Polynomialzeitreduktion

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 3. Mai 2023
Vorlesungsfolien

Effizient lösbare Probleme

Wiederholung aus dem Kapitel „Analyse von Algorithmen“:
Wir bezeichnen ein Problem als effizient lösbar, wenn es in Polynomialzeit gelöst werden kann. Für ein solches Problem existiert ein Algorithmus mit einer Laufzeit $O(n^c)$

- $n =$ Eingabegröße (z.B. in Bits)
- $c =$ konstanter Exponent

Effizient lösbare Probleme werden auch als handhabbar (*tractable*) bezeichnet.

Cobham–Edmonds Annahme:

- Die Annahme, Handhabbarkeit mit Lösbarkeit in Polynomialzeit gleichzusetzten, geht auf Alan Cobham and Jack Edmonds zurück, die das in den 1960er-Jahren vorgeschlagen haben.
- Diese Annahme hat sich weitgehend durchgesetzt und die Informatikforschung der letzten 50 Jahre geprägt.

Edmonds 1963, Cobham 1965

PATHS, TREES, AND FLOWERS

JACK EDMONDS

1. Introduction. A *graph* G for purposes here is a finite set of elements called *vertices* and a finite set of elements called *edges* such that each edge *meets* exactly two vertices, called the *end-points* of the edge. An edge is said to *join* its end-points.

A *matching* in G is a subset of its edges such that no two meet the same vertex. We describe an efficient algorithm for finding in a given graph a matching of maximum cardinality. This problem was posed and partly solved by C. Berge; see Sections 3.7 and 3.8.

THE INTRINSIC COMPUTATIONAL DIFFICULTY OF FUNCTIONS

ALAN COBHAM
I.B.M. Research Center, Yorktown Heights, N. Y., U.S.A.

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? I grant I have put the first of these questions rather loosely; nevertheless, I think the answer, ought to be: *yes*. It is the second, which asks for a justification of this answer which provides the challenge.

Diskussion

Rechtfertigung der Annahme:

- In der Praxis haben polynomielle Algorithmen meist kleine Konstanten und kleine Exponenten.
- Das Überwinden der exponentiellen Schranke von Brute-Force-Algorithmen legt meist eine wichtige Struktur des Problems offen.

Ausnahmen/Kritik:

- Einige polynomielle Algorithmen haben große Konstanten und/oder große Exponenten und sind praktisch unbrauchbar.
- Einige Algorithmen mit exponentieller (oder schlechterer) Laufzeit werden oft benutzt, da Worst-Case-Instanzen sehr selten auftreten oder die Instanzen klein genug sind.

Probleme klassifizieren: P or not P?

Ziel: Klassifiziere Probleme in solche, die in Polynomialzeit gelöst werden können und in solche, die nicht in Polynomialzeit gelöst werden können.

Erfordert nachweislich mehr als polynomielle Zeit:

- Hält eine gegebene Turingmaschine nach höchstens k Schritten?
- Gegeben sei eine Brettbelegung für eine n-mal-n Generalisierung von Schach. Kann Schwarz garantiert gewinnen?

Probleme klassifizieren: P or not P?

Kein polynomieller Algorithmus bekannt, aber auch kein Nachweis, dass mehr als polynomielle Zeit erforderlich:

- Gegeben ein Graph G und eine Zahl k, enthält G mindestens k Knoten, die paarweise nicht adjazent sind?
- Lassen sich die Knoten eines gegebenen Graphen mit 3 Farben färben, sodass Paare adjazenter Knoten unterschiedliche Farben haben?
- Ist eine gegebene aussagenlogische Formel erfüllbar?

Für viele fundamentale Problem wurde noch keine Klassifikation (polynomiell/exponnentiell) gefunden.

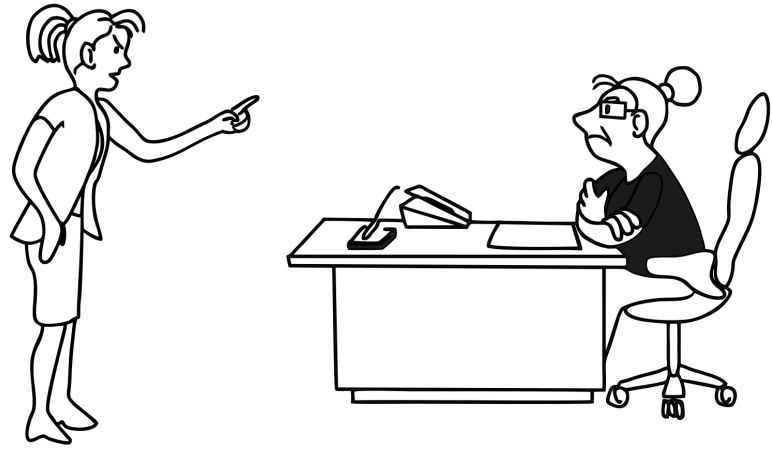
Das ist sehr unzufriedenstellend!

Wie sollen wir damit umgehen, wenn wir ein Problem nicht in Polynomialzeit lösen können?

Keine gute Lösung

“I can’t find an efficient algorithm, I guess I’m just too dumb.”

Wäre die beste Lösung, aber oft nicht möglich



“I can’t find an efficient algorithm, because no such algorithm is possible!”

Gute Kompromisslösung

“I can’t find an efficient algorithm, but neither can all these famous people.”

Äquivalenz bezüglich Lösbarkeit in Polynomialzeit

Dieser Foliensatz: Wir zeigen, dass viele dieser fundamentalen Probleme vom Berechnungsaufwand her im gewissen Sinne äquivalent und eigentlich nur unterschiedliche Erscheinungsformen eines einzigen wirklich schweren Problems sind.

Ein wichtiges Konzept dabei sind so genannte Polynomialzeitreduktionen, mit denen ein Problem in ein anderes „übersetzt“ werden kann.

Ja/Nein-Problem

Einschränkung: Einfachheitshalber beschränken wir uns auf Ja/Nein-Probleme.

Ja/Nein-Problem (*decision problem*):

- Ein Ja/Nein-Problem ist ein Problem, für das die Lösung eine Ja- oder Nein-Antwort ist.
- Im Gegensatz dazu wird bei funktionalen Problemen oder Optimierungsproblemen eine Lösung (Lösungsmenge) oder mehr als ein Bit retourniert.

Beispiel und Unterscheidung zu anderen Problemen:

- Ja/Nein-Problem: Gibt es für einen gewichteten Graphen einen Spannbaum mit Kosten $\leq k$? Das kann mit Ja oder Nein beantwortet werden.
- Funktionales Problem: Finde in einem gewichteten Graphen einen Spannbaum mit Kosten $\leq k$.
- Optimierungsproblem: Finde in einem gewichteten Graphen einen Spannbaum mit minimalen Kosten (MST).

Polynomialzeitreduktionen

Frage: Angenommen, wir können ein Problem in Polynomialzeit lösen. Welche anderen Probleme können wir in Polynomialzeit lösen?

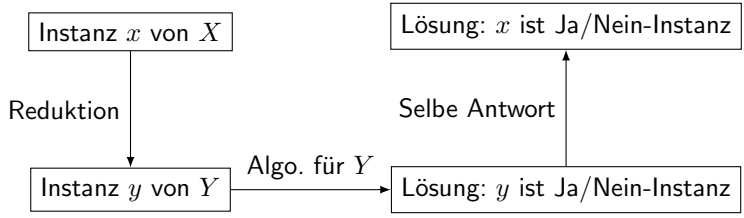
Frage: Wie stellen wir eine Verbindung zu anderen Problemen her?

Reduktion: Problem X kann polynomiell auf ein Problem Y reduziert werden.

Polynomialzeitreduktionen

Informell: Kann X polynomiell auf Y reduziert werden, und wir können Y effizient lösen, dann können wir auch X effizient lösen.

