Ludwig-Maximilians-Universität München

WS 2015/2016

Martin Hofmann, Ulrich Schöpp

Komplexitätstheorie

Vorlesungsmitschrieb von

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

Stand: 14. Dezember 2015, 10:16

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!

Website der Vorlesung: `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

<u>Definition</u>

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 $\Sigma = \{0, 1\}, Q = \{q_0, \dots q_4\}$

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	$\square$	$\square$	$\square$	$\square$	S	S	q_1
q_0	0	$\square$	0	0	R	R	q_3
q_0	1	$\square$	1	$\square$	S	S	q_2
q_3	$\square$	$\square$	$\_$	$\_$	$\_$	$\_$	q_2
q_3	0	$\square$	0	0	R	R	q_3
q_3	1	$\square$	1	$\square$	S	L	q_4
q_4	0	0	$\_$	$\_$	$\_$	$\_$	q_2
q_4	1	0	1	0	R	L	q_4
q_4	0	$\square$	$\square$	$\square$	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 $L \subseteq (\Sigma \setminus \{\Box\})^*$ falls gilt:

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

Eine Turingmaschine **entscheidet** eine Sprache $L \subseteq (\Sigma \setminus \{\Box\})^*$ 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

<u>**Definition**</u>

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

<u>**Satz**</u>

Das Halteproblem ist unentscheidbar.

<u>Beweis</u>

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

<u>**Definition**</u>

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.

<u>Definition</u>

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

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

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

<u>Satz</u>

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

<u>Definition</u>

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**:

<u>Satz</u>

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.

<u>Satz</u>

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.

<u>Definition</u>

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.

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

$$\exists L' \in P \text{ sodass gilt } x \in L \Leftrightarrow \exists z \text{ ``Loesung'' mit } |h| \leq L'$$

$$\text{wobei } p \text{ ein Polynom ist}$$

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

<u>**Definition**</u>

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

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

<u>**Satz**</u>

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

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

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

<u>Beweis</u>

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 effizienter Lösbarkeit.

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 stures Durchprobieren, auch Bruteforcing genannt, 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 $(l_{j,1}) \dots (l_{j,k})$,
  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-te 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 $P =$

$\{L(M_i) \mid i \geq 0\}$.

<u>**Lemma**</u>

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

- $f(n) inZEitp(n)$ 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 $\mathcal{O}(\Sigma_{m<n} p(m)) = poly(n)$.

Offensichtlich ist $A = A_f \neq L(M_i)$, 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.

Vorlesung vom 16.11.15

3.7 Orakel-Turingmaschinen

Orakel-Turingmaschinen als Mittel zu zeigen, dass die Beweismethoden Diagonalisierung[1] und Simulation[2] nicht helfen, um $P = NP$ zu entscheiden.

Definition

Eine **Orakel-Turingmaschine** T hat ein zusätzliches Band (Orakelband) und drei zusätzliche Zustände q_Q (Frage), q_{yