Ludwig-Maximilians-Universität München

WS 2015/2016

Martin Hofmann, Ulrich Schöpp

Komplexitätstheorie

Vorlesungsmitschrieb von

Philipp Moers
<p.moers@campus.lmu.de>
<soziflip@gmail.com>

Last updated: 15. November 2015, 16:37

Zusammenfassung

Die Komplexitätstheorie beschäftigt sich mit der Klassifikation von Algorithmen und Berechnungsproblemen nach ihrem Ressourcenverbrauch, z.B. Rechenzeit oder benötigtem Speicherplatz. Probleme mit gleichartigem Ressourcenverbrauch werden zu Komplexitätsklassen zusammengefasst. Die bekanntesten Komplexitätsklassen sind sicherlich P und NP, die die in polynomieller Zeit deterministisch bzw. nicht-deterministisch lösbaren Probleme umfassen.

P und NP sind jedoch nur zwei Beispiele von Komplexitätsklassen. Andere Klassen ergeben sich etwa bei der Untersuchung der effizienten Parallelisierbarkeit von Problemen, der Lösbarkeit durch zufallsgesteuerte oder interaktive Algorithmen, der approximativen Lösung von Problemen, um nur einige Beispiele zu nennen.

Anmerkung

Dies ist ein inoffizieller Vorlesungsmitschrieb. Als solcher erhebt er keinen Anspruch auf (NP-) Vollständigkeit oder Korrektheit. Nutzung, Anmerkungen und Korrekturen sind jedoch durchaus erwünscht!

Vorlesungs-Website: `http://www.tcs.ifi.lmu.de/lehre/ws-2015-16/kompl`

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Theoretische Informatik, Berechenbarkeit und insbesondere Komplexitätstheorie ist der Informatiker-Shit schlechthin. Let's do it!

1.2 Literatur

Die Vorlesung basiert hauptsächlich auf folgendem Buch:

- Bovet, Crescenzi. Introduction to the Theory of Complexity. Prentice Hall. New York. 1994.

Weiterhin ist folgende Literatur gegeben:

- C. Papadimitriou. Computational Complexity. Addison-Wesley. Reading. 1995.
- I. Wegener. Komplexitätstheorie: Grenzen der Effizienz von Algorithmen. Springer. 2003.
- S. Arora und B. Barak. Complexity Theory: A Modern Approach.

Zur Motivation:

- Heribert Vollmer. Was leistet die Komplexitätstheorie für die Praxis? Informatik Spektrum 22 Heft 5, 1999.

- Stephen Cook: The Importance of the P versus NP Question. Journal of the ACM (Vol. 50 No. 1)

Vorlesung vom 12.10.15

2 Turingmaschinen, Berechenbarkeit und Komplexität

2.1 Turingmaschinen

Definition

Eine **Turingmaschine** T mit k Bändern ist ein 5-Tupel

$$T = (Q, \Sigma, I, q_0, F)$$

- Q ist eine endliche Menge von Zuständen
- Σ ist eine endliche Menge von Bandsymbolen, $\square \in \Sigma$
- I ist eine Menge von Quintupeln der Form (q, s, s', m, q') mit $q, q' \in Q$ und $s, s' \in \Sigma^k$ und $m \in \{L, R, S\}^k$
- $q_0 \in Q$ Startzustand
- $F \subseteq Q$ Endzustände

$\square$ ist das Leerzeichen oder **Blanksymbol**.

T heißt **deterministisch** genau dann, wenn für jedes $q \in Q$ und $s \in \Sigma^k$ genau ein Quintupel der Form $(q, s, \_, \_, \_) \in I$ existiert. Sonst heißt T **nichtdeterministisch**.

Eine Turingmaschine heißt **Akzeptormaschine** genau dann, wenn zwei Zustände $q_A, q_R \in F$ speziell markiert sind. q_A signalisiert Akzeptanz, q_R signalisiert Verwerfen der Eingabe.

Eine Turingmaschine heißt **Transducermaschine** genau dann, wenn ein zusätzliches Band ausgezeichnet ist (das Ausgabeband).

Beispiel

Akzeptormaschine T für Sprache $L = \{0^n 1^n | n \geq 0\}$ wobei $\Sigma = \{0, 1\}, Q = \{q_0, \ldots q_4\}$

T wird deterministisch sein. $T = (Q, \Sigma, I, q_0, F), q_A = q_1, q_R = q_2, F = \{q_1, q_2\}, k = 2$

q	s_1	s_2	s_1'	s_2'	m_1	m_2	q'
q_0	□	□	□	□	S	S	q_1
q_0	0	□	0	0	R	R	q_3
q_0	1	□	1	□	S	S	q_2
q_3	□	□	_	_	_	_	q_2
q_3	0	□	0	0	R	R	q_3
q_3	1	□	1	□	S	L	q_4
q_4	0	0	_	_	_	_	q_2
q_4	1	0	1	0	R	L	q_4
q_4	0	□	□	□	S	S	q_1
_	_	_	_	_	_	_	q_2

Die **globale Konfiguration** (oder der **Zustand**) einer Turingmaschine beinhaltet die Beschriftung aller Bänder, den internen Zustand ($\in Q$) und die Positionen aller k

Lese-/Schreibköpfe. Globale Konfigurationen können als endliche Wörter über einem geeigneten Alphabet (z.B. $\{0,1\}$) codiert werden.

Eine Turingmaschine **akzeptiert** eine Eingabe genau dann, wenn eine Berechnungsfolge ausgehend von dieser Eingabe existiert und in einem Zustand aus F endet.

Eine Turingmaschine **akzeptiert** eine Sprache $L \subseteq (\Sigma \setminus \{\square\})^*$ falls gilt:

$$\text{Die Turingmaschine akzeptiert } w \Leftrightarrow w \in L$$

Eine Turingmaschine **entscheidet** eine Sprache $L \subseteq (\Sigma \setminus \{\square\})^*$ genau dann, wenn sie sie akzeptiert und eine/die Berechnung in q_A endet.

Vorlesung vom 15.10.15

Zu **Mehrband-Turingmaschinen**:

Bisher waren die Bänder beidseitig unendlich. Ab jetzt und im Buch sind sie nur noch einseitig unendlich.

Satz

Eine Mehrband-Turingmaschine mit k Bändern kann durch eine Einband-Turingmaschine simuliert werden.

Dies benötigt quadratischen Mehraufwand.

Beweis

Die Beweisidee nutzt für das alte Alphabet Σ das neue Alphabet $\Sigma^k \times \{0,1\}^k$, das die Zeichen auf den Bändern und, ob der Lese-/Schreibkopf an dieser Position steht, speichert.

q.e.d.

Definition

Eine **universelle Turingmaschine** erhält als Eingabe (M, x), wobei M die Beschreibung einer Turingmaschine in geeignetem Binärformat und x die Eingabe für M ist. Sie berechnet dann die Ausführung von M auf x.

2.2 Halteproblem

Definition

Gegeben Turingmaschine und Eingabe (M, x). Das Problem, zu entscheiden, ob M angewendet auf x hält oder nicht, heißt **Halteproblem**.

Satz

Das Halteproblem ist unentscheidbar.

Beweis

Angenommen, es gäbe eine Turingmaschine M_{HALT}, die das Halteproblem entscheidet.

Dann könnten wir auch eine neue Turingmaschine M_D konstruieren:
Simuliere Eingabe M auf M selbst und schaue, ob sie hält. Falls ja, dann gehe in Endlosschleife. Falls nicht, halte an.

Für $M = M_D$ ergibt sich nun ein Widerspruch: Falls sie hält, hält sie nicht. Falls sie nicht hält, hält sie.

q.e.d.

2.3 Rechenzeit

Definition

Die **Rechenzeit** definiert man wie folgt:

Gegeben eine Turingmaschine M und Eingabe x.

$TIME_M(x)$ ist die Dauer (Anzahl der Schritte) der Berechnung von M auf x.

Im Beispiel der Maschine für $L = \{0^n 1^n | n \geq 0\}$ ist $TIME_M(x) = |x|$ (Länge des Strings).