Wir reduzieren dazu eine Instanz x von X auf eine Instanz y vom Y, dann lösen wir y. Ist y eine Ja-Instanz von Y, dann ist x auch eine Ja-Instanz von X.



Polynomialzeitreduktionen

Definition: Eine Polynomialzeitreduktion von Problem X auf Problem Y ist ein Algorithmus R, der für jede Instanz x von X eine Instanz y von Y berechnet, sodass folgendes gilt:

1. x ist eine Ja-Instanz von X genau dann, wenn y eine Ja-Instanz von Y ist.
2. Der Algorithmus R läuft in Polynomialzeit (d.h. es gibt eine Konstante c, sodass der Algorithmus R die Instanz y in Zeit $O(n^c)$ berechnet, wobei n die Eingabegröße für x ist).

Notation: Wir schreiben $X \leq_P Y$, falls es eine Polynomialzeitreduktion von X auf Y gibt. In dem Fall sagen wir auch, dass „X auf Y polynomiell reduzierbar ist“.

Hinweis: Falls $X \leq_P Y$, dann können wir auch sagen, dass „Y mindestens so schwer ist wie X“.

Lösung durch Reduktion

Idee: Wir lösen ein Problem durch Polynomialzeitreduktion auf ein anderes Problem, das handhabbar ist.

Es gilt: Wenn $X \leq_P Y$ und Y kann in polynomieller Zeit gelöst werden, dann kann auch X in polynomieller Zeit gelöst werden.

Beweis:

- Sei R ein Algorithmus der X auf Y reduziert, der in Zeit $O(n^a)$ läuft (für Instanzen von X der Größe n, $a \geq 1$).
- Sei A ein Algorithmus der Y in Zeit $O(n^b)$ löst (für Instanzen von Y der Größe n, $b \geq 1$).
- Sei nun x eine Instanz von X der Größe n.
- Wir wenden R an und erhalten in $O(n^a)$ Zeit eine Instanz y von Y.
- die Größe von y ist höchstens $O(n^a)$, da y in Zeit $O(n^a)$ erzeugt wurde.
- Wir lösen y mit A in Zeit $O((n^a)^b) = O(n^{ab})$.
- Insgesamt haben wir x in Zeit $O(n^a) + O(n^{ab}) = O(n^{ab})$ gelöst, das ist polynomiell in n. $\square$

Nicht-Handhabbarkeit zeigen

Idee: Wir zeigen, dass ein Problem nicht handhabbar ist, in dem wir ein anderes nicht handhabbares Problem darauf reduzieren.

Es gilt: Wenn $X \leq_P Y$ und X kann nicht in polynomieller Zeit gelöst werden, dann kann Y nicht in polynomieller Zeit gelöst werden.

Beweis:

- Angenommen Y wäre in polynomieller Zeit lösbar.
- $X \leq_P Y$ bedeutet, dass wir X in polynomieller Zeit auf Y reduzieren können.
- Da die Reduktion $X \leq_P Y$ und das Lösen von Y in polynomieller Zeit ausführbar sind, ist X in polynomieller Zeit lösbar. Das ist ein Widerspruch zur Annahme. $\square$

Äquivalenz: Wenn $X \leq_P Y$ und $Y \leq_P X$, dann schreiben wir $X \equiv_P Y$.

Transitivität

Es gilt: Ist $X \leq_P Y$ und $Y \leq_P Z$, dann folgt daraus $X \leq_P Z$.

Beweis:

- Sei R_1 der Algorithmus für $X \leq_P Y$, der in $O(n^a)$ läuft, $a \geq 1$.
- Sei R_2 der Algorithmus für $Y \leq_P Z$, der in $O(n^b)$ läuft $b \geq 1$.
- Sei n die Größe von x
- Dann benötigt der Aufruf von R_1 auf x höchstens $O(n^a)$ Zeit.
- Weiters erzeugt R_1 eine Ausgabe x' mit einer Größe von $O(n^a)$.
- Daher benötigt der darauffolgende Aufruf von R_2 auf x' eine Laufzeit von $O(n^{ab})$.
- Damit ist aber die Reduktion von X auf Z in polynomieller Zeit möglich. $\square$

Polynomialzeitreduktionen angeben

- Im Folgenden werden wir einige Beispiele von Polynomialzeitreduktionen betrachten.
- Wenn wir eine solche Reduktion angeben, müssen wir zwei Eigenschaften sicherstellen (siehe die Definition einer Polynomialzeitreduktion)
  1. die Reduktion ist korrekt, d.h., Ja-Instanzen werden auf Ja-Instanzen abgebildet, und Nein-Instanzen auf Nein-Instanzen
  2. die Reduktion ist polynomiell, d.h., die Reduktion kann in Polynomialzeit ausgeführt werden.
- In manchen Fällen ist Korrektheit einfach zu zeigen, in anderen Fällen die Polynomialzeit.

Reduktion durch einfache Äquivalenz

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion durch Kodierung mit Gadgets.

Independent Set

Independent Set: Ein Independent Set eines Graphen $G = (V, E)$ ist eine Teilmenge $S \subseteq V$, in der es keine zwei adjazenten Knoten gibt.

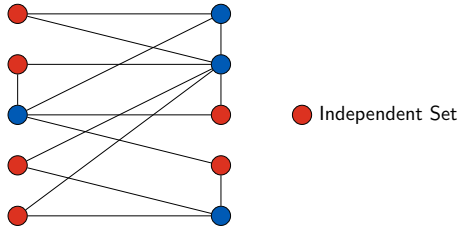
INDEPENDENT SET: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Independent Set S, sodass $|S| \geq k$ gilt?

Wichtig: Die Zahl k ist Teil der Eingabe und keine Konstante.

Hinweis: Ja/Nein-Probleme werden ab jetzt mit dieser Schreibweise (INDEPENDENT SET) gekennzeichnet.

Independent Set

Independent Set: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Independent Set S, sodass $|S| \geq k$ gilt?

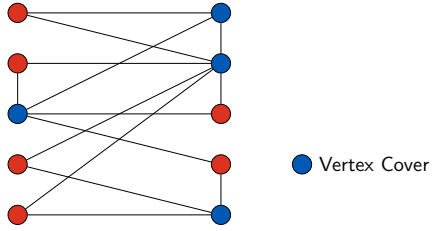


Beispiel: Existiert ein Independent Set der Größe ≥ 6? Ja.
Beispiel: Existiert ein Independent Set der Größe ≥ 7? Nein.

Vertex Cover

Vertex Cover: Ein Vertex Cover eines Graphen $G = (V, E)$ ist eine Menge $S \subseteq V$, sodass jede Kante des Graphen zu mindestens einem Knoten aus S inzident ist.

VERTEX COVER: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Vertex Cover S von G, sodass $|S| \leq k$?

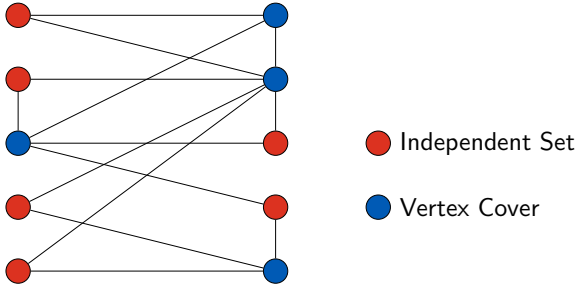


Beispiel: Existiert ein Vertex Cover der Größe ≤ 4? Ja.
Beispiel: Existiert ein Vertex Cover der Größe ≤ 3? Nein.

Vertex Cover und Independent Set

Wir wollen zeigen, dass VERTEX COVER $\equiv_P$ INDEPENDENT SET. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.



Vertex Cover und Independent Set

Wir wollen zeigen, dass VERTEX COVER $\equiv_P$ INDEPENDENT SET. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.

