

# 9. Polynominalzeitreduktionen

## Einleitung

### Effizient lösbare Probleme (Wiederholung)

- **Definition:** Ein Problem gilt als **effizient lösbar**, wenn es durch einen Algorithmus gelöst werden kann, dessen Laufzeit durch ein **Polynom** der Eingabegröße beschränkt ist.
  - Laufzeit:  $O(n^c)$ 
    - $n$ : Eingabegröße (z.B. Anzahl der Bits)
    - $c$ : Konstanter Exponent
- **Synonym:** Effizient lösbare Probleme werden auch als **handhabbar (tractable)** bezeichnet.
- **Cobham-Edmonds-Annahme:**
  - Vorschlag von Alan Cobham und Jack Edmonds in den 1960er-Jahren.
  - **Gleichsetzung von Handhabbarkeit mit Lösbarkeit in Polynomialzeit.**
  - Hat die Informatikforschung der letzten 50 Jahre maßgeblich beeinflusst und sich weitgehend durchgesetzt.

### Diskussion zur Cobham-Edmonds-Annahme

- **Rechtfertigung der Annahme:**
  - **Praxisbezug:** Polynomiale Algorithmen weisen in der Regel kleine Konstanten und niedrige Exponenten auf.
  - **Strukturelle Einsicht:** Der Übergang von exponentiellen (z.B. Brute-Force) zu polynomialem Algorithmen deutet oft auf das Erkennen einer fundamentalen Struktur des Problems hin.
- **Ausnahmen/Kritik:**
  - **Ineffiziente polynomiale Algorithmen:** Es existieren polynomiale Algorithmen mit sehr großen Konstanten oder hohen Exponenten, die in der praktischen Anwendung unbrauchbar sein können.
  - **Praktische Relevanz exponentieller Algorithmen:** Algorithmen mit exponentieller oder noch schlechterer Laufzeit finden dennoch Anwendung, wenn:
    - Worst-Case-Eingaben extrem selten auftreten.
    - Die Größe der zu lösenden Problemfälle ausreichend klein ist.

# Probleme klassifizieren: P or not P?

## Ziel der Klassifizierung von Problemen

- Unterscheidung zwischen Problemen, die **in Polynomialzeit lösbar** sind, und solchen, die **nicht in Polynomialzeit lösbar** sind (NP-Probleme)

☰ Beispiele für Probleme, die nachweislich mehr als polynomiale Zeit erfordern:

- Halteproblem mit Schranke:** Hält eine gegebene Turingmaschine nach höchstens  $k$  Schritten?
- Verallgemeinertes Schachgewinnproblem:** Gegeben sei eine Brettbelegung für eine  $n \times n$  Generalisierung von Schach. Kann Schwarz garantiert gewinnen?

☰ Probleme, deren Klassifizierung (P oder NP?) unbekannt ist

- Maximum Independent Set:** Gegeben ein Graph  $G$  und eine Zahl  $k$ , enthält  $G$  mindestens  $k$  Knoten, die paarweise nicht adjazent sind?
- 3-Färbbarkeit:** Lassen sich die Knoten eines gegebenen Graphen mit 3 Farben färben, sodass Paare adjazenter Knoten unterschiedliche Farben haben?
- SAT (Erfüllbarkeitsproblem der Aussagenlogik):** Ist eine gegebene aussagenlogische Formel erfüllbar?

**Anmerkung:** Für viele fundamentale Probleme konnte noch keine eindeutige Klassifizierung (polynomial oder exponentiell) gefunden werden. Dies ist ein unbefriedigender Zustand in der theoretischen Informatik.

## Umgang mit Problemen, die nicht in Polynomialzeit lösbar sind:

- Wie gehen wir damit um, wenn wir ein Problem nicht in Polynomialzeit lösen können?
- Man soll nicht sagen, dass man zu blöd ist den Algo zu finden, sondern soll sagen, dass am selbst + die ganzen anderen Personen nichts gefunden haben.

## Ja/Nein Probleme

siehe hier

- Vereinfachung:** Zur einfacheren Betrachtung konzentrieren wir uns auf **Ja/Nein-Probleme (decision problems)**.
- Definition Ja/Nein-Problem:** Ein Problem, dessen Lösung entweder **Ja** oder **Nein** ist.
- Unterscheidung zu anderen Problemtypen:**
  - Funktionales Problem:** Liefert eine **Lösung** oder eine **Lösungsmenge** als Antwort.

- **Optimierungsproblem:** Ziel ist es, eine **optimale Lösung** (z.B. Minimum oder Maximum) zu finden.

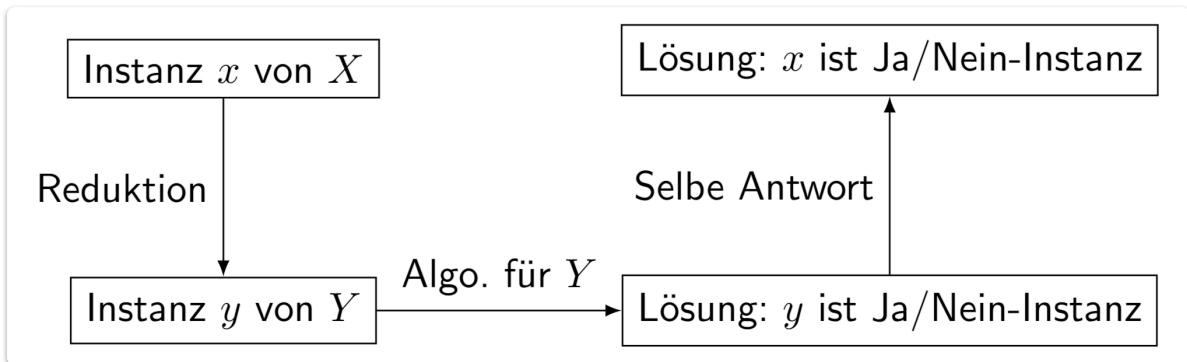
### ☰ Beispiel und Unterscheidung:

- **Ja/Nein-Problem:** Gibt es für einen gewichteten Graphen einen Spannbaum mit Kosten  $\leq k$ ? (Antwort: Ja oder Nein)
- **Funktionales Problem:** Finde in einem gewichteten Graphen einen Spannbaum mit Kosten  $\leq k$ . (Antwort: Der Spannbaum selbst, falls er existiert)
- **Optimierungsproblem:** Finde in einem gewichteten Graphen einen Spannbaum mit minimalen Kosten (MST). (Antwort: Der Spannbaum mit den geringsten Gesamtkosten)

# Polynominalzeitreduktionen

Wir haben 2 Probleme und wollen das Problem  $X$  auf das Problem  $Y$  polynomiell reduzieren.

- **Informelle Beschreibung:** Wenn ein Problem  $X$  **polynomialzeitreduzierbar** auf ein Problem  $Y$  ist, bedeutet das:
  - Falls wir einen **effizienten Algorithmus** (Polynomialzeit) für  $Y$  haben,
  - dann können wir auch  $X$  **effizient lösen**.
- **Vorgehensweise der Reduktion:**
  1. Gegeben eine **Instanz  $x$  von Problem  $X$** .
  2. Transformiere  $x$  in eine **Instanz  $y$  von Problem  $Y$**  in Polynomialzeit.
  3. Löse die Instanz  $y$  von  $Y$  mit dem effizienten Algorithmus für  $Y$ .
  4. **Schlussfolgerung:**
    - Wenn  $y$  eine **Ja-Instanz** von  $Y$  ist, dann ist auch  $x$  eine **Ja-Instanz** von  $X$ .
    - Wenn  $y$  eine **Nein-Instanz** von  $Y$  ist, dann ist auch  $x$  eine **Nein-Instanz** von  $X$ .



## ⌚ Definition

siehe

Eine **Polynominalzeitreduktion** von Problem  $X$  auf Problem  $Y$  ist ein Algorithmus  $R$ , der für jede Instanz  $x$  von  $X$  eine Instanz  $y$  von  $Y$  berechnet, so dass die folgenden zwei Bedingungen erfüllt sind:

1. **Äquivalenz der Ja-Instanzen:**  $x$  ist eine Ja-Instanz von  $X$  genau dann, wenn  $y$  eine Ja-Instanz von  $Y$  ist.
2. **Effizienz der Reduktion:** Der Algorithmus  $R$  hat eine **Laufzeit in Polynomialzeit**. Das bedeutet, es existiert eine Konstante  $c$ , sodass  $R$  die Instanz  $y$  in einer Zeit von  $O(n^c)$  berechnet, wobei  $n$  die Eingabegröße der Instanz  $x$  ist.

## Notation

Wir schreiben  $X \leq_P Y$ , um auszudrücken, dass es eine Polynominalzeitreduktion von  $X$  auf  $Y$  gibt. In diesem Fall sagen wir auch: " **$X$  ist auf  $Y$  polynomiell reduzierbar**".

## Hinweis zur Interpretation

Falls  $X \leq_P Y$  gilt, kann man auch sagen, dass "Y mindestens so schwer ist wie X"

## Lösung von Problemen durch Reduktion

siehe

- **Idee:** Nutze eine Polynominalzeitreduktion auf ein bereits als handhabbar bekanntes Problem.
- **Grundsatz:** Wenn  $X \leq_P Y$  und Y in Polynomialzeit lösbar ist, dann ist auch X in Polynomialzeit lösbar.
- **Beweis:**
  - Sei  $R$  der Reduktionsalgorithmus von X nach Y mit Laufzeit  $O(n^a)$  für eine Instanz  $x$  von X der Größe  $n$  ( $a \geq 1$ ).
  - Sei  $A$  der Algorithmus zum Lösen von Y mit Laufzeit  $O(n^b)$  für eine Instanz  $y$  von Y der Größe  $n$  ( $b \geq 1$ ).
  - Gegeben eine Instanz  $x$  von X der Größe  $n$ :
    1. Anwenden von  $R$  auf  $x$  erzeugt eine Instanz  $y$  von Y in  $O(n^a)$  Zeit.
    2. Die Größe von  $y$  ist höchstens  $O(n^a)$ , da sie in dieser Zeit erzeugt wurde.
    3. Lösen von  $y$  mit  $A$  benötigt  $O((n^a)^b) = O(n^{ab})$  Zeit.
  - Die Gesamtlaufzeit zur Lösung von  $x$  ist  $O(n^a) + O(n^{ab}) = O(n^{ab})$ , was ein Polynom in  $n$  ist.  $\square$

## Nachweis der Nicht-Handhabbarkeit durch Reduktion

siehe

- **Idee:** Reduziere ein bereits als nicht handhabbar bekanntes Problem auf das zu untersuchende Problem.
- **Grundsatz:** Wenn  $X \leq_P Y$  und X nicht in Polynomialzeit lösbar ist, dann kann auch Y nicht in Polynomialzeit lösbar sein.
- **Beweis (durch Widerspruch):**
  - Angenommen, Y wäre in Polynomialzeit lösbar.
  - Da  $X \leq_P Y$ , existiert eine Polynominalzeitreduktion von X auf Y.
  - Durch die Kombination der Polynominalzeitreduktion und des Polynomialzeitalgorithmus für Y könnte X ebenfalls in Polynomialzeit gelöst werden.
  - Dies widerspricht der Annahme, dass X nicht in Polynomialzeit lösbar ist.

## ⌚ Definition - Polynomialzeit-Äquivalenz

Wenn  $X \leq_P Y$  und  $Y \leq_P X$ , dann schreiben wir  $X \equiv_P Y$ . Dies bedeutet, dass X und Y bezüglich ihrer Schwierigkeit in Polynomialzeit äquivalent sind.

### ⓘ Satz - Transitivität der PZR

siehe

- Ist  $X \leq_P Y$  und  $Y \leq_P Z$ , dann folgt daraus  $X \leq_P Z$ .
- **Beweis:**
  - Sei  $R_1$  der Reduktionsalgorithmus von X nach Y mit Laufzeit  $O(n^a)$  ( $a \geq 1$ ).
  - Sei  $R_2$  der Reduktionsalgorithmus von Y nach Z mit Laufzeit  $O(n^b)$  ( $b \geq 1$ ).
  - Sei  $x$  eine Instanz von X der Größe  $n$ .
  - $R_1(x)$  erzeugt eine Instanz  $x'$  von Y in höchstens  $O(n^a)$  Zeit, wobei die Größe von  $x'$   $O(n^a)$  ist.
  - $R_2(x')$  erzeugt eine Instanz  $z$  von Z in  $O((n^a)^b) = O(n^{ab})$  Zeit.
  - Somit existiert eine Polynomialzeitreduktion (die Komposition von  $R_1$  und  $R_2$ ) von X nach Z mit einer Laufzeit von  $O(n^a) + O(n^{ab}) = O(n^{ab})$ , was polynomiell in  $n$  ist.

### ⓘ Angeben von Polynomialzeitreduktion

siehe

Beim Definieren einer Polynomialzeitreduktion von einem Problem X auf ein Problem Y müssen zwei zentrale Eigenschaften nachgewiesen werden:

#### 1. Korrektheit der Reduktion:

- Ja-Instanzen von X müssen auf Ja-Instanzen von Y abgebildet werden.
- Nein-Instanzen von X müssen auf Nein-Instanzen von Y abgebildet werden.

#### 2. Polynomialität der Reduktion:

- Der Reduktionsalgorithmus muss in Polynomialzeit ausführbar sein.

Die Schwierigkeit des Nachweises kann variieren: In manchen Fällen ist die Korrektheit offensichtlich, während in anderen die Polynomialität leichter zu zeigen ist.

# Independent set und Vertex Cover

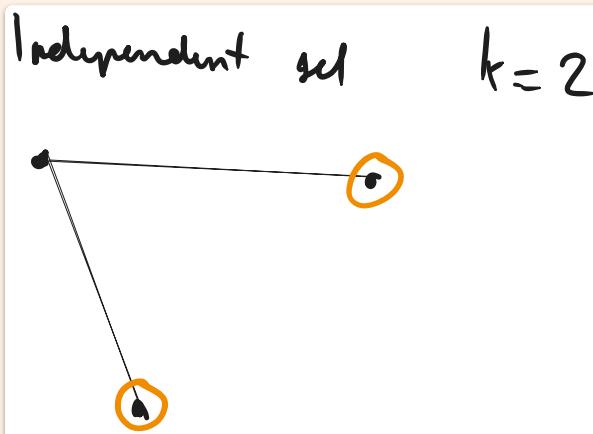
## Independent set

### ⌚ Definition

- Ein *Independent Set* (oder auch *unabhängige Menge*) eines Graphen  $G = (V, E)$  ist eine Teilmenge  $S \subseteq V$  der Knoten, in der es **keine zwei adjazenten Knoten** gibt.
- **Beispiel:** Wenn Knoten A und B durch eine Kante verbunden sind, können nicht beide in einem Independent Set sein.

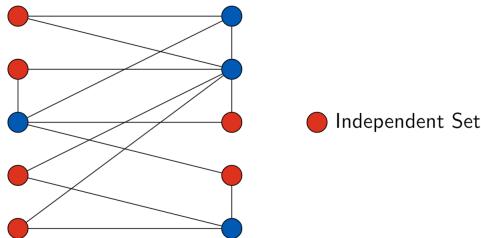
### ⌚ Problemstellung

- **Gegeben:** Ein Graph  $G = (V, E)$  und eine ganze Zahl  $k$ .
- **Frage:** Gibt es ein Independent Set  $S$ , sodass  $|S| \geq k$  gilt?
  - **Wichtig:** Die Zahl  $k$  ist Teil der Eingabe und keine Konstante. Sie variiert also von Problem zu Problem.
- **Hinweis:** Ja/Nein-Probleme werden ab jetzt mit dieser Schreibweise (z.B. **INDEPENDENT SET**) gekennzeichnet.



### ☰ Weiteres Beispiel

**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  $\geq 6$ ? Ja.

Beispiel: Existiert ein Independent Set der Größe  $\geq 7$ ? Nein.

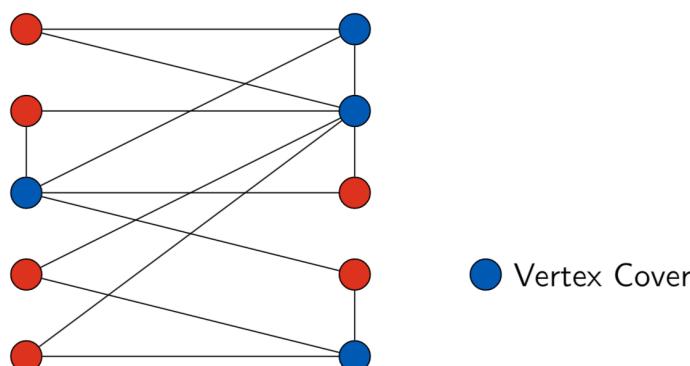
## Vertex Cover

### ⌚ Definition

- Ein *Vertex Cover* (oder auch *Knotenüberdeckung*) eines Graphen  $G = (V, E)$  ist eine Menge  $S \subseteq V$  von Knoten, sodass jede Kante des Graphen zu mindestens einem Knoten aus  $S$  inzidiert ist.
  - Anders ausgedrückt: Wenn man alle Knoten in  $S$  markiert, muss jede Kante im Graphen mindestens einen markierten Endpunkt haben.

### ⌚ Problemstellung

- Gegeben:** Ein Graph  $G = (V, E)$  und eine ganze Zahl  $k$ .
- Frage:** Gibt es ein Vertex Cover  $S$  von  $G$ , sodass  $|S| \leq k$  gilt?



- Beispiel für die Abbildung:**

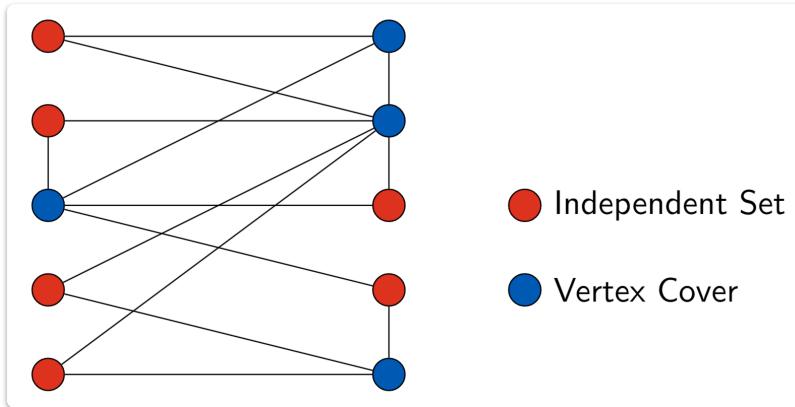
- Existiert ein Vertex Cover der Größe  $\leq 4$ ? Ja.
- Existiert ein Vertex Cover der Größe  $\leq 3$ ? Nein.

## Konversionslemma: Vertex Cover und Independent Set

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

Sei  $G = (V, E)$  ein ungerichteter Graph mit Knotenmenge  $V$  und Kantenmenge  $E$ . Sei  $S \subseteq V$  eine Teilmenge der Knoten und  $C = V - S$  das Komplement von  $S$  in  $V$ . Dann gilt:

$S$  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.  $\square$

$\Leftarrow:$

- 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:  $\text{VERTEX COVER} \equiv_P \text{INDEPENDENT SET}$ .

Dies bedeutet, dass die Probleme **Vertex Cover** und **Independent Set** äquivalent sind. Das

bedeutet, wenn man ein effizientes Verfahren (z.B. eine polynomielle Zeitlösung) für das eine Problem hat, kann man es auch für das andere Problem anwenden.

### ✓ Beweis der Äquivalenz

Um die Äquivalenz zu zeigen, müssen wir beide Richtungen beweisen:

**Beweisidee:** Wir zeigen, dass eine Instanz vom einen ins andere umgewandelt werden kann, sodass die Lösung vom umgewandelten die Lösung vom vorigen zeigt.

## 1. VERTEX COVER $\leq_P$ INDEPENDENT SET

- Sei  $(G, k)$  eine Instanz von VERTEX COVER.
  - $G$  ist ein Graph.
  - $k$  ist die gewünschte Größe des Vertex Covers.
- Sei  $n$  die Anzahl der Knoten von  $G$ .
- In Polynomialzeit generieren wir  $(G, n - k)$  als Instanz von INDEPENDENT SET.
  - Das bedeutet, wir suchen ein Independent Set der Größe  $n - k$  im selben Graphen  $G$ .
- **Die Reduktion ist korrekt:**
  - Ein Graph  $G$  hat genau dann ein  $G$  Vertex Cover der Größe  $\leq k$ , wenn  $G$  ein Independent Set der Größe  $\geq n - k$  hat.
  - (Dies folgt aus dem Konversionslemma, welches die Beziehung zwischen Vertex Cover und Independent Set herstellt.)
- **Die Reduktion ist klarerweise polynomiell:**
  - Sie ersetzt lediglich  $k$  durch  $n - k$ , was in konstanter Zeit geschieht.

## 2. INDEPENDENT SET $\leq_P$ VERTEX COVER

- Sei  $(G, k)$  eine Instanz von INDEPENDENT SET.
  - $G$  ist ein Graph.
  - $k$  ist die gewünschte Größe des Independent Sets.
- Sei  $n$  die Anzahl der Knoten von  $G$ .
- In Polynomialzeit generieren wir  $(G, n - k)$  als Instanz von VERTEX COVER.
  - Das bedeutet, wir suchen ein Vertex Cover der Größe  $n - k$  im selben Graphen  $G$ .
- **Die Reduktion ist korrekt:**
  - Ein Graph  $G$  hat genau dann ein  $G$  Independent Set der Größe  $\geq k$ , wenn  $G$  ein Vertex Cover der Größe  $\leq n - k$  hat.
  - (Auch dies folgt aus dem Konversionslemma.)
- **Die Reduktion ist klarerweise polynomiell:**

- Auch hier wird lediglich  $k$  durch  $n - k$  ersetzt.

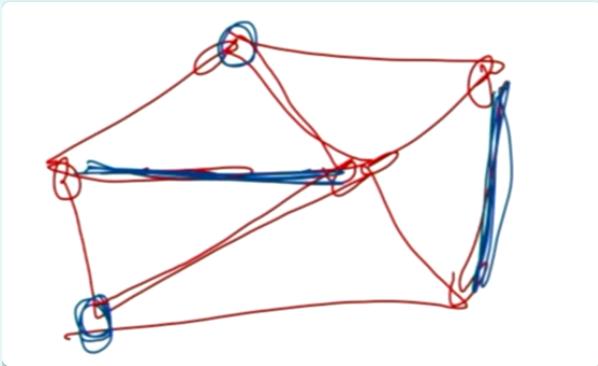
Wir haben also beide Richtungen gezeigt, und es folgt die **Äquivalenz** zwischen Vertex Cover und Independent Set.

# Beispiel: Spannbäume und Nicht-Blockierer

## ⌚ Definition - Nicht-Blockierer

- Gegeben ist ein Graph  $G = (V, E)$  mit reellwertigen Kantengewichten  $c_e = c_{uv}$  für  $e = (u, v) \in E$ .
- Ein **Nicht-Blockierer** ( $N$ ) 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. (Ein Nicht-Blockierer "blockiert" also keinen Pfad, d.h., es gibt immer einen Pfad, der die Kanten in  $N$  meidet.)
- **Kosten eines Nicht-Blockierers:** Die Kosten eines Nicht-Blockierers sind gegeben durch  $\sum_{e \in N} c_e$ .
- Ein **maximaler Nicht-Blockierer** ist ein Nicht-Blockierer mit größten Kosten.

Wenn man jeden Knoten von jedem Knoten immer noch erreichen kann auch wenn man eine Kante weglassen würde. Beispielsweise bei Straßennetz und Baustelle:



## ❓ Problem MNB (Maximaler Nicht-Blockierer)

- **Definition:** Gegeben ist ein gewichteter Graph  $G$  und eine Zahl  $k$ . Besitzt  $G$  einen Nicht-Blockierer mit Kosten  $\geq k$ ?
- **Frage:** Ist das Problem MNB in Polynomialzeit lösbar?

## Reduktion auf Spannbäume

Wir zeigen, dass MNB in Polynomialzeit lösbar ist, indem wir es auf ein bekanntes Problem (MST\*) reduzieren.

## ⌚ Definition Spannbaum

- **Definition:** Gegeben ist ein Graph  $G = (V, E)$  mit reellwertigen Kantengewichten  $c_e = c_{uv}$  für  $e = (u, v) \in E$ .

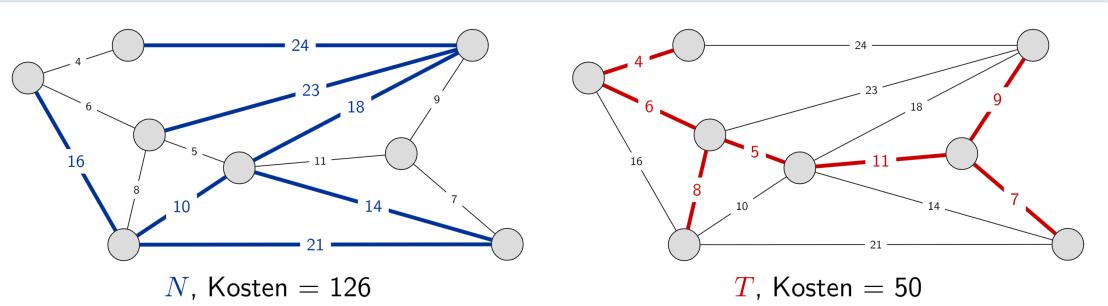
- Ein **Spannbaum** ( $T$ ) ist eine Teilmenge der Kanten  $T \subseteq E$ , sodass  $G_T = (V, T)$  ein Baum ist. (Ein Baum ist ein zusammenhängender, zyklenfreier Graph.)
- **Kosten eines Spannbaums:** Die Kosten des Baumes sind gegeben durch  $\sum_{e \in T} c_e$ .
- Ein **minimaler Spannbaum** ist ein Spannbaum mit kleinsten Kosten.

### ② Problem MST\* (Minimaler Spannbaum mit Kosten)

- **Definition:** Gegeben ist ein gewichteter Graph  $G$  und eine Zahl  $k$ . Besitzt  $G$  einen Spannbaum mit Kosten  $\leq k$ ?
- **Lösbarkeit:** Wir wissen aus dem Kapitel „Greedy-Algorithmen“, dass  $MST^*$  in Polynomialzeit gelöst werden kann (z.B. mit dem Kruskal- oder Prim-Algorithmus).

### ① Konversionslemma

- Sei  $G = (V, E)$  ein gewichteter Graph.
- Sei  $N \subseteq E$  ein Nicht-Blockierer und  $T = E - N$  (d.h.,  $T$  sind alle Kanten, die *nicht* im Nicht-Blockierer sind).
- **Aussage:**  $N$  ist ein maximaler Nicht-Blockierer genau dann, wenn  $T$  ein minimaler Spannbaum ist.
- **Kostenbeziehung:** Die Kosten von  $N$  sind genau  $K' := \sum_{e \in E} c_e$  minus den Kosten von  $T$ .
  - $\text{Kosten}(N) = \sum_{e \in E} c_e - \text{Kosten}(T)$



## Äquivalenz von MNB und MST\*

Es gilt:  $MNB \equiv_P MST^*$  (MNB ist polynomiell äquivalent zu MST\*).

### ✓ Beweis der Äquivalenz

1.  $MNB \leq_P MST^*$ 
  - Wir reduzieren eine Instanz  $(G, k)$  von MNB auf die Instanz  $(G, K_{\text{gesamt}} - k)$  von  $MST^*$ .
    - $K_{\text{gesamt}}$  ist die Summe der Gewichte aller Kanten in  $G$ .

- Wenn wir einen Nicht-Blockierer mit Kosten  $\geq k$  suchen, entspricht das dem Finden eines Spannbaums mit Kosten  $\leq K_{gesamt} - k$ .
- **Intuition:** Ein Nicht-Blockierer mit hohen Kosten bedeutet, dass die *verbleibenden* Kanten (die nicht im Nicht-Blockierer sind und den Spannbaum bilden) geringe Kosten haben müssen.

## 2. $MST^* \leq_P MNB$

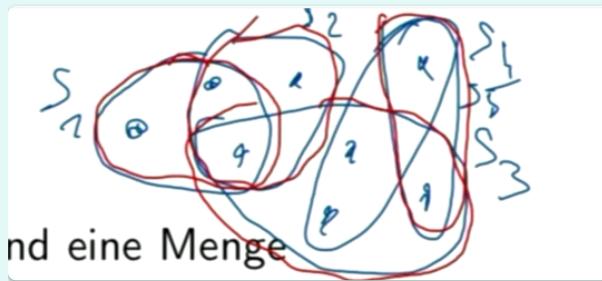
- Wir reduzieren eine Instanz  $(G, k)$  von  $MST^*$  auf die Instanz  $(G, K_{gesamt} - k)$  von MNB.
  - Wenn wir einen Spannbaum mit Kosten  $\leq k$  suchen, entspricht das dem Finden eines Nicht-Blockierers mit Kosten  $\geq K_{gesamt} - k$ .
- **Intuition:** Ein Spannbaum mit geringen Kosten bedeutet, dass die Kanten, die *nicht* im Spannbaum sind (und den Nicht-Blockierer bilden), hohe Kosten haben müssen.

Es folgt daher: **MNB ist in Polynomialzeit lösbar**, da es auf  $MST^*$  reduziert werden kann, welches in Polynomialzeit lösbar ist.

# Set Cover (Mengenüberdeckungsproblem)

## ⌚ Definition des Set Cover

- Gegeben sei:
  - Eine Menge  $U$  von Elementen (auch "Universum" genannt).
  - Eine Menge  $S = \{S_1, S_2, \dots, S_m\}$  von Teilmengen von  $U$ . Jede  $S_i$  ist also  $S_i \subseteq U$ .
- Ein **Set Cover** ( $C$ ) ist eine Teilmenge  $C \subseteq S$ .
  - Dies bedeutet,  $C$  ist eine Menge von Mengen aus  $S$ .
- Die Bedingung für ein Set Cover ist, dass die **Vereinigung aller Mengen in  $C$  dem Universum  $U$  entspricht**.
  - Formal:  $\bigcup_{X \in C} X = U$ .
- Man sagt auch:  $C$  ist ein Set Cover von  $S$ .



## ❓ Das Set Cover Problem (Entscheidungsproblem)

- **Gegeben:**
  - Eine Menge  $U$  von Elementen.
  - Eine Menge  $S = \{S_1, S_2, \dots, S_m\}$  von Teilmengen von  $U$ .
  - Eine ganze Zahl  $k$ .
- **Frage:** Existiert eine Teilmenge  $C \subseteq S$  mit  $|C| \leq k$ , sodass die Vereinigung von  $C$  gleich  $U$  ist?
  - Das Problem fragt also, ob es ein Set Cover gibt, das aus **höchstens  $k$**  der gegebenen Teilmengen besteht.

## ☰ Beispielhafte Anwendungen des Set Cover Problems

Das Set Cover Problem ist ein klassisches Problem in der Informatik und hat viele praktische Anwendungen.

- **Szenario:** Softwareentwicklung
  - Es gibt  $m$  verfügbare Softwarekomponenten.

- Menge  $U$  besteht aus  $n$  Eigenschaften, die unser Softwaresystem haben sollte. (z.B. "Datenbankzugriff", "Benutzeroauthentifizierung", "Reporting-Funktion").
  - Die  $i$ -te Softwarekomponente bietet eine Menge  $S_i \subseteq U$  von Eigenschaften an. (z.B. Komponente 1 bietet "Datenbankzugriff" und "Caching", Komponente 2 bietet "Benutzeroauthentifizierung" und "Verschlüsselung").
  - **Ziel:** Erreiche alle  $n$  Eigenschaften mit *maximal*  $k$  Komponenten.
  - Dies entspricht genau der Fragestellung des Set Cover Problems: Finde die kleinste Anzahl von Komponenten, deren kombinierte Eigenschaften alle gewünschten Eigenschaften abdecken.
- 

### Weiteres Beispiel:

Möglichst wenige Pizzen aus Speisekarten aussuchen, dass möglichst viele Zutaten drauf sind.

---

### Weiteres Beispiel:

$$\begin{aligned} U &= \{1, 2, 3, 4, 5, 6, 7\} \\ k &= 2 \\ S_1 &= \{3, 7\} & S_4 &= \{2, 4\} \\ S_2 &= \{3, 4, 5, 6\} & S_5 &= \{5\} \\ S_3 &= \{1\} & S_6 &= \{1, 2, 6, 7\} \end{aligned}$$

## Vertex Cover auf Set Cover reduzieren

**Behauptung:** VERTEX COVER  $\leq_P$  SET COVER.

**Beweis:** Gegeben sei eine Vertex Cover Instanz  $(G, k)$  mit  $G = (V, E)$ .

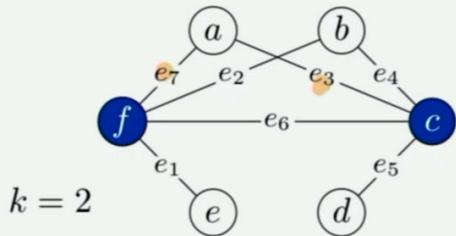
Wir konstruieren eine Instanz von SET COVER.

- $k = k$ ,
- $U = E$ ,
- 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*.

Wir wollen hier das **Vertex Cover als Set Cover darstellen**.

Für jeden Knoten nehmen wir alle Kanten die zu dem Knoten inzident sind. Beispiel Kante  $e_3$  und  $e_7$

### VERTEX COVER



### SET COVER

$$\begin{aligned} U &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \quad k = 2 \\ S_a &= \{e_3, e_7\}, \quad S_b = \{e_2, e_4\} \\ S_c &= \{e_3, e_4, e_5, e_6\}, \quad S_d = \{e_5\} \\ S_e &= \{e_1\}, \quad S_f = \{e_1, e_2, e_6, e_7\} \end{aligned}$$

und das machen wir jetzt für alle Knoten. Und das  $k$  übernehmen wir einfach und lassen wir gleich.

- **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  $C = \{v_1, \dots, v_k\} \subseteq V$  ein Vertex Cover von  $G$ . Dann ist  $\mathcal{C} = \{S_{v_1}, \dots, S_{v_k}\}$  ein Set Cover von  $\mathcal{S}$ .
- $\Leftarrow$ : Sei  $\mathcal{C} = \{S_{v_1}, \dots, S_{v_k}\}$  ein Set Cover von  $\mathcal{S}$ . Dann ist  $C = \{v_1, \dots, v_k\} \subseteq V$  ein Vertex Cover von  $G$ .  $\square$

### Bemerkung

Jede Setcover Instanz die wir durch Reduktion bekommen, hat bestimmte Eigenschaften.

Nicht jede Instanz wird diese Eigenschaften haben. Eine die wir bei unserem Beispiel herauslesen können ist, dass jede Kante in genau 2 Mengen vorkommt. Weil wir für jeden Knoten die Menge aller inzidenten Kanten finden und jede ist zu genau 2 inzident also wird das stimmen. Wenn wir aber im allgemeinen das machen, muss das nicht stimmen.

- 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

Mit Gadgets sind in diesem Fall diese Dreiecke gemeint und ist ein allgemeiner Begriff um kleine Bausteine zu beschreiben, die dann eine ganze Instanz implementieren. Aber es gibt keine genau Definition von Gadgets.

## Erfüllbarkeitsproblem (satisfiability)

siehe [GDS](#)

- **Literal:** Eine boolesche Variable ( $x_i$ ) oder ihre Negation ( $\neg x_i$ ).
  - Beispiele:  $x_i, \neg x_i$
- **Klausel:** Eine Disjunktion (logisches ODER,  $\vee$ ) von Literalen.
  - Beispiel:  $C_j = x_1 \vee x_2 \vee x_3$
- **Konjunktive Normalform (KNF):** Eine aussagenlogische Formel  $\Phi$ , bei der Klauseln konjunktiv (logisches UND,  $\wedge$ ) 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  $\Phi$ , falls jede Klausel von  $\Phi$  mindestens eine Variable  $x$  mit  $f(x) = \text{true}$  oder eine negierte Variable  $\neg x$  mit  $f(x) = \text{false}$  enthält.

### ⌚ Das Erfüllbarkeitsproblem (SAT)

- **SAT (satisfiability):** Gegeben ist eine KNF-Formel  $\Phi$ . Gibt es eine Wahrheitsbelegung, die  $\Phi$  erfüllt?
- **3-SAT:** SAT, bei dem jede Klausel genau 3 Literale enthält.
  - Jedes Literal muss sich auf eine unterschiedliche Variable beziehen.
  - Beispiel:  $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
  - **Erfüllende Wahrheitsbelegung:**  $f(x_1) = \text{true}, f(x_2) = \text{true}, f(x_3) = \text{false}$ .

## Bedeutung von SAT

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

1. **Mächtige heuristische Algorithmen (SAT-Solver):** Für SAT existieren leistungsstarke 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. **Nicht-Handhabbarkeit für allgemeine Instanzen:** Andererseits wird SAT für allgemeine Instanzen als nicht-handhabbar angesehen (SAT ist "NP-vollständig", was auf den

nächsten Folien erläutert wird).

- → Daher ist SAT ein beliebtes Problem, das auf andere Probleme reduziert wird, um deren Nicht-Handhabbarkeit zu zeigen.

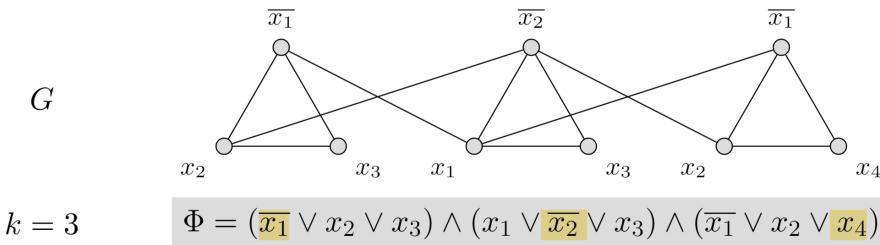
### 3-Sat auf Independent Set reduzieren

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

**Beweis:** Gegeben sei eine Instanz  $\Phi$  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.



- Als  $k$  definieren wir immer die Anzahl der Klauseln

**Korrektheit:**  $G$  enthält ein Independent Set der Größe  $k$  genau dann wenn  $\phi$  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  $x = \text{true}$  setzen, falls  $x \in S$ , und  $x = \text{false}$  setzen, falls  $\neg x \in S$ .
- Wegen der Kanten zwischen den Dreiecken kann es zu keinen widersprüchlichen Belegungen ein und derselben Variable kommen.
- Wir setzen die übrigen Variablen beliebig auf  $\text{true}$  oder  $\text{false}$ .
- Diese Wahrheitsbelegung erfüllt alle Klauseln.

( $\Leftarrow$ ) Gegeben sei eine Wahrheitsbelegung  $f$ , die  $\Phi$  erfüllt.

- Wir konstruieren ein Independent Set  $S$ , indem wir von jedem Dreieck einen Knoten  $l$  mit  $l = x$  und  $f(x) = \text{true}$  oder einen Knoten  $l$  mit  $l = \neg x$  und  $f(x) = \text{false}$  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*.

# 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.

Können wir die beiden Probleme jetzt aufeinander reduzieren?

Das Ja/Nein Problem kann sehr einfach auf das Optimierungsproblem reduzieren, weil man einfach den Algorithmus ausführen kann von dem OP und dann schauen ob  $k$  kleiner/größer ist, aber umgekehrt ist das schon bisschen schwerer.

## 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.

**Ziel:** Wir möchten mittels  $VC(G, k)$  ein kleinstes Vertex Cover finden.

**Verwendete Eigenschaften für jeden Graphen  $G = (V, E)$ :**

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.

## Der Algorithmus OptVC( $G$ )

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

```
OptVC( $G$ ):  

for  $k \leftarrow 0$  bis  $n - 1$   

    if VC( $G, k$ )  

        return FindVC( $G, k$ )
```

```
FindVC( $G, k$ ):  

if  $k = 0$   

    return  $\emptyset$   

else  

    foreach  $v \in V(G)$   

        if VC( $G - v, k - 1$ )  

            return  $\{v\} \cup$  FindVC( $G - v, k - 1$ )
```

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

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

# Definition von NP

## NP-Probleme

### ⌚ Definition eines NP-Problems

- Ein Ja/Nein-Problem (oder Entscheidungsproblem) ist ein **NP-Problem**, falls wir Ja-Instanzen mit Hilfe eines **Zertifikats** effizient überprüfen können.
  - "Effizient" bedeutet hier in Polynomialzeit.

### Zertifikat

- Ein **Zertifikat**  $t$  für eine Instanz  $x$  ist ein beliebiger Input.
- Die Größe  $m$  des Zertifikats  $t$  muss polynomiell in der Größe  $n$  von  $x$  beschränkt sein, das heißt  $m \leq p(n)$  für ein Polynom  $p$ .
  - Dies stellt sicher, dass das Zertifikat nicht übermäßig groß ist und somit effizient verarbeitet werden kann.

### Zertifizierer

- Ein **Zertifizierer**  $C(x, t)$  ist ein Polynomialzeitalgorithmus.
- Er überprüft, ob eine Ja-Instanz  $x$  mit Hilfe eines Zertifikats  $t$  gültig ist.
  - Der Zertifizierer nimmt die Instanz  $x$  und das potenzielle Zertifikat  $t$  als Eingabe und entscheidet, ob  $t$  ein gültiger "Beweis" dafür ist, dass  $x$  eine Ja-Instanz ist.

### ⓘ Anmerkung zur Notation

- **NP** steht für „nicht-deterministisch polynomielle“ Zeit.
  - Dies bezieht sich auf die ursprüngliche Definition, bei der ein nicht-deterministischer Turing-Automat das Problem in Polynomialzeit lösen kann. Die hier gegebene Definition über Zertifikate ist jedoch äquivalent und oft intuitiver.
- Für ein NP-Problem  $X$  sagen wir auch: „ $X$  ist in NP“.

### Eigenschaften des Zertifizierers $C(x, t)$

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

- **Für jede Ja-Instanz  $x$  gibt es ein Zertifikat  $t$**  (von polynomieller Länge), welches den Zertifizierer zum Akzeptieren bringt.

- Wenn eine Instanz tatsächlich eine "Ja"-Antwort hat, muss es einen "Beweis" (Zertifikat) geben, den der Zertifizierer als gültig erkennt.
- Man kann sich das so vorstellen: "Der Zertifizierer kann überzeugt werden."
- Für keine Nein-Instanz  $x$  gibt es ein Zertifikat  $t$ , welches den Zertifizierer zum Akzeptieren bringt.
- Wenn eine Instanz eine "Nein"-Antwort hat, darf es keinen "Beweis" (Zertifikat) geben, der den Zertifizierer fälschlicherweise dazu bringt, die Instanz als "Ja" zu akzeptieren.
- Man kann sich das so vorstellen: "Der Zertifizierer kann nicht ausgetrickst werden."

### ☰ Beispiel dazu: SAT und HAM-Cycle

**SAT:** Gegeben sei eine Formel  $\Phi$  in konjunktiver Normalform. Ist diese Formel erfüllbar?

**Zertifikat:** Eine Wahrheitsbelegung  $f$  für die  $n$  booleschen Variablen, die  $\Phi$  erfüllt.

**Zertifizierer:** Überprüfe, ob  $f$  die Formel  $\Phi$  erfüllt.

**Beispiel:**

$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4)$ ; Instanz  $s$

$f(x_1) = \text{true}$ ,  $f(x_2) = \text{true}$ ,  $f(x_3) = \text{false}$ ,  $f(x_4) = \text{true}$  Zertifikat  $t$

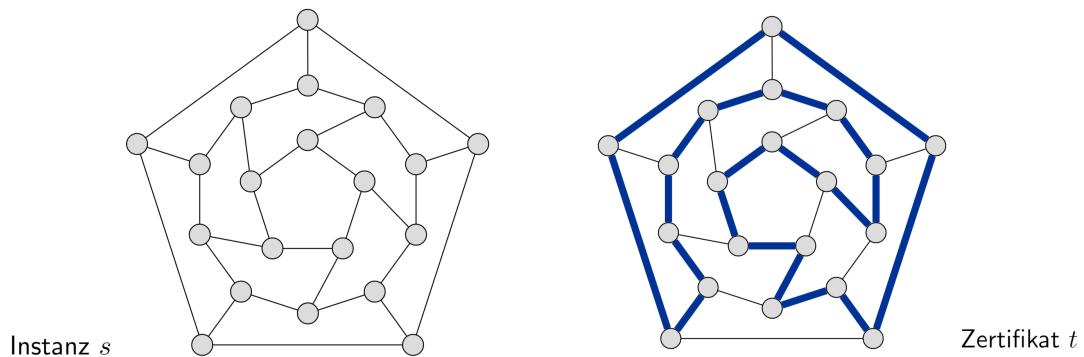
**Schlussfolgerung:** SAT ist in NP.

**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:** Ein 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:**



**Schlussfolgerung:** HAM-CYCLE ist in NP.

## Quiz zur Definition:

**Frage 6:** Welche der folgenden Probleme sind in NP?

- ✓ (A) INDEPENDENT SET
- ✗ (B) Gegeben ein Graph  $G$ , finde ein kleinstes Vertex Cover von  $G$ .
- ✗ (C) Gegeben ein Graph  $G$  und  $k > 0$ . Hat jedes Vertex Cover von  $G$  mindestens  $k$  Knoten?
- ✓ (D) Gegeben die Präferenzen von  $n$  Kindern und  $n$  Gastfamilien, existiert ein Stable Matching?

- B stimmt nicht weil kein ja/nein
- C stimmt nicht, weil man nicht Existenz beweisen muss --> alle anschauen --> nicht zertifizierbar.

## Weitere Eigenschaften zu P und NP

### ① Unterscheidung P und NP

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

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

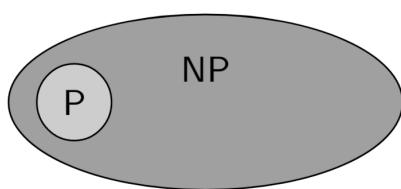
Es gilt:  $P \subseteq NP$ .

Beweis: Wir betrachten ein beliebiges Problem  $X$  in P.

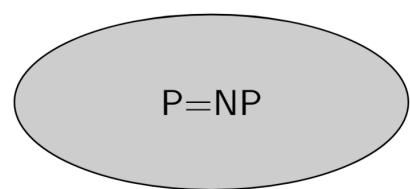
- Nach Definition existiert ein Polynomialzeit-Algorithmus  $A(s)$ , der  $X$  löst.
- Zertifikat:  $t = \emptyset$ , Zertifizierer  $C(s, t) = A(s)$ .  $\square$

### ② Gilt $P = NP$ ? [Cook 1971, Levin 1973]

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



Falls  $P \neq NP$



Falls  $P = NP$

- **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

## ⌚ Definition - NP-Schwer

- 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 - NP-vollständig

- 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.

## ⓘ 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.
- ( $\Rightarrow$ ) 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 (reduziere  $X$  auf  $Y$  in Polynomialzeit, lösse  $Y$  in Polynomialzeit). Das impliziert  $NP \subseteq P$ . Wir wissen bereits, dass  $P \subseteq NP$ . Daher  $P = NP$ .

## Gibt es ein NP-vollständiges Problem?

**Allgemeine Überlegung:** Das ist nicht von vornherein klar. Es könnte z.B. mehrere schwerste NP-Probleme geben, die nicht jeweils aufeinander reduzierbar sind.

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

## NP-Vollständigkeit nachweisen

- **Anmerkung:** Sobald ein Problem als NP-vollständig nachgewiesen wurde, können andere Probleme einfacher als NP-vollständig erkannt werden, da sie wie Dominosteine "umfallen".

### Rezept zum Nachweis der NP-Vollständigkeit eines Problems $Y$ :

1. **Schritt 1:** Zeige, dass  $Y$  in NP ist.
2. **Schritt 2:** Wähle ein NP-vollständiges Problem  $X$ .
3. **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.

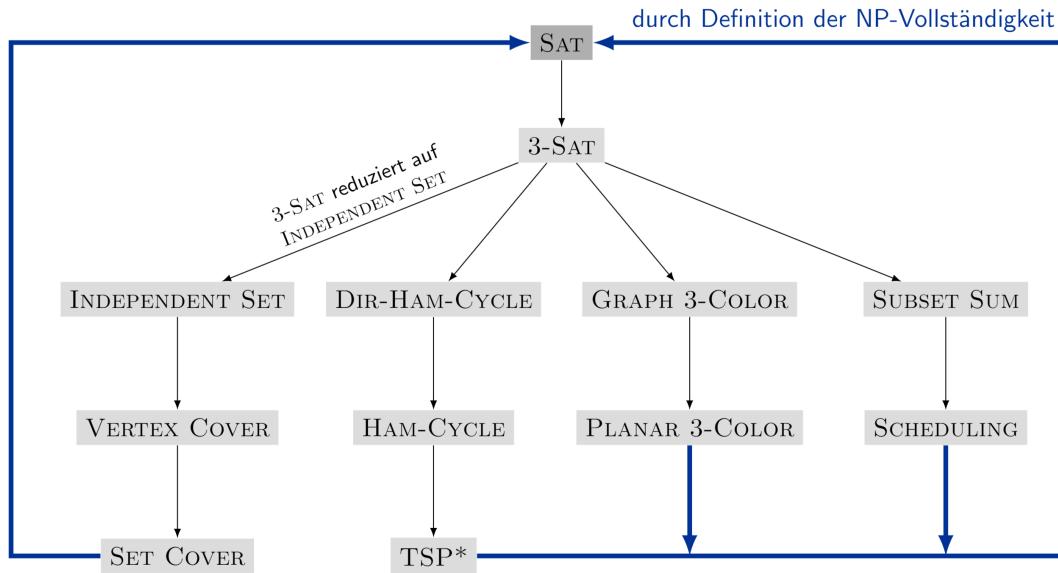
#### Beweis

Sei  $W$  ein beliebiges Problem in NP.

Dann gilt  $W \leq_p X \leq_p Y$ .

- $W \leq_p X$ : durch Definition von NP-Vollständigkeit von  $X$  (da  $X$  NP-vollständig ist, kann jedes Problem in NP polynomial-zeitreduziert werden auf  $X$ ).
- $X \leq_p Y$ : durch Annahme (unser Beweisschritt 3).
- Durch Transitivität gilt:  $W \leq_p Y$ .
  - (Transitivität bedeutet hier, wenn  $W$  auf  $X$  reduziert werden kann und  $X$  auf  $Y$ , dann kann  $W$  auch direkt auf  $Y$  reduziert werden.)
- Daher ist  $Y$  NP-vollständig.

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

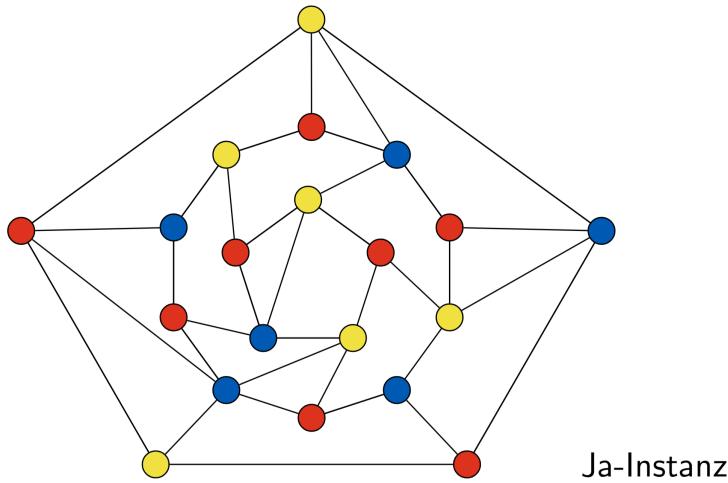


## 3-Color (3-Knotenfärbung)

Da wir nur bis hier in der vo gekommen sind werde ich hier einfach nur die Folien einblenden:

### 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?



## 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:**  $\text{3-COLOR} \leq_P k\text{-REGISTER-ALLOCATION}$  für eine beliebige Konstante  $k \geq 3$ .

## 3-COLOR

**Behauptung:**  $\text{3-SAT} \leq_P \text{3-COLOR}$ .

**Beweis:** Gegeben sei die 3-SAT-Instanz  $\Phi$ . Wir konstruieren eine Instanz von 3-COLOR, die genau dann 3-färbbar ist wenn  $\Phi$  erfüllbar ist.

## 3-COLOR

**Konstruktion:**

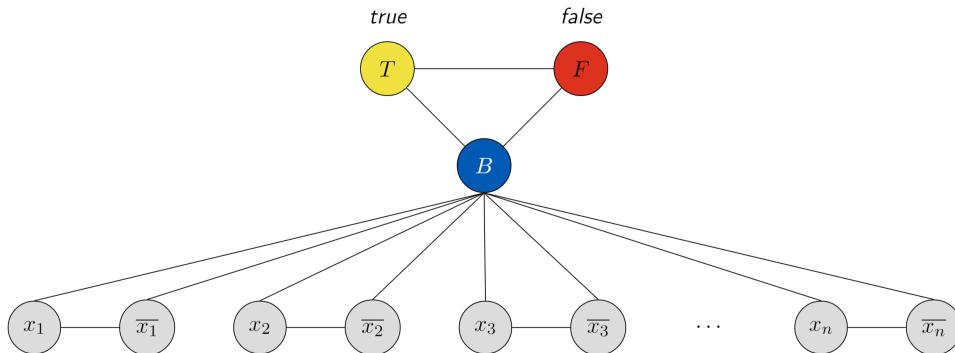
- Für jedes Literal wird ein Knoten erzeugt.
  - Erzeuge drei neue Knoten  $T$ ,  $F$ ,  $B$ . Verbinde diese Knoten zu einem Dreieck und verbinde jedes Literal mit  $B$ .
  - Verbinde jedes Literal mit seiner Negation.
  - Für jede Klausel wird ein **Gadget** mit 6 Knoten hinzugefügt; das Gadget simuliert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

## 3-COLOR

**Behauptung:** Graph ist 3-färbbar genau dann, wenn  $\Phi$  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.

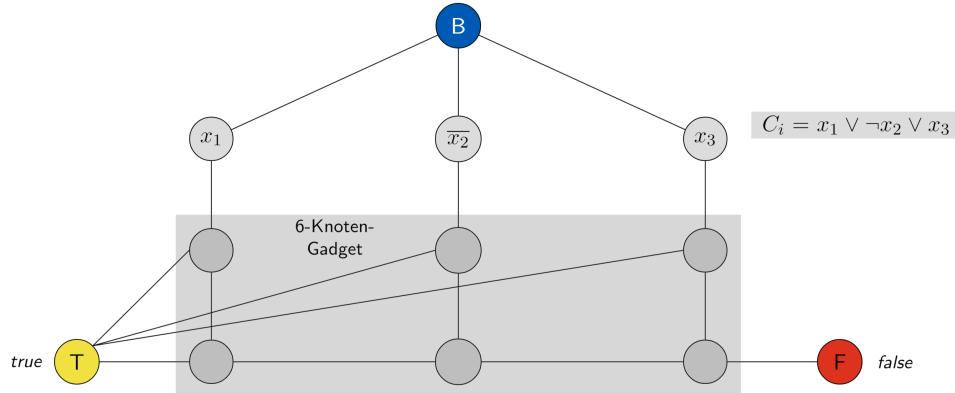


## 3-COLOR

**Behauptung:** Graph ist 3-färbbar genau dann, wenn  $\Phi$  erfüllbar ist.

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

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



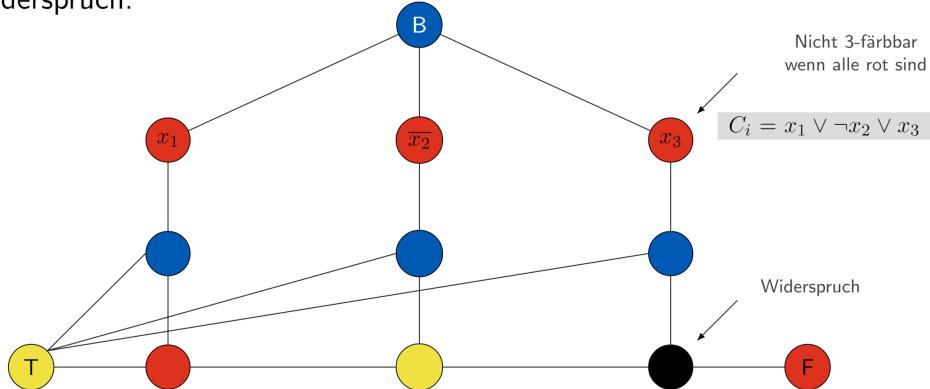
- $T, F, B$  Knoten verschoben und Kanten nicht eingezeichnet.

## 3-COLOR

**Behauptung:** Graph ist 3-färbbar genau dann, wenn  $\Phi$  erfüllbar ist.

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

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



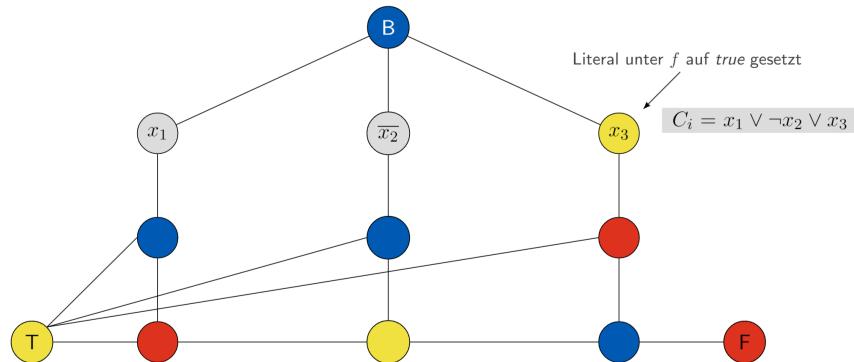
- Daher:  $\Phi$  ist erfüllbar, und die Richtung  $\Rightarrow$  ist gezeigt.

## 3-COLOR

**Behauptung:** Graph ist 3-färbbar genau dann, wenn  $\Phi$  erfüllbar ist.

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

- Betrachte eine erfüllende Wahrheitsbelegung  $f$ .
- Färbe Knoten  $T$  gelb,  $F$  rot und  $B$  blau.
- Färbe Literal gelb, wenn es unter  $f$  *true* ist, 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]

## 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 OPTVC: Gegeben ist ein Graph  $G$ , berechne ein kleinstes Vertex Cover von  $G$ .

**Theorem:** Angenommen  $P \neq NP$ . Dann gibt es keinen Polynomialzeitalgorithmus für 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 Optimierungsprobleme und funktionale Probleme als „NP-schwer“ bezeichnet, wenn aus ihrer Lösbarkeit in Polynomialzeit  $P = NP$  folgt.

Wir können also beispielsweise sagen, dass 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 → Approximationsalgorithmen, Heuristische Algorithmen.
- Löse Problem in Polynomialzeit → Algorithmen mit exponentieller Laufzeit.
- Löse beliebige Instanzen des Problems → Identifizierte effizient lösbar Spezialfälle.