Algorithmen und Datenstrukturen
Kapitel 6.2

Komplexitätstheorie
P, *NP* und *NPC*

Frank Heitmann
heitmann@informatik.uni-hamburg.de

18. November 2015

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.

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.

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.

Exkurs: Reduktionen allgemein

Exkurs (für Interessierte)

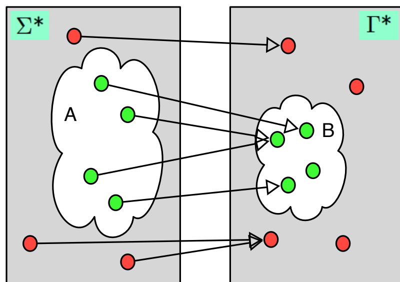
- Allgemeiner ist zu zwei Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ eine Reduktion eine Funktion $f : \Sigma^* \to \Gamma^*$ mit $x \in A$ gdw. $f(x) \in B$ für alle $x \in \Sigma^*$.
- Den Sprachen können also verschiedene Alphabete zugrunde liegen und die Reduktion muss (zunächst) 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 (für Interessierte)

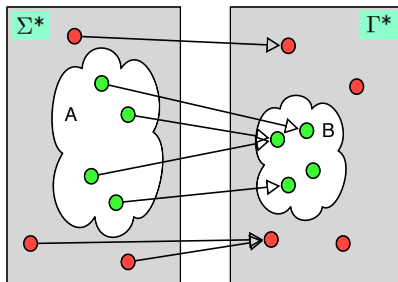
- Allgemeiner ist zu zwei Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ eine Reduktion eine Funktion $f : \Sigma^* \to \Gamma^*$ mit $x \in A$ gdw. $f(x) \in B$ für alle $x \in \Sigma^*$.
- Den Sprachen können also verschiedene Alphabete zugrunde liegen und die Reduktion muss (zunächst) 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



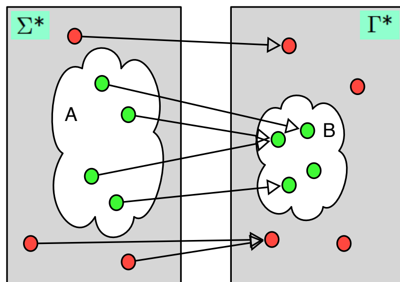
- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion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

Reduktionen: Erläuterungen



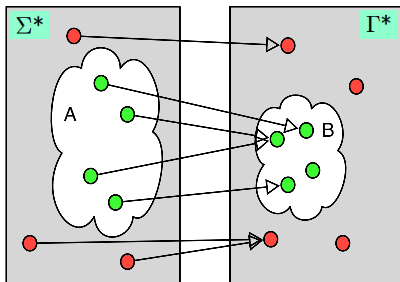
- Ja-Instanzen ($x \in A$) auf Ja-Instanzen ($f(x) \in B$) abbilden,
- Nein-Instanzen ($x \notin A$) auf Nein-Instanzen ($f(x) \notin B$).
- Gleiche Antwort auf die Fragen '$x \in A$?' und '$f(x) \in B$?'
- Viele Ja-Instanzen können auf *eine* Ja-Instanz abgebildet werden. (Daher auch als 'many-one'-Reduktion 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 (Lemma 34.3 im (Cormen))

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.
- Korrekt da: $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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.
- Korrekt da: $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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.
- Korrekt da: $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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.
- Korrekt da: $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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.
- Korrekt da: $f(x) \in L_2$ gilt gdw. $x \in L_1$ gilt.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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 (Lemma 34.3 im (Cormen))

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 (Lemma 34.3 im (Cormen))

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 (Lemma 34.3 im (Cormen))

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: $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$.

□

Unterroutinen in Polynomialzeit

Seien A und B Polynomialzeitalgorithmen, x eine Eingabe für A.

- Die Ausgabe von A ist polynomial in $|x|$. Nutzt man diese als Eingabe für z.B. B, so hat man weiterhin eine polynomiale Laufzeit. (Würde A eine superpolynomiale Ausgabe produzieren, so würde B (auf dieser als Eingabe) nicht mehr umbedingt in Polynomialzeit arbeiten.)
- Ruft B A als Unteroutine eine konstante Anzahl von Malen auf, so ist der gesamte Algorithmus immer noch in P.
- Gibt es in B hingegen eine poynomiale Anzahl von Aufrufen von A, so kann der entstehende Algorithmus eine exponentielle Laufzeit haben.

Unterroutinen in Polynomialzeit

Seien A und B Polynomialzeitalgorithmen, x eine Eingabe für A.

- Die Ausgabe von A ist polynomial in $|x|$. Nutzt man diese als Eingabe für z.B. B, so hat man weiterhin eine polynomiale Laufzeit. (Würde A eine superpolynomiale Ausgabe produzieren, so würde B (auf dieser als Eingabe) nicht mehr umbedingt in Polynomialzeit arbeiten.)
- Ruft B A als Unteroutine eine konstante Anzahl von Malen auf, so ist der gesamte Algorithmus immer noch in P.
- Gibt es in B hingegen eine poynomiale Anzahl von Aufrufen von A, so kann der entstehende Algorithmus eine exponentielle Laufzeit haben.