Satz

Das **Speedup-Theorem** besagt, dass zu jeder Turingmaschine M eine äquivalente Turingmaschine M' konstruiert werden kann, sodass

$$TIME_{M'}(x) \leq \frac{1}{k} * TIME_M(x)$$

wobei $k \in \mathbb{N} \setminus \{0\}$ fest gewählt ist.

Zum Beispiel ist bei $k = 7$ die neue Turingmaschine siebenmal so schnell.

Beweis

Gegeben M mit Alphabet Σ.

Dann wird M' mit Alphabet Σ^k konstruiert. Ein Symbol von M' repräsentiert k aufeinanderfolgende Symbole von M, d.h. M' kann k Schritte von M ein einem einzigen ausführen.

q.e.d.

Anmerkung:

Die Schritte werden in der Praxis also schon aufwändiger, die definierte Metrik $TIME$ erfasst das nur nicht. No Magic here.

Definition

Sei $f : \mathbb{N} \to \mathbb{N}$.

Dann definieren wir $DTIME(f)$ als Menge aller Entscheidungsprobleme (oder Berechnungsprobleme) A, zu denen eine deterministische Turingmaschine M existiert, sodass M A entscheidet und die Rechenzeit in $\mathcal{O}(f(n))$ liegt.

$$DTIME(f) = \{A \mid \exists M : M \text{ entscheidet } A \text{ und } \forall x \in \Sigma^* : TIME_M(x) = \mathcal{O}(f(|x|))\}$$

Satz

Matrixmultiplikation liegt in $\mathcal{O}(n^3)$, also in $DTIME(\sqrt{n}^3)$, wenn die Länge der Matrix auf dem Band n ist. Sie liegt sogar in $\mathcal{O}(n^{2.78})$

Offen ist die Frage, ob sie in $DTIME(\sqrt{n}^2)$ liegt.

2.4 Komplexitätsklassen

Definition

Eine Menge der Form $DTIME(f(n))$ heißt **deterministische Zeitkomplexitätsklasse**. Analog heißt $NTIME(f(n))$ für nichtdeterministische Turingmaschinen **nichtdeterministische Zeitkomplexitätsklasse**.

Wir betrachten zu gegebener Funktion $f : \mathbb{N} \to \mathbb{N}$ durch Turingmaschine M folgenden Algorithmus:

Rechne M auf Eingabe M selbst für $f(M)$ Schritte. Falls M sich bis dahin akzeptiert, verwerfe die Eingabe. Falls sie sich verwirft oder bis dahin nicht gehalten hat, akzeptiere die Eingabe.

Das durch diesen Algorithmus beschriebene Problem

$$K_f = \{M \mid M \text{ akzeptiert sich selbst nicht in höchstens } f(|M|) \text{ Schritten.}\}$$

ist "offensichtlich" entscheidbar. Die Rechenzeit für diese Entscheidung muss aber im

allgemeinen $f(|M|)$ übersteigen.

Wäre M_f eine Turingmaschine, die K_f entscheidet und außerdem $TIME_{M_f} \leq f(|x|)$ für alle x, dann führt die Anwendung von M_f auf M_f selbst zum Widerspruch (wie beim Halteproblem).

f muss dazu selbst in Zeit $\mathcal{O}(f(n))$ berechenbar und monoton steigend sein. Man nennt f dann **zeitkonstruierbar**.

Durch geschickte Ausnutzung dieses Arguments erhält man den **Zeit-Hierarchie-Satz**:

Satz

Falls $f : \mathbb{N} \to \mathbb{N}$ zeitkonstruierbar ist, dann gilt:

$$DTIME(f(n)) \subset DTIME(f(n) * \log^2(f(n)))$$

wobei $\subset$ eine echte Teilmengenbeziehung bezeichnet.

Anmerkung:

"Vernünftige" Funktionen wie 2^n, $\log(n)$, $\sqrt{n}$ etc. sind zeitkonstruierbar.

Satz

Nach Borodin und Trakhtenbrot gilt das **Gap-Theorem**:

Für eine totale, berechenbare Funktion $g : \mathbb{N} \to \mathbb{N}$ mit $g(n) \geq n$ gibt es immer eine totale, berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$, sodass gilt:

$$DTIME(f) = DTIME(g \circ f)$$

Es gibt also in der Hierarchie der Komplexitätsklassen beliebig große Lücken.

Definition

Wichtige Komplexitätsklassen:

$$P = \bigcup_{k \geq 1} DTIME(n^k)$$

$$E = \bigcup_{k \geq 1} DTIME(2^{kn})$$

$$EXP = \bigcup_{k \geq 1} DTIME(2^{n^k})$$

Nach dem **Zeit-Hierarchie-Satz** gilt:

$$P \subset E \subset EXP$$

Vorlesung vom 19.10.15

Definition

Nichtdeterministische Zeitkomplexität Sei T eine nichtdeterministische Turingmaschine.

Für $x \in \Sigma$ ist $NTIME_T(x)$

1. definiert genau dann, wenn alle Berechnungen von T auf x halten
2. Falls definiert und $x \in L(T)$ (d.h. es gibt eine akzeptierende Berechnung von T auf x) definiert als Länge der kürzesten akzeptierenden Berechnungen von T auf x.
3. Falls überhaupt definiert $x \notin L(T)$ so ist $NTIME_{(x)}$ die Länge der kürzesten Berechnung.

Definition

Nichtdeterministische Komplexitätsklassen

$$NTIME(f(n)) = \{L | \exists T \text{ mit } L(T) = L \text{ und } NTIME_T(x) = \mathcal{O}(f(|x|))\}$$

Es gibt einen nichtdeterministischen Zeithierachiesatz.

$$NP = \bigcup_{k \geq 1} NTIME(n^k)$$

$$NE = \bigcup_{k \geq 1} NTIME(2^{kn})$$

$$NEXP = \bigcup_{k \geq 1} NTIME(2^{n^k})$$

Nichtdeterminismus kann durch erschöpfende Suche deterministsch simuliert werden.
z.B. $NP \subseteq EXP$

Allgemein:

$$NTIME(f(n)) \subseteq DTIME(2^{\mathcal{O}(f(n)}))$$

(zeitkonstruierbar)

2.5 polynomielle Verifizierbarkeit

Definition

Charakterisierung von NP durch **polynomielle Verifizierbarkeit** PV.

$L \subseteq \Sigma^*.L \in PV$ genau dann, wenn

$\exists L' \in P$ sodass gilt $x \in L \Leftrightarrow \exists z$ "Loesung" mit $|h| \leq L'$

wobei p ein Polynom ist

Satz

$$NP = PV$$

Beweis

"$\subseteq$":

$L \in NP$. Sei T eine nichtdeterministische Turingmaschine für L mit Laufzeit $p(n)$.

$$L' = \{(x,y)|y \text{ codiert eine akzeptierende Berechnung von } T \text{ auf } x\}$$

1. $x \in L \Leftrightarrow \exists y : |y| \leq (p(|x|))^2.(x,y,) \in L'$
2. $L' \in P$

"$\supseteq$":

Gegeben $L, L' \in P$.

Eine nichtdeterministische Turingmaschine T für L rät zunächst y und prüft dann $(x,y) \in L'$

q.e.d.

EXP im Gegensartz zu NP bzw. PV umfasst auch Probleme mit exponentiell großen Lösungen bzw solchen wo die Verifikation einer Lösung einen exponentiellen Aufwand macht.

3 NP und P

3.1 Padding

Definition

Sei $L \subseteq \Sigma^*$ eine Sprache.

$$padd(L) = \{1^l O x | x \in L, l = 2^{|x|}\}$$

Satz

Es gilt: $L \in DTIME(f(s^n))$, dann ist

$$padd(L) \in DTIME(f(n))$$

Blase f zeitkonstant und insbesondere $f(n) \geq n$

Beweis

Sei T eine deterministische Turingmaschine für L und $DTIME_T(x) \leq \in xf(2^n)$

Die folgende Maschine T' entscheided $padd(L)$:

Gegeben Eingabe y, schreibe $y + 1^l Ox$ und prüfe ob $l = 2^{|x|}$ geht in Zeit $\mathcal{O}(|y|)$

Aufwand: $cf(2^{|x|}) \leq c * f(|y|)$

Gesamtaufwand: $\leq c * f(|y|) + |y| = \mathcal{O}(f(|y|))$

falls $f(n) \geq n$ (zeitkonstruierbar)

q.e.d.

Satz

Umgekehrt gilt auch:

Wenn $padd(L) \in DTIME(f(n))$ dann $L \in DTIME(f(s^{n+1}))$

Beweis

Sei T eine Turingmaschine für $padd(L)$ mit $DTIME_T(y) \leq cf(|y|)$

Wir bauen eine Maschine für L: Gegeben Eingabe x, bilde $y = 1^{2^{|x|}} Ox$.

Aufwand: $\mathcal{O}(2^{|x|})$ Setze T auf y an. Aufwand: $c * f(|y|) = c * f(2^{|x|} + |x| + 1)$

Gesamtaufwand: $\mathcal{O}(f(2^{|x|+1}))$

q.e.d.

3.2 Was wenn P = NP

Satz

Folgerung:

$$P = NP \Rightarrow E = NE$$

Beweis

Sei $P = NP$ und $L \in NE$ und T eine Maschine mit Aufwand n^{kk} wobei k fest, n Länge der Eingabe.

$L \in NTIME(n^{nk}) = NTIME((2^n)^k)$ Also $padd(L) \in NTIME(2^k)$ also $padd(L) \in NP$ und nach Annahme $padd(L) \in P$

Also $padd(L) \in DTIME(n^{k'})$ also $L \in DTIME((2^{n+1})^{k'}) = DTIME(2^{k'n+k'}) = DTIME(2^{k'n}) \subseteq E$

q.e.d.

Slogan: Gleichheit von Komplexitätsklassen vererbt sich nach oben.

Mit anderen Paddingfunktion zeigt man ebenso:

$$P = NP \Rightarrow EXP = NEXP$$

$$E = NE \Rightarrow EXP = NEXP$$

Kontrapositiv ausgedrückt:

$$E \neq NE \Rightarrow P \neq NP$$

etc.

Es koennte sein, dass $P \neq NP$ aber doch $E = NE$.

Slogan: Trennung von Komplexitätklassen vererbt sich (durch Padding) von oben nach unten.

3.3 Sogenannte Effizienz

P wird gemeinhin gleichgesetzt mit "effizient loesbar".

Wachstumsverhalten:

p Polynom. $\Rightarrow$

$$\exists c > 0 : p(2n) \leq c * p(n)$$

Bei Verdopplung des Inputs wird der Output also ver-c-facht.

Häufig hat eine Bruteforce-Lösung ("stures Durchprobieren") exponentiellen und eine echte algorithmische Lösung hat polynomiellen Aufwand.

Vorlesung vom 09.11.15

3.4 Polynomialzeitreduktionen

Definition

$f : \Sigma^* \to \Sigma^*$ beziehungweise für Binärkodierung $f : \mathbb{N} \to \mathbb{N}$

ist in FP (eine **in Polynomialzeit berechenbare Funktion**) genau dann, wenn eine polynomialzeitbeschränkte (deterministische) Transducer-Maschine existiert, die f berechnet.

Gegeben Input $x \in \Sigma^*$, dann hält die Maschine nach $\leq p(|x|)$ Schritten mit Ergebnis $f(x)$, wobei p ein Polynom ist.

Beispiel

Musterbeispiele:

- Alle Polynome sind in FP, zum Beispiel $f(x) = x^3 + 10x^2 + x$
- Charakteristische Funktionen aller Probleme in P sind in FP.
- Matrixmultiplikation ist in FP.

Gegenbeispiele:

- $f(x) = 2^x$ ist nicht in FP, denn $|2^x| = x + 1 \geq 2^{|x|+1} + 1 = \Omega(2^{|x|})$

- Charakteristische Funktionen von Problemen in $EXP \setminus P$ sind nicht in FP.

