Formale Grundlagen der Informatik 1
Kapitel 13
NP-Vollständigkeit

Frank Heitmann
heitmann@informatik.uni-hamburg.de

19. Mai 2015

Die Klassen P und NP

$$
\begin{aligned}
P &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine} \\
&\qquad p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\} \\
&= \cup_{i \geq 1} DTIME(n^i) \\
NP &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine} \\
&\qquad p(n)\text{-zeitbeschränkte NTM } A \text{ mit } L(A) = L\} \\
&= \cup_{i \geq 1} NTIME(n^i)
\end{aligned}
$$

Die Klassen P und NP

Probleme in P gelten als effizient lösbar.

Unser Ziel heute:

- Für Probleme, für die wir keinen Algorithmus in P finden,
- eine Technik entwickeln, mit der wir sagen können, dass es dies auch nicht gibt!
- (Solch eine Technik gibt es aktuell nicht, aber es gibt eine Technik, mit der wir sagen können, dass dies zumindest sehr unwahrscheinlich ist.)

Die Klasse P

$$
\begin{aligned}
P &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine} \\
&\qquad p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\} \\
&= \cup_{i \geq 1} DTIME(n^i)
\end{aligned}
$$

Anmerkung

- Ein Algorithmus (eine TM) A akzeptiert eine Sprache L in *polynomialer Zeit*, wenn sie von A akzeptiert wird und zusätzlich ein $k \in \mathbb{N}$ existiert, so dass jedes $x \in L$ mit $|x| = n$ in Zeit $O(n^k)$ akzeptiert wird.
- Akzeptieren und entscheiden ist hier aber egal, wie der nächste Satz zeigt ...

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Ein Problem in P

Typisches Beispiel eines Problems in P:

$$\texttt{PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus höchstens } k \text{ Kanten besteht.} \end{array} \}$$

Verifikation in polynomialer Zeit

$$\texttt{L-PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus } \textit{mindestens } k \text{ Kanten besteht.} \end{array} \}$$

Anmerkung

- Erscheint schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.
- Dies führt zu einer alternativen Definition von *NP* ...

Verifikation in polynomialer Zeit

$$\texttt{L-PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus } \textit{mindestens } k \text{ Kanten besteht.} \end{array} \}$$

Anmerkung

- Erscheint schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.
- Dies führt zu einer alternativen Definition von *NP* ...

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein deterministischer Algorithmus mit zwei Argumenten $x, y \in \Sigma^*$, wobei x die gewöhnliche Eingabe und y ein *Zertifikat* ist. A *verifiziert* x, wenn es ein Zertifikat y gibt mit $A(x, y) = 1$. Die von A verifizierte Sprache ist

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein deterministischer Algorithmus mit zwei Argumenten $x, y \in \Sigma^*$, wobei x die gewöhnliche Eingabe und y ein *Zertifikat* ist. A *verifiziert* x, wenn es ein Zertifikat y gibt mit $A(x, y) = 1$. Die von A verifizierte Sprache ist

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Die Klasse *NP*

In *NP* sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (*NP*)

$L \in NP$ gdw. ein Verifikationsalgorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x,y) = 1 \text{ gilt}\end{array} \}$$

gilt.

Die Klasse *NP*

In *NP* sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (*NP*)

$L \in NP$ gdw. ein Verifikationsalgorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x,y) = 1 \text{ gilt}\end{array} \}$$

gilt.

Die Klasse *NP*

In *NP* sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (*NP*)

$L \in NP$ gdw. ein Verifikationsalgorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x,y) = 1 \text{ gilt}\end{array} \}$$

gilt.

Nichtdeterminismus vs. Verifikation

Verifikation

Ein Verifikationsalgorithmus erhält neben der eigentlichen Eingabe noch ein Zertifikat und kann damit überprüfen, dass die Eingabe in der Sprache ist (zumindest, wenn es ein gutes Zertifikat ist).

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten' und so in einem Zustand z.B. eine Variable auf 0 und auf 1 setzen.

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Nichtdeterminismus vs. Verifikation

Verifikation

Ein Verifikationsalgorithmus erhält neben der eigentlichen Eingabe noch ein Zertifikat und kann damit überprüfen, dass die Eingabe in der Sprache ist (zumindest, wenn es ein gutes Zertifikat ist).

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten' und so in einem Zustand z.B. eine Variable auf 0 und auf 1 setzen.

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Nichtdeterminismus vs. Verifikation

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent

Bemerkung

- Um den Satz zu beweisen, überlegt man sich, wie man das Mengenpartitionsproblem von eben mit einem Verifikations- und wie mit einem nichtdeterministischen Algorithmus löst und generalisiert die Idee.
- Die Verifikationsidee wird oft in der Algorithmik benutzt.

NP-Probleme lösen

Frage

Wie löst man nun *NP* Probleme *deterministisch*?

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

NP-Probleme lösen

Frage

Wie löst man nun *NP* Probleme *deterministisch*?

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

NP-Probleme lösen

Frage

Wie löst man nun *NP* Probleme *deterministisch*?

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

Beweis.

Beweisskizze/Idee: Ist $L \in NP$, so gibt es einen Verifikationsalgorithmus in $O(n^k)$ (n ist die Eingabelänge). Das Zertifikat y hat eine Länge in $O(n^c)$. Man geht alle $2^{O(n^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgortihmus aus. Dieses Verfahren ist in $2^{O(n^k)}$. □

NP-Probleme lösen

Frage

Wie löst man nun *NP* Probleme *deterministisch*?

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

Beweis.

Beweisskizze/Idee: Ist $L \in NP$, so gibt es einen Verifikationsalgorithmus in $O(n^k)$ (n ist die Eingabelänge). Das Zertifikat y hat eine Länge in $O(n^c)$. Man geht alle $2^{O(n^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgortihmus aus. Dieses Verfahren ist in $2^{O(n^k)}$. □

NP-Probleme lösen

Frage

Wie löst man nun *NP* Probleme *deterministisch*?

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

Beweis.

Beweisskizze/Idee: Ist $L \in NP$, so gibt es einen Verifikationsalgorithmus in $O(n^k)$ (n ist die Eingabelänge). Das Zertifikat y hat eine Länge in $O(n^c)$. Man geht alle $2^{O(n^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgortihmus aus. Dieses Verfahren ist in $2^{O(n^k)}$. □

NP-Probleme lösen

Nebenbemerkung

Zu der Laufzeitschranke von eben:

- Das Verfahren ist eigentlich in $2^{O(n^c)} \cdot O(n^k)$.
- Wegen $n^k \leq 2^{n^k}$ kann man dies nach oben mit $2^{O(n^c)} \cdot 2^{O(n^k)}$ abschätzen.
- Nun darf man $k \geq c$ annehmen, da sonst der Verifikationsalgorithmus eine Laufzeit hätte bei der er sich gar nicht das ganze Zertifikat ansehen kann. Damit kann man nach oben durch $2^{O(n^k)} \cdot 2^{O(n^k)}$ abschätzen.
- Dies ist gleich $2^{2 \cdot O(n^k)} = 2^{O(n^k)}$.

Mengenpartitionsproblem - Deterministisch

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Algorithmus 1 Suchraum durchsuchen (deterministisch!)

```
1: for all A ⊆ S do
2:     if Σ_{x∈A} x = Σ_{x∈Ā} x then
3:         return true
4:     end if
5: end for
6: return false
```

Laufzeit ist *deterministisch* in $O(2^{|S|})$

Mengenpartitionsproblem - Nichtdeterministisch

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Algorithmus 2 Suchraum durchsuchen (nichtdeterministisch!)

1: Rate ein $A \subseteq S$
2: **if** $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ **then**
3: **return true**
4: **end if**
5: **return false**

Laufzeit ist *nichtdeterministisch* in $O(|S|)$ (also in NP).

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

P vs. *NP*

Alle oben genannten Probleme sind

- in *NP* - und damit schnell *nichtdeterministisch* lösbar

Die besten bekannten deterministischen Algorithmen

- benötigen aber exponentielle Laufzeit!

Geht es wirklich nicht schneller?!?

P vs. *NP*

Alle oben genannten Probleme sind

- in *NP* - und damit schnell *nichtdeterministisch* lösbar

Die besten bekannten deterministischen Algorithmen

- benötigen aber exponentielle Laufzeit!

Geht es wirklich nicht schneller?!?

P vs. *NP*

Alle oben genannten Probleme sind

- in *NP* - und damit schnell *nichtdeterministisch* lösbar

Die besten bekannten deterministischen Algorithmen

- benötigen aber exponentielle Laufzeit!

Geht es wirklich nicht schneller?!?

Das Problem

Ziel und Hindernis

- Falls nicht, würden wir gerne so etwas zeigen können wie "dieses Problem lässt sich für kein k in $O(n^k)$ lösen".
- Leider haben wir aktuell keine Möglichkeit untere (Zeit-)Schranken für Probleme zu zeigen!

Wir geben aber nicht auf und behelfen uns mit einem anderen Ansatz...

- Wir zeigen, dass Probleme eine bestimmte Eigenschaft haben, so dass, wenn das Problem doch in P lösbar ist, sehr unwahrscheinliche Dinge folgen würde.
- Dazu etablieren wir eine Art zu sagen, dass ein Problem zu "den schwierigsten Problemen seiner Klasse" gehört.
- Und dazu brauchen wir den Begriff der Reduktion ...

Das Problem

Ziel und Hindernis

- Falls nicht, würden wir gerne so etwas zeigen können wie "dieses Problem lässt sich für kein k in $O(n^k)$ lösen".
- Leider haben wir aktuell keine Möglichkeit untere (Zeit-)Schranken für Probleme zu zeigen!

Wir geben aber nicht auf und behelfen uns mit einem anderen Ansatz...

- Wir zeigen, dass Probleme eine bestimmte Eigenschaft haben, so dass, wenn das Problem doch in P lösbar ist, sehr unwahrscheinliche Dinge folgen würde.
- Dazu etablieren wir eine Art zu sagen, dass ein Problem zu "den schwierigsten Problemen seiner Klasse" gehört.
- Und dazu brauchen wir den Begriff der Reduktion ...

Das Problem

Ziel und Hindernis

- Falls nicht, würden wir gerne so etwas zeigen können wie "dieses Problem lässt sich für kein k in $O(n^k)$ lösen".
- Leider haben wir aktuell keine Möglichkeit untere (Zeit-)Schranken für Probleme zu zeigen!

Wir geben aber nicht auf und behelfen uns mit einem anderen Ansatz...

- Wir zeigen, dass Probleme eine bestimmte Eigenschaft haben, so dass, wenn das Problem doch in P lösbar ist, sehr unwahrscheinliche Dinge folgen würde.
- Dazu etablieren wir eine Art zu sagen, dass ein Problem zu "den schwierigsten Problemen seiner Klasse" gehört.
- Und dazu brauchen wir den Begriff der Reduktion ...

Das Problem

Ziel und Hindernis

- Falls nicht, würden wir gerne so etwas zeigen können wie "dieses Problem lässt sich für kein k in $O(n^k)$ lösen".
- Leider haben wir aktuell keine Möglichkeit untere (Zeit-)Schranken für Probleme zu zeigen!

Wir geben aber nicht auf und behelfen uns mit einem anderen Ansatz...

- Wir zeigen, dass Probleme eine bestimmte Eigenschaft haben, so dass, wenn das Problem doch in P lösbar ist, sehr unwahrscheinliche Dinge folgen würde.
- Dazu etablieren wir eine Art zu sagen, dass ein Problem zu "den schwierigsten Problemen seiner Klasse" gehört.
- Und dazu brauchen wir den Begriff der Reduktion ...

Das Problem

Ziel und Hindernis

- Falls nicht, würden wir gerne so etwas zeigen können wie "dieses Problem lässt sich für kein k in $O(n^k)$ lösen".
- Leider haben wir aktuell keine Möglichkeit untere (Zeit-)Schranken für Probleme zu zeigen!

Wir geben aber nicht auf und behelfen uns mit einem anderen Ansatz...

- Wir zeigen, dass Probleme eine bestimmte Eigenschaft haben, so dass, wenn das Problem doch in P lösbar ist, sehr unwahrscheinliche Dinge folgen würde.
- Dazu etablieren wir eine Art zu sagen, dass ein Problem zu "den schwierigsten Problemen seiner Klasse" gehört.
- Und dazu brauchen wir den Begriff der Reduktion ...

Reduktionen

Definition (Reduktion)

Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 *in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

für alle $x \in \{0,1\}^*$ gilt. Hierfür schreiben wir dann $L_1 \leq_p L_2$. f wird als Reduktionsfunktion, ein Algortihmus der f berechnet als Reduktionsalgorithmus bezeichnet.

Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin L_2$.)

Reduktionen

Definition (Reduktion)

Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 *in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

für alle $x \in \{0,1\}^*$ gilt. Hierfür schreiben wir dann $L_1 \leq_p L_2$. f wird als Reduktionsfunktion, ein Algortihmus der f berechnet als Reduktionsalgorithmus bezeichnet.

Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin L_2$.)

Reduktionen

Definition (Reduktion)

Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 *in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

für alle $x \in \{0,1\}^*$ gilt. Hierfür schreiben wir dann $L_1 \leq_p L_2$. f wird als Reduktionsfunktion, ein Algortihmus der f berechnet als Reduktionsalgorithmus bezeichnet.

Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin L_2$.)

Reduktionen

Definition (Reduktion)

Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 *in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

für alle $x \in \{0,1\}^*$ gilt. Hierfür schreiben wir dann $L_1 \leq_p L_2$. f wird als Reduktionsfunktion, ein Algortihmus der f berechnet als Reduktionsalgorithmus bezeichnet.

Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin L_2$.)

Exkurs: Reduktionen allgemein

Exkurs

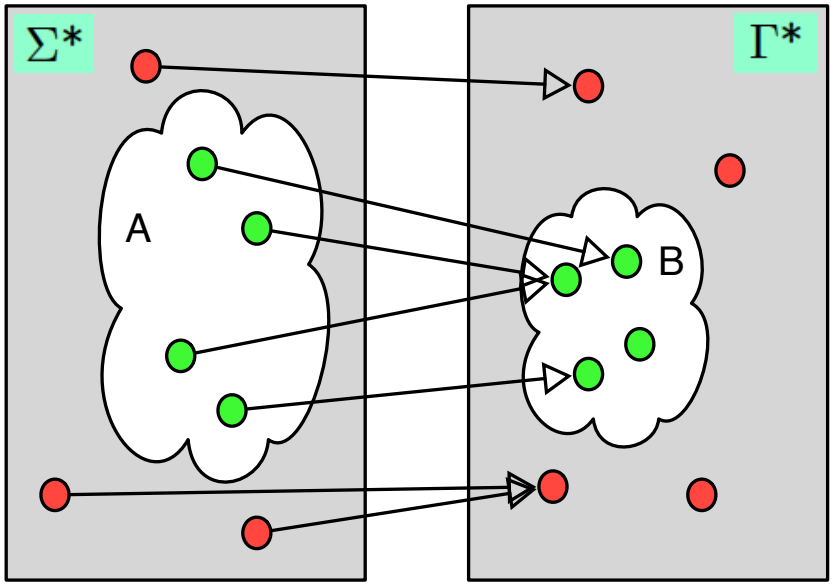
- Allgemeiner ist zu zwei Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ eine Reduktion eine Funktion $f : \Sigma^* \to \Gamma^*$ mit $x \in A$ gdw. $f(x) \in B$ für alle $x \in \Sigma^*$.
- Den Sprachen können also verschiedene Alphabete zugrunde liegen und die Reduktion muss (zunächst) nicht in Polynomialzeit möglich sein.
- Man kann dann unterschiedliche Zeitreduktionen einführen und so z.B. auch P-vollständige Probleme definieren (was dann die schwierigsten Probleme in P sind).