Beweis: $\Rightarrow$

- Betrachte eine beliebige Kante $(u, v) \in E$. Wir wollen zeigen, dass mindestens einer der beiden Knoten u, v in C liegt.
- Weil S ein Independent Set ist, muss mindestens einer der beiden Knoten u, v nicht in S liegen. Also liegt mindestens einer der beiden Knoten in C.
- Daher ist C ein Vertex Cover. $\square$

Vertex Cover und Independent Set

Wir wollen zeigen, dass Vertex Cover $\equiv_P$ Independent Set. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.

Beweis: $\Leftarrow$

- Betrachte zwei Knoten $u \in S$ und $v \in S$. Wir wollen zeigen, dass u und v nicht adjazent sind.
- Aus $u \in S$ und $v \in S$ folgt $u \notin C$ und $v \notin C$.
- u und v können nicht adjazent sein, ansonsten wäre C kein Vertex Cover (weil es die Kante (u, v) nicht überdeckt).
- Also ist S ein Independent Set. $\square$

Also gilt das Lemma.

Vertex Cover und Independent Set

Es gilt: Vertex Cover $\equiv_P$ Independent Set.

Beweis:
Zuerst zeigen wir Vertex Cover $\leq_P$ Independent Set.

- Sei (G, k) eine Instanz von Vertex Cover. Sei n die Anzahl der Knoten von G.
- In Polynomialzeit generieren wir $(G, n - k)$, eine Instanz von Independent Set.
- Die Reduktion ist *korrekt*, da G ein Vertex Cover $\leq k$ hat genau dann wenn G ein Independent Set der Größe $\geq n - k$ hat (folgt aus dem Konversionslemma).
- Die Reduktion ist klarerweise *polynomiell*, da sie ja nur n durch $n - k$ ersetzten braucht. $\square$

Vertex Cover und Independent Set

Es gilt: VERTEX COVER $\equiv_P$ INDEPENDENT SET.

Beweis:

Weiters zeigen wir INDEPENDENT SET $\leq_P$ VERTEX COVER

- Sei (G, k) eine Instanz von INDEPENDENT SET. Sei n die Anzahl der Knoten von G.
- In Polynomialzeit generieren wir $(G, n-k)$ eine Instanz von VERTEX COVER.
- Die Reduktion ist korrekt, da G ein Independent Set $\geq k$ hat genau dann wenn G ein Vertex Cover Set der Größe $\leq n-k$ hat (folgt aus dem Konversionslemma).
- Die Reduktion ist klarerweise *polynomiell*, da sie ja nur n durch $n-k$ ersetzten braucht. $\square$

Wir haben also beide Richtungen gezeigt, und es folgt die Äquivalenz.

Beispiel: Spannbäume und Nicht-Blockierer

NICHT-BLOCKIERER: Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$. Ein *Nicht-Blockierer* ist eine Teilmenge der Kanten $N \subseteq E$, sodass es für alle Knotenpaare $u, v \in V$ einen u-v-Pfad in G gibt, der keine Kante aus N enthält.
Die *Kosten* des Nicht-Blockierers ist gegeben durch $\sum_{e \in N} c_e$.
Ein *maximaler Nicht-Blockierer* ist ein Nicht-Blockierer mit größten Kosten.

MNB: Gegeben ist ein gewichteter Graph G und eine Zahl k. Besitzt G einen Nicht-Blockierer mit Kosten $\geq k$?

Ist das Problem MNB in Polynomialzeit lösbar?

Beispiel: Spannbäume und Nicht-Blockierer

Wir zeigen, dass MNB in Polynomialzeit lösbar ist, indem wir es auf ein bekanntes Problem reduzieren.

SPANNBAUM: Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$. Ein *Spannbaum* ist eine Teilmenge der Kanten $T \subseteq E$, sodass $G_T = (V, T)$ ein Baum ist.
Die *Kosten* des Baumes ist gegeben durch $\sum_{e \in T} c_e$.
Ein *minimaler Spannbaum* ist ein Spannbaum mit kleinsten Kosten.

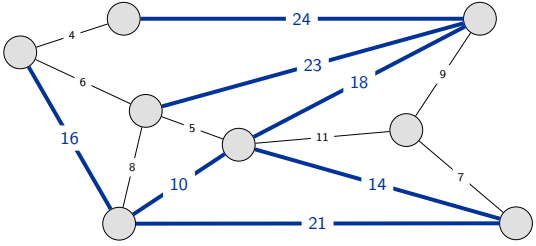
MST^*: Gegeben ist ein gewichteter Graph G und eine Zahl k. Besitzt G einen Spannbaum mit Kosten $\leq k$?

Wir wissen aus dem Kapitel „Greedy-Algorithmen“, dass MST^* in Polynomialzeit gelöst werden kann.

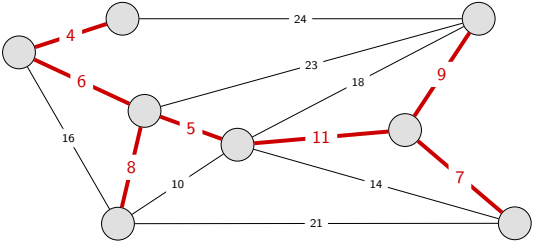
Beispiel: Spannbäume und Nicht-Blockierer

Konversionslemma: Sei $G = (V, E)$ ein gewichteter Graph, $N \subseteq E$ und $T = E - N$. Dann ist N ein maximaler Nicht-Blockierer genau dann wenn T ein minimaler Spannbaum ist. Die Kosten von N sind genau $K := \sum_{e \in E} c_e$ minus der Kosten von T.

Wir lassen den sehr einfachen Beweis als Übung.



N, Kosten $= 126$



T, Kosten $= 50$

Beispiel: Spannbäume und Nicht-Blockierer

Konversionslemma: Sei $G = (V, E)$ ein gewichteter Graph, $N \subseteq E$ und $T = E - N$. Dann ist N ein maximaler Nicht-Blockierer genau dann wenn T ein minimaler Spannbaum ist. Die Kosten von N sind genau $K := \sum_{e \in E} c_e$ minus der Kosten von T.

Wir lassen den sehr einfachen Beweis als Übung.

Es gilt: $\mathrm{MNB} \equiv_P \mathrm{MST}^*$

Beweis:

- $\mathrm{MNB} \leq_P \mathrm{MST}^*$: Wir reduzieren eine Instanz (G, k) von MNB auf die Instanz $(G, K - k)$ von MST^*.
- $\mathrm{MST}^* \leq_P \mathrm{MNB}$: Wir reduzieren eine Instanz (G, k) von MST^* auf die Instanz $(G, K - k)$ von MNB. $\square$

Es folgt daher: MNB ist in Polynomialzeit lösbar.

Reduktion eines Spezialfalls auf den allgemeinen Fall.

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion durch Kodierung mit Gadgets.

Set Cover (Mengenüberdeckungsproblem)

Set Cover: Gegeben sei eine Menge U von Elementen und eine Menge $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ von Teilmengen von U. Ein Set Cover ist eine Teilmenge $\mathcal{C} \subseteq \mathcal{S}$, also eine Menge von Mengen, deren Vereinigung U entspricht. $\mathcal{C}$ ist ein Set Cover von $\mathcal{S}$.

SET COVER: Gegeben sei eine Menge U von Elementen, eine Menge $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ von Teilmengen von U und eine ganze Zahl k. Existiert eine Teilmenge $\mathcal{C} \subseteq \mathcal{S}$ mit $|\mathcal{C}| \leq k$, sodass die Vereinigung von $\mathcal{C}$ gleich U ist?