Unterroutinen in Polynomialzeit

Seien A und B Polynomialzeitalgorithmen, x eine Eingabe für A.

- Die Ausgabe von A ist polynomial in $|x|$. Nutzt man diese als Eingabe für z.B. B, so hat man weiterhin eine polynomiale Laufzeit. (Würde A eine superpolynomiale Ausgabe produzieren, so würde B (auf dieser als Eingabe) nicht mehr umbedingt in Polynomialzeit arbeiten.)
- Ruft B A als Unteroutine eine konstante Anzahl von Malen auf, so ist der gesamte Algorithmus immer noch in P.
- Gibt es in B hingegen eine poynomiale Anzahl von Aufrufen von A, so kann der entstehende Algorithmus eine exponentielle Laufzeit haben.

Unterroutinen in Polynomialzeit

Seien A und B Polynomialzeitalgorithmen, x eine Eingabe für A.

- Die Ausgabe von A ist polynomial in $|x|$. Nutzt man diese als Eingabe für z.B. B, so hat man weiterhin eine polynomiale Laufzeit. (Würde A eine superpolynomiale Ausgabe produzieren, so würde B (auf dieser als Eingabe) nicht mehr umbedingt in Polynomialzeit arbeiten.)
- Ruft B A als Unteroutine eine konstante Anzahl von Malen auf, so ist der gesamte Algorithmus immer noch in P.
- Gibt es in B hingegen eine poynomiale Anzahl von Aufrufen von A, so kann der entstehende Algorithmus eine exponentielle Laufzeit haben.

Probleme durch andere lösen

Satz (Lemma 34.3 im (Cormen))

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 Reduktions*algorithmus*) auch L_1. Das Problem L_1 zu lösen ist also darauf 'reduziert' worden, das Problem L_2 zu lösen.

Übergang zur *NP*-Vollständigkeit...

- Ist $A \leq_p B$, so ist A höchstens so schwierig wie B (daher die $\leq$-Notation).
- 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 (daher die $\leq$-Notation).
- 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 (daher die $\leq$-Notation).
- 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 (Theorem 34.4 im (Cormen))

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 (Theorem 34.4 im (Cormen))

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 (Theorem 34.4 im (Cormen))

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 (Theorem 34.4 im (Cormen))

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 (Theorem 34.4 im (Cormen))

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

Anmerkung (zur Nachbereitung)

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.

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.
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 $L \in NPC$.

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:

$$\phi = (x_1 \lor x_2) \land (\neg x_1 \land (\neg x_2 \lor \neg x_1))$$

Führt mit $x_1 = 0$ und $x_2 = 1$ zu

$$\begin{aligned}
\phi &= (0 \lor 1) \land (\neg 0 \land (\neg 1 \lor \neg 0)) \\
&= (0 \lor 1) \land (1 \land (0 \lor 1)) \\
&= 1 \land (1 \land 1) \\
&= 1 \land 1 \\
&= 1
\end{aligned}$$

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... □

Hinweis

Schön erklärter Beweis in *Introduction to Automata Theory, Languages, and Computation* von Hopcroft, Motwani, Ullman; Addison-Wesley 2001 (2.ed.) und Skizze im [Cormen].

$SAT \in NPC$

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... □

Hinweis

Schön erklärter Beweis in *Introduction to Automata Theory, Languages, and Computation* von Hopcroft, Motwani, Ullman; Addison-Wesley 2001 (2.ed.) und Skizze im [Cormen].

$SAT \in NPC$

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... □

Hinweis

Schön erklärter Beweis in *Introduction to Automata Theory, Languages, and Computation* von Hopcroft, Motwani, Ullman; Addison-Wesley 2001 (2.ed.) und Skizze im [Cormen].

$SAT \in NPC$

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... □

Hinweis

Schön erklärter Beweis in *Introduction to Automata Theory, Languages, and Computation* von Hopcroft, Motwani, Ullman; Addison-Wesley 2001 (2.ed.) und Skizze im [Cormen].

$SAT \in NPC$

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... □

Hinweis

Schön erklärter Beweis in *Introduction to Automata Theory, Languages, and Computation* von Hopcroft, Motwani, Ullman; Addison-Wesley 2001 (2.ed.) und Skizze im [Cormen].

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)

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

Definition (3CNF)

$\mathtt{3CNF} = \{\langle\phi\rangle \mid \phi \in$ *CNF* und jede Klausel hat genau drei verschiedene Literale$\}$

Theorem

CNF und 3CNF sind NP-vollständig.

Beweis.

$\mathtt{CNF}, \mathtt{3CNF} \in NP$ ist klar. Man kann dann $\mathtt{SAT} \leq_p \mathtt{CNF}$ und $\mathtt{CNF} \leq_p \mathtt{3CNF}$ zeigen (siehe wieder das Buch von Hopcroft et al.). □

Definition (CNF)

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

Definition (3CNF)

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

Theorem

CNF und 3CNF sind NP-vollständig.

Beweis.

$\mathtt{CNF}, \mathtt{3CNF} \in NP$ ist klar. Man kann dann $\mathtt{SAT} \leq_p \mathtt{CNF}$ und $\mathtt{CNF} \leq_p \mathtt{3CNF}$ zeigen (siehe wieder das Buch von Hopcroft et al.). □

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 und 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 das Buch von Hopcroft et al.). □

Clique

Definition (Clique)

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

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

Satz

$\texttt{CLIQUE}$ 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 $\texttt{CLIQUE} \in NP$. □

$\texttt{CLIQUE} \in NPC$

Satz

CLIQUE ist NP-vollständig.

Beweis.

Nun zeigen wir noch $\texttt{3CNF} \leq_p \texttt{CLIQUE}$. 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, m) 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 m der Instanz von CLIQUE entspricht der Anzahl k 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, m) 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 m der Instanz von CLIQUE entspricht der Anzahl k der Klauseln von ϕ. □

$\texttt{CLIQUE} \in NPC$

Satz

$\texttt{CLIQUE}$ ist NP-vollständig.

Beweis.

Nun zeigen wir noch $\texttt{3CNF} \leq_p \texttt{CLIQUE}$. Sei dazu $\phi = C_1 \wedge \ldots \wedge C_k$ eine Instanz von $\texttt{3CNF}$ 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, m) von $\texttt{CLIQUE}$ 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 m der Instanz von $\texttt{CLIQUE}$ entspricht der Anzahl k der Klauseln von ϕ. $\square$

CLIQUE $\in NPC$ - Illustration

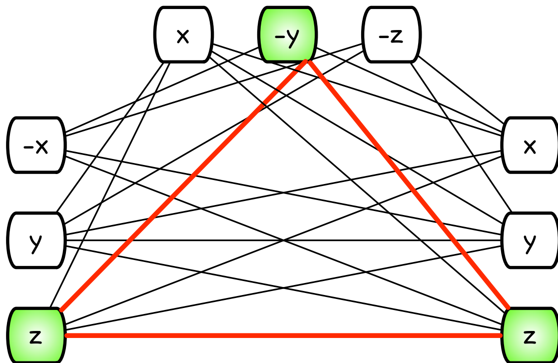
Konstruktion zu

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

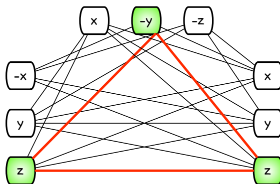
CLIQUE $\in NPC$ - Illustration

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 $O(V^2) = O(\phi^2)$. $\square$

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. □

SubsetSum

Das Teilsummenproblem

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

Der *NP*-Vollständigkeitsbeweis für SubsetSum kann im [Cormen] nachgelesen werden (dort sind auch noch weitere Beweise zu finden).

SubsetSum

Das Teilsummenproblem

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

Der *NP*-Vollständigkeitsbeweis für SubsetSum kann im [Cormen] nachgelesen werden (dort sind auch noch weitere Beweise zu finden).

Weitere Probleme in *NPC*

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

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, CLIQUE und SUBSETSUM gibt es viele weitere *NP*-vollständige Probleme:

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 $c : V \rightarrow \{1, \ldots, k\}$ 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 $c : V \to \{1, \dots, k\}$ 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?

Ausblick: Mehr zu *NPC*

Es gibt verschiedene 'Methoden', um Probleme als *NP*-vollständig nachzuweisen. Dazu gehören:

- Restriktion
- Local Replacement
- Gadget Construction

Anmerkung

Um dies besser zu verstehen, lohnt es sich ein paar Beweise selbst zu lesen (je mehr desto besser, aber irgendwann ist genug) und insbesondere Beweise selbst zu versuchen! Gute Literatur dazu ist das oben angesprochene Buch von Hopcroft et al. für den Einstieg und dann z.B. das Buch von Garey und Johnson *Computers and Intractability*.

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... ;-)

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)
- ...

Zusammenfassung - Überblick

0. Problem, Codierung, Laufzeit
1. *P* (effizient lösbar)
2. *NP* (Verifikation/Nichtdeterminismus)
3. Reduktion/*NP*-Vollständigkeit

Wiederholung: Codierung, P

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Wiederholung: NP

Definition (NP)

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

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

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Fazit

Man weiß über NP nur sehr wenig gesichert, aber viele Vermutungen kann man gut untermauern. Insb. hilft die Klasse 'unhandbare' Probleme zu klassifizieren.

Reduktion - Wiederholung

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.

NP-vollständig - Wiederholung

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*.

Reduktion - Sätze

Satz (Lemma 34.3 im (Cormen))

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

Satz

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

Theorem (Theorem 34.4 im (Cormen))

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

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.*

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.
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 $L \in NPC$.

NP-vollständige Probleme

Definition (SAT)

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

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 und jede Klausel hat genau drei verschiedene Literale$\}$

Definition (Clique)

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

...

Anhang

Anhang

Anhang

Wir zeigen noch, dass die folgenden beiden Probleme *NP*-vollständig sind:

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 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?