEPENDENT SET, SUBSET SUM und PARTITION – jeweils mit der passenden Größenrechnung.

? Abruf – erst selbst beantworten

A16. Wenn die ETH *falsch* ist – folgt daraus $P = NP$? Begründe.

A17. Skizziere die Größenrechnung für die 3-COLOR-Schranke. An welcher Stelle kommt das Sparsification-Lemma ins Spiel?

9 Damit leben: gute Näherungen

★ In diesem Kapitel

Wenn ein Optimierungsproblem (vermutlich) keine schnelle exakte Lösung hat, berechnet man in Polynomialzeit eine *beweisbar gute* Näherung. Wir lernen, was „beweisbar gut“ heißt, und sehen die wichtigsten Algorithmen – den TSP-Algorithmus zeichnen wir Schritt für Schritt.

9.1 Was heißt „gute Näherung“?

■ Definition 9.1 Multiplikative Güte

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

- bei Minimierung (TSP, Scheduling): $A(I) \leq \alpha \cdot \text{OPT}(I)$ – die Lösung ist höchstens α -mal so teuer wie das Optimum;
- bei Maximierung (Knapsack): $\text{OPT}(I) \leq \alpha \cdot A(I)$ – die Lösung liefert mindestens ein α -tel des Optimums.

Die Güte heißt *scharf*, wenn es Instanzen gibt, die den Faktor (fast) erreichen – sie ist dann nicht zu pessimistisch.

★ 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 bestmögliche. Kleineres α ist besser; $\alpha = 1$ wäre exakt optimal.

Manche Probleme erlauben sogar eine einstellbare Genauigkeit – eine ganze Familie von Algorithmen, bei der du $\alpha = 1 + \varepsilon$ so nah an 1 wählst, wie du willst.

■ Definition 9.2 PTAS, EPTAS, FPTAS

Eine Familie (A_ε) mit Güte $1 + \varepsilon$ für jedes $\varepsilon > 0$ heißt:

- *PTAS*, wenn die Laufzeit für jedes feste ε polynomiell in n ist (ε darf im Exponenten stecken, z. B. $n^{1/\varepsilon}$);
- *EPTAS*, wenn die Laufzeit $f(1/\varepsilon) \cdot \text{poly}(n)$ ist – ε ist aus dem Exponenten von n heraus;
- *FPTAS*, wenn die Laufzeit polynomiell in n und $1/\varepsilon$ ist – die stärkste Form.

Ein Wort noch: *pseudopolynomiell* heißt eine Laufzeit, die polynomiell in den *Zahlenwerten* der Eingabe ist – was wegen des Stolpersteins aus Kapitel 2 *nicht* dasselbe ist wie polynomiell in der Eingabelänge.

9.2 TSP ist im Allgemeinen gar nicht approximierbar

▲ Satz 9.3 Kein Näherungsalgorithmus für allgemeines TSP

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

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

Der Ausweg ist eine realistische Zusatzannahme: das *metrische* TSP. Die 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. Echte Entfernungen erfüllen das immer. Und hier funktioniert ein schöner Algorithmus.

9.3 ΔTSP_1 : der Algorithmus als Bilderfolge

☞ Idee: Der Dreisprung

ΔTSP_1 baut zuerst ein billiges Gerüst, das alle Städte verbindet (einen *minimalen Spannbaum*), macht daraus einen Rundweg über alle Kanten (*Eulerkreis*, indem alle Knotengrade gerade werden) und kürzt diesen zur Tour ab. Wir gehen es an einem festen Beispiel durch – vier Bilder, ein Algorithmus.

Unser Beispiel: die Städte A, B, C, D, E mit $d(A, E) = 1$, $d(B, C) = 1$, $d(A, C) = 2$, $d(C, D) = 2$ (die übrigen Distanzen sind 2 oder 3). Das Layout der fünf Städte bleibt in allen Bildern gleich.