Beispielhafte Anwendung:

- m verfügbare Softwarekomponenten.
- Menge U von n Eigenschaften, die unser Softwaresystem haben sollte.
- Die i-te Softwarekomponente bietet eine Menge $S_i \subseteq U$ von Eigenschaften an.
- Ziel: Erreiche alle n Eigenschaften mit maximal k Komponenten.

Set Cover: Beispiel

Beispiel:

$U = \{1, 2, 3, 4, 5, 6, 7\}$
$k = 2$

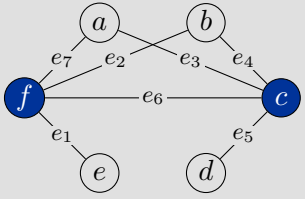
$S_1 = \{3, 7\}$	$S_4 = \{2, 4\}$
$S_2 = \{3, 4, 5, 6\}$	$S_5 = \{5\}$
$S_3 = \{1\}$	$S_6 = \{1, 2, 6, 7\}$

Vertex Cover auf Set Cover reduzieren

Behauptung: Vertex Cover $\leq_P$ Set Cover.
Beweis: Gegeben sei eine Vertex Cover Instanz (G, k) mit $G = (V, E)$.
Wir konstruieren eine Instanz von Set Cover.

- $k = k$,
- $U = E$,
- Für jedes $v \in V$ erzeugen wir eine Menge $S_v = \{e \in E : \ e \text{ inzident zu } v\}$
- $\mathcal{S}$ ist die Menge aller S_v für $v \in V$.
- Die Reduktion ist klarerweise *polynomiell*.

Vertex Cover



$k = 2$

Set Cover
$U = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ $k = 2$
$S_a = \{e_3, e_7\}$, $S_b = \{e_2, e_4\}$
$S_c = \{e_3, e_4, e_5, e_6\}$, $S_d = \{e_5\}$
$S_e = \{e_1\}$, $S_f = \{e_1, e_2, e_6, e_7\}$

Vertex Cover auf Set Cover reduzieren

Behauptung: Vertex Cover $\leq_P$ Set Cover.
Beweis: Gegeben sei eine Vertex Cover Instanz (G, k) mit $G = (V, E)$.
Wir konstruieren eine Instanz von Set Cover.

- $k = k$,
- $U = E$,
- Für jedes $v \in V$ erzeugen wir eine Menge $S_v = \{e \in E : \ e \text{ inzident zu } v\}$
- $\mathcal{S}$ ist die Menge aller S_v für $v \in V$.
- Die Reduktion ist klarerweise *polynomiell*.
- *Korrektheit:* G hat ein Vertex Cover der Größe $\leq k$ genau dann wenn $\mathcal{S}$ ein Set Cover der Größe $\leq k$ hat.
- $\Rightarrow$: Sei $C = \{v_1, \ldots, v_k\} \subseteq V$ ein Vertex Cover von G. Dann ist $\mathcal{C} = \{S_{v_1}, \ldots, S_{v_k}\}$ ein Set Cover von $\mathcal{S}$.
- $\Leftarrow$: Sei $\mathcal{C} = \{S_{v_1}, \ldots, S_{v_k}\}$ ein Set Cover von $\mathcal{S}$. Dann ist $C = \{v_1, \ldots, v_k\} \subseteq V$ ein Vertex Cover von G. $\square$

Bemerkung

Nicht jede Set Cover Instanz kann durch die Reduktion von Vertex Cover entstehen.

Daher sprechen wir hier von einer „Reduktion eines Spezialfalls auf den allgemeinen Fall“.

Reduktion mit „Gadgets“

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion mit „Gadgets“.

Erfüllbarkeit (satisfiability)

Literal: Eine boolesche Variable oder ihre Negation: x_i oder $\overline{x_i}$

Klausel: Eine Disjunktion (logisches Oder) von Literalen:
Beispiel: $C_j = x_1 \vee \overline{x_2} \vee x_3$

Konjunktive Normalform (KNF): Eine aussagenlogische Formel Φ, bei der Klauseln konjunktiv (logisches Und) verknüpft werden.
Beispiel: $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

Wahrheitsbelegung (*truth assignment*): Eine Wahrheitsbelegung ist eine Funktion f, die jeder Variable einen Wahrheitswert *true* oder *false* zuordnet.

Erfüllen einer Formel: Eine Wahrheitsbelegung f erfüllt eine KNF-Formel Φ, falls jede Klausel von Φ mindestens eine Variable x mit $f(x) =$ *true* oder eine negierte Variable $\overline{x}$ mit $f(x) =$ *false* enthält.

Das Erfüllbarkeitsproblem (SAT)

SAT (*satisfiability*): Gegeben ist eine KNF-Formel Φ. Gibt es eine Wahrheitsbelegung, die Φ erfüllt?

3-SAT: SAT, bei dem jede Klausel genau 3 Literale enthält.

- *Jedes Literal muss sich auf eine unterschiedliche Variable beziehen.*

Beispiel: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$
Erfüllende Wahrheitsbelegung: $f(x_1) = \textit{true}$, $f(x_2) = \textit{true}$, $f(x_3) = \textit{false}$.

Bedeutung von SAT

Das Erfüllbarkeitsproblem (SAT) ist in zweierlei Hinsicht von großer Bedeutung:

1. Für SAT gibt es mächtige heuristische Algorithmen (SAT-solver), die große „strukturierte“ Instanzen lösen können.
   → Daher ist SAT ein beliebtes Problem um andere Probleme darauf zu reduzieren (*Lösung durch Reduktion*).
2. Andererseits wird SAT für allgemeine Instanzen als nicht-handhabbar angesehen (SAT ist „NP-vollständig“, was wir auf den nächsten Folien erläutern werden).
   → Daher ist SAT ein beliebtes Problem, das auf andere Probleme reduziert wird, um deren *Nicht-Handhabbarkeit zu zeigen*.

3-SAT auf INDEPENDENT SET reduzieren

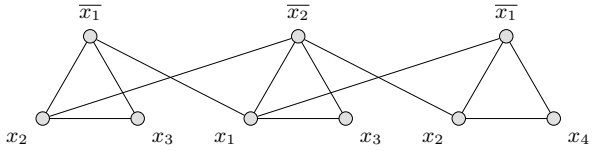
Behauptung: 3-SAT $\leq_P$ INDEPENDENT SET.

Beweis: Gegeben sei eine Instanz Φ von 3-SAT mit k Klauseln. Wir konstruieren eine Instanz (G, k) von INDEPENDENT SET.

- G enthält 3 Knoten für jede Klausel (einen für jedes Literal).
- Verbinde 3 Literale in einer Klausel zu einem Dreieck.
- Verbinde ein Literal mit jeder seiner Negationen.

Die Reduktion ist klarerweise *polynomiell*.

G



$k = 3$

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

3-SAT auf INDEPENDENT SET reduzieren

Korrektheit: G enthält ein Independent Set der Größe k genau dann wenn Φ erfüllbar ist.

Beweis: ($\Rightarrow$) Sei S ein Independent Set der Größe k.

- S enthält genau einen Knoten pro Dreieck (S kann höchstens einen Knoten pro Dreieck enthalten ansonsten ist S nicht unabhängig; S muss mindestens einen Knoten pro Dreieck enthalten, da $|S| = k$ und es k Dreiecke gibt)
- Wir konstruieren eine Wahrheitsbelegung der Variablen indem wir $x = \textit{true}$ setzen, falls $x \in S$ und $x = \textit{false}$ setzen, falls $\overline{x} \in S$.
- Wegen der Kanten zwischen den Dreiecken kann es zu keinen widersprüchlichen Belegungen ein und der selben Variable kommen.
- Wir setzen die übrigen Variablen beliebig auf *true* oder *false*.
- Diese Wahrheitsbelegung erfüllt alle Klauseln. $\square$

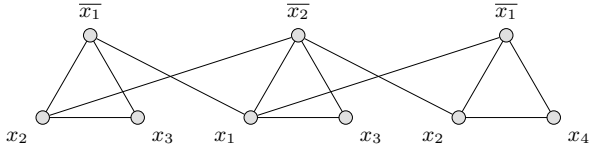
3-Sat auf Independent Set reduzieren

Korrektheit: G enthält ein Independent Set der Größe k genau dann wenn Φ erfüllbar ist.

