Formale Grundlagen der Informatik 1
Kapitel 21
P und NP

Frank Heitmann
heitmann@informatik.uni-hamburg.de

28. Juni

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

Wir haben im letzten Kapitel gesehen, dass wir statt mit Akzeptanz auch mit Entscheidbarkeit arbeiten können. Damit ist dann:

- Eine Sprache / ein Problem L ist in P, wenn ein Algorithmus / eine DTM A und ein Polynom p existiert, so dass A bei einer Eingabe der Länge n nach $p(n)$ Schritten spätestens anhält und dann korrekt akzeptiert oder ablehnt.
- Entsprechend für NP mit einem nichtdeterministischem Algorithmus / einer nichtdeterministischen TM. (Der Beweis hierfür geht analog!)

Turingmaschinen und Algorithmen

Zu Turingmaschinen und Algorithmen

Man beachte noch einmal die Ähnlichkeit zwischen Turingmaschinen und Algorithmen. (Man denke auch noch einmal an die Church-Turing-These und die erweiterte Church-Turing-These!)
Man kann also auch mit Algorithmen arbeiten, sollte sich dann aber auf elementare Operationen beschränken und einige Besonderheiten beachten (z.B. muss man Platz anders messen).

Abschlusseigenschaften

Satz (Abschlusseigenschaften)

1. *Gilt* $L_1, L_2 \in P$*, dann auch* $L_1 \cup L_2, L_1 \cap L_2, \overline{L_1}, L_1 \cdot L_2, L_1^* \in P$.
2. *Gilt* $L_1, L_2 \in NP$*, dann auch* $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, L_1^* \in NP$.

Beweis.

Beweis für Vereinigung und Komplement? Für Komplementabschluss von P: Sei $L_1 \in P$ und sei A_1 ein Algorithmus, der L_1 entscheidet. Um $\overline{L_1} \in P$ zu zeigen, ist es lediglich nötig die Ausgabe von A_1 zu invertieren. Ein Algorithmus A für $\overline{L_1}$ startet bei Eingabe x also zunächst A_1 (mit Eingabe x) und invertiert dann das Ergebnis, d.h. liefert A_1 als Rückgabe 1, so liefert A eine 0, liefert A_1 eine 0, so liefert A eine 1.
Gilt also $x \in L_1$ (in diesem Fall liefert A_1 eine 1), so liefert A eine 0, was $x \notin \overline{L_1}$ bedeutet (was wegen $x \in L_1$ korrekt ist). Entsprechend für $x \notin L_1$.
Dass A in polynomieller Zeit läuft, wenn A_1 dies tut, ist klar, da nur ein weiterer Schritt nötig ist. □

Ein Problem in P

Typisches Beispiel eines Problems in P:

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

Mit Dijkstras Algorithmus kann man dieses Problem in Polynomialzeit lösen.

Ein Problem in *NP*

Typisches Beispiel eines Problems in *NP*:

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

Wie löst man dieses Problem mit einem nichtdeterministischen Algorithmus in Polynomialzeit?

Entscheidungs- und Optimierungsprobleme

Anmerkung

Wie auch schon bei der Entscheidbarkeit macht es auch bei den Klassen P und NP keinen wichtigen Unterschied, ob man die Entscheidungsprobleme (gibt es einen (kurzen) Pfad der Länge höchstens k?) oder die Optimierungsprobleme (berechne den kürzesten Pfad) betrachtet. Sie sind mit vertretbaren Aufwand ineinander überführbar. Für uns ist es hier aber praktischer mit Sprachen und daher mit den Entscheidungsproblemen zu arbeiten.

Weitere Probleme in *NP*

Definition (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?

Folgender Algorithmus löst das Problem: Bei Eingabe G und k rate zunächst nichtdeterministisch k Knoten von G (z.B. indem ein Zähler mitgeführt wird oder indem für jeden Knoten von G geraten wird, ob er gewählt werden soll oder nicht und im Anschluss überprüft wird, dass genau k Knoten gewählt wurden). Dann überprüfe deterministisch für diese k Knoten, dass sie paarweise nicht miteinander verbunden sind. Gehe hierzu die Kanten durch und überprüfe, dass nie beide Enden einer Kante zu den gewählten Knoten gehören.

Weitere Probleme in *NP*

Definition (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?