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

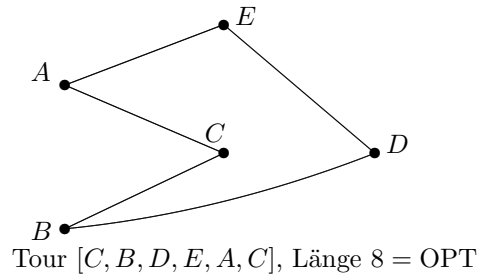
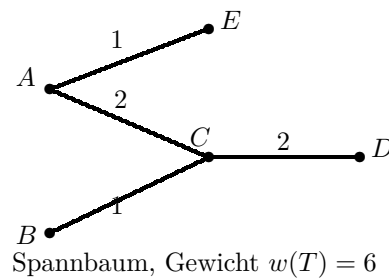


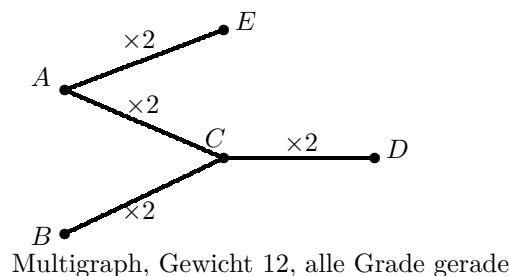
Bild 2 – Schritt 1: minimaler Spannbaum. Der Algorithmus verbindet alle Städte so billig wie möglich zu einem Baum (keine Kreise). Das sind die Kanten $A-E$, $B-C$, $A-C$, $C-D$ mit Gesamtgewicht 6.



★ Intuition

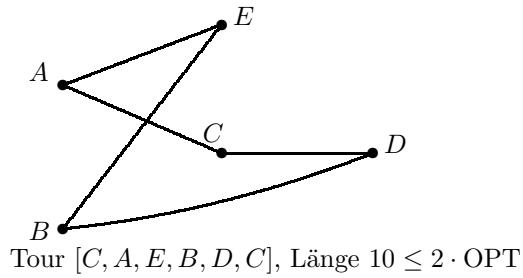
Warum ein Spannbaum? Weil er *garantiert billiger* ist als die optimale Tour: Nimm die optimale Tour und lösche eine ihrer Kanten – übrig bleibt ein Weg durch alle Städte, also ein (spezieller) Spannbaum. Der *minimale* Spannbaum ist höchstens so teuer wie dieser. Also $w(T) \leq \text{OPT}$. Das ist der Kern der Güte-Garantie.

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



Ein Eulerkreis in diesem Multigraphen ist z. B. $[C, A, E, A, C, B, C, D, C]$ – er benutzt jede der verdoppelten Kanten einmal und hat Länge 12. Beachte: Manche Städte (hier A, C) werden dabei *mehrfach* besucht.

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



★ Intuition

Warum ist Abkürzen nie schädlich? Weil eine Abkürzung zwei Kanten durch eine direkte Kante ersetzt, und die *Dreiecksungleichung* garantiert, dass der direkte Weg nicht länger ist. Deshalb bleibt die Tour $\leq 12 = 2w(T) \leq 2\text{OPT}$. Hier: Länge 10, garantiert ≤ 16 , tatsächlich nah an $\text{OPT} = 8$.

9.4 Christofides: schlauer verdoppeln

ΔTSP_1 verdoppelt *alle* Kanten – das kostet zusätzlich etwa $w(T) \approx \text{OPT}$, daher die Güte 2. Christofides bemerkt: Wir müssen gar nicht alle Grade reparieren, sondern nur die *ungeraden*. Und die repariert man viel billiger.

🔧 Idee: Nur die schiefen Knoten reparieren

Ein Eulerkreis braucht nur, dass *jeder* Knoten geraden Grad hat. Im Spannbaum haben aber ohnehin nur *manche* Knoten ungeraden Grad. Statt alles zu verdoppeln, verbindet Christofides genau diese ungeraden Knoten durch ein *minimales perfektes Matching* – die billigste Art, sie paarweise zu verkuppeln. Das kostet höchstens $\text{OPT}/2$, und daraus folgt Güte 1,5.

Bild 1 – der Spannbaum (wie zuvor). Wir starten mit demselben minimalen Spannbaum wie bei ΔTSP_1 .

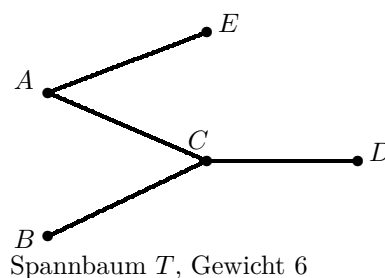


Bild 2 – die ungeraden Knoten finden. Wir zählen die Grade im Baum: A hat Grad 2 (gerade), alle anderen ungerade. Die ungeraden Knoten $X = \{B, C, D, E\}$ sind mit einem Ring markiert.

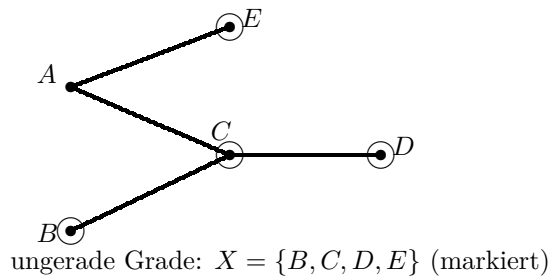


Bild 3 – das Matching hinzufügen. Unter den möglichen Paarungen der vier Knoten ist $\{B-C, D-E\}$ die billigste (Kosten $1 + 2 = 3$). Diese zwei Matching-Kanten (dünn) kommen zum Baum dazu. Jetzt haben *alle* Knoten geraden Grad.

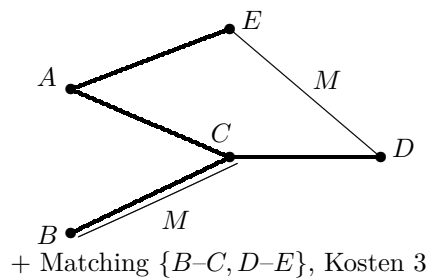
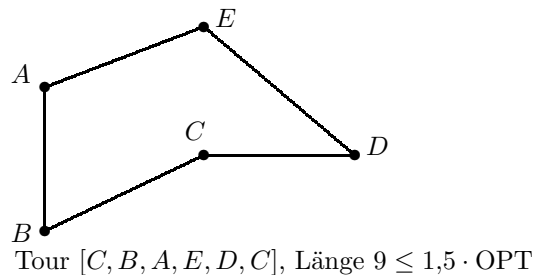


Bild 4 – Eulerkreis und abkürzen. Aus $T + M$ gewinnt man einen Eulerkreis und kürzt ihn ab. Ergebnis: die Tour $[C, B, A, E, D, C]$ mit Länge **9** – besser als die 10 von ΔTSP_1 .



✗ Stolperstein

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

9.5 Knapsack: warum pure Gier scheitert

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

📖 Idee: Greedy – und sein Versagen

Der naheliegende Weg (*Greedy*): sortiere nach „Profit pro Gewicht“ und pack der Reihe nach ein, was noch passt. Das ist oft gut, hat aber *keine* Garantie.

● Beispiel 9.4 Greedy kann beliebig schlecht sein

Zwei Gegenstände: einer mit Gewicht 1 und Profit 1, ein anderer mit Gewicht B und Profit $B - 1$. Die Kapazität ist B . Greedy schaut auf die Dichte: der kleine hat Dichte 1, der große Dichte $(B - 1)/B < 1$. Also nimmt Greedy den kleinen – Profit 1. Optimal wäre der große – Profit $B - 1$. Das Verhältnis $B - 1$ wächst mit B ins Unendliche. Keine feste Güte.

🔧 Idee: Die Reparatur: Modified Greedy (MGA), Güte 2

Rechne zwei Kandidaten aus – das Greedy-Ergebnis *und* „nimm nur den einen profitabelsten Gegenstand“ – und gib den besseren aus. Warum das reicht: Greedy verliert seinen Profit immer nur an dem *einen* „Split-Item“, das gerade nicht mehr passte. Dessen Profit fängt der Einzel-Gegenstand ab. Damit ist man nie schlechter als halb so gut wie das Optimum. Noch näher kommen der *Sahni*-Algorithmus (probiert alle kleinen Vorauswahlen, ein PTAS) und das *FPTAS* (rundet die Profite und löst exakt per dynamischer Programmierung, Güte $1 + \varepsilon$).

9.6 Scheduling: einfache Regeln, sichtbar im Gantt-Bild

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

■ Definition 9.5 List Scheduling und LPT

List Scheduling: Gehe die Jobs der Reihe nach durch und lege jeden auf die *momentan am wenigsten belastete* Maschine. *LPT* (Longest Processing Time): Sortiere die Jobs zuerst *absteigend*, wende dann List Scheduling an.

Das folgende Gantt-Bild macht sichtbar, warum die Reihenfolge zählt. Drei Jobs der Größen 1, 1, 2 auf zwei Maschinen. Die Balkenlänge ist die Zeit.

● Beispiel 9.6 List Scheduling gegen LPT

List Scheduling (Reihenfolge 1, 1, 2):

M_1

1	2
---	---

 Makespan 3

M_2

1

LPT (sortiert 2, 1, 1):

M_1

2

 Makespan 2

0 1 2 3 → Zeit

Links stapelt List Scheduling die große 2 auf eine schon belegte Maschine – Makespan 3. Rechts legt LPT die große 2 zuerst, die zwei Einsen wandern auf M_2 – Makespan 2, das Optimum.

▲ Satz 9.7 Die Güten – und die Idee dahinter

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

Die Idee (List Scheduling): Betrachte die Maschine mit der höchsten Last und den *letzten* Job J_k , der auf ihr landete. In dem Moment, als J_k zugewiesen wurde, war diese Maschine die *leerste* – also waren *alle* Maschinen mindestens so voll wie $L - p_k$. Mit den zwei Universalschranken $\text{OPT} \geq \frac{1}{m} \sum p_i$ (die Gesamtlast verteilt sich auf m Maschinen) und $\text{OPT} \geq p_{\max}$ (der größte Job muss irgendwo laufen) folgt die Schranke.

9.7 MAX-3-SAT: Güte 2 fast geschenkt

☞ Idee: Zwei Belegungen genügen

Beim Optimierungsproblem MAX-3-SAT will man *möglichst viele* Klauseln erfüllen. Erstaunlich einfacher Algorithmus mit Güte 2: Werte die Formel nur *zweimal* aus – einmal mit „alle Variablen falsch“ (β_0), einmal mit „alle wahr“ (β_1) – und nimm die bessere.

Warum das reicht: Jede Klausel enthält mindestens ein Literal, entweder ein positives oder ein negatives. Ein positives Literal wird von β_1 erfüllt, ein negatives von β_0 . Also erfüllt *jede* Klausel mindestens eine der beiden Belegungen. Damit erfüllen β_0 und β_1 zusammen $\geq m$ Klauseln, die bessere von beiden also $\geq m/2$. Da $\text{OPT} \leq m$, ist das mindestens $\text{OPT}/2$.

? Abruf – erst selbst beantworten

A18. Nenne die vier Schritte von ΔTSP_1 und sage bei jedem, warum er die Güte-2-Schranke stützt.

A19. Warum ist Christofides besser als ΔTSP_1 , obwohl beide mit demselben Spannbaum starten?

A20. Warum genügt beim Greedy-Rucksack der zusätzliche Vergleich mit dem „besten Einzel-Gegenstand“, um von „unbeschränkt schlecht“ auf Güte 2 zu kommen?

10 Jenseits von NP: das Halteproblem

★ In diesem Kapitel

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

■ Definition 10.1 Halteproblem

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

Das Halteproblem ist *unentscheidbar* – kein Algorithmus löst es, mit keiner noch so großen Laufzeit. Das ist ein tiefes, bewiesenes Resultat (Turing). Und trotzdem ist es NP-schwer.