Beweis: ($\Leftarrow$) Gegeben sei eine Wahrheitsbelegung f die Φ erfüllt.

- Wir konstruieren ein Independent Set S indem wir von jedem Dreieck einen Knoten ℓ mit $\ell = x$ und $f(x) = \textit{true}$ oder einen Knoten ℓ mit $\ell = \overline{x}$ und $f(x) = \textit{false}$ wählen. (So einen Knoten ℓ gibt es immer, da f erfüllend)
- S ist unabhängig, weil die Kanten zwischen den Dreiecken jeweils zwischen einer Variable und ihrer Negation verlaufen.
- $|S| = k$, weil wir von jedem Dreieck einen Knoten wählen. $\square$

G



$k = 3$

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Rückblick

Grundlegende Reduktionsstrategien:

- Einfache Äquivalenz:
  Independent Set $\equiv_P$ Vertex Cover.
- Spezieller Fall auf allgemeinen Fall:
  Vertex Cover $\leq_P$ Set Cover.
- Kodierung mit Gadgets:
  3-Sat $\leq_P$ Independent Set.

Transitivität: Wenn $X \leq_P Y$ und $Y \leq_P Z$, dann $X \leq_P Z$.
Beweisidee: Verbinde zwei Algorithmen.

Beispiel: 3-Sat $\leq_P$ Independent Set $\leq_P$ Vertex Cover $\leq_P$ Set Cover.

Optimierungsprobleme

Ja/Nein-Problem: Existiert ein Vertex Cover der Größe $\leq k$?
Optimierungsproblem: Finde kleinstes Vertex Cover.

Klar:

- Ja/Nein-Problem kann mit dem Optimierungsproblem gelöst werden.
- Berechne kleinstes Vertex Cover C und antworte „Ja“ falls $|C| \leq k$ ist.

Umgekehrte Richtung: Löse Optimierungsproblem mittels (mehrmaligem) Lösen des Ja/Nein-Problems.

Optimierungsproblem für Vertex Cover lösen

Ausgangslage: Es existiert ein Algorithmus VC(G,k), der das Ja/Nein-Problem für ein Vertex Cover der Größe $\leq k$ löst.

Wir möchten mittels VC(G,k) ein kleinstes Vertex Cover finden.

Dazu verwenden wir die folgenden Eigenschaften, die für jeden Graphen $G = (V, E)$ gelten:

1. Sei $v \in V$. Falls C ein Vertex Cover von $G - v$ ist, dann ist $C \cup \{v\}$ ein Vertex Cover von G.
2. Falls G ein Vertex Cover der Grösse $k \geq 1$ besitzt, dann gibt es ein $v \in V$ sodass $G - v$ ein Vertex Cover der Größe $k - 1$ besitzt.

Optimierungsproblem für Vertex Cover lösen

Der Algorithmus OptVC(G) berechnet ein kleinstes Vertex Cover von G mittels mehrmaligen Aufrufs von VC(G,k).

```
OptVC(G):
for k ← 0 bis n − 1
    if VC(G,k)
        return FindVC(G,k)

FindVC(G,k):
if k = 0
    return ∅
else
    foreach v ∈ V(G)
        if VC(G − v,k − 1)
            return {v} ∪ FindVC(G − v,k − 1)
```

Komplexität: Insgesamt wird VC(G,k) höchstens $O(n) + O(n^2) = O(n^2)$ mal aufgerufen.

Optimierungsprobleme

- Analog kann für viele andere Optimierungsprobleme vorgegangen werden.
- Das rechtfertigt unseren Fokus auf Ja/Nein-Probleme.

Definition von NP

NP-Probleme

NP-Probleme: Ein Ja/Nein-Problem ist ein NP-Problem, falls wir Ja-Instanzen mit Hilfe eines Zertifikats effizient überprüfen können.

Zertifikat: Ein Zertifikat t für eine Instanz x ist ein beliebiger Input dessen Größe m polynomiell in der Größe n von x beschränkt ist, das heißt $m \leq p(n)$ gilt für ein Polynom p.

Zertifizierer: Einen Polynomialzeitalgorithmus $C(x, t)$, der Ja-Instanzen x mit Hilfe von Zertifikaten t überprüft, nennt man Zertifizierer.

Anmerkung zur Notation: NP steht für „nicht-deterministisch polynomielle“ Zeit.

Für ein NP-Problem X sagen wir auch „X ist in NP“.

NP-Probleme

Genauer gesagt, der Zertifizierer $C(x, t)$ soll folgende Eigenschaften haben:

- Für jede Ja-Instanz x gibt es ein Zertifikat t (polynomieller Länge), welches den Zertifizierer zum akzeptieren bringt.
  („Zertifizierer kann überzeugt werden“)
- Für keine Nein-Instanz x gibt es ein Zertifikat t, welches den Zertifizierer zum akzeptieren bringt.
  („Zertifizierer kann nicht ausgetrickst werden“)

NP-Probleme

Beispiel VERTEX COVER:

- Für eine Ja-Instanz (G, k) nehmen wir ein Vertex Cover S der Größe $\leq k$ als Zertifikat.
- Wir können in Polynomialzeit überprüfen, ob S tatsächlich ein Vertex Cover der Größe $\leq k$ ist.
  - Die Größe von S lässt sich in Polynomialzeit überprüfen.
  - Danach überprüft man für jede Kante in G, ob diese zumindest einen Endpunkt in S hat. Das lässt sich wiederum in Polynomialzeit überprüfen.

Schlussfolgerung: VERTEX COVER ist ein NP-Problem
(anders gesagt, VERTEX COVER ist in NP).

Beispiel für Zertifizierer und Zertifikate: SAT

SAT: Gegeben sei eine Formel Φ in konjunktiver Normalform. Ist diese Formel erfüllbar?

Zertifikat: Eine Wahrheitsbelegung f für die n booleschen Variablen die Φ erfüllt.

Zertifizierer: Überprüfe ob f die Formel Φ erfüllt.

Beispiel:

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)\,;$$

Instanz s

$$f(x_1) = 1,\ f(x_2) = 1,\ f(x_3) = 0,\ f(x_4) = 1$$

Zertifikat t

Schlussfolgerung: SAT ist in NP.

Beispiel für Zertifizierer und Zertifikate: Hamiltonkreisproblem

HAM-CYCLE: Gegeben sei ein ungerichteter Graph G. Existiert ein Kreis C in G der alle Knoten von G genau einmal enthält?

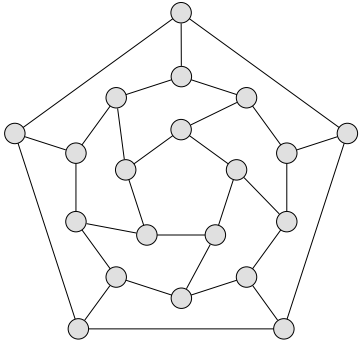
So ein Kreis wird als *Hamiltonkreis* bezeichnet.

Zertifikat: Eine Hamiltonkreis.

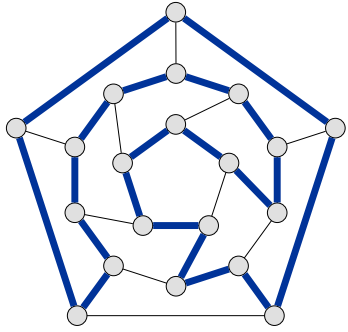
Zertifizierer: Überprüfe, ob der Hamiltonkreis jeden Knoten in V genau einmal enthält und dass es eine Kante zwischen jedem Paar von direkt aufeinander folgenden Knoten in dem Hamiltonkreis und auch vom ersten zum letzten Knoten gibt.

