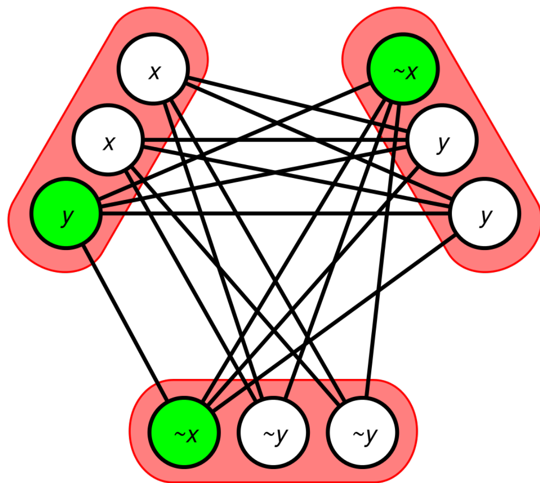
Kapitel 2

Komplexität



Inhalt

2 Komplexität

- Probleme, Komplexitätsmaße, Laufzeiten
- Komplexität der Linearen Programmierung
- Die Klassen $\mathcal{P}$ und $\mathcal{NP}$
- $\mathcal{NP}$-Vollständigkeit
- Komplexität und kombinatorische Optimierung

Probleme

Wir benötigen eine einigermaßen präzise Definition des Begriffs Problem.

Definition 2.1

- Ein Problem ist eine allgemeine Fragestellung, bei der mehrere Parameter offen gelassen sind und für die eine Lösung oder Antwort gesucht wird.
- Ein Problem ist dadurch definiert, dass alle seine Parameter beschrieben werden und dass genau angegeben wird, welche Eigenschaften eine Antwort (Lösung) haben soll.
- Sind alle Parameter eines Problems mit expliziten Daten belegt, dann sprechen wir von einer Probleminstanz.

Beispielproblem

Beispiel 2.2

- Das Travelling-Salesman-Problem ist ein Problem im Sinne von Definition 2.1.
- Seine offenen Parameter sind die Anzahl der Städte und die Entfernungen zwischen diesen Städten.
- Eine Entfernungstabelle definiert eine Probleminstanz für das Travelling-Salesman-Problem.
- Eine kürzeste Rundreise ist eine Lösung.

Bemerkung:

- Der Begriff Problem bezeichnet demnach eine abstrakte Problemstellung.
- Eine konkrete Problemstellung ist eine Probleminstanz.

Lösungsalgorithmus

- Mathematisch betrachten wir ein Problem Π als die Menge aller Probleminstanzen.
- Im mathematischen Sinn ist also das Travelling-Salesman-Problem die Menge aller TSP-Instanzen.

Definition 2.3

Ein Algorithmus löst ein Problem Π, wenn er für jede Probleminstanz $\mathcal{I} \in \Pi$ eine Lösung findet.

Bemerkung: Hier bleibt noch die Frage offen, wie solch eine Lösung aussehen kann.

Kodierungsschemata

- Ziel ist es, möglichst effiziente Verfahren zur Lösung von Problemen zu finden.
- Mittels Komplexitätsmaßen machen wir den Begriff der Effizienz messbar.
- am wichtigsten: Zeit- und Speicherplatzkomplexität
- Die Laufzeit eines Algorithmus hängt in der Regel von der Größe einer Probleminstanz ab, d. h. vom Umfang der Eingabedaten.
- Wir müssen also beschreiben, wie wir Probleminstanzen repräsentieren.
- Kodierungsschemata leisten diese Aufgabe.

Kodierung von Zahlen, Vektoren und Matrizen

- Ganze Zahlen kodieren wir binär.
- Die binäre Darstellung einer nichtnegativen ganzen Zahl n benötigt $\lceil \log_2(n+1) \rceil$ Bits. Hinzu kommt ein Bit für das Vorzeichen.
- Die Kodierungslänge $\langle n \rangle$ einer ganzen Zahl n ist dann

$$\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1.$$

- Jede rationale Zahl r hat eine Darstellung $r = \frac{p}{q}$ mit $p, q \in \mathbb{Z}$, p, q sind teilerfremd und $q > 0$.
- Die Kodierungslänge von $r = \frac{p}{q}$ ist dann

$$\langle r \rangle = \langle p \rangle + \langle q \rangle.$$

- Die Kodierungslänge eines Vektors $\mathbf{x} \in \mathbb{Q}^n$ ist

$$\langle \mathbf{x} \rangle = \sum_{j=1}^{n} \langle x_j \rangle.$$

- Die Kodierungslänge einer Matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ ist

$$\langle \mathbf{A} \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle.$$

- Kodierungslängen für andere Datenstrukturen (insbesondere Graphen) leiten wir aus den so gegebenen Kodierungslängen her.

Rechnermodell

- Wir benötigen ein Rechnermodell, um Speicher- und Laufzeitberechnungen durchführen zu können.
- Die Komplexitätstheorie nutzt hierzu Turing- oder RAM-Maschinen. Wir nutzen ein einfacheres Modell.
- Ein Algorithmus A soll Probleminstanz $\mathcal{I}$ des Problems Π lösen. Probleminstanzen liegen immer in kodierter Form vor. $\langle \mathcal{I} \rangle$ bezeichnet die Kodierungslänge von $\mathcal{I}$.
- Der Algorithmus liest diese Daten und beginnt Operationen auszuführen, d. h. Zahlen zu berechnen, zu speichern, zu löschen, usw.
- Die Anzahl der Speicherzellen, die während der Ausführung des Algorithmus A mindestens einmal benutzt werden, nennen wir den Speicherbedarf von A zur Lösung von $\mathcal{I}$.

- Die Laufzeit von A zur Lösung von $\mathcal{I}$ basiert auf der Anzahl der elementaren Operationen, die A bis zur Terminierung ausführt.
- Elementare Operationen sind

  Lesen, Schreiben, Initialisieren, Addieren, Subtrahieren, Multiplizieren, Dividieren und Vergleichen

  von rationalen Zahlen.
- Für die Laufzeitberechnung muss nun jede elementare Operation mit der Kodierungslänge der beteiligten Zahlen multipliziert werden.
- Die Laufzeit von A zur Lösung von $\mathcal{I}$ ist die Anzahl der ausgeführten elementaren Rechenoperationen multipliziert mit der längsten aufgetretenen Kodierungslänge einer Zahl.
- Wir tun also so, als hätten wir alle elementaren Operationen mit der am längsten kodierten Zahl ausgeführt.

Laufzeit- und Speicherplatzbedarf

Definition 2.4

Sei A ein Algorithmus zur Lösung eines Problems Π.

1. Die Funktion $f_A : \mathbb{N} \to \mathbb{N}$, definiert durch

$$f_A(n) = \max \{\text{Laufzeit von } A \text{ zur Lösung von } \mathcal{I} | \mathcal{I} \in \Pi \text{ und } \langle \mathcal{I} \rangle \leq n\},$$

   heißt Laufzeitfunktion von A.

2. Die Funktion $s_A : \mathbb{N} \to \mathbb{N}$, definiert durch

$$s_A(n) = \max\{\text{Speicherbedarf von } A \text{ zur Lösung von } \mathcal{I} | \mathcal{I} \in \Pi \text{ und } \langle \mathcal{I} \rangle \leq n\},$$

   heißt Speicherplatzfunktion von A.

Fortsetzung Definition.

3. Der Algorithmus A hat polynomielle Laufzeit (kurz: ist polynomiell), wenn es ein Polynom $p : \mathbb{N} \rightarrow \mathbb{N}$ gibt mit
$$f_A(n) \leq p(n) \quad \text{für alle } n \in N.$$
Wir schreiben $f_A = O(n^k)$, falls das Polynom p den Grad k hat.
4. Der Algorithmus A hat polynomiellen Speicherplatzbedarf, falls es ein Polynom $q : \mathbb{N} \rightarrow \mathbb{N}$ gibt mit
$$s_A(n) \leq q(n) \quad \text{für alle } n \in N.$$

Bemerkung: Ein Algorithmus, dessen Speicherplatzfunktion nicht durch ein Polynom beschränkt ist, kann nicht polynomiell sein.

Komplexität des Simplexalgorithmus

- Wir betrachten die lineare Programmierung (LP) als Problem.
- Eine Lösung im Sinne von Definition 2.1 ist eine optimale Lösung im Sinne der linearen Programmierung.
- Wie viel Rechenzeit benötigen wir, um mit dem Simplexalgorithmus eine optimale Lösung zu berechnen?
- Wir teilen die Frage auf:
  - Wie aufwendig ist eine Iteration?
  - Wie viele Iterationen sind notwendig?

Aufwand für eine Iteration (1)

- kanonische Normalform, m Nebenbedingungen, n Variablen
- Pivotspalte bestimmen: n Vergleiche
- Pivotzeile bestimmen: m Divisionen und Vergleiche
- Pivotzeile normalisieren: $n+1$ Divisionen
- Tableau anpassen: $m \cdot (n+1)$ Divisionen und Subtraktionen

Fazit: $O(mn)$ Operationen für eine Simplexiteration

☞ Dies ist noch kein Beweis für die polynomielle Laufzeit einer Simplexiteration (in der Theorie).

Warum? Kodierungslänge der Werte im Tableau!

Aufwand für eine Iteration (2)

- Kodierungslänge für ein LP: $\langle \mathbf{A} \rangle + \langle \mathbf{b} \rangle + \langle \mathbf{c} \rangle$
- Man kann zeigen, dass die Einträge im Simplextableau im Betrag durch

$$2^{\langle \mathbf{A} \rangle + \langle \mathbf{b} \rangle + \langle \mathbf{c} \rangle - 2n}$$

  beschränkt bleiben.
- Ihre Kodierungslänge bleibt also polynomiell in der Kodierungslänge des LP.
- Einen Beweis hierfür findet man bei Borgwardt (siehe Literaturhinweise).

Fakt 2.5

Eine Iteration des Simplexalgorithmus hat polynomielle Laufzeit.

Klee–Minty-Polytop

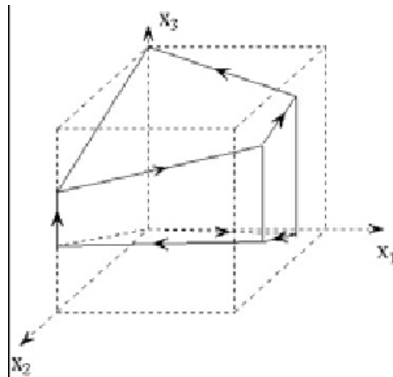
Definition 2.6

Das *d*-dimensionale Klee–Minty-Polytop ist durch folgendes Ungleichungssystem definiert:

$$\begin{array}{rcrcrcrcrcl}
x_1 & & & & & & & & & \leq & 5 \\
4x_1 & + & x_2 & & & & & & & \leq & 25 \\
8x_1 & + & 4x_2 & + & x_3 & & & & & \leq & 125 \\
\vdots & & \vdots & + & \vdots & & & & & \leq & \vdots \\
2^d x_1 & + & 2^{d-1} x_2 & + & \cdots & & 4x_{d-1} & + & x_d & \leq & 5^d \\
 & & & & & & & & x_1, x_2, \ldots, x_d & \geq & 0
\end{array}$$

Bemerkungen zum Klee–Minty-Polytop

- Das Klee–Minty-Polytop ist ein verzerrter Würfel.
- Das Klee–Minty-Polytop hat genauso viele d-dimensionale Seiten und Ecken wie der d-dimensionale Würfel,
- also $2d$ viele d-dimensionale Seiten und 2^d Ecken.



Nichtpolynomialität des Simplexalgorithmus (1)

Fakt 2.7

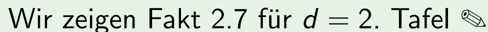
Für die Zielfunktion

$$\max 2^{d-1}x_1 + 2^{d-2}x_2 + \cdots + 2x_{d-1} + x_d$$

durchläuft der Simplexalgorithmus mit der Pivotregel von Dantzig (negatives Element der Zielfunktionszeile mit größtem Betrag) alle 2^d Ecken des d-dimensionalen Klee–Minty-Polytops.

Beispiel 2.8

Wir zeigen Fakt 2.7 für $d = 2$. Tafel ✎



Nichtpolynomialität des Simplexalgorithmus (2)

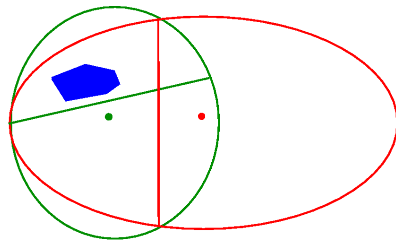
- Folgerung: Der Simplexalgorithmus (mit der Pivotregel von Dantzig) hat im Worst-Case keine polynomielle Laufzeit.
- Auch für alle anderen bekannten Pivot-Regeln gibt es analoge Resultate.
- Noch ungeklärt ist, ob es wirklich keine polynomielle Simplexvariante gibt.

Average-Case:

- Probabilistische Analysen liefern, dass für den Erwartungswert der Anzahl an Pivotschritten $O(m^{\frac{1}{n-1}} n^3)$ gilt (siehe Borgwardt).
- Andere Analysen kommen auf noch bessere Schranken.
- Damit hat der Simplexalgorithmus im Mittel polynomielle Laufzeit.

Ellipsoidalgorithmus

- Leonid Khachiyan (1979)
- polynomieller Algorithmus der entscheidet, ob ein Polyeder $P = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ leer ist oder nicht
- Idee: man konstruiert immer kleiner werdende Ellipsoide, die das Polyeder umfassen
- Nach einer maximalen Iterationszahl muss das Zentrum des Ellipsoids im Polyeder liegen (wenn nicht leer).
- Die maximale Iterationszahl ist polynomiell in der Kodierungslänge von $\mathbf{A}$ und $\mathbf{b}$.



Äquivalenz zwischen Optimalität und Zulässigkeit

- Die Frage nach einer optimalen Lösung für ein LP ist genauso „schwierig" wie die Frage nach einer Lösung für ein Ungleichungssystem.
- Das LP (P) $\max \mathbf{c}^T\mathbf{x}$ u.d.N. $\mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ hat genau dann eine optimale Lösung, wenn das folgende Ungleichungssystem (D) eine Lösung hat.

$$
\begin{aligned}
&\mathbf{A}\mathbf{x} \leq \mathbf{b} \\
&\mathbf{A}^T\mathbf{u} \geq \mathbf{c} \\
&\mathbf{c}^T\mathbf{x} \geq \mathbf{b}^T\mathbf{u} \\
&\mathbf{x} \geq \mathbf{0}, \mathbf{u} \geq \mathbf{0}
\end{aligned}
$$

- Für jede Lösung $(\tilde{\mathbf{x}}, \tilde{\mathbf{u}})$ von (D) ist $\tilde{\mathbf{x}}$ eine optimale Lösung von (P).
- Begründung: Dualitätssätze

Polynomialität der linearen Programmierung

Folgerung 2.9

Es existiert ein polynomieller Algorithmus für die lineare Programmierung.

- Der Ellipsoidalgorithmus ist von großer theoretischer Bedeutung, für die Praxis aber zu ineffizient.
- Weitere polynomielle Alternativen sind Innere-Punkte-Methoden.
- Ausgehend von einem Punkt im Innern folgt man dem Gradienten der Zielfunktion.
- Probleme:
  - Schrittweite?
  - Am Rand führt die Richtung des Gradienten evtl. in die Unzulässigkeit.
- Hier keine weitere Betrachtung solcher Algorithmen.

Entscheidungsprobleme

Definition 2.10

Ein Entscheidungsproblem ist ein Problem, das nur zwei mögliche Lösungen (Ausgaben) besitzt, nämlich „ja“ oder „nein“.