Der eben vorgestellte Algorithmus ist korrekt, denn gibt es k Knoten, die ein Independent Set bilden, so müssen diese k Knoten bei einer Rechnung auch geraten werden und gibt es keine k Knoten, die ein Indepenednet Set bilden, so kann auch keines geraten werden. Ferner arbeitet der Algorithmus in Polynomialzeit, denn die Rate-Phase geht in $O(V)$ und die anschliessende deterministische Phase in $O(E)$, also sind wir insgesamt in $O(V + E)$, was polynomiell in der Eingabe ist. Damit ist das Problem in *NP*.

Weitere Probleme in *NP*

Definition (SAT)

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

Wir raten für jedes Aussagesymbol in ϕ eine Belegung (0 oder 1). Nach dieser Rate-Phase überprüfen wir deterministisch, ob die so geratene Belegung ϕ wahr macht. Falls ja, wird akzeptiert, sonst abgelehnt. Der Algorithmus ist korrekt (gibt es eine erfüllende Belegung, so wird sie gefunden, gibt es keine, kann keine gefunden werden) und arbeitet in Polynomialzeit, also ist das Problem wieder in *NP*.

Weitere Probleme in *NP*

Definition (Lagerhaus-Problem)

Gegeben ist ein ungerichteter Graph $G = (V, E)$, eine Entfernung $d \in \mathbb{N}$ und ein Budget $g \in \mathbb{N}$. Ist es möglich auf g Knoten Lagerhäuser zu platzieren, so dass jeder Knoten des Graphen maximal d Kanten von einem Lagerhaus entfernt ist?

Wir raten nichtdeterministisch g Knoten (so wie beim Independent Set Problem). Im Anschluss bestimmt man für jeden der gewählten Knoten z.B. mit einer Breitensuche alle Knoten, die bis zu d Kanten entfernt sind. Jeder so besuchte Knoten wird markiert (ggf. mehrfach, aber die Markierung wird nicht rückgängig gemacht). Dann wird noch überprüft, ob jeder Knoten aus G markiert ist und akzeptiert, falls ja. Korrektheit? Laufzeit?

Die Klassen P und NP

Probleme in P gelten als effizient lösbar. Man trifft etliche solcher Probleme in typischen Algorithmenvorlesungen (Sortieren, Suchen usw.) Das Ziel ist es meist, einen möglichst guten Algorithmus zu finden, also einen der statt in $O(n^3)$ dann vielleicht in $O(n^2)$ läuft. Die obigen Probleme in *NP* scheinen aber nicht effizient (in *P*) lösbar zu sein.

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! Es gibt aber eine Technik, mit der wir sagen können, dass ein Polynomialzeit-Algorithmus zumindest sehr unwahrscheinlich ist.)*

Ablauf

Der weitere Ablauf:

- Eine alternative Definition von *NP*
- Probleme in *NP* deterministisch lösen
- Technik für oben angesprochenes entwickeln

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.

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.

Definition (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?

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 Problem von eben mit einem Verifikations- und wie mit einem nichtdeterministischen Algorithmus löst und generalisiert die Idee.
- Die Verifikationssicht 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.

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 $2^{2 \cdot O(n^k)} = 2^{O(n^k)}$.

P vs. *NP*

Abgesehen von PATH sind alle anfangs betrachteten Probleme (L-PATH, Independent Set, SAT, Lagerhaus-Problem)

- 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 ...

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 Algorithmus der f berechnet als Reduktionsalgorithmus bezeichnet.