☆ Aha

Man reduziert SAT darauf. Zu einer Formel φ baut man ein Programm M_φ , das *alle* Belegungen der Reihe nach durchprobiert und genau dann anhält, wenn es eine erfüllende findet. Dann gilt: φ erfüllbar $\iff M_\varphi$ hält. Der Clou liegt im Zeitverhalten: M_φ läuft im Nein-Fall *ewig* – aber die *Konstruktion* von M_φ aus φ ist billig (polynomiell), und nur darauf kommt es bei einer Reduktion an. Also ist das Halteproblem mindestens so schwer wie SAT. In NP liegt es aber nicht – es ist ja nicht einmal entscheidbar. **NP-schwer ist nicht dasselbe wie NP-vollständig.**

11 Das große Ganze

★ In diesem Kapitel

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

11.1 Die Reise auf einen Blick

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

11.2 Wahr oder falsch? (Die Begründung zählt)

Wenn du das Fundament verstanden hast, kannst du jede dieser Aussagen begründen.

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

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

☆ Aha

Du hast jetzt das *Warum* beisammen: was ein Problem schwer macht, wie Schwere von einem Problem zum anderen fließt, wo die Grenzen liegen, und was man tut, wenn exakte Lösungen zu teuer sind. Das ist das Fundament des ganzen Kurses.

Der nächste Schritt. Dieses Heft hat dir das Verständnis gegeben (Schritt 1 und 2 deines Lernmodells). Jetzt kommt die Anwendung (Schritt 3 und 4): Nimm die Präsenz-, Haus- und Klausuraufgaben und führe dort selbst aus, was hier als Idee stand – Reduktionen *vollständig* beweisen (beide Richtungen!), Güte-Schranken herleiten, Worst-Case-Instanzen bauen. Viel Erfolg.

A Lösungen zu den Abruf-Fragen

Erst selbst beantwortet, dann hier vergleichen – nur so wirkt das aktive Abrufen.

A1. Ein konstanter Faktor wie 1000 ($\approx 2^{10}$) verschiebt die exponentielle Kurve nur um eine *additive* Konstante: Du schaffst rund 20 Knoten mehr – ein einziges Mal. Danach wächst das Problem exponentiell weiter, die Hardware bleibt gleich. Exponentielles Wachstum lässt sich nicht durch schnellere Hardware einholen.

A2. „Kennen keinen“ = niemand hat bisher einen Algorithmus gefunden (eine Wissenslücke). „Gibt keinen“ = die Nichtexistenz ist bewiesen. Für die Clique gilt nur das Erste; ein Nichtexistenzbeweis fehlt (die Frage P vs. NP ist offen).

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

A4. Bis K zu zählen kostet K Schritte. Aber K ist als Eingabe nur $O(\log K)$ Zeichen lang, sein Wert ist exponentiell in dieser Länge. Die Laufzeit ist also exponentiell in der Eingabegröße.

A5. Zertifikat: eine Knotenmenge C . **Verifizierer:** prüfe, ob jede Kante mindestens einen Endpunkt in C hat ($O(|E|)$), und ob $|C| \leq k$. **Korrektheit:** Gibt es ein Vertex Cover der Größe $\leq k$, wird es akzeptiert; sonst keines. **Laufzeit** polynomiell. (Rate-Sicht: C raten, dann prüfen.)

A6. Der Verifizierer bekommt das Zertifikat *geschenkt* und muss es nur prüfen. Der Löser hat niemanden, der ihm die Lösung vorlegt – er muss sie selbst finden, und dafür können exponentiell viele Kandidaten in Frage kommen. Prüfen beginnt einen Schritt weiter als Lösen.

A7. $\text{CLIQUE} \leq_p \text{VERTEX COVER}$ – vom bekannt schweren CLIQUE auf das neue VC. So erbt VC die Schwere. Die Gegenrichtung würde über VC nichts aussagen (sie zeigte nur „höchstens so schwer wie das schwere Clique“).