Beispiel 2.11

Die Fragen

- Enthält ein Graph $G = (V, E)$ einen Kreis?
- Enthält ein Graph $G = (V, E)$ einen hamiltonschen Kreis?
- Ist ein Graph $G = (V, E)$ bipartit?
- Ist die Zahl $n \in \mathbb{N}$ eine Primzahl?

sind Entscheidungsprobleme.

Die Klasse $\mathcal{P}$

Definition 2.12

Die Klasse aller Entscheidungsprobleme, für die ein polynomieller Lösungsalgorithmus existiert, wird mit $\mathcal{P}$ bezeichnet.

Bemerkungen:

- Wir müssten $\mathcal{P}$ eigentlich abhängig von einem Kodierungsschema und einem Rechnermodell definieren.
- Wir beziehen die Definition daher auf das hier definierte Kodierungsschema und Rechenmodell.
- Das Entscheidungsproblem "Enthält ein Graph $G = (V, E)$ einen Kreis?" gehört zur Klasse $\mathcal{P}$.

Verifikation

- Wir wollen herausfinden, ob ein Graph $G = (V, E)$ einen hamiltonschen Kreis enthält.
- Angenommen, G hat einen hamiltonschen Kreis, und ein Orakel nennt uns eine Knotenfolge $K = (v_1, v_2, \ldots, v_n, v_1)$ und behauptet, dies sei ein hamiltonscher Kreis.
- Dann können wir jetzt prüfen, ob K tatsächlich ein Hamiltonkreis von G ist.
- Dazu müssen wir testen, ob
  - jeder Knoten v in V genau einmal in K auftritt und
  - $\{v_i, v_{i+1}\} \in E$ für $i = 1, \ldots, n-1$ und $\{v_n, v_1\} \in E$ gilt.
- Dies ist offensichtlich in polynomieller Zeit möglich.
- Somit können wir die Korrektheit der „ja“-Antwort polynomiell verifizieren.

Die Klasse $\mathcal{NP}$

Definition 2.13

Ein Entscheidungsproblem Π gehört zur Klasse $\mathcal{NP}$, wenn es die folgenden Eigenschaften hat:

1. Für jede Probleminstanz $\mathcal{I} \in \Pi$, für die die Lösung „ja“ lautet, gibt es mindestens ein Objekt Q, mit dessen Hilfe die Korrektheit der „ja“-Antwort überprüft werden kann.
2. Es gibt einen Algorithmus, der
   - Probleminstanzen $\mathcal{I} \in \Pi$ und Zusatzobjekte Q liest und
   - der in einer Laufzeit, die polynomiell in $\langle \mathcal{I} \rangle$ ist, überprüft, ob Q ein Objekt ist, aufgrund dessen Existenz eine „ja“-Antwort gegeben werden muss.

Beispiele für Probleme in $\mathcal{NP}$

Beispiel 2.14

1. Hat $G = (V, E)$ einen Kreis?
2. Hat $G = (V, E)$ einen hamiltonschen Kreis?
3. Ist $n \in \mathbb{N}$ keine Primzahl?
   Hier liefert das Orakel ein Produkt zweier Zahlen $\neq 1$.
4. Gibt es für $G = (V, E)$ eine Färbung mit k Farben?

Bemerkungen zur Klasse $\mathcal{NP}$

- Definition 2.13 sagt nichts darüber aus, wie das Zusatzobjekt Q zu finden ist. Es wird lediglich postuliert, dass es existiert.
- Die Laufzeit des Algorithmus in Definition 2.13 (2) ist polynomiell in $\mathcal{I}$. Da der Algorithmus Q lesen muss, folgt, dass $\langle Q \rangle$ polynomiell in $\mathcal{I}$ sein muss.
- Die Definition der Klasse $\mathcal{NP}$ ist unsymmetrisch: Die Definition impliziert nicht, dass wir auch für Probleminstanzen mit „nein"-Antworten Objekte Q und polynomielle Algorithmen mit den genannten Eigenschaften finden können.

Nichtdeterministischer polynomieller Algorithmus

- $\mathcal{NP}$ ist abgeleitet von „nichtdeterministischer polynomieller Algorithmus“.
- Nichtdeterministische Algorithmen können eine Lösung raten (Orakel).
- Ablauf:
  1. Ein Lösungsvorschlag Q wird geraten.
  2. Gibt es keine Lösung: STOP!
  3. Der Lösungsvorschlag Q wird überprüft.
  4. Ist Q eine Lösung, dann antwortet der Algorithmus mit „ja“.
- Es gilt $\mathcal{P} \subseteq \mathcal{NP}$, denn für Probleme in $\mathcal{P}$ existieren Algorithmen, die ohne Lösungsvorschlag Q in polynomieller Zeit eine Antwort liefern.

Die Klasse co-$\mathcal{NP}$

Definition 2.15

Die Klasse co-$\mathcal{NP}$ besteht aus den Entscheidungsproblemen, die Negationen von Problemen $\Pi \in \mathcal{NP}$ sind.

- Zu co-$\mathcal{NP}$ gehören z. B.:
  1. Hat $G = (V, E)$ keinen Kreis?
  2. Hat $G = (V, E)$ keinen hamiltonschen Kreis?
  3. Ist $n \in \mathbb{N}$ eine Primzahl?
- 1. und 3. sind auch $\in \mathcal{NP}$, sogar $\in \mathcal{P}$.
- Von 2. weiß man dies nicht.
- Es gilt $\mathcal{P} \subseteq$ co-$\mathcal{NP}$.

$\mathcal{P} = \mathcal{NP}$?

- Eigentlich sollte man meinen, dass Algorithmen, die eine Lösung raten können, mächtiger sind als übliche Algorithmen.
- Trotzdem ist die Frage „$\mathcal{P} = \mathcal{NP}$?“ immer noch ungelöst.
- Millennium-Problem, 1 Mio. US$ Preisgeld
- Vermutung: $\mathcal{P} \neq \mathcal{NP}$
- Könnte man dies bestätigen, können wir für viele praxisrelevante Probleme niemals effiziente Algorithmen finden (für große Probleminstanzen).

Weitere offene Fragen

- Wir wissen: $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co-}\mathcal{NP}$
- Gilt $\mathcal{P} = \mathcal{NP} \cap \text{co-}\mathcal{NP}$?
- Gilt $\mathcal{NP} = \text{co-}\mathcal{NP}$?

Polynomielle Transformation

- Wir wollen nun innnerhalb der Klasse $\mathcal{NP}$ eine Klasse von besonders schwierigen Problemen auszeichnen.

Definition 2.16

Gegeben seien die Entscheidungsprobleme Π und Π'.

Eine polynomielle Transformation von Π in Π' ist ein polynomieller Algorithmus, der aus einer Probleminstanz $\mathcal{I} \in \Pi$ eine Probleminstanz $\mathcal{I}' \in \Pi'$ erzeugt, so dass folgendes gilt:

Die Antwort für $\mathcal{I}$ ist genau dann „ja“, wenn die Antwort für $\mathcal{I}'$ „ja“ ist.

Wir sagen dann, dass Π polynomiell transformierbar in Π' ist und schreiben

$$\Pi \leq_p \Pi'.$$

Konsequenz

Damit gilt Folgendes:

- Wenn Π polynomiell transformierbar in Π' ist und für Π' ein polynomieller Lösungsalgorithmus existiert,
- dann können wir auch Π in polynomieller Laufzeit lösen.
- Hierzu transformieren wir einfach eine Instanz $\mathcal{I} \in \Pi$ in eine Instanz $\mathcal{I}' \in \Pi'$ und wenden den Lösungsalgorithmus für Π' an.
- Sowohl die Transformation als auch der Lösungsalgorithmus für Π' sind polynomiell, somit auch die Kombination.

$\mathcal{NP}$-vollständig

Definition 2.17

Ein Entscheidungsproblem Π heißt $\mathcal{NP}$-vollständig, wenn folgendes gilt:

1. $\Pi \in \mathcal{NP}$
2. Für alle Probleme $\Pi' \in \mathcal{NP}$ gilt:
   Π' ist polynomiell transformierbar in Π.

Ein $\mathcal{NP}$-vollständiges Problem hat demnach die folgende Eigenschaft:

- Wenn Π in polynomieller Zeit gelöst werden kann, dann kann auch jedes andere Problem in $\mathcal{NP}$ in polynomieller Zeit gelöst werden.
- Π ist $\mathcal{NP}$-vollständig und $\Pi \in \mathcal{P} \Longrightarrow \mathcal{P} = \mathcal{NP}$.

Satz von Cook

- Konsequenz: Bezüglich polynomieller Lösbarkeit ist kein Problem schwieriger als ein $\mathcal{NP}$-vollständiges.
- Aber gibt es überhaupt $\mathcal{NP}$-vollständige Probleme? Ja!

Definition 2.18

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) lautet:

- Gegeben ist eine aussagenlogische Formel Φ in konjunktiver Normalform (KNF) mit Variablen $x_1, \ldots, x_n$.
- Existiert eine Belegung der Variablen, so dass Φ wahr wird?

Satz 2.19 (Cook (1971))

SAT ist $\mathcal{NP}$-vollständig.

Beweis für die $\mathcal{NP}$-Vollständigkeit eines Problems

- Wie können wir beweisen, dass ein Entscheidungsproblem Π $\mathcal{NP}$-vollständig ist?
- Hierfür die Definition der $\mathcal{NP}$-Vollständigkeit zu nutzen, ist sehr unhandlich.
- Der Satz von Cook ist überaus hilfreich.

Folgerung 2.20

Es sei Π ein Entscheidungsproblem. Dann ist Π genau dann $\mathcal{NP}$-vollständig, wenn gilt:

1. $\Pi \in \mathcal{NP}$ *und*
2. $SAT \leq_p \Pi$.

3-SAT ist $\mathcal{NP}$-vollständig

Definition 2.21

Das Problem 3-SAT lautet:

- Existiert für eine aussagenlogische Formel Φ in KNF und genau drei Literalen pro Klausel (3KNF) eine Belegung der Variablen, so dass Φ wahr wird?

Lemma 2.22

3-SAT ist $\mathcal{NP}$-vollständig.

Beweis.

1. Wir müssen 3-SAT $\in \mathcal{NP}$ zeigen.
    - Für jede Instanz $\mathcal{I} \in$ 3-SAT gilt $\mathcal{I} \in$ SAT.
    - Ein nichtdeterministischer polynomieller Algorithmus für SAT löst somit auch 3-SAT-Instanzen.
    - Damit folgt 3-SAT $\in \mathcal{NP}$.

Fortsetzung Beweis.

2. Wir müssen SAT $\leq_p$ 3-SAT zeigen.
   - Wir zeigen nun, wie eine beliebige KNF-Formel Φ in eine Formel Ψ mit genau drei Literalen pro Klausel umgewandelt werden kann (3KNF).
   - Es sei C eine Klausel mit nur einem Literal, also $C = x_i$ oder $C = \neg x_i$.

     Dann führen wir zwei neue Variablen y_i, z_i ein und ersetzen C durch die Klauseln

     $$C_1 = x_i \vee y_i \vee z_i, C_2 = x_i \vee y_i \vee \neg z_i, C_3 = x_i \vee \neg y_i \vee z_i, C_4 = x_i \vee \neg y_i \vee \neg z_i.$$

   - Analog für $C = \neg x_i$.
   - Es gilt $C \Leftrightarrow C_1 \wedge C_2 \wedge C_3 \wedge C_4$.
   - Es sei C eine Klausel mit zwei Literalen, o.B.d.A. $C = x_i \vee x_j$.

     Dann führen wir eine neue Variable y_{ij} ein und ersetzen C durch die Klauseln

     $$C_1 = x_i \vee x_j \vee y_{ij}, \quad C_2 = x_i \vee x_j \vee \neg y_{ij}.$$

     Es gilt $C \Leftrightarrow C_1 \wedge C_2$.

Fortsetzung Beweis.

2. Fortsetzung SAT $\leq_p$ 3-SAT.
   - Es sei C eine Klausel von Φ mit vier Literalen, z. B.

$$C = x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4.$$

   - Wir führen eine neue Variable z ein und ersetzen C durch die beiden Klauseln

$$C_1 = x_1 \vee \neg x_2 \vee z \quad \text{und} \quad C_2 = \neg x_3 \vee x_4 \vee \neg z.$$

   - Es gilt: $C \Leftrightarrow C_1 \wedge C_2$.
   - Diese Technik funktioniert auch allgemein, um eine Klausel mit k Literalen durch zwei Klauseln mit 3 und $k-1$ Literalen zu ersetzen.
   - Sukzessive Anwendung liefert einen polynomiellen Transformationsalgorithmus.

Transitivität der polynomiellen Transformierbarkeit

- Statt SAT können wir nun auch 3-SAT in Folgerung 2.20 verwenden.
- Das gilt natürlich nicht nur für 3-SAT, sondern jedes $\mathcal{NP}$-vollständige Problem.

Folgerung 2.23

Es sei Π ein Entscheidungsproblem. Dann ist Π genau dann $\mathcal{NP}$-vollständig, wenn gilt:

1. $\Pi \in \mathcal{NP}$ *und*
2. *es existiert ein $\mathcal{NP}$-vollständiges Problem Π' mit $\Pi' \leq_p \Pi$.*

VC ist $\mathcal{NP}$-vollständig

Definition 2.24

Das Knotenüberdeckungsproblem (vertex cover, VC) lautet:

- Gegeben sei ein Graph $G = (V, E)$ und eine natürliche Zahl k.
- Existiert eine Teilmenge $U \subseteq V$ mit $|U| \leq k$, so dass jede Kante $e \in E$ mit mindestens einem Knoten aus U inzident ist?

Satz 2.25

VC ist $\mathcal{NP}$-vollständig.

Beweis.

1. Für eine Teilmenge $U \subseteq V$ können wir in polynomieller Zeit prüfen, ob die Knotenüberdeckungseigenschaft erfüllt ist.
2. Wir zeigen 3-SAT $\leq_p$ VC.

Fortsetzung Beweis.

- Es sei $\Phi = C_1 \wedge \ldots \wedge C_m$ eine 3KNF Formel, die aus den Klauseln C_i und den Variablen $x_1, \ldots, x_n$ besteht.
- Wir müssen einen Graphen $G = (V, E)$ und ein $k \in \mathbb{N}$ konstruieren, so dass G genau dann eine Knotenüberdeckung der Größe $\leq k$ hat, wenn Φ erfüllbar ist.
- G besteht aus drei Arten von Komponenten.
- Belegungskomponenten:
  - Für jede Variable x_i definieren wir die Komponente $T_i = (V_i, E_i)$ mit $V_i = \{x_i, \neg x_i\}$ und $E_i = \{\{x_i, \neg x_i\}\}$.
  - Jede Knotenüberdeckung enthält somit mindestens einen der beiden Knoten x_i und $\neg x_i$.

Fortsetzung Beweis.