Wahrscheinliches Gegenbeispiel:

$f(n) =$ größter Teiler von n außer n selbst. Algorithmus: Alle Zahlen von 1 bis n durchlaufen und testen, immer merken, wenn größerer Teiler gefunden.
Laufzeit: $\Omega(n) = \Omega(2^{|n|})$ (Länge der Eingabe statt Zahl selbst)

Satz

FP ist unter Komposition (Hintereinanderausführung) abgeschlossen.

Beweis

Seien $f : \Sigma^* \to \Sigma^*, g : \Sigma^* \to \Sigma^* \in FP$

Wir definieren $h(x) = g(f(x))$

Programm für h: $y = f(x);\ z = g(y);\ return\ z;$
Gesamtaufwand: $O(p_f(|x|) + p_g(p_f(|x|))$, wobei p_f und p_g Polynome sind, die die Laufzeit für Algorithmen für f bzw. g beschränken.

q.e.d.

Definition

Polynomielle Reduzierbarkeit

Seien $L_1, L_2 a \subseteq \Sigma^*$.

Wir sagen L_1 ist polynomiell auf L_2 reduzierbar (FP-**reduzierbar**) genau dann, wenn $f \in FP$ existiert, sodass $x \in L_1 \Leftrightarrow f(x) \in L_2$.

Aus Algorithmus für L_2 erhält man einen für L_1, indem man $f(x)$ berechnet und prüft, ob das Ergebnis in L_2 liegt.

In Zeichen: $L_1 \leq_p L_2$ oder $L_1 \leq L_2$, auch $f : L_1 \leq L_2$

Beispiel

3COL $= \{G = (V, E) \mid G$ kann mit 3 Farben gefärbt werden, d.h. $\exists c$: $V \to \{r, g, b\}$ sodass $\forall (u, u') \in E : c(v) \neq c(u')\}$

SAT $= \{\phi \mid \phi$ aussagenlogische Formel die erfüllbar ist$\}$

Behauptung: 3COL $\leq$ SAT

$$f((V,E)) = \left(\bigwedge_{v \in V} \bigvee_{y \in \{r,g,b\}} x_{v,y} \right) \wedge$$

$$\left(\bigwedge_{v \in V} \bigwedge_{c \in \{r,g,b\}} \bigwedge_{c' \in \{r,g,b\}} x_{v,c} \to \neg x_{v,c'} \right) \wedge$$

$$\left(\bigwedge_{(v,v') \in E} \bigwedge_{y \in \{r,g,b\}} x_{v,y} \to \neg x_{v',y} \right)$$

Offensichtlich ist $f \in FP$ und $G \in$ 3COL $\Leftrightarrow f(G) \in$ SAT ,
also 3COL $\leq$ SAT.

Beispiel

KNFSAT := wie SAT, aber auf konjunktive Normalform eingeschränkt.

Es gilt trivialerweise KNFSAT $\leq$ SAT, aber auch SAT $\leq$ KNFSAT (Durch

Einführung von Abkürzungen von Teilformeln).

Beispiel

3SAT := KNFSAT eingeschränkt auf Klauseln mit 3 Literalen.

Es gilt KNFSAT $\leq$ 3SAT.

Man kennt keine Reduktion von 3SAT auf 2SAT.

Beispiel

NODE-COVER := $\{G = (V, E), n \mid \exists U \subseteq V : |U| \leq n$ und $\forall (v, v') \in E : v \in U \vee v' \in U\}$

Es gilt NODE-COVER $\leq$ KNFSAT.

und - was schwieriger zu zeigen ist - KNFSAT $\leq$ NODE-COVER:

Gegeben: KNF ϕ mit m Variablen $x_1 \ldots x_m$ und k Klauseln $C_1 \ldots C_k$ wobei $C_j = l_{j,1} \vee \cdots \vee l_{j,k}$
Falls 3SAT, so sind alle $k_j = 3$.
Die $l_{j,i}$ sind Literale, d.h. negierte oder nicht-negierte Variablen.

Wir konstruieren Graphen $G = (V, E)$ wie folgt:

- Für jede Variable x_t zwei Knoten $x_t, \neg x_t$
- Für jede Klausel C_j k_j Knoten $(l_{j,1}) \ldots (l_{j,k})$,
  Insgesamt $2m\Sigma_{j=1}^{k} k_j$ Knoten
- Kanten:
  - $(x_t, \neg x_t)$

– Vollständiger Graph für $l_{j,1}$ bis $l_{j,k}$

– $(x_t, l_{j,i})$ bzw. $(\neg x_t, l_{j,i})$, falls $l_{j,i} = x_t$ bzw. $l_{j,i} = \neg x_t$

3.5 NP-Härte und NP-Vollständigkeit

Definition

$L \subseteq \Sigma^*$ ist NP-**hart** (NP-schwer, NP-schwierig) genau dann, wenn

$$\forall L' \in NP : L' \leq_p L$$

Satz

HALT (Halteproblem) ist NP-hart.

Beweis

Gegeben: $L' \in NP$. Baue deterministische Turingmaschine M, sodass $M(x)$ hält genau dann, wenn $x \in L'$ Brute-force Suche, Laufzeit exponentiell.
$x \in L' \Leftrightarrow (M, x) \in$ HALT
also ist $f(x) = (M, x)$ eine Reduktion von L' auf HALT $f : L' \leq$ HALT .

q.e.d.

Definition

$L \subseteq \Sigma^*$ ist **NP-vollständig** genau dann, wenn L NP-hart ist und $L \in NP$.

Satz

HALT $\notin NP$.

Satz

Satz von Cook

SAT ist NP-vollständig.

Beweis

SAT $\in NP$: trivial.

Sei $L \in NP$ gegeben und o.B.d.A M eine nichtdeterministische Turingmaschine für L mit einem Band $M = (\Sigma, Q, q_0, F, I)$ und p ein Polynom, das die Laufzeit von M beschränkt. Gegeben weiterhin $x = x_1 \dots x_n$ Input.

Gesucht: aussagenlogische Formel $\phi = f(x)$, sodass ϕ erfüllbar ist genau dann, wenn M akzeptiert x. q muss aus x in polynomieller Zeit berechenbar sein, d.h. $f \in FP$.

M akzeptiert x genau dann, wenn eine akzeptierende Berechnung von M auf x existiert. Solch eine Berechnung hat höchstens $p(n)$ Schritte und o.B.d.A genau $p(n)$ Schritte. Die Bandbeschriftung zu jedem dieser $p(n)$ Schritte besteht aus höchstens $p(n)$ Symbolen und o.B.d.A genau $p(n)$ Symbolen.

Die Formel ϕ verwendet die Variablen

- Q_t^i: Zur Zeit t ist M im Zustand i.
- $P_{s,t}^i$: Zur Zeit t enthält Bandposition s das i. Symbol.
- $S_{s,t}$: Zur Zeit t ist der Kopf in Position s.

$$q = A \wedge B \wedge C \wedge D \wedge E \wedge F$$

Details im Buch...

q.e.d.

3SAT, NODE-COVER sind auch NP-vollständig.

Allgemein gilt: L NP-vollständig und $L' \in NP, L \leq L'$ so folgt L' NP-vollständig.

Anmerkung: $\leq$ ist transitiv, da FP unter Komposition abgeschlossen ist.

Anmerkung: 3COL, TRAVELINGSALESMAN, SUBSETSUM etc. sind auch NP-vollständig.

Vorlesung vom 12.11.15

3.6 Etwas zwischen P und NP

Satz

Satz von Ladner

Falls $P \neq NP$, dann

$$\exists A \in NP \setminus P : A \text{ nicht NP-vollständig.}$$

A liegt also "echt" zwischen P und NP-vollständig.