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

A9. (1) eine Reduktion von einem bekannten NP-vollständigen Problem *auf* das neue; (2) der Nachweis, dass das neue Problem in NP liegt. Vergessen wird am häufigsten (2) – ohne ihn hat man nur NP-Schwere, nicht Vollständigkeit.

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

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

A12. Für eine Knotenmenge C gilt: C ist Clique in $G \iff C$ ist Independent Set im Komplementgraphen $\bar{G} \iff V \setminus C$ ist Vertex Cover in $\bar{G}$. Die drei Probleme sind dasselbe, betrachtet über den Komplementgraphen und die Komplement-Knotenmenge.

A13. Bei Basis $n + 1$ und höchstens n Summanden entsteht an keiner Stelle ein Übertrag – jede Ziffer zählt für sich. Dann bedeutet „Summe trifft das Ziel“ genau „jede Stelle wird genau einmal überdeckt“. Bei Basis 2 gäbe es Überträge, und eine falsche Auswahl könnte das Ziel zufällig treffen.

A14. Die beiden Zusatzzahlen sind $N - K$ und $K + 1$; ihre Summe ist $N + 1$. Eine Hälfte hat aber nur den Wert N . Zwei Zahlen mit Summe $N + 1 > N$ passen also nie zusammen in dieselbe Hälfte – sie werden immer getrennt.

A15. Beispiel: G mit $V = \{1, 2, 3, 4\}$ und Kanten $\{1, 2\}, \{2, 3\}, \{1, 3\}$ (ein Dreieck plus isolierter Knoten 4), $n = 4, k = 3$. Clique $C = \{1, 2, 3\}$. Im Komplement $\bar{G}$ (Kanten $\{1, 4\}, \{2, 4\}, \{3, 4\}$) ist $V \setminus C = \{4\}$ ein Vertex Cover der Größe $n - k = 1$ – Knoten 4 berührt alle Kanten von $\bar{G}$.

A16. 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.

A17. Die Reduktion $3\text{-SAT} \leq_p 3\text{-COLOR}$ baut aus n Variablen und m Klauseln einen Graphen mit $O(n+m)$ Knoten. Ein $2^{o(N)}$ -Algorithmus für 3-COLOR ($N = \text{Knotenzahl}$) ergäbe damit einen $2^{o(n+m)}$ -Algorithmus für 3-SAT, und wegen $n + m = O(m)$ einen $2^{o(m)}$ -Algorithmus. Genau hier kommt das *Sparsification-Lemma* ins Spiel: Es verbietet $2^{o(m)}$ für 3-SAT – der Widerspruch ist komplett.

A18. (1) Minimaler Spannbaum T : er ist billiger als die optimale Tour, $w(T) \leq \text{OPT}$. (2) Kanten verdoppeln: Gesamtlänge $2w(T) \leq 2\text{OPT}$. (3) Eulerkreis: nutzt die geraden Grade, ändert die Länge nicht. (4) Abkürzen: verlängert wegen der Dreiecksungleichung nicht. Zusammen: $d(R) \leq 2\text{OPT}$.

A19. ΔTSP_1 verdoppelt *alle* Baumkanten und zahlt dafür zusätzlich etwa $w(T) \approx \text{OPT}$ – Güte 2. Christofides fügt stattdessen nur ein minimales perfektes Matching auf den ungerad-gradigen Knoten hinzu, das höchstens $\text{OPT}/2$ kostet. Deshalb landet er bei $w(T) + \text{OPT}/2 \leq 1,5 \text{OPT}$.

A20. Greedy verliert seinen Profit immer nur an dem einen „Split-Item“, das gerade nicht mehr in den Rucksack passte. Dessen Profit ist höchstens der maximale Einzelprofit. Der zweite Kandidat (bester Einzel-Gegenstand) deckt genau diesen Fall ab. Also ist das Bessere von beiden mindestens halb so gut wie das Optimum – Güte 2.