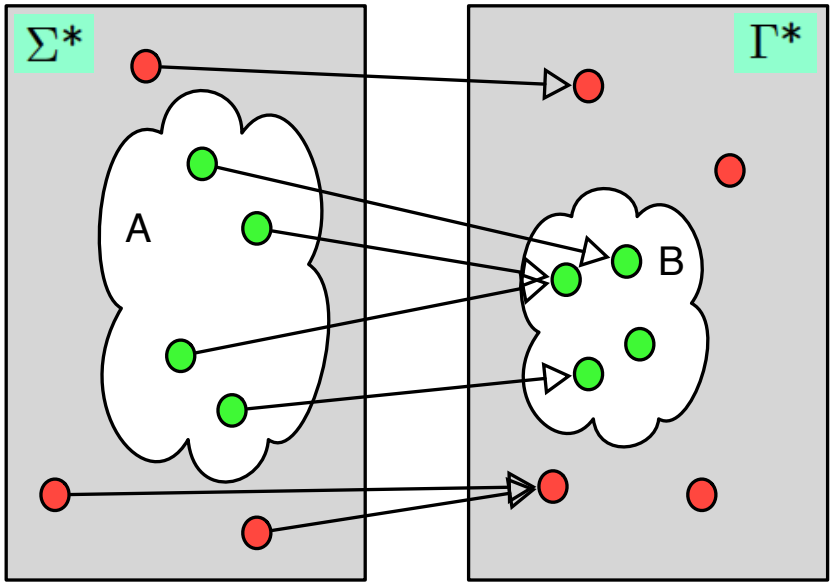
Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin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

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).

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: Hinweis

Hinweis

Eine Reduktion ist im Kern auch einfach ein Algorithmus. Es werden Probleminstanzen eines Problems in Probleminstanzen eines anderen Problems umgewandelt. Im Kern findet also eine (algorithmische) Konvertierung statt. Eine Java-Routine, die einen String in eine Zahl umwandelt, tut im Grunde ähnliches.
Einige Beispiele für Reduktionen kennen wir im Grunde schon...

Reduktionen: Beispiel

Definition (TAUT und KONT)

$$\texttt{TAUT} = \{\langle\phi\rangle \mid \phi \text{ ist eine aussagenlogische Tautologie}\}$$
$$\texttt{KONT} = \{\langle\phi\rangle \mid \phi \text{ ist eine aussagenlogische Kontradiktion}\}$$

Wir wollen $\texttt{TAUT} \leq_p \texttt{KONT}$ zeigen. Die Reduktion muss aus einer Instanz F von TAUT eine Instanz G von KONT machen. Die Reduktion ist einfach: Zu F wird $G := \neg F$ konstruiert. Zu zeigen ist nun:

1. Wenn $F \in \texttt{TAUT}$, dann $G \in \texttt{KONT}$.
2. Wenn $G \in \texttt{KONT}$, dann $F \in \texttt{TAUT}$.
3. Die Reduktion geht in Polynomialzeit.

Die sind bei diesem einfachen Beispiel alle klar, mündlich ...

Motivation

Der Sinn bei den Reduktion ist (u.a.), dass wir nun Probleme durch Algorithmen für andere Probleme lösen können. Im obigen Beispielen:

- Um eine Formel auf Tautologie zu prüfen, können wir nun
- die Formel algorithmisch durch die Reduktion in eine andere Formel umwandeln und
- diese auf Unerfüllbarkeit testen.

Hat man einen Unerfüllbarkeitstester (aber keinen Tautologietester), so kann man den nun nutzen (und hat damit jetzt auch einen Tautologietester).

Reduktionen: Beispiel

Definition (FOLG und KONT)

$$\begin{aligned} \texttt{FOLG} &= \{\langle \phi, \psi \rangle \mid \phi \models \psi \text{ (aussagenlogisch)}\} \\ \texttt{KONT} &= \{\langle \phi \rangle \mid \phi \text{ ist eine aussagenlogische Kontradiktion}\} \end{aligned}$$

Eine Reduktion $\texttt{FOLG} \leq_p \texttt{KONT}$ gelingt indem (ϕ, ψ) auf $\phi \wedge \neg\psi$ abgebildet wird. Zu zeigen ist wieder

1. Wenn $(\phi, \psi) \in \texttt{FOLG}$, dann $\phi \wedge \neg\psi \in \texttt{KONT}$.
2. Wenn $\phi \wedge \neg\psi \in \texttt{KONT}$, dann $(\phi, \psi) \in \texttt{FOLG}$.
3. Die Reduktion geht in Polynomialzeit.

Die sind bei diesem Beispiel wieder alle bekannt/klar, mündlich ...

Reduktionen: Beispiel

Definition (3CNF)

Sei 3CNF die Menge der (Kodierungen von) erfüllbaren aussagenlogischen Formeln, die in KNF sind und derart, dass jede Klausel genau drei verschiedene Literale enthält.

Definition (Clique)

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

Wir wollen $\texttt{3CNF} \leq_p \texttt{CLIQUE}$ zeigen ...

3CNF $\leq_p$ CLIQUE - Die Reduktion

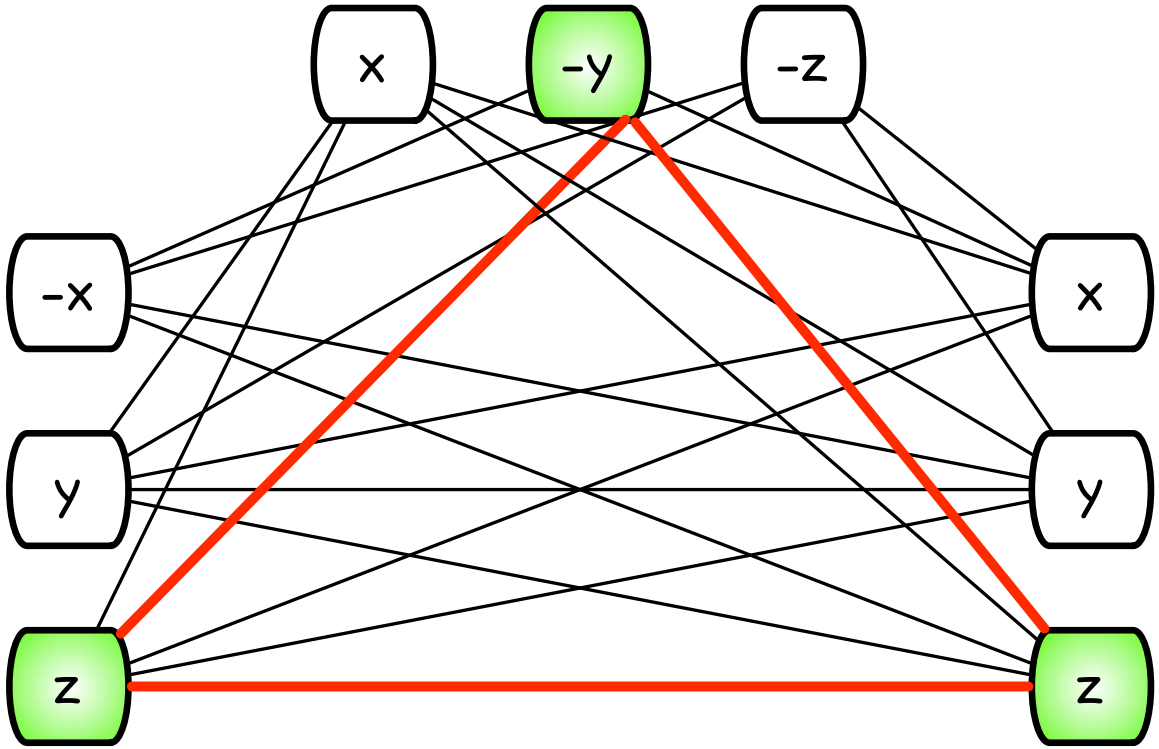
Beweis

Wir zeigen 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.

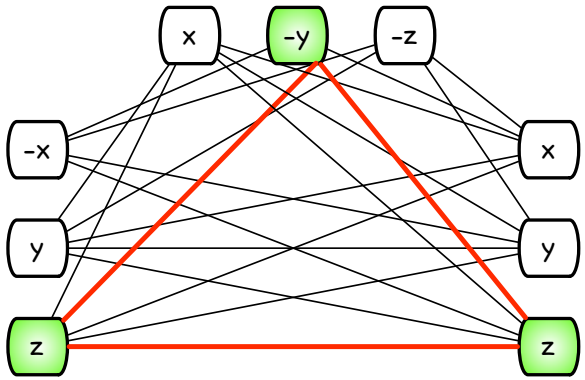
3CNF $\leq_p$ CLIQUE - Illustration

Konstruktion zu

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



3CNF $\leq_p$ CLIQUE - 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 $O(V^2) = O(\phi^2)$.

3CNF $\leq_p$ CLIQUE - Korrektheit

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.

3CNF $\leq_p$ CLIQUE - Korrektheit

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. □

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* $L_1 \in P$*.*

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.

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, führen wir nächstes Mal mit Hilfe der Reduktionen den Begriff der NP-Vollständigkeit ein.