Definition

Diagonalisierung

Um zu zeigen, dass eine Sprache A nicht in einer Klasse $\mathcal{C}$ ist, beziehungweise um solch ein A zu konstruieren, kann man eine effektive (FP) Aufzählung von Turingmaschine $(M_i)_i$ verwenden, sodass $\mathcal{C} = \{L(M_i) \mid i \geq 0\}$ und dann dafür sorgen, beziehungweise zeigen, dass $\forall i : A \neq L(M_i)$ beziehungweise $\forall i : A \triangle L(M_i) \neq 0$.
Das heißt $\forall i \exists x : (x \in A \wedge x \notin L(M_i)) \vee (x \notin A \wedge x \in L(M_i))$

Lemma

Es existiert eine FP-Funktion $i \mapsto M_i$, sodass $DTIME_{M_i}(x) \leq (|x| + 2)^2$ und $P = \{L(M_i) \mid i \geq 0\}$.

Lemma

Es existiert eine FP-Funktion $i \mapsto f_i$ wobei f_i eine Übersetzermaschine ist und $FP = \{f_i \mid i \geq 0\}$ und $DTIME_{f_i}(x) \leq (|x|+2)^i$. Insbesondere $|f_i(x)| \leq (|x|+2)^i$.

Es ist klar, dass $A \in NP$ aber $A \notin P$ und A nicht NP-vollständig, wenn

- $A \in NP$
- $\forall i \exists x : x \in A \triangle L(M_i)$
- $\forall i \exists x : x \in SAT \wedge f_i(x) \notin A$ oder $x \notin SAT \wedge f_i(x) \in A$

Das heißt f_i ist keine Reduktion von SAT auf A.

Wir konstruieren A in der folgenden Form:

$$A = \{x \mid x \in SAT \wedge f(|x|) \text{ gerade.}\}$$

f wird sogleich rekursiv definiert derart, dass dieses A die Bedingungen 1, 2 und 3 erfüllt.

Man sollte also versuchen sicherzustellen, dass

- $f(n) in ZEit p(n)$ berechenbar für Polynom p (Bedingung 1).
- Für alle i existiert x mit
  $x \in SAT$ und $f(|x|)$ gerade und $x \notin L(M_i)$
  oder
  ($x \notin SAT$ oder $f(|x|)$ ungerade) und $x \in L(M_i)$
- Für alle i existiert x, sodass $x \in SAT$ und ($f(|f_i(x)|)$ ungerade oder $f_i(x) \notin SAT$

oder
$x \notin SAT$ und $f(|f_i(x)|)$ gerade und $f_i(x) \in SAT$

f wird jetzt rekursiv definiert.
Wir schreiben $A_f = \{x \mid x \in SAT \wedge f_($|x| gerade.$\}$

$$
\begin{aligned}
f(n+1) = IF \;\; & (2+\log\log n)^{f(n)} \geq \log n \\
THEN \;\; & f(n) \\
ELIF \;\; & \exists x : |x| \leq \log\log n \text{ und } x \in L(M_i) \wedge x \notin A_f \text{ oder } x \notin L(M_i) \wedge x \in A_f \\
THEN \;\; & f(n)+1 \text{ ebe } f(n) \\
ELIF \;\; & \exists x : |x| \leq \log\log n \\
THEN \;\; & f(n)+1
\end{aligned}
$$

Um $f(n+1)$ zu berechnen wird rekursiv nur auf Werte $f(m)$ mit $m \leq n$ zugegriffen, also ist f eine totale Funktion.

Es genügt, ein Polynom $p(n)$ zu finden, sodass in Zeit $p(n)$ der Wert $f(n+1)$ aus $f(0), f(1), \ldots f(n)$ bestimmt werden kann.
Die Laufzeit für $f(n)$ ist nämlich dann $\mathcal{O}(\Sigma_{m<n} p(m)) = poly(n)$.

Klar ist, dass $A = A_f \neq L(M_i)$ falls $f(n) = 2i+1$ für ein n, denn dann war $f(n') = 2i$ für ein $n' < n$ und $f(n'+1) = 2i+1$ also die Suche in Fall 2 erfolgreich.

Ebenso ist f_i keine Reduktion: $SAT \leq A_f$ falls $f(n) = (2i+1)+1$ für ein n.

Das heißt wir müssen zeigen, dass f surjektiv ist, d.h. dass jeder Fall irgendwann erfolgreich abgeschlossen wird.

Details dazu auf der Website.