Beispiel für Zertifizierer und Zertifikate: Hamiltonkreisproblem

Beispiel:



Instanz s



Zertifikat t

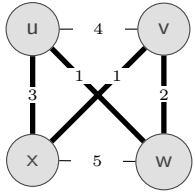
Schlussfolgerung: HAM-CYCLE ist in NP.

Travelling Salesman Problem (TSP)

Travelling Salesman Problem (TSP):

- Man sucht eine Reihenfolge für den Besuch mehrerer Orte (eine „Tour“), sodass die gesamte Reisestrecke eines Handlungsreisenden möglichst kurz ist (Minimierungsproblem).
- Die Strecke zwischen zwei Orten u und v ist als reelle Zahl $d(u, v)$ gegeben.
- Der erste Ort sollte gleich dem letzten Ort sein.

Beispiel: 4 Orte, minimale Tour der Länge 7.



Travelling Salesman Problem (TSP)

Wir bezeichnen mit TSP^* das Ja/Nein-Problem, das nach einer Tour der Länge $\leq k$ fragt.

Reduktion: $\textsc{Ham-Cycle} \leq_P \mathrm{TSP}^*$.

Beweis:

- Gegeben sei eine Instanz $G = (V, E)$ von $\textsc{Ham-Cycle}$.
- Erzeuge n Städte mit einer Distanzfunktion

$$d(u,v) = \begin{cases} 1 & \text{wenn} (u,v) \in E \\ 2 & \text{wenn} (u,v) \notin E \end{cases}$$

- TSP-Instanz besitzt eine Tour der Länge n genau dann, wenn G einen Hamiltonkreis besitzt. $\square$

P, NP

P: Ja/Nein-Probleme, für die polynomielle Algorithmen existieren.

NP: Ja/Nein-Probleme, für die polynomielle Zertifizierer existieren.

Es gilt: P $\subseteq$ NP.

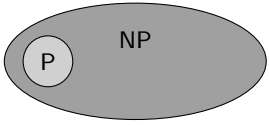
Beweis: Wir betrachten ein beliebiges Problem X in P.

- Nach Definition existiert ein Polynomialzeit-Algorithmus $A(s)$, der X löst.
- Zertifikat: $t = \emptyset$, Zertifizierer $C(s, t) = A(s)$. $\square$

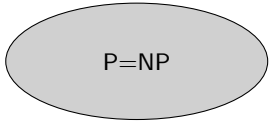
P = NP?

Gilt P = NP? [Cook 1971, Levin 1973]

- Ist das Ja/Nein-Problem so leicht wie das Zertifizierunsgproblem?
- Für die Beantwortung der Frage ist 1 Million US Dollar ausgeschrieben (Clay Mathematics Institute).



Falls P $\neq$ NP



Falls P = NP

P = NP?

Gilt P = NP? [Cook 1971, Levin 1973]

Falls ja: Effiziente Algorithmen für Vertex Cover, Ham-Cycle, TSP*, Sat, ...

Falls nein: Keine effizienten Algorithmen für Vertex Cover, Ham-Cycle, TSP*, Sat, ...

Vorherrschende Meinung zu P = NP: Wahrscheinlich „nein“

NP-Vollständigkeit

NP-Vollständigkeit

Definition: Ein Ja/Nein-Problem Y ist NP-schwer, falls für jedes Problem X in NP gilt, dass $X \leq_p Y$.

Das heißt, jedes NP-Problem X kann in Polynomialzeit auf Y reduziert werden.

NP-schwere Probleme sind also gewissermaßen „mindestens so schwer“ wie alle Probleme in NP.

Definition: Ein Problem Y ist NP-vollständig, falls es sowohl in NP liegt als auch NP-schwer ist.

Die NP-vollständigen Probleme sind also gewissermaßen die „schwersten“ Probleme in NP.

Nach dieser Definition können nur Ja/Nein-Probleme NP-vollständig sein.

NP-Vollständigkeit

Theorem: Sei Y ein NP-vollständiges Problem.
Y ist in polynomieller Zeit lösbar genau dann, wenn P $=$ NP.

Beweis: ($\Leftarrow$) Wenn P $=$ NP, dann kann Y in polynomieller Zeit gelöst werden, da Y sich in NP befindet.
Beweis: ($\Rightarrow$) Angenommen, Y kann in polynomieller Zeit gelöst werden.

- Sei X ein beliebiges Problem in NP. Da $X \leq_p Y$, können wir X in Polynomialzeit lösen. Das impliziert NP $\subseteq$ P.
- Wir wissen bereits, dass P $\subseteq$ NP. Daher P $=$ NP. $\square$

Gibt es ein NP-vollständiges Problem?

Allgemeine Überlegung:

- Das ist nicht von vorne herein klar.
- Es könnte z.B. mehrere schwerste NP-Probleme geben, die nicht jeweils auf einander reduzierbar sind.

Theorem: SAT ist NP-vollständig. [Cook 1971, Levin 1973]

Cook 1971, Levin 1973

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of finding a certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will

ПРОБЛЕМЫ ПЕРЕДАЧИ ИНФОРМАЦИИ

Том IX *1973* *Вып. 3*

КРАТКИЕ СООБЩЕНИЯ

УДК 519.14

УНИВЕРСАЛЬНЫЕ ЗАДАЧИ ПЕРЕБОРА

Л. А. Левин

В статье рассматривается несколько известных массовых задач «переборного типа» и доказывается, что эти задачи можно решать лишь за такое время, за которое можно решать вообще любые задачи указанного типа.

NP-Vollständigkeit nachweisen

Anmerkung: Sobald wir ein erstes Problem als NP-vollständig nachgewiesen haben, fallen die anderen wie Dominosteine.

Rezept um die NP-Vollständigkeit eines Problems Y nachzuweisen:

- Schritt 1. Zeige dass Y in NP ist.
- Schritt 2. Wähle ein NP-vollständiges Problem X.
- Schritt 3. Beweise, dass $X \leq_p Y$.

Rechtfertigung: Wenn X ein NP-vollständiges Problem ist und Y ein Problem in NP mit der Eigenschaft $X \leq_p Y$, dann ist Y NP-vollständig.

NP-Vollständigkeit nachweisen

Rechtfertigung: Wenn X ein NP-vollständiges Problem ist und Y ein Problem in NP mit der Eigenschaft $X \leq_p Y$, dann ist Y NP-vollständig.

Beweis: Sei W ein beliebiges Problem in NP.
Dann gilt $W \leq_p X \leq_p Y$.

□ *durch Definition von NP-vollständig* □ *durch Annahme*

- Durch Transitivität, $W \leq_p Y$.
- Daher ist Y NP-vollständig. □

[Karp 1972] Karp hat mit einem einflussreichen Artikel wesentlich zur Verbreitung der Theorie der NP-Vollständigkeit beigetragen. Er hat 1985 dafür den Turingpreis erhalten.

Karp 1972

REDUCIBILITY AMONG COMBINATORIAL PROBLEMS†

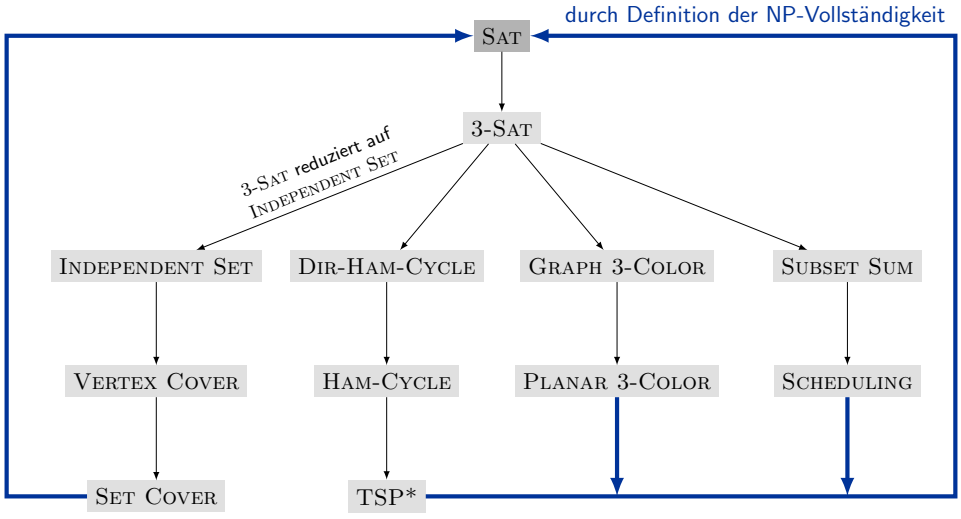
Richard M. Karp

University of California at Berkeley

Abstract: A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.

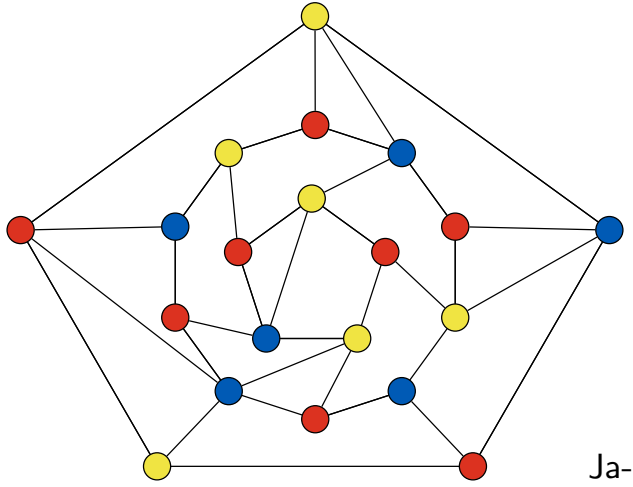
NP-Vollständigkeit

Beobachtung: Alle Probleme in der Abbildung sind NP-vollständig und lassen sich polynomiell auf einander reduzieren.



Beispiel: 3-Color (3-Knotenfärbung)

3-Color: Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit den Farben Rot, Gelb und Blau so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?



Ja-Instanz

Anwendung: Register-Allocation (Registerzuteilung)

Registerzuteilung: Ist die Zuteilung von Registern zu Programmvariablen. Es dürfen nicht mehr als k Register benutzt werden und zwei Programmvariablen, die zur gleichen Zeit benötigt werden, werden nicht dem gleichen Register zugewiesen.

Register-Interferenz-Graph:

- Knoten sind Variablen in einem Programm.
- Es existiert eine Kante zwischen u und v, wenn es eine Operation gibt, bei der u und v zur gleichen Zeit benötigt werden.

Beobachtung: [Chaitin 1982] Das Problem der Registerzuteilung kann genau dann gelöst werden, wenn der Register-Interferenz-Graph k-färbbar ist.

Fakt: 3-Color $\leq_P$ k-Register-Allocation für eine beliebige Konstante $k \geq 3$.

3-Color

Behauptung: 3-Sat $\leq_P$ 3-Color.

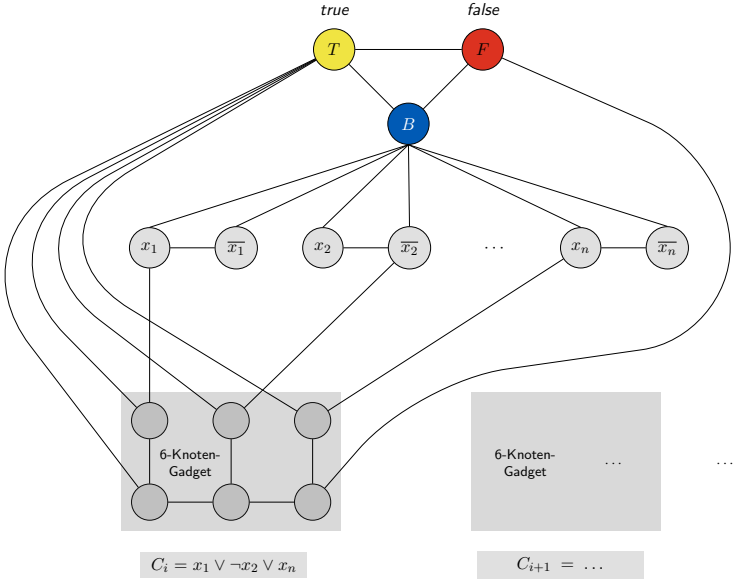
Beweis: Gegeben sei die 3-Sat-Instanz Φ. Wir konstruieren eine Instanz von 3-Color, die genau dann 3-färbbar ist wenn Φ erfüllbar ist.

Konstruktion:

- Für jedes Literal wird ein Knoten erzeugt.
- Erzeuge drei neue Knoten T, F, B. Verbinde diese Knoten zu einem Dreieck und verbinde jedes Literal mit B.
- Verbinde jedes Literal mit seiner Negation.
- Für jede Klausel wird ein Gadget mit 6 Knoten hinzugefügt.
- Jedes Gadget wird mit den Knoten, die den Literalen der entsprechenden Klausel entsprechen, und den Knoten T, F, B verbunden.

  □ *Siehe Abbildungen auf den nächsten Folien.*

3-Color (Visualisierung der Konstruktion)



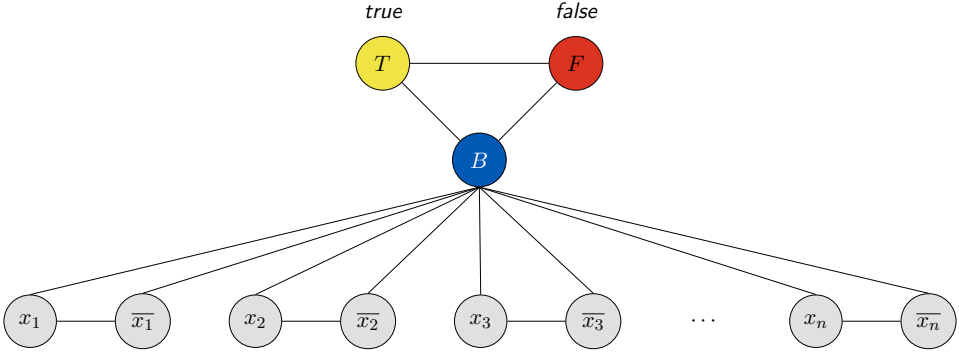
- *Nächste Folien: Beweis fokusiert sich auf Teile der Konstruktion*

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.

Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

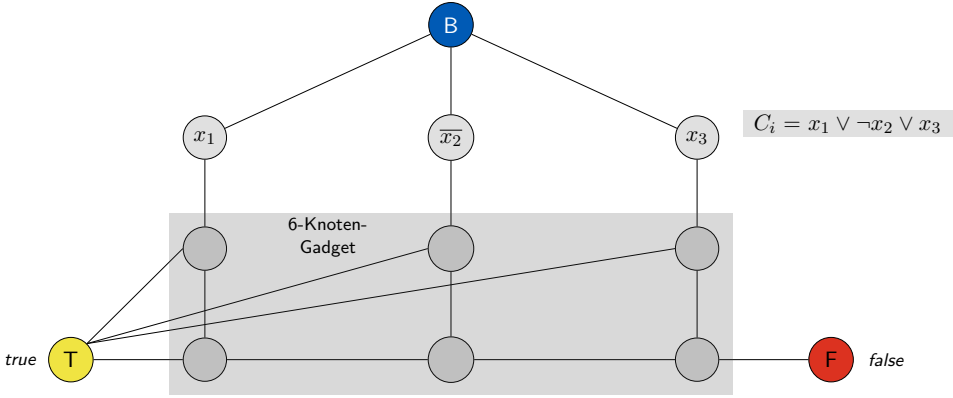
- Betrachte eine 3-Färbung. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass sie den Knoten T gelb, den Knoten F rot, und den Knoten B blau färbt.
- Daher ist jeder Literalknoten rot oder gelb gefärbt.
- Das definiert eine Wahrheitsbelegung f, die die gelben Literale auf *true*, und die roten Literale auf *false* setzt.