Exkurs: Reduktionen allgemein

Exkurs

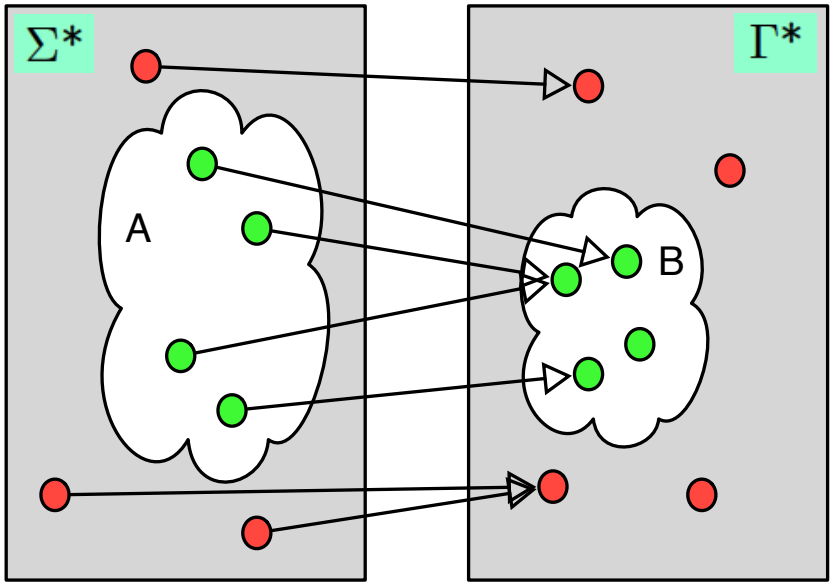
- Allgemeiner ist zu zwei Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ eine Reduktion eine Funktion $f : \Sigma^* \to \Gamma^*$ mit $x \in A$ gdw. $f(x) \in B$ für alle $x \in \Sigma^*$.
- Den Sprachen können also verschiedene Alphabete zugrunde liegen und die Reduktion muss (zunächst) nicht in Polynomialzeit möglich sein.
- Man kann dann unterschiedliche Zeitreduktionen einführen und so z.B. auch P-vollständige Probleme definieren (was dann die schwierigsten Probleme in P sind).

Reduktionen: Erläuterungen



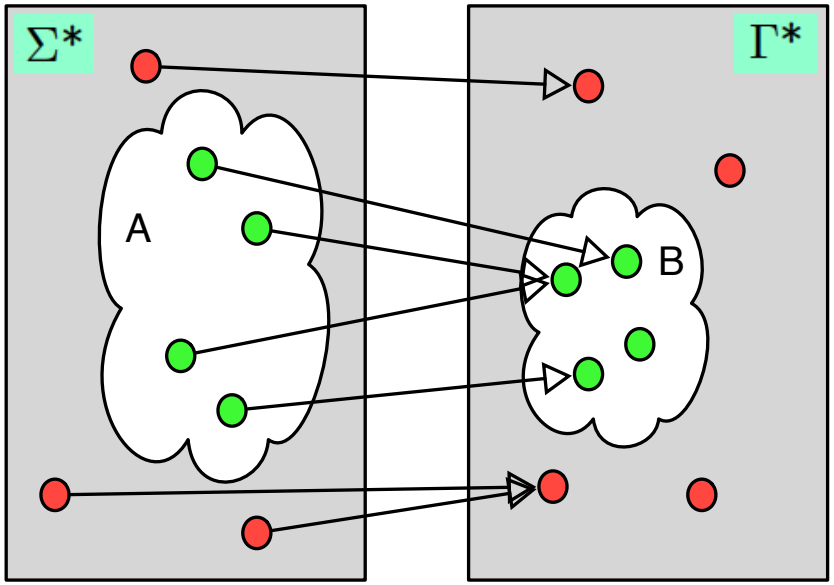
- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion bezeichnet.)

Reduktionen: Erläuterungen



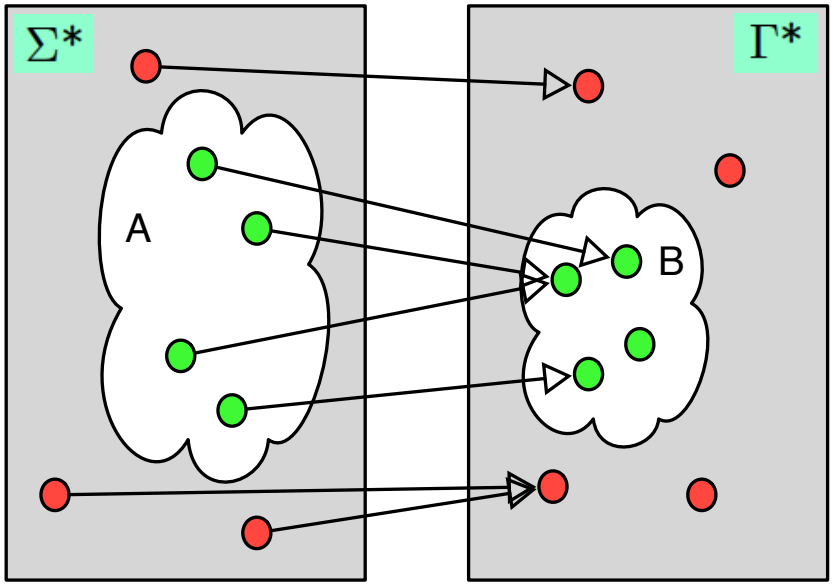
- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion bezeichnet.)

Reduktionen: Erläuterungen



- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion bezeichnet.)

Reduktionen: Erläuterungen



- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion bezeichnet.)

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, dann folgt aus $L_2 \in P$ auch $L_1 \in P$.

Beweis.

- Wegen $L_1 \leq_p L_2$ gibt es eine Reduktionsfunktion f mit $x \in L_1$ gdw. $f(x) \in L_2$, die in Polynomialzeit berechenbar ist.
- Wegen $L_2 \in P$ kann L_2 von einem Algorithmus A_2 in Polynomialzeit entschieden werden.
- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Wegen $L_1 \leq_p L_2$ gibt es eine Reduktionsfunktion f mit $x \in L_1$ gdw. $f(x) \in L_2$, die in Polynomialzeit berechenbar ist.
- Wegen $L_2 \in P$ kann L_2 von einem Algorithmus A_2 in Polynomialzeit entschieden werden.
- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Wegen $L_1 \leq_p L_2$ gibt es eine Reduktionsfunktion f mit $x \in L_1$ gdw. $f(x) \in L_2$, die in Polynomialzeit berechenbar ist.
- Wegen $L_2 \in P$ kann L_2 von einem Algorithmus A_2 in Polynomialzeit entschieden werden.
- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Wegen $L_1 \leq_p L_2$ gibt es eine Reduktionsfunktion f mit $x \in L_1$ gdw. $f(x) \in L_2$, die in Polynomialzeit berechenbar ist.
- Wegen $L_2 \in P$ kann L_2 von einem Algorithmus A_2 in Polynomialzeit entschieden werden.
- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Wegen $L_1 \leq_p L_2$ gibt es eine Reduktionsfunktion f mit $x \in L_1$ gdw. $f(x) \in L_2$, die in Polynomialzeit berechenbar ist.
- Wegen $L_2 \in P$ kann L_2 von einem Algorithmus A_2 in Polynomialzeit entschieden werden.
- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, dann folgt aus $L_2 \in P$ auch $L_1 \in P$.

Beweis.

- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- A_1 arbeitet in Polynomialzeit: f kann in Polynomialzeit berechnet werden und daher ist $|f(x)| \in O(|x|^c)$ (c eine Konstante). Die Laufzeit von A_2 ist dann durch $O(|f(x)|^d) = O(|x|^{c \cdot d})$ beschränkt. Insgesamt arbeitet A_1 also in Polynomialzeit: $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- A_1 arbeitet in Polynomialzeit: f kann in Polynomialzeit berechnet werden und daher ist $|f(x)| \in O(|x|^c)$ (c eine Konstante). Die Laufzeit von A_2 ist dann durch $O(|f(x)|^d) = O(|x|^{c \cdot d})$ beschränkt. Insgesamt arbeitet A_1 also in Polynomialzeit: $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- A_1 arbeitet in Polynomialzeit: f kann in Polynomialzeit berechnet werden und daher ist $|f(x)| \in O(|x|^c)$ (c eine Konstante). Die Laufzeit von A_2 ist dann durch $O(|f(x)|^d) = O(|x|^{c \cdot d})$ beschränkt. Insgesamt arbeitet A_1 also in Polynomialzeit: $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Beweis.

- Der Algorithmus A_1, der L_1 in Polynomialzeit entscheidet arbeitet auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:
  - Berechne $f(x)$.
  - Nutze A_2, um $f(x) \in L_2$ zu entscheiden.
- A_1 arbeitet in Polynomialzeit: f kann in Polynomialzeit berechnet werden und daher ist $|f(x)| \in O(|x|^c)$ (c eine Konstante). Die Laufzeit von A_2 ist dann durch $O(|f(x)|^d) = O(|x|^{c \cdot d})$ beschränkt. Insgesamt arbeitet A_1 also in Polynomialzeit: $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$.

□

Probleme durch andere lösen

Satz

Seien $L_1, L_2 \subseteq \{0,1\}^*$ *mit* $L_1 \leq_p L_2$*, dann folgt aus* $L_2 \in P$ *auch* $L_1 \in P$*.*

Anmerkung

Mit obigen Satz, kann man ein Problem (L_1) durch ein anderes (L_2) lösen. Darum auch Reduktion: Statt einen Algortihmus für L_1 zu finden und so L_1 zu lösen, findet man einen für L_2 und löst so nicht nur L_2, sondern (dank des Reduktionsalgorithmus) auch L_1. Das Problem L_1 zu lösen ist also darauf 'reduziert' worden das Problem L_2 zu lösen.
(Alternativ kann eine Reduktion als eine *Transformation* von einem Problem in ein anderes angesehen werden.)

Übergang zur *NP*-Vollständigkeit...

- Ist $A \leq_p B$, so ist A höchstens so schwierig wie B.
- Reduziert man nun *jede* Sprache aus NP auf eine (neue) Sprache L, so ist L mindestens so schwierig wie *ganz* NP, denn löst man L, kann man jedes Problem in NP lösen.
- Das macht $L \in P$ *sehr* unwahrscheinlich, weil dann $P = NP$ gelten würde.

Anmerkung

Höchstens/mindestens bezieht sich auf polynomiellen Mehraufwand, der hier (im Falle von Problemen in P und NP) als akzeptabel angesehen wird.

Übergang zur *NP*-Vollständigkeit...

- Ist $A \leq_p B$, so ist A höchstens so schwierig wie B.
- Reduziert man nun *jede* Sprache aus NP auf eine (neue) Sprache L, so ist L mindestens so schwierig wie *ganz* NP, denn löst man L, kann man jedes Problem in NP lösen.
- Das macht $L \in P$ *sehr* unwahrscheinlich, weil dann $P = NP$ gelten würde.

Anmerkung

Höchstens/mindestens bezieht sich auf polynomiellen Mehraufwand, der hier (im Falle von Problemen in P und NP) als akzeptabel angesehen wird.

Übergang zur *NP*-Vollständigkeit...

- Ist $A \leq_p B$, so ist A höchstens so schwierig wie B.
- Reduziert man nun *jede* Sprache aus NP auf eine (neue) Sprache L, so ist L mindestens so schwierig wie *ganz* NP, denn löst man L, kann man jedes Problem in NP lösen.
- Das macht $L \in P$ *sehr* unwahrscheinlich, weil dann $P = NP$ gelten würde.

Anmerkung

Höchstens/mindestens bezieht sich auf polynomiellen Mehraufwand, der hier (im Falle von Problemen in P und NP) als akzeptabel angesehen wird.

NP-vollständig

Definition

Eine Sprache $L \subseteq \{0,1\}^*$ wird als *NP*-vollständig bezeichnet, wenn

1. $L \in NP$ und
2. $L' \leq_p L$ für jedes $L' \in NP$ gilt.

Kann man für L zunächst nur die zweite Eigenschaft beweisen, so ist L *NP*-schwierig (-schwer/-hart).
Alle *NP*-vollständigen Probleme bilden die Komplexitätsklasse *NPC*.

NP-vollständig

Definition

Eine Sprache $L \subseteq \{0,1\}^*$ wird als *NP*-vollständig bezeichnet, wenn

1. $L \in NP$ und
2. $L' \leq_p L$ für jedes $L' \in NP$ gilt.

Kann man für L zunächst nur die zweite Eigenschaft beweisen, so ist L *NP*-schwierig (-schwer/-hart).

Alle *NP*-vollständigen Probleme bilden die Komplexitätsklasse *NPC*.

NP-vollständig

Definition

Eine Sprache $L \subseteq \{0,1\}^*$ wird als *NP*-vollständig bezeichnet, wenn

1. $L \in NP$ und
2. $L' \leq_p L$ für jedes $L' \in NP$ gilt.

Kann man für L zunächst nur die zweite Eigenschaft beweisen, so ist L *NP*-schwierig (-schwer/-hart).
Alle *NP*-vollständigen Probleme bilden die Komplexitätsklasse *NPC*.

Zwei wichtige Theoreme (I)

Theorem

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Beweis.

Sei $L \in NPC \cap P$. Sei nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem letzten Satz $L' \in P$. □

Anmerkung

Äquivalente Formulierung: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$.

Zwei wichtige Theoreme (I)

Theorem

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Beweis.

Sei $L \in NPC \cap P$. Sei nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem letzten Satz $L' \in P$. □

Anmerkung

Äquivalente Formulierung: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$.

Zwei wichtige Theoreme (I)

Theorem

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Beweis.

Sei $L \in NPC \cap P$. Sei nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem letzten Satz $L' \in P$. □

Anmerkung

Äquivalente Formulierung: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$.

Zwei wichtige Theoreme (I)

Theorem

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Beweis.

Sei $L \in NPC \cap P$. Sei nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem letzten Satz $L' \in P$. □

Anmerkung

Äquivalente Formulierung: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$.

Zwei wichtige Theoreme (I)

Theorem

Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.

Beweis.

Sei $L \in NPC \cap P$. Sei nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem letzten Satz $L' \in P$. □

Anmerkung

Äquivalente Formulierung: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$.

Zur Nachbereitung

Wichtige Anmerkung

Der letzte Satz rechtfertigt die Aussage, dass ein Problem in *NPC* (also ein *NP*-vollständiges Problem) höchstwahrscheinlich nicht effizient lösbar ist (also in P ist), da dann $P = NP$ gelten würde und damit alle Probleme in *NP* (darunter auch all die komplizierten aus *NPC*) effizient lösbar (in P) wären.

Begründung

Dass wir an $P = NP$ nicht glauben liegt daran, dass in *NPC* sehr viele Probleme liegen, an denen schon seit sehr langer Zeit gearbeitet wird und für keines davon kennen wir einen effizienten Algorithmus (einen in P).

Zwei wichtige Theoreme (II)

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.

Beweis.

Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Zwei wichtige Theoreme (II)

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.

Beweis.

Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Zwei wichtige Theoreme (II)

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.

Beweis.

Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Zwei wichtige Theoreme (II)

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.

Beweis.

Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Zwei wichtige Theoreme (II)

Satz

Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.

Beweis.

Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Zwei wichtige Theoreme (III)

Theorem

Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.

Beweis.

Wegen $L' \in NPC$ gilt $L'' \leq_p L'$ für jedes $L'' \in NP$. Aus $L' \leq_p L$ und dem vorherigen Satz folgt dann $L'' \leq_p L$, L ist also NP-schwierig. Ist zusätzlich $L \in NP$, so ist L nach Definition NP-vollständig. □

Zwei wichtige Theoreme (III)

Theorem

Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.

Beweis.

Wegen $L' \in NPC$ gilt $L'' \leq_p L'$ für jedes $L'' \in NP$. Aus $L' \leq_p L$ und dem vorherigen Satz folgt dann $L'' \leq_p L$, L ist also NP-schwierig. Ist zusätzlich $L \in NP$, so ist L nach Definition NP-vollständig. □

Zwei wichtige Theoreme (III)

Theorem

Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.

Beweis.

Wegen $L' \in NPC$ gilt $L'' \leq_p L'$ für jedes $L'' \in NP$. Aus $L' \leq_p L$ und dem vorherigen Satz folgt dann $L'' \leq_p L$, L ist also *NP*-schwierig. Ist zusätzlich $L \in NP$, so ist L nach Definition *NP*-vollständig. □

Zwei wichtige Theoreme (III)

Theorem

Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.

Beweis.

Wegen $L' \in NPC$ gilt $L'' \leq_p L'$ für jedes $L'' \in NP$. Aus $L' \leq_p L$ und dem vorherigen Satz folgt dann $L'' \leq_p L$, L ist also NP-schwierig. Ist zusätzlich $L \in NP$, so ist L nach Definition NP-vollständig. □

Verfahren

Methode zum Beweis der *NP*-Vollständigkeit einer Sprache L:

1. Zeige $L \in NP$.
2. Wähle ein $L' \in NPC$ aus.
3. Gib einen Algorithmus an, der ein f berechnet, das jede Instanz $x \in \{0,1\}^*$ von L' auf eine Instanz $f(x)$ von L abbildet (also eine Reduktion).
4. Beweise, dass f die Eigenschaft $x \in L'$ gdw. $f(x) \in L$ für jedes $x \in \{0,1\}^*$ besitzt.
5. Beweise, dass f in Polynomialzeit berechnet werden kann.

Anmerkung

Die letzten drei Punkte zeigen $L' \leq_p L$. Mit dem vorherigen Satz folgt daraus und aus den ersten beiden Punkten $L \in NPC$.

Verfahren

Anmerkung

Um im Verfahren eben ein $L' \in NPC$ auswählen zu können, muss man aber erstmal welche haben! Je mehr man kennt, desto besser ist es später, aber ein erstes brauchen wir und dort werden wir tatsächlich *alle* Probleme aus *NP* auf dieses reduzieren müssen!

Zusammenfassung

Zusammenfassung bisher:

- In P sind jene Probleme, die in Polynomialzeit lösbar sind,
  - d.h. es gibt ein Polynom p, so dass bei einer Eingabe der Länge n maximal $p(n)$ Schritte benötigt werden.
  - Probleme in P gelten als effizient lösbar.
- In NP sind Probleme, die *nichtdeterministisch* in Polynomialzeit lösbar sind.
- Ein NP Problem kann deterministisch auf jeden Fall in Exponentialzeit gelöst werden.
- Um zu zeigen, dass es wahrscheinlich nicht schneller geht, zeigt man dass das Problem NP-vollständig ist.
- Begriffe: P, NP, Reduktion, NP-schwierig, NP-vollständig.

Fragen

Welche Aussage gilt unter der Annahme $P \neq NP$?

1. $P \subsetneq NP \subsetneq NPC$
2. $P \subsetneq NPC \subsetneq NP$
3. $P \subsetneq NP$ und $NPC \subsetneq NP$ und $P \cap NPC = \emptyset$
4. $P \subsetneq NPC$ und $NP \subsetneq NPC$

Fragen

Sei *NPH* die Klasse der *NP*-schwierigen Probleme. Was gilt?

1. $NPH \subseteq NPC$
2. $NPC \subseteq NPH$
3. $NPC \cap NPH = \emptyset$
4. $NPC \cap NPH \neq \emptyset$ aber auch $NPC \setminus NPH \neq \emptyset$ und $NPH \setminus NPC \neq \emptyset$

Fragen

Sei $L_{NPC} \in NPC$ und die Komplexität von $L_?$ unbekannt. Welche Reduktion müssen Sie zeigen, um $L_?$ als *NP*-vollständig nachzuweisen?

1. $L_?$ auf L_{NPC} reduzieren
2. L_{NPC} auf $L_?$ reduzieren
3. Beide oben genannten Reduktionen
4. Welche Richtung ist wegen der Eigenschaft der Reduktion ($x \in L_1$ gdw. $f(x) \in L_2$) egal.

Fragen

Sei nochmal $L_{NPC} \in NPC$ und $L_?$ ein Problem mit unbekannter Komplexität. Was wissen Sie, wenn Sie doch $L_? \leq_p L_{NPC}$ zeigen?

1. Nichts! (Zumindest nichts hilfreiches!)
2. $L_? \in NPC$
3. $L_? \in NP$
4. L_{NPC} "erbt" die Komplexität von $L_?$, wenn wir diese ermittelt haben.

Fragen

Zur Nachbereitung

1. 3. ist richtig. Alle anderen entfallen, da sie $P \subset NPC$ enthalten, woraus $P = NP$ folgt.
2. 2. ist richtig. Jedes *NP*-vollständige Problem ist auch *NP*-schwierig, da für *NP*-schwierig weniger verlangt wird.
3. 2. ist richtig. Zudem muss noch $L_? \in NP$ gezeigt werden.
4. 3. ist richtig. Man kann in Polynomialzeit mit der Reduktion $L_?$ auf L_{NPC} reduzieren und dann (in *NP*) L_{NPC} lösen. Damit hat man dann einen *NP* Algorithmus für $L_?$.

SAT - Ein erstes NPC Problem!

Definition (Aussagenlogische Formeln)

- $x_1, x_2, \ldots$ sind Boolsche Variablen, die den Wert 0 oder 1 annehmen können.
- Es gibt die Verknüpfungen:
  - Negation $\neg$ mit $\neg 0 = 1$ und $\neg 1 = 0$.
  - Disjunktion $\vee$ mit $(x_1 \vee x_2) = 1$ gdw. mindestens eine der beiden Variablen 1 ist.
  - Konjunktion $\wedge$ mit $(x_1 \wedge x_2) = 1$ gdw. beide Variablen 1 sind.
- Gegeben eine Formel ϕ, ist eine Belegung eine Funktion, die jeder Variablen in ϕ einen Wert aus $\{0, 1\}$ zu weist. Kann man die Formel dann mit obigen Verknüpfungen zu 1 auswerten, so ist die Formel *erfüllbar*.

SAT - Ein erstes NPC Problem!

Definition (Aussagenlogische Formeln)

- $x_1, x_2, \ldots$ sind Boolsche Variablen, die den Wert 0 oder 1 annehmen können.
- Es gibt die Verknüpfungen:
  - Negation $\neg$ mit $\neg 0 = 1$ und $\neg 1 = 0$.
  - Disjunktion $\vee$ mit $(x_1 \vee x_2) = 1$ gdw. mindestens eine der beiden Variablen 1 ist.
  - Konjunktion $\wedge$ mit $(x_1 \wedge x_2) = 1$ gdw. beide Variablen 1 sind.
- Gegeben eine Formel ϕ, ist eine Belegung eine Funktion, die jeder Variablen in ϕ einen Wert aus $\{0, 1\}$ zu weist. Kann man die Formel dann mit obigen Verknüpfungen zu 1 auswerten, so ist die Formel *erfüllbar*.

SAT - Ein erstes NPC Problem!

Definition (Aussagenlogische Formeln)

- $x_1, x_2, \ldots$ sind Boolsche Variablen, die den Wert 0 oder 1 annehmen können.
- Es gibt die Verknüpfungen:
  - Negation $\neg$ mit $\neg 0 = 1$ und $\neg 1 = 0$.
  - Disjunktion $\vee$ mit $(x_1 \vee x_2) = 1$ gdw. mindestens eine der beiden Variablen 1 ist.
  - Konjunktion $\wedge$ mit $(x_1 \wedge x_2) = 1$ gdw. beide Variablen 1 sind.
- Gegeben eine Formel ϕ, ist eine Belegung eine Funktion, die jeder Variablen in ϕ einen Wert aus $\{0, 1\}$ zu weist. Kann man die Formel dann mit obigen Verknüpfungen zu 1 auswerten, so ist die Formel *erfüllbar*.

SAT - Ein erstes NPC Problem!

Definition (Aussagenlogische Formeln)

- $x_1, x_2, \ldots$ sind Boolsche Variablen, die den Wert 0 oder 1 annehmen können.
- Es gibt die Verknüpfungen:
  - Negation $\neg$ mit $\neg 0 = 1$ und $\neg 1 = 0$.
  - Disjunktion $\vee$ mit $(x_1 \vee x_2) = 1$ gdw. mindestens eine der beiden Variablen 1 ist.
  - Konjunktion $\wedge$ mit $(x_1 \wedge x_2) = 1$ gdw. beide Variablen 1 sind.
- Gegeben eine Formel ϕ, ist eine Belegung eine Funktion, die jeder Variablen in ϕ einen Wert aus $\{0, 1\}$ zu weist. Kann man die Formel dann mit obigen Verknüpfungen zu 1 auswerten, so ist die Formel *erfüllbar*.

SAT - Ein erstes NPC Problem!

Definition (Aussagenlogische Formeln)

- $x_1, x_2, \ldots$ sind Boolsche Variablen, die den Wert 0 oder 1 annehmen können.
- Es gibt die Verknüpfungen:
  - Negation $\neg$ mit $\neg 0 = 1$ und $\neg 1 = 0$.
  - Disjunktion $\vee$ mit $(x_1 \vee x_2) = 1$ gdw. mindestens eine der beiden Variablen 1 ist.
  - Konjunktion $\wedge$ mit $(x_1 \wedge x_2) = 1$ gdw. beide Variablen 1 sind.
- Gegeben eine Formel ϕ, ist eine Belegung eine Funktion, die jeder Variablen in ϕ einen Wert aus $\{0, 1\}$ zu weist. Kann man die Formel dann mit obigen Verknüpfungen zu 1 auswerten, so ist die Formel *erfüllbar*.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Ein Beispiel:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \wedge (\neg x_2 \vee \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned} \phi &= (0 \vee 1) \wedge (\neg 0 \wedge (\neg 1 \vee \neg 0)) \\ &= (0 \vee 1) \wedge (1 \wedge (0 \vee 1)) \\ &= 1 \wedge (1 \wedge 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Anmerkung

Man beachte, dass wir *keine* Klammerersparnisregeln einführen und dass wir hier nicht mit Auswertungen etc. arbeiten, wie man es formal tun würde (um Syntax und Semantik zu unterscheiden). Für uns reicht hier das intuitive Verständnis.