- Testkomponenten:
  - Für jede Klausel $C_i \in \Phi$ definieren wir eine Testkomponente $S_j = (V'_j, E'_j)$, die ein Dreieck bildet:

$$\begin{aligned} V'_j &= \{a_1[j], a_2[j], a_3[j]\}, \\ E'_j &= \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\}. \end{aligned}$$

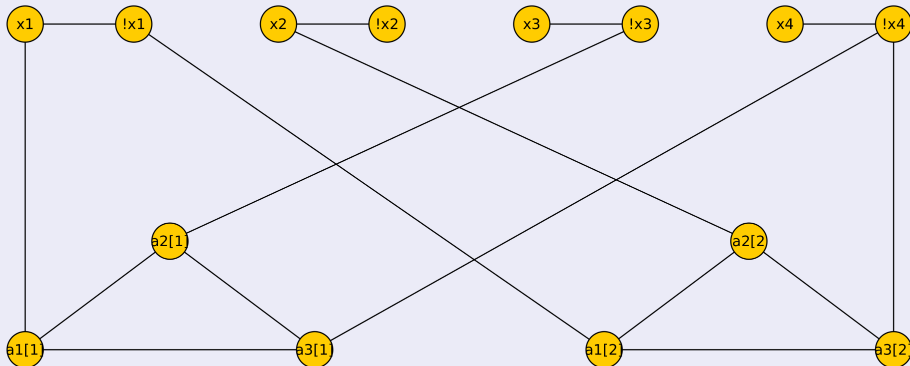
  - Jede Knotenüberdeckung enthält mindestens zwei Knoten aus S_j.
- Kommunikationskomponenten:
  - verbinden die Belegungskomponenten mit den Testkomponenten
  - Für eine Klausel $C_j = u_j \vee v_j \vee w_j$, wobei u_j, v_j, w_j Literale sind, setzen wir

$$E''_j = \{\{a_1[j], u_j\}, \{a_2[j], v_j\}, \{a_3[j], w_j\}\}.$$

  - D. h. wir verbinden das ;ue*i*-te Literal einer Klausel mit dem *i*-ten Knoten der zugehörigen Testkomponente.

Fortsetzung Beweis.

- Sei $k = n + 2m$ und $G = (V, E)$ mit
  - $V = \left(\bigcup_{i=1}^{n} V_i\right) \cup \left(\bigcup_{j=1}^{m} V_j'\right)$
  - $E = \left(\bigcup_{i=1}^{n} E_i\right) \cup \left(\bigcup_{j=1}^{m} E_j'\right) \cup \left(\bigcup_{j=1}^{m} E_j''\right)$.
- Beispiel für $\Phi = (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$:



Fortsetzung Beweis.

- „$\Rightarrow$": Sei $B : \{x_1, \ldots, x_n\} \to \{\texttt{true}, \texttt{false}\}$ eine Belegung, die Φ wahr macht.
- Die Knotenüberdeckung besteht dann aus:
  - x_i, wenn $B(x_i) = \texttt{true}$ und $\neg x_i$, wenn $B(x_i) = \texttt{false}$.
  - Damit sind dann alle Kanten E_i aus den Belegungskomponenten abgedeckt.
  - Außerdem ist jeweils mindestens eine Kante aus den Mengen E_j'' (Kommunikationskomponenten) abgedeckt, denn Φ erfüllt jede Klausel.
  - Um auch die jeweils beiden anderen Kanten der Mengen E_j'' abzudecken, wählen wir die beiden entsprechenden Knoten aus V_j' (Testkomponente).
  - Damit sind dann auch die Kanten E_j' aus den Testkomponenten abgedeckt.
- Die Größe der Knotenüberdeckung ist $n + 2m$.

Fortsetzung Beweis.

- „$\Leftarrow$“: Sei $U \subseteq V$ eine Knotenüberdeckung von G mit $|U| \leq n + 2m$.
- siehe Bemerkungen bei Belegungs- und Testkomponenten: eine Knotenüberdeckung muss mindestens $n + 2m$ Knoten umfassen.
- $\Rightarrow |U| = n + 2m$ und U enhält pro Belegungskomponente genau einen Knoten und pro Testkomponente genau zwei Knoten.
- Gilt $x_i \in U$, dann setzen wir $B(x_i) = \texttt{true}$, ansonsten $B(x_i) = \texttt{false}$.
- Von den drei Kanten in der j-ten Kommunikationskomponente können nur zwei durch $V_j' \cap U$ abgedeckt werden.
- Also wird die dritte Kante durch einen Knoten aus einer Belegungskomponente abgedeckt.
- Die entsprechende Belegung macht damit die j-te Klausel wahr.
- Dies gilt für alle j. Also wird die Formel Φ erfüllt.

Hamiltonkreisproblem

Definition 2.26

Das Hamiltonkreisproblem (HC) lautet:

- Gegeben ist ein Graph $G = (V, E)$.
- Enthält G einen hamiltonschen Kreis?

Das gerichtete Hamiltonkreisproblem (DHC) lautet:

- Gegeben ist ein gerichteter Graph $G = (V, E)$.
- Enthält G einen gerichteten hamiltonschen Kreis?

Das (gerichtete) Hamiltonwegprobem (HP bzw. DHP) lautet:

- Gegeben ist ein (gerichteter) Graph $G = (V, E)$.
- Enthält G einen (gerichteten) hamiltonschen Weg?

DHC ist $\mathcal{NP}$-vollständig

Satz 2.27

DHC ist $\mathcal{NP}$-vollständig.

Beweis

1. Für eine Knotenfolge $(v_1, v_2, \ldots, v_n, v_1)$ können wir in polynomieller Zeit prüfen, ob diese einen gerichteten Hamiltonkreis von G bildet.
2. Wir zeigen 3-SAT $\leq_p$ DHC.
   - Es sei $\Phi = C_1 \wedge \ldots \wedge C_m$ eine 3KNF-Formel, die aus den Klauseln C_i und den Variablen $x_1, \ldots, x_n$ besteht.
   - Wir konstruieren einen gerichteten Graphen $G = (V, E)$.
   - Wir definieren für jede Variable x_i einen Knoten.
   - Wir definieren für jede Klausel eine Komponente, die aus sechs Knoten besteht.
   - Also $|V| = n + 6m$.
   - Jeder Variablenknoten x_i hat genau zwei ausgehende und zwei eingehende Kanten.

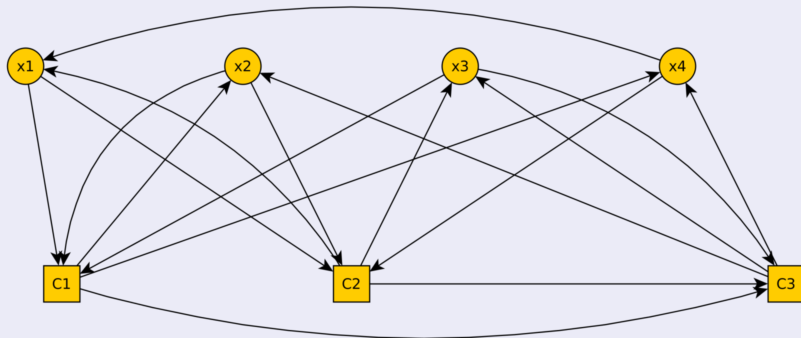
Fortsetzung Beweis.

2. Fortsetzung 3-SAT $\leq_p$ DHC.
   - Jede Klauselkomponente hat genau drei eingehende und drei ausgehende Kanten.
   - Die beiden ausgehenden Kanten aus einem Variablenknoten repräsentieren die Literale x_i und $\neg x_i$.
   - Die erste ausgehende Kante aus dem Knoten x_i ist l-te eingehende Kante der j-ten Klauselkomponente, wenn die j-te Klauselkomponente die erste ist, in der x_i auftritt, wobei x_i das l-te Literal in dieser Klausel ist.
   - Die l-te ausgehende Kante der j-ten Klauselkomponente ist dann die l'-te eingehende Kante in die j'-te Klauselkomponente, wenn die j'-te Klausel die zweite ist, in der x_i auftritt, wobei x_i das l'-te Literal in dieser Klausel ist, usw.
   - Wenn es kein weiteres Vorkommen von x_i gibt, dann ist die aus der Klauselkomponente ausgehende Kante die erste in den Variablenknoten x_{i+1} eingehende Kante, bzw. in den Variablenknoten x_1 für $i = n$.
   - Die zweite aus einem Variablenknoten ausgehende Kante hat die gleiche Funktion für das negative Literal $\neg x_i$.

Fortsetzung Beweis.

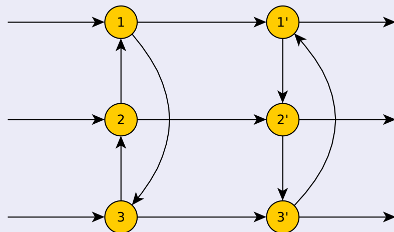
Beispiel für die Formel

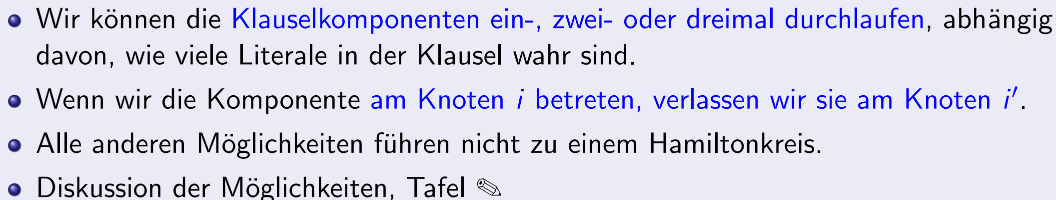
$$\Phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3).$$



Fortsetzung Beweis.

Konstruktion der Klauselkomponenten:





- Wir können die Klauselkomponenten ein-, zwei- oder dreimal durchlaufen, abhängig davon, wie viele Literale in der Klausel wahr sind.
- Wenn wir die Komponente am Knoten i betreten, verlassen wir sie am Knoten i'.
- Alle anderen Möglichkeiten führen nicht zu einem Hamiltonkreis.
- Diskussion der Möglichkeiten, Tafel ✎

Fortsetzung Beweis.

- "$\Rightarrow$": Sei $B : \{x_1, \ldots, x_n\} \to \{\texttt{true}, \texttt{false}\}$ eine Belegung, die Φ wahr macht.
- Dann starten wir den Hamiltonkreis am Variablenknoten x_1 und
- beginnen mit der Kante, die der Variablenbelegung entspricht.
- Die erreichten Klauselkomponenten werden so durchlaufen wie diskutiert,
- wobei wir berücksichtigen, wie viele Literale pro Klausel wahr sind.
- Wir erreichen Variablenknoten x_2 und fahren entsprechend fort.
- Damit konstruieren wir einen gerichteten Hamiltonkreis.

Fortsetzung Beweis.

- “$\Leftarrow$”: Sei ein Hamiltonkreis gegeben.
- Wir durchlaufen ihn beginnend beim Knoten x_1.
- Abhängig von der Kante, welche der Hamiltonkreis wählt, belegen wir x_1.
- Dann durchlaufen wir die erste Klauselkomponente, die das erfüllt x_1-Literal enthält.
- Gemäß unseren Überlegungen wird die Komponente so verlassen, dass wir die zweite Klauselkomponente erreichen, usw.,
- bis wir den Variablenknoten x_2 erreichen.
- Auf diese Weise konstruieren wir eine Belegung der Variablen.
- Wir haben nur Klauselkomponenten durchlaufen, die erfüllte Literale enthalten.
- Da wir einen Hamiltonkreis haben, sind wir durch alle Klauselkomponenten gelaufen, es sind also alle erfüllt.

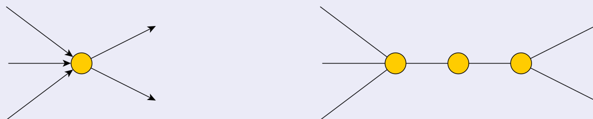
HC ist $\mathcal{NP}$-vollständig

Satz 2.28

HC ist $\mathcal{NP}$-vollständig.

Beweis.

1. Für eine Knotenfolge $(v_1, v_2, \ldots, v_n, v_1)$ können wir in polynomieller Zeit prüfen, ob diese einen Hamiltonkreis von G bildet.
2. Wir zeigen $DHC \leq_p HC$.
   - Hierzu führen wir die folgende Ersetzung für jeden Knoten durch:



   - "$\Rightarrow$": offensichtlich
   - "$\Leftarrow$": Für einen Hamiltonkreis im ungerichteten Graphen legen wir auf Basis des gerichteten Graphen eine Richtung fest.

Traveling-Salesman-Problem

Definition 2.29

Das Traveling-Salesman-Problem (TSP) lautet:

- Gegeben sei ein vollständiger Graph $G = (V, E)$, eine Funktion $c : E \to \mathbb{N}_0$ und ein $k \in \mathbb{N}_0$.
- Enthält G einen Hamiltonkreis mit einer Länge $\leq k$?

Folgerung 2.30

TSP ist $\mathcal{NP}$-vollständig.

Beweis

Übungsaufgabe.

Summenproblem

Definition 2.31

Das Summenproblem (SUM) lautet:

- Gegeben ist eine Menge $A = \{a_1, a_2, \ldots, a_n\} \subseteq \mathbb{N}_0$ und eine Zahl $S \in \mathbb{N}_0$.
- Existiert eine Teilmenge $B \subseteq A$ mit $\sum_{b \in B} b = S$?

Lemma 2.32

SUM ist $\mathcal{NP}$-vollständig.

Beweis.

1. Für eine Menge $B = \{b_1, \ldots, b_k\} \subseteq A$ können wir in polynomieller Zeit prüfen, ob $\sum_{i=1}^{k} b_i = S$ gilt.
2. Wir zeigen 3-SAT $\leq_p$ SUM.

Fortsetzung Beweis.

- Es sei $\Phi = C_1 \wedge \ldots \wedge C_m$ eine 3KNF-Formel, die aus den Klauseln C_i und den Variablen $x_1, \ldots, x_n$ besteht.
- Sei $S = \underbrace{44\cdots4}_{m \text{ Ziffern}}\underbrace{11\cdots1}_{n \text{ Ziffern}}$.
- Die Menge A umfasst $2m + 2n$ viele Zahlen a_i, b_i, c_j, d_j mit $1 \leq i \leq n$ und $1 \leq j \leq m$.
- Alle Zahlen haben wie S auch $m + n$ viele Ziffern.
- Die Zahl a_i repräsentiert das positive Literal x_i.
  - a_i hat genau dann im linken Block bei der j-ten Ziffern eine 1, wenn x_i in der j-ten Klausel auftritt, sonst eine 0.
  - Im rechten Block hat a_i genau an Position i eine 1, sonst 0en.
- Die Zahl b_i ist analog aufgebaut für das Literal $\neg x_i$.
- Die Zahl c_j hat im linken Block an der j-ten Stelle eine 1, sonst überall 0en.
- Die Zahl d_j hat im linken Block an der j-ten Stelle eine 2, sonst überall 0en.

Fortsetzung Beweis.

- Beispiel: $\Phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, also $m = 3, n = 4$.

$$
\begin{array}{rclrcl}
S & = & 444\,1111 & & & \\
\hline
a_1 & = & 100\,1000 & b_1 & = & 011\,1000 \\
a_2 & = & 010\,0100 & b_2 & = & 101\,0100 \\
a_3 & = & 100\,0010 & b_3 & = & 001\,0010 \\
a_4 & = & 000\,0001 & b_4 & = & 010\,0001 \\
\hline
c_1 & = & 100\,0000 & d_1 & = & 200\,0000 \\
c_2 & = & 010\,0000 & d_2 & = & 020\,0000 \\
c_3 & = & 001\,0000 & d_3 & = & 002\,0000
\end{array}
$$

Fortsetzung Beweis.

- "$\Rightarrow$": Sei eine Belegung gegeben, die Φ wahr macht.
- Durch Summenbildung enstehen keine Überträge!
- Dann wählen wir für jedes $1 \leq i \leq n$ von den Zahlen a_i und b_i genau eine aus, abhängig von der Belegung für x_i.
- Damit entsteht in der Summe an allen n Stellen des rechten Blocks eine 1.
- Da die Belegung alle Klauseln erfüllt, haben wir in der Summe an jeder Stelle j des linken Blocks eine 1, 2 oder 3 (abhängig davon, wie viele Literale die j-te Klausel erfüllen).
- Damit können wir für jedes j aus den Zahlen c_j und d_j so auswählen, dass wir in der Summe auf eine 4 kommen.

Fortsetzung Beweis.

- "$\Leftarrow$": Es gebe eine Teilmenge $B \subseteq \{a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_m, d_1, \ldots, d_m\}$ deren Summe gleich S ist.
- Dann muss für jedes i genau eine der beiden Zahlen a_i oder b_i in B enthalten sein (um auf eine 1 an der i-ten Stelle zu kommen).
- Je nachdem welche der beiden Zahlen enthalten ist, belegen wir x_i.
- Da c_j und d_j in der Summe an der j-ten Stelle eine 3 ergeben, muss für jedes j mindestens eine Zahl enthalten sein, die an der j-ten Stelle im linken Block eine 1 hat.
- Damit erfüllt die obige Belegung die j-te Klausel.
- Dies gilt für alle $1 \leq j \leq m$, womit die Formel Φ erfüllt ist.

Rucksackproblem

Definition 2.33

Das Rucksackproblem (KP) lautet:

- Gegeben sind Zahlen $w_1, \ldots, w_n \in \mathbb{N}$ und $p_1, \ldots, p_n \in \mathbb{N}$ sowie Zahlen $W, P \in \mathbb{N}$.
- Existiert eine Teilmenge $I \subseteq \{1, \ldots, n\}$ für die gilt $\sum_{i \in I} w_i \leq W$ und $\sum_{i \in I} p_i \geq P$?

Satz 2.34

KP ist $\mathcal{NP}$-vollständig.

Beweis.

1. Offensichtlich.
2. Mit $w_i = p_i = a_i$ und $P = W = S$ ergibt sich SUM $\leq_p$ KP.

Beweistechniken für $\mathcal{NP}$-Vollständigkeit

Wir wollen zeigen, dass ein Problem Π $\mathcal{NP}$-vollständig ist.

1. Restriktion
   Wir zeigen, dass Π ein $\mathcal{NP}$-vollständiges Problem Π' als Spezialfall enthält.
2. Lokale Ersetzung
   Wir ersetzen einen bestimmten Aspekt eines $\mathcal{NP}$-vollständigen Problems Π', um damit Π zu modellieren.
3. Komponentenentwurf
   Wir konstruieren Komponenten für Π, um damit ein $\mathcal{NP}$-vollständiges Problem Π' lösen zu können.

Restriktion

- Der Beweis für KP ist Restriktion.
- KP mit $p_i = w_i$ und $P = W$ ist SUM.

Beispiel 2.35

Problem Isomorphie von Untergraphen (subgraph isomophism):

- Gegeben seien zwei Graphen $G = (V_1, E_1)$ und $H = (V_2, E_2)$.
- Enthält G einen Untergraphen, der isomorph zu H ist?

Dieses Problem enthält CLIQUE als Spezialfall.

Hierzu wählt man für H einen vollständigen Graphen mit k Knoten.

Subgraph Isomorphism für einen vollständigen Graphen H ist CLIQUE.

Lokale Ersetzung

Wir ersetzen einen bestimmten Aspekt eines $\mathcal{NP}$-vollständigen Problems Π', um damit Π zu modellieren.

Beispiele:

- SAT $\leq_p$ 3-SAT
  Wir haben beliebige Klauseln durch Klauseln mit genau drei Literalen ersetzt.
- DHC $\leq_p$ HC
  Wir haben einzelne Knoten durch Knotentripel ersetzt.

Komponentenentwurf

Beispiele:

- Knotenüberdeckung (VC)
  3-SAT $\leq$ VC
- gerichtetes Hamiltonkreisproblem (DHC)
  3-SAT $\leq$ DHC
- Summenproblem (SUM)
  3-SAT $\leq$ SUM

Optimierungsprobleme und $\mathcal{NP}$ (1)

Die Definitionen 2.29 und 2.33 zeigen beispielhaft, wie wir aus einem Optimierungsproblem ein Entscheidungsproblem machen können.

- Ist Π ein Maximierungsproblem (Minimierungsproblem), so legen wir zusätzlich zu jeder Instanz $\mathcal{I} \in \Pi$ noch eine Schranke B fest und fragen:
  - Gibt es für $\mathcal{I}$ eine Lösung, deren Wert nicht kleiner (nicht größer) als B ist?

Definition 2.36

Wir nennen ein Optimierungsproblem $\mathcal{NP}$-schwer, wenn das zugeordnete Entscheidungsproblem $\mathcal{NP}$-vollständig ist.

Optimierungsprobleme und $\mathcal{NP}$ (2)

- Alle $\mathcal{NP}$-schweren Optimierungsprobleme sind mindestens so schwierig wie die $\mathcal{NP}$-vollständigen Probleme.
- Begründung: Könnten wir ein $\mathcal{NP}$-schweres Optimierungsproblem in polynomieller Zeit lösen, dann könnten wir auch das zugehörige Entscheidungsproblem in polynomieller Zeit lösen.
  - Wir berechnen den Wert w einer Optimallösung und vergleichen ihn mit B.
  - Ist bei einem Maximierungsproblem $w \geq B$, so antworten wir „ja“, ansonsten „nein“.

Optimierungsprobleme und $\mathcal{NP}$ (3)

- Häufig kann man Entscheidungsprobleme dazu benutzen, um Optimierungsprobleme zu lösen.
- Wir betrachten als Beispiel das TSP-Entscheidungsproblem mit n Städten.
- Ist s die kleinste vorkommende Entfernung, so ziehen wir von allen Entfernungen s ab.
- Damit hat dann die kürzeste Entfernung den Wert 0.
- Ist t die größte der (modifizierten Entfernungen), so folgt, dass keine Tour länger als $n \cdot t$ ist.
- Wir fragen nun den Algorithmus zur Lösung des TSP-Entscheidungsproblems, ob es eine Rundreise gibt, deren Länge nicht größer als $\frac{nt}{2}$ ist.
- Ist das so, fragen wir, ob es eine Rundreise gibt, deren Länge höchstens $\frac{nt}{4}$ ist, andernfalls fragen wir, ob es eine Rundreise gibt mit Länge höchstens $\frac{3nt}{4}$.

- Wir fahren auf diese Weise fort, bis wir das Interval für die Tourlänge einer optimalen Lösung auf eine einzelne mögliche Zahl reduziert haben.
- Diese Zahl ist dann die Länge einer kürzesten Rundreise.
- Insgesamt müssen wir zur Lösung des Optimierungsproblems das TSP-Entscheidungsproblem $(\lceil \log_2(nt) \rceil + 1)$-mal aufrufen.
- binäre Suche

$\mathcal{NP}$-Äquivalenz

Definition 2.37

Ein Optimierungsproblem Π heißt $\mathcal{NP}$-leicht, wenn ein Entscheidungsproblem $\Pi' \in \mathcal{NP}$ existiert, so dass Π durch polynomial viele Aufrufe eines Algorithmus zur Lösung von Π' gelöst werden kann.

Ein Optimierungsproblem Π heißt $\mathcal{NP}$-äquivalent, wenn Π sowohl $\mathcal{NP}$-leicht als auch $\mathcal{NP}$-schwer ist.

- $\mathcal{NP}$-leichte Probleme sind also nicht schwerer als die Probleme in $\mathcal{NP}$.
- Ein $\mathcal{NP}$-äquivalentes Problem ist genau dann in polynomieller Zeit lösbar, wenn $\mathcal{P} = \mathcal{NP}$ gilt.

Kombinatorische Optimierung

- Wir könnten ein allgemeines kombinatorisches Optimierungsproblem wie folgt definieren:
  - Gegeben seien eine endliche Menge $\mathcal{I}$ und eine Funktion $f : \mathcal{I} \to \mathbb{R}$, die jedem Element von $\mathcal{I}$ einen Wert zuordnet.
  - Gesucht als Lösung ist ein Element $I^* \in \mathcal{I}$, so dass $f(I^*)$ möglicht groß (oder klein) ist.
- Eine Problemformulierung dieser Art ist aber relativ sinnlos, denn es können kaum vernünftige mathematische Aussagen über das Problem getroffen werden.
- Algorithmisch ist dieses Problem trivial in linearer Laufzeit (in $\mathcal{I}$) lösbar: man durchlaufe alle Elemente $I \in \mathcal{I}$ und werte sie aus.

- In der Regel betrachten wir kombinatorische Optimierungsprobleme mit linearer Zielfunktion.
  - Gegeben sei eine endliche Grundmenge E, eine Menge $\mathcal{I} \subseteq \mathcal{P}(E)$ von zulässigen Lösungen und eine Funktion $c : E \to \mathbb{R}$.
  - Für jede Menge $F \subseteq E$ definieren wir ihren Wert

$$c(F) = \sum_{e \in F} c(e).$$

  - Wir suchen eine Menge $I^* \in \mathcal{I}$, so dass $c(I^*)$ minimal (oder maximal) wird.
- Die Menge $\mathcal{I}$ der zulässigen Lösungen wird dabei üblicherweise implizit angegeben.

$$\mathcal{I} = \{I \subseteq E | I \text{ hat Eigenschaft } \Pi\}$$

- Typischerweise wächst die Anzahl der Elemente von $\mathcal{I}$ dabei exponentiell in der Größe von E.

Gewichtetes VC als kombinatorisches Optimierungsproblem

Beispiel 2.38

- Gegeben sei ein Graph $G = (V, E)$ und eine Kostenfunktion $c : V \to \mathbb{R}$, die jedem Knoten $v \in V$ ein Gewicht zuordnet.
- Dann lautet die Menge $\mathcal{I}$ der zulässigen Lösungen:

$$\mathcal{I} = \{U \subseteq V | \forall e = \{v, w\} \in E : v \in U \vee w \in U\}.$$

- Zielfunktion für ein $I \in \mathcal{I}$:

$$\sum_{v \in I} c(v).$$

Binary Programming

In den meisten Fällen, die wir betrachten, können wir $\mathcal{I}$ durch eine Menge von m linearen Gleichungen oder Ungleichungen ausdrücken.

Definition 2.39

Ein Optimierungsproblem mit

- n Variablen $x_1, \ldots, x_n$,
- Zielfunktion $\sum_{j=1}^{n} c_j x_j$
- Nebenbedingungen $\sum_{j=1}^{n} a_{ij} x_j \theta b_i$ mit $\theta \in \{\leq, \geq, =\}$ und $i = 1, \ldots, m$
- und $x_j \in \{0, 1\}$.

heißt 0-1-Optimierungsproblem (BP).

- engl.: binary programming oder 0-1-programming
- Typischerweise sind dabei c_j, a_{ij}, b_j ganzzahlig.
- kurz: $\min/\max \mathbf{c}^T\mathbf{x}$ u.d.N. $\mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \in \{0,1\}^n$.

Gewichtetes VC als 0-1-Programm

Beispiel 2.40

Das Problem aus Beispiel 2.38 können wir wie folgt definieren:

- Sei $V = \{v_1, \ldots, v_n\}$. Für jeden Knoten $v_i \in V$ definieren wir eine Variable $x_i \in \{0, 1\}$.
- $c_j := c(v_j)$
- Zielfunktion:

$$\min \sum_{j=1}^{n} c_j x_j$$

- Nebenbedingungen: Für jede Kante $e = \{v_i, v_j\} \in E$ ensteht die Ungleichung

$$x_i + x_j \geq 1.$$

BP ist $\mathcal{NP}$-äquivalent

Folgerung 2.41

BP ist $\mathcal{NP}$-äquivalent.

Beweis.

- VC ist $\mathcal{NP}$-vollständig.
- Beispiel 2.40 zeigt, wie VC polynomiell in BP transformiert werden kann (setze $c_j = 1$).
- Für die Entscheidungsvariante von BP kann in polynomieller Zeit geprüft werden, ob ein Lösungsvorschlag $\mathbf{x} \in \{0,1\}^n$ die Bedingungen
  - $\mathbf{Ax} \leq \mathbf{b}$ und
  - $\mathbf{c}^T\mathbf{x} \leq k$ oder $\geq k$ erfüllt.

Folgerung 2.42

ILP (siehe Lineare Optimierung, Definition 1.8) ist $\mathcal{NP}$-äquivalent.

SAT und BP

Beispiel 2.43

Gegeben sei eine aussagenlogische Formel $\Phi = C_1 \wedge \ldots \wedge C_m$ in KNF.

Dann können wir die Frage, ob Φ erfüllbar ist, mit einem 0-1-Programm entscheiden.

- Wir führen für jede aussagenlogische Variable eine algebraische Variable $x_i \in \{0, 1\}$ ein.
- Aus jeder Klausel machen wir folgendermaßen eine Ungleichung:
  - positives Literal wird algebraisch zu x_i
  - negatives Literal wird algebraisch zu $(1 - x_i)$
  - Wir bilden die Summe dieser Terme.
  - Ungleichung: immer ≥ 1
- Beispiel: Aus $x_1 \vee \neg x_2 \vee \neg x_3$ wird

$$x_1 + (1 - x_2) + (1 - x_3) \geq 1 \quad \Leftrightarrow \quad -x_1 + x_2 + x_3 \leq 1.$$

Worum geht's?

Eine Analogie von Vašek Chvátal

"In den kommunistischen Ländern des Ostblocks in den 60'er und 70'er Jahren war es möglich, intelligent, ehrenhaft und ein Mitglied der kommunistischen Partei zu sein, aber es war nicht möglich, alle drei Eigenschaften gleichzeitig zu verkörpern."

- tschechisch-kanadischer Mathematiker
- arbeitet vor allem auf den Gebieten lineare und ganzzahlige Programmierung, Graphentheorie und Kombinatorik

Algorithmischer Wunschzettel

Für die Problemlösung wünschen wir uns Algorithmen, die

1. optimale Lösungen berechnen,
2. für jede mögliche Instanz und
3. in polynomieller Zeit.

Für $\mathcal{NP}$-schwere Probleme können wir dies nicht:

- aktuell nicht, weil niemand solch einen Algorithmus kennt,
- prinzipiell nicht, falls $\mathcal{P} \neq \mathcal{NP}$ gilt.

Was tun?

- Wir verzichten auf die polynomielle Zeit im Worst-Case.
  - klassische kombinatorische Optimierung bzw. ganzzahlige Programmierung.
  - Ziel: Für möglichst große Probleme in akzeptabler Rechenzeit eine optimale Lösung finden.
  - ☞ Kapitel 3 bis 5
- Wir betrachten nicht alle Instanzen.
  - Wir versuchen Spezialfälle zu finden, die in polynomieller Zeit optimal gelöst werden können.
  - problem- statt methodenorientiert, theoretisch interessant, in der Praxis wenig brauchbar
  - ☞ Diesen Ansatz verfolgen wir nicht weiter.
- Wir verzichten auf die Optimalität.
  - Es genügt uns, wenn eine Lösung fast optimal ist.
  - Verwendung von Heuristiken
  - Wir versuchen auch Aussagen über die Güte einer Lösung herzuleiten.
  - ☞ Kapitel 6 und 7

Zusammenfassung

- Simplexalgorithmus ist im Worst-Case nicht polynomiell, aber im Average-Case.
- Mit dem Ellipsoidalgorithmus existiert ein polynomieller Algorithmus für die lineare Programmierung.
- Klasse $\mathcal{NP}$: Polynomiell Lösungsvorschläge überprüfen können.
- polynomielle Transformation
- Kern für den Beweis der $\mathcal{NP}$-Vollständigkeit von Π: Ein $\mathcal{NP}$-vollständiges Π' polynomiell in Π transformieren.
- BP und ILP sind $\mathcal{NP}$-vollständig.