3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.
Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

- Behauptung: Zumindest ein Literal in jeder Klausel wird von f auf *true* gesetzt.

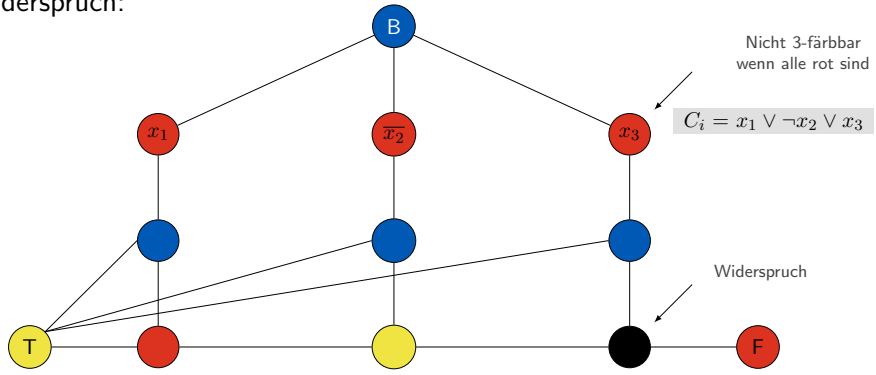


- T, F, B *Knoten verschoben und Kanten nicht eingezeichnet.*

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.
Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

- Die Annahme, alle Literale einer Klausel sind auf *false* gesetzt, ergibt einen Widerspruch:



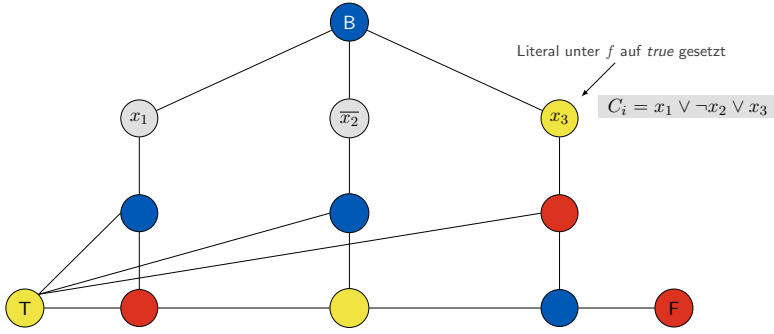
- Daher: Φ ist erfüllbar, und die Richtung $\Rightarrow$ ist gezeigt.

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.

Beweis: ($\Leftarrow$) Angenommen die Formel Φ ist erfüllbar.

- Betrachte eine erfüllende Wahrheitsbelegung f.
- Färbe Knoten T gelb, F rot und B blau.
- Färbe Literal gelb, wenn es unter f *true*, ansonsten rot.
- Es ist nicht schwer zu sehen, dass die Färbung auf alle Knoten in den Gadgets erweitert werden kann.



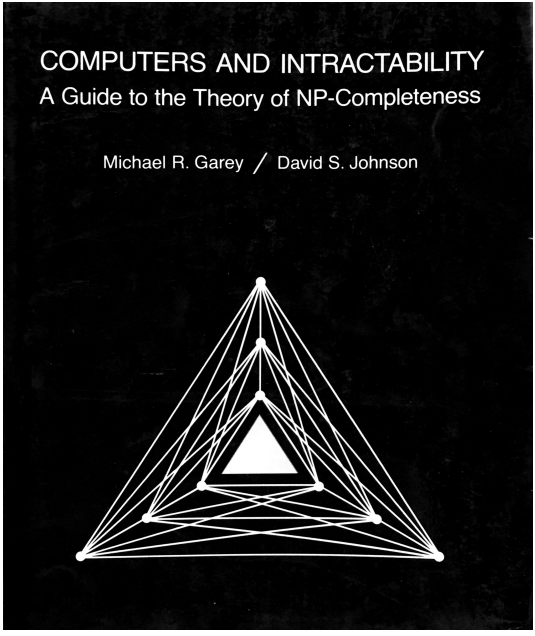
- Daher: der Graph is 3-färbbar, und die Richtung $\Leftarrow$ ist gezeigt.

Auswirkung der NP-Vollständigkeit

Auswirkung der NP-Vollständigkeit:

- Primärer intellektueller Beitrag der Informatik zu anderen Disziplinen.
- 6,000 Zitate pro Jahr (Titel, Abstract, Keywords).
  - Mehr als „Compiler“, „Betriebssysteme“, „Datenbanken“
- Breites Anwendungsspektrum und Klassifizierungsstärke.
- Schon Ende der 1970er Jahre waren hunderte Probleme als NP-vollständig erkannt
- Einflussreiches Buch von Garey und Johnson [1979]

Garey und Johnson 1979



Anwendung auf Optimierungsprobleme

NP-Vollständigkeit ist nur für Ja/Nein-Probleme definiert.

NP-Schwere kann aber in folgender Weise auch auf funktionale Probleme und Optimierungsprobleme angewandt werden.

Beispiel. Betrachte folgendes Problem $\textsc{OptVC}$: Gegeben ist ein Graph G, berechne ein kleinstes Vertex Cover von G.

Theorem: Angenommen P $\neq$ NP. Dann gibt es keinen Polynomialzeitalgorithmus für $\textsc{OptVC}$.

Anwendung auf Optimierungsprobleme

Theorem: Angenommen P $\neq$ NP. Dann gibt es keinen Polynomialzeitalgorithmus für OptVC.

Beweis: (durch Widerspruch)

- Angenommen es gäbe einen Polynomialzeitalgorithmus A für OptVC.
- Wir verwenden A, um das Ja/Nein-Problem Vertex-Cover in Polynomialzeit zu lösen:
- Auf eine gegebene Instanz (G, k) von Vertex-Cover wenden wir A an, und berechnen in Polynomialzeit ein kleinstes Vertex Cover C von G.
- Falls $|C| \leq k$ antworten wir Ja, ansonsten Nein.
- Da Vertex-Cover NP-vollständig ist, folgt P = NP, ein Widerspruch zur Annahme. $\square$

Terminologie

Es werden oft auch Optmierungsprobleme und funktionale Probleme als „NP-schwer“ bezeichnet, wenn aus ihrere Lösbarkeit in Polynomialzeit $\mathrm{P} = \mathrm{NP}$ folgt.

Wir können also beispielsweise sagen, dass $\textsc{OptVC}$ NP-schwer ist (wir sagen aber nicht, es sei NP-vollständig, da nicht in NP).

NP-Vollständigkeit meistern

Frage: Angenommen, wir müssen ein NP-vollständiges Problem lösen. Wie sollen wir vorgehen?
Antwort: Die Theorie besagt, dass es unwahrscheinlich ist, einen polynomiellen Algorithmus zu finden.

Man muss eine der gewünschten Eigenschaften aufgeben:

- Löse Problem optimal $\rightarrow$ Approximationsalgorithmen, Heuristische Algorithmen.
- Löse Problem in Polynomialzeit $\rightarrow$ Algorithmen mit exponentieller Laufzeit.
- Löse beliebige Instanzen des Problems $\rightarrow$ Identifiziere effizient lösbare Spezialfälle.