SAT - Ein erstes NPC Problem!

Definition (SAT)

$\texttt{SAT} = \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Formel}\}$

Anmerkung

Das *Erfüllbarkeitsproblem der Aussagenlogik*, SAT, ist historisch das erste *NP*-vollständige Problem (und immer noch sehr nützlich).

$SAT \in NPC$

Theorem

SAT ist NP-vollständig.

Beweis.

Um SAT $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.
Um zu zeigen, dass SAT vollständig ist für NP müssen wir alle Probleme aus NP auf SAT reduzieren. Dies führt hier zu weit. Die Idee mündlich... □

Literaturhinweis

Schön erklärter Beweis in [HMU].

$SAT \in NPC$

Theorem

SAT ist NP-vollständig.

Beweis.

Um SAT $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.

Um zu zeigen, dass SAT vollständig ist für NP müssen wir alle Probleme aus NP auf SAT reduzieren. Dies führt hier zu weit. Die Idee mündlich... □

Literaturhinweis

Schön erklärter Beweis in [HMU].

$SAT \in NPC$

Theorem

SAT ist NP-vollständig.

Beweis.

Um SAT $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.
Um zu zeigen, dass SAT vollständig ist für NP müssen wir alle Probleme aus NP auf SAT reduzieren. Dies führt hier zu weit. Die Idee mündlich... □

Literaturhinweis

Schön erklärter Beweis in [HMU].

$SAT \in NPC$

Theorem

SAT ist NP-vollständig.

Beweis.

Um SAT $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.
Um zu zeigen, dass SAT vollständig ist für NP müssen wir alle Probleme aus NP auf SAT reduzieren. Dies führt hier zu weit. Die Idee mündlich... □

Literaturhinweis

Schön erklärter Beweis in [HMU].

$SAT \in NPC$

Theorem

SAT ist NP-vollständig.

Beweis.

Um SAT $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.
Um zu zeigen, dass SAT vollständig ist für NP müssen wir alle Probleme aus NP auf SAT reduzieren. Dies führt hier zu weit. Die Idee mündlich... □

Literaturhinweis

Schön erklärter Beweis in [HMU].

Der weitere Weg...

Anmerkung (zur Nachbereitung)

Hat man nun erstmal ein *NP*-vollständiges Problem, so kann man - siehe den Plan oben - nun dieses benutzen, um es auf neue Probleme zu reduzieren und diese so als *NP*-vollständig nachzuweisen. Der umständliche Weg *alle NP*-Probleme auf ein neues zu reduzieren entfällt so (bzw. man kriegt dies insb. wegen der Transitivität von $\leq_p$ geschenkt). Je größer dann der Vorrat an *NP*-vollständigen Problemen ist, desto größer ist die Auswahl an Problemen, von denen man eine Reduktion auf ein neues Problem, dessen Komplexität noch unbekannt ist, versuchen kann. (Schritt 2 in obigem Plan.)

Konjunktive Normalform

Wir wollen zwei weitere Probleme als gegeben vorausetzen, dazu:

Definition (Konjunktive Normalform)

- Ein *Literal* L ist eine positive oder negative (d.h. negierte) aussagenlogische Variable, also z.B. $L = x_3$ oder $L = \neg x_2$.
- Eine *Klausel* ist eine oder-Verknüpfung von Literalen.
- Eine aussagenlogische Formel ϕ ist in konjunktiver Normalform (KNF), wenn sie die Form $\phi = (L_{11} \vee L_{21} \vee \ldots \vee L_{i_1 1}) \wedge \ldots \wedge (L_{1j} \vee L_{2j} \vee \ldots \vee L_{i_j j})$ besitzt, also eine und-Verknüpfung von Klauseln ist.

Anmerkung

Praktisch zum Modellieren: Ich hätte gerne Eigenschaft A oder B und dann noch C oder 'nicht A' und dann noch ...

Definition (CNF)

$\texttt{CNF} = \{\langle\phi\rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Formel in KNF}\}$

Definition (3CNF)

$\texttt{3CNF} = \{\langle\phi\rangle \mid \phi \in$
CNF, jede Klausel hat genau drei verschiedene Literale$\}$

Theorem

CNF und 3CNF sind NP-vollständig.

Beweis.

$\texttt{CNF}, \texttt{3CNF} \in NP$ ist klar. Man kann dann $\texttt{SAT} \leq_p \texttt{CNF}$ und $\texttt{CNF} \leq_p \texttt{3CNF}$ zeigen (siehe wieder [HMU]). □

Definition (CNF)

$\texttt{CNF} = \{\langle\phi\rangle \mid \phi$ ist eine erfüllbare aussagenlogische Formel in KNF$\}$

Definition (3CNF)

$\texttt{3CNF} = \{\langle\phi\rangle \mid \phi \in$
CNF, jede Klausel hat genau drei verschiedene Literale$\}$

Theorem

CNF und 3CNF sind NP-vollständig.

Beweis.

$\texttt{CNF}, \texttt{3CNF} \in NP$ ist klar. Man kann dann $\texttt{SAT} \leq_p \texttt{CNF}$ und $\texttt{CNF} \leq_p \texttt{3CNF}$ zeigen (siehe wieder [HMU]). □

Definition (CNF)

$\texttt{CNF} = \{\langle\phi\rangle \mid \phi$ ist eine erfüllbare aussagenlogische Formel in KNF$\}$

Definition (3CNF)

$\texttt{3CNF} = \{\langle\phi\rangle \mid \phi \in$
CNF, jede Klausel hat genau drei verschiedene Literale$\}$

Theorem

CNF und 3CNF sind NP-vollständig.

Beweis.

$\texttt{CNF}, \texttt{3CNF} \in NP$ ist klar. Man kann dann $\texttt{SAT} \leq_p \texttt{CNF}$ und $\texttt{CNF} \leq_p \texttt{3CNF}$ zeigen (siehe wieder [HMU]). □

Clique

Nun wollen wir ein Problem tatsächlich als *NP*-vollständig nachweisen...

Definition (Clique)

$\texttt{CLIQUE} = \{\langle G, k\rangle \mid G \text{ enthält einen } K^k \text{ als Teilgraphen}$

Dabei ist ein K^k eine k-Clique, d.h. ein (Teil-)Graph auf k Knoten, wobei alle Knoten paarweise miteinander verbunden sind.

Clique

Nun wollen wir ein Problem tatsächlich als *NP*-vollständig nachweisen...

Definition (Clique)

$\texttt{CLIQUE} = \{\langle G, k\rangle \mid G \text{ enthält einen } K^k \text{ als Teilgraphen}$

Dabei ist ein K^k eine k-Clique, d.h. ein (Teil-)Graph auf k Knoten, wobei alle Knoten paarweise miteinander verbunden sind.

Clique

Definition (Clique)

$\texttt{CLIQUE} = \{\langle G, k\rangle \mid G \text{ enthält einen } K^k \text{ als Teilgraphen}$

Satz

CLIQUE ist NP-vollständig.

Beweis

Als Zertifikat nehmen wir eine Menge $V' \subseteq V(G)$ von Knoten, die eine Clique bilden. Dieses Zertifikat ist polynomial in der Eingabelänge und zudem lässt sich leicht in polynomialer Zeit prüfen, ob alle Knoten verbunden sind, indem man für je zwei Knoten u, v aus V' einfach testet, ob $\{u, v\}$ eine Kante in $E(G)$ ist. Dies zeigt $\texttt{CLIQUE} \in NP$.

Clique

Definition (Clique)

$\texttt{CLIQUE} = \{\langle G, k\rangle \mid G \text{ enthält einen } K^k \text{ als Teilgraphen}$

Satz

CLIQUE ist NP-vollständig.

Beweis

Als Zertifikat nehmen wir eine Menge $V' \subseteq V(G)$ von Knoten, die eine Clique bilden. Dieses Zertifikat ist polynomial in der Eingabelänge und zudem lässt sich leicht in polynomialer Zeit prüfen, ob alle Knoten verbunden sind, indem man für je zwei Knoten u, v aus V' einfach testet, ob $\{u, v\}$ eine Kante in $E(G)$ ist. Dies zeigt $\texttt{CLIQUE} \in NP$.

CLIQUE $\in$ *NPC*

Satz

CLIQUE ist NP-vollständig.

Beweis

Nun zeigen wir noch 3CNF $\leq_p$ CLIQUE. Sei dazu $\phi = C_1 \wedge \ldots \wedge C_k$ eine Instanz von 3CNF mit k Klauseln. Seien ferner l_1^r, l_2^r, l_3^r für $r = 1, 2, \ldots, k$ die drei verschiedenen Literale in der Klausel C_r. Wir konstruieren eine Instanz (G, k) von CLIQUE wie folgt: Zu jeder Klausel $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ nehmen wir ein Tripel v_1^r, v_2^r, v_3^r in V auf. Zwei Knoten v_i^s und v_j^t sind nun genau dann miteinander verbunden, wenn $s \neq t$ gilt und die zugehörigen Literale nicht zueinander komplementär sind (d.h. das eine ein positives das andere ein negatives Literal der selben Variable ist). Der Wert k der Instanz von CLIQUE entspricht der Anzahl der Klauseln von ϕ.

CLIQUE $\in$ *NPC*

Satz

CLIQUE ist NP-vollständig.

Beweis

Nun zeigen wir noch 3CNF $\leq_p$ CLIQUE. Sei dazu $\phi = C_1 \wedge \ldots \wedge C_k$ eine Instanz von 3CNF mit k Klauseln. Seien ferner l_1^r, l_2^r, l_3^r für $r = 1, 2, \ldots, k$ die drei verschiedenen Literale in der Klausel C_r. Wir konstruieren eine Instanz (G, k) von CLIQUE wie folgt: Zu jeder Klausel $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ nehmen wir ein Tripel v_1^r, v_2^r, v_3^r in V auf. Zwei Knoten v_i^s und v_j^t sind nun genau dann miteinander verbunden, wenn $s \neq t$ gilt und die zugehörigen Literale nicht zueinander komplementär sind (d.h. das eine ein positives das andere ein negatives Literal der selben Variable ist). Der Wert k der Instanz von CLIQUE entspricht der Anzahl der Klauseln von ϕ.

CLIQUE $\in$ *NPC*

Satz

CLIQUE ist NP-vollständig.

Beweis

Nun zeigen wir noch 3CNF $\leq_p$ CLIQUE. Sei dazu $\phi = C_1 \wedge \ldots \wedge C_k$ eine Instanz von 3CNF mit k Klauseln. Seien ferner l_1^r, l_2^r, l_3^r für $r = 1, 2, \ldots, k$ die drei verschiedenen Literale in der Klausel C_r. Wir konstruieren eine Instanz (G, k) von CLIQUE wie folgt: Zu jeder Klausel $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ nehmen wir ein Tripel v_1^r, v_2^r, v_3^r in V auf. Zwei Knoten v_i^s und v_j^t sind nun genau dann miteinander verbunden, wenn $s \neq t$ gilt und die zugehörigen Literale nicht zueinander komplementär sind (d.h. das eine ein positives das andere ein negatives Literal der selben Variable ist). Der Wert k der Instanz von CLIQUE entspricht der Anzahl der Klauseln von ϕ.

CLIQUE $\in$ *NPC* - Illustration

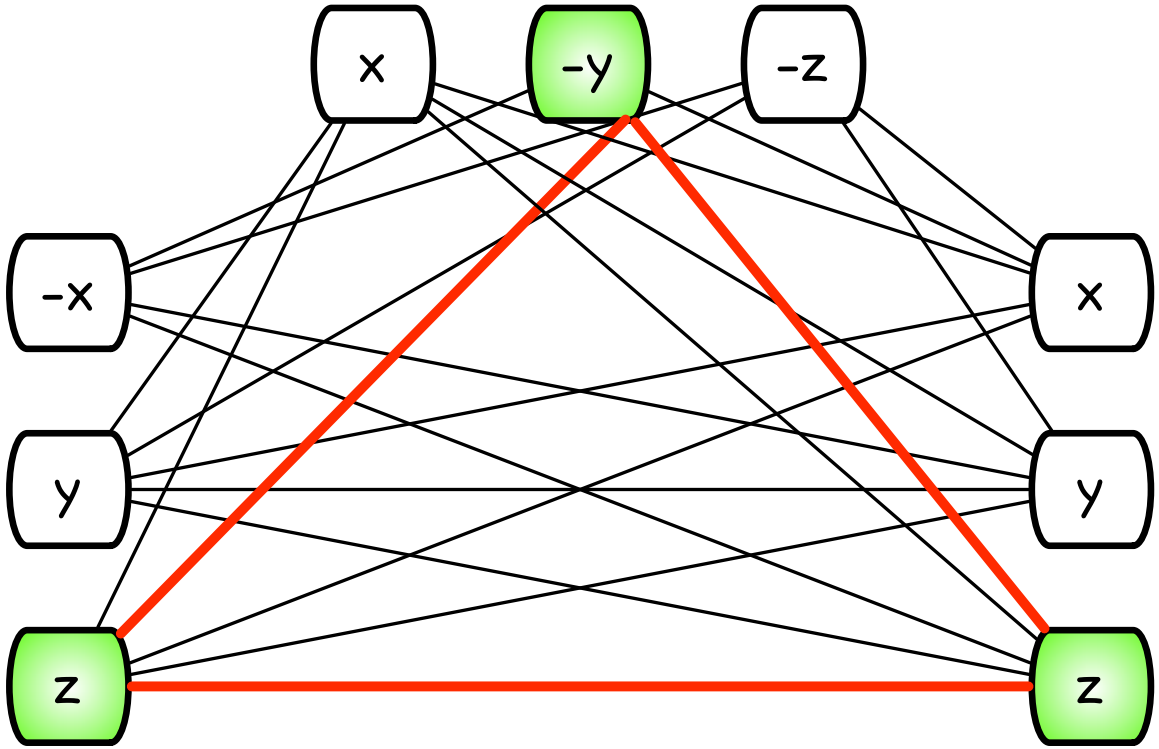
Konstruktion zu

$$\phi = (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z)$$

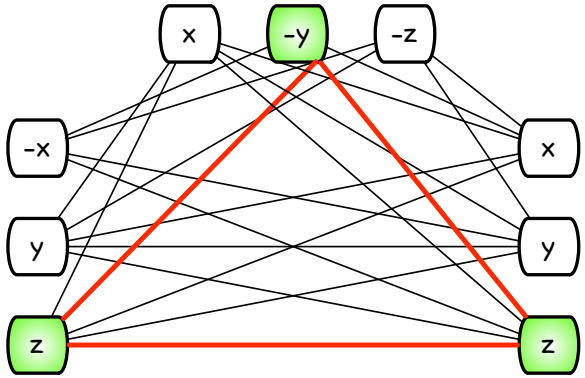
CLIQUE $\in$ *NPC* - Illustration

Konstruktion zu

$$\phi = (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z)$$



CLIQUE $\in$ *NPC* - Konstruktion in *P*



Beweis

Diese Konstruktion ist in Polynomialzeit möglich, da man durch einmal lesen der Formel die auftretenden Variablen und Klauseln kennt und so die Knoten erzeugen kann. Die Kanten erzeugt man dann schlimmstenfalls in dem man jeden Knoten mit allen Knoten der anderen Tripel vergleicht und prüft ob die zugehörigen Literale komplementär sind. Sind sie es nicht, fügt man eine Kante hinzu. Dies geht dann in $O(V^2) = O(\phi^2)$.

CLIQUE $\in$ *NPC*

Beweis

Wir müssen noch zeigen, dass dies wirklich eine Reduktion ist, der gegebene Graph also genau dann eine Clique enthält, wenn die Formel erfüllbar ist. Sei die Formel erfüllbar, dann gibt es eine Belegung die in jeder Klausel mindestens ein Literal wahr macht. Nimmt man nun aus jeder Klausel eines dieser wahren Literale und dann die jeweils zugehörigen Knoten aus den Tripeln so hat man eine k-Clique, denn es sind k Knoten (da es k Klauseln sind) und zu zwei Knoten v_i^r, v_j^s gilt $r \neq s$ (da die Literale aus verschiedenen Klauseln, die Knoten also aus verschiedenen Tripeln gewählt wurden) und ferner sind die zu den Knoten gehörigen Literale nicht komplementär, da die Belegung dann nicht beide wahr machen könnte. Die Knoten sind also durch eine Kante verbunden.

CLIQUE $\in$ *NPC*

Beweis

Wir müssen noch zeigen, dass dies wirklich eine Reduktion ist, der gegebene Graph also genau dann eine Clique enthält, wenn die Formel erfüllbar ist. Sei die Formel erfüllbar, dann gibt es eine Belegung die in jeder Klausel mindestens ein Literal wahr macht. Nimmt man nun aus jeder Klausel eines dieser wahren Literale und dann die jeweils zugehörigen Knoten aus den Tripeln so hat man eine k-Clique, denn es sind k Knoten (da es k Klauseln sind) und zu zwei Knoten v_i^r, v_j^s gilt $r \neq s$ (da die Literale aus verschiedenen Klauseln, die Knoten also aus verschiedenen Tripeln gewählt wurden) und ferner sind die zu den Knoten gehörigen Literale nicht komplementär, da die Belegung dann nicht beide wahr machen könnte. Die Knoten sind also durch eine Kante verbunden.

CLIQUE $\in$ *NPC*

Beweis

Wir müssen noch zeigen, dass dies wirklich eine Reduktion ist, der gegebene Graph also genau dann eine Clique enthält, wenn die Formel erfüllbar ist. Sei die Formel erfüllbar, dann gibt es eine Belegung die in jeder Klausel mindestens ein Literal wahr macht. Nimmt man nun aus jeder Klausel eines dieser wahren Literale und dann die jeweils zugehörigen Knoten aus den Tripeln so hat man eine k-Clique, denn es sind k Knoten (da es k Klauseln sind) und zu zwei Knoten v_i^r, v_j^s gilt $r \neq s$ (da die Literale aus verschiedenen Klauseln, die Knoten also aus verschiedenen Tripeln gewählt wurden) und ferner sind die zu den Knoten gehörigen Literale nicht komplementär, da die Belegung dann nicht beide wahr machen könnte. Die Knoten sind also durch eine Kante verbunden.

CLIQUE $\in$ *NPC*

Beweis

Wir müssen noch zeigen, dass dies wirklich eine Reduktion ist, der gegebene Graph also genau dann eine Clique enthält, wenn die Formel erfüllbar ist. Sei die Formel erfüllbar, dann gibt es eine Belegung die in jeder Klausel mindestens ein Literal wahr macht. Nimmt man nun aus jeder Klausel eines dieser wahren Literale und dann die jeweils zugehörigen Knoten aus den Tripeln so hat man eine k-Clique, denn es sind k Knoten (da es k Klauseln sind) und zu zwei Knoten v_i^r, v_j^s gilt $r \neq s$ (da die Literale aus verschiedenen Klauseln, die Knoten also aus verschiedenen Tripeln gewählt wurden) und ferner sind die zu den Knoten gehörigen Literale nicht komplementär, da die Belegung dann nicht beide wahr machen könnte. Die Knoten sind also durch eine Kante verbunden.

CLIQUE $\in$ *NPC*

Beweis.

Gibt es andersherum eine k-Clique V', so muss jeder Knoten aus einem anderen Tripel sein, da die Knoten in einem Tripel nicht miteinander verbunden sind. Wir können nun dem zu einem Knoten $v_i^r \in V'$ zugehörigem Literal l_i^r den Wert 1 zuweisen ohne dadurch dem Literal und seinem Komplement den Wert 1 zuzuweisen, da dann zwei Knoten in V' sein müssten, die nicht miteinander verbunden wären (was nicht sein kann, da V' eine Clique ist). Damit ist dann jede Klausel erfüllt, da aus jedem Tripel ein Knoten und damit aus jeder Klausel ein Literal beteiligt ist und wir haben somit eine erfüllende Belegung. Damit ist alles gezeigt. □

CLIQUE $\in$ *NPC*

Beweis.

Gibt es andersherum eine k-Clique V', so muss jeder Knoten aus einem anderen Tripel sein, da die Knoten in einem Tripel nicht miteinander verbunden sind. Wir können nun dem zu einem Knoten $v_i^r \in V'$ zugehörigem Literal l_i^r den Wert 1 zuweisen ohne dadurch dem Literal und seinem Komplement den Wert 1 zuzuweisen, da dann zwei Knoten in V' sein müssten, die nicht miteinander verbunden wären (was nicht sein kann, da V' eine Clique ist). Damit ist dann jede Klausel erfüllt, da aus jedem Tripel ein Knoten und damit aus jeder Klausel ein Literal beteiligt ist und wir haben somit eine erfüllende Belegung. Damit ist alles gezeigt. □

Weitere Probleme in *NPC*

Neben `SAT`, `CNF`, `3CNF` und `CLIQUE` gibt es viele weitere *NP*-vollständige Probleme:

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Das Knotenüberdeckungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Knotenüberdeckung der Größe k, d.h. eine Teilmenge $V' \subseteq V$ mit $|V| = k$ derart, dass für alle $\{u, v\} \in E$ $u \in V'$ oder $v \in V'$ (oder beides) gilt?

Weitere Probleme in *NPC*

Neben `SAT`, `CNF`, `3CNF` und `CLIQUE` gibt es viele weitere *NP*-vollständige Probleme:

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Das Knotenüberdeckungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Knotenüberdeckung der Größe k, d.h. eine Teilmenge $V' \subseteq V$ mit $|V| = k$ derart, dass für alle $\{u, v\} \in E$ $u \in V'$ oder $v \in V'$ (oder beides) gilt?

Weitere Probleme in *NPC*

Neben `SAT`, `CNF`, `3CNF` und `CLIQUE` gibt es viele weitere *NP*-vollständige Probleme:

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Das Knotenüberdeckungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Knotenüberdeckung der Größe k, d.h. eine Teilmenge $V' \subseteq V$ mit $|V| = k$ derart, dass für alle $\{u, v\} \in E$ $u \in V'$ oder $v \in V'$ (oder beides) gilt?

Weitere Probleme in *NPC*

Independent Set

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G ein *Independent Set* der Größe k, d.h. k Knoten bei denen keine zwei miteinander verbunden sind?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

Das Hamilton-Kreis-Problem

Gegeben ist ein ungerichteter Graph $G = (V, E)$. Besitzt G einen Hamilton-Kreis, d.h. einen einfachen Kreis, der alle Knoten aus V enthält?

Weitere Probleme in *NPC*

Independent Set

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G ein *Independent Set* der Größe k, d.h. k Knoten bei denen keine zwei miteinander verbunden sind?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

Das Hamilton-Kreis-Problem

Gegeben ist ein ungerichteter Graph $G = (V, E)$. Besitzt G einen Hamilton-Kreis, d.h. einen einfachen Kreis, der alle Knoten aus V enthält?

Weitere Probleme in *NPC*

Independent Set

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G ein *Independent Set* der Größe k, d.h. k Knoten bei denen keine zwei miteinander verbunden sind?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

Das Hamilton-Kreis-Problem

Gegeben ist ein ungerichteter Graph $G = (V, E)$. Besitzt G einen Hamilton-Kreis, d.h. einen einfachen Kreis, der alle Knoten aus V enthält?

P vs. *NP*

Zusammenhänge zwischen *P* und *NP*:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der *NP*-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine *NP*-vollständige Sprache in *P* liegt.
- Alle oben genannten Probleme sind *NP*-vollständig...

P vs. *NP*

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Alle oben genannten Probleme sind NP-vollständig...

P vs. *NP*

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Alle oben genannten Probleme sind NP-vollständig...

Take Home Message

Take Home Message

Gegeben ein Problem für das ihr einen Algortihmus entwickeln sollt. Fallen euch nach einiger Zeit und etlichem Nachdenken stets nur Algorithmen ein, die **im Prinzip den ganzen Suchraum durchgehen**, so ist das Problem vermutlich *NP*-vollständig (oder schlimmer). (Hängt natrürlich u.a. von eurer Erfahrung im Algorithmenentwurf ab.) Ein Nachweis der *NP*-Vollständigkeit ist natürlich trotzdem nett und sinnvoll... ;-)

Für Interessierte

Mehr zu NP-vollständigen Problemen findet Ihr in [HMU] und in dem Buch *Computers and Intractability* von Garey und Johnson.

Ausblick: Wie löst man die Probleme dennoch?

Wir wissen jetzt, dass es Probleme in *NPC* gibt und dass es dort viele wichtige Probleme gibt. Wir wissen aber auch, dass man diese Probleme deterministisch nur sehr schlecht (in $2^{O(n^k)}$) lösen kann.

Dennoch gibt es Möglichkeiten diese Probleme zu attackieren:

- Einschränkung der Eingabe (z.B. Bäume statt Graphen)
- Approximationsalgorithmen (man kriegt nicht das Optimum)
- Randomisierte Algorithmen (z.B. manchmal kein Ergebnis)
- Heuristiken (Laufzeit und Güte des Ergebnisses meist unklar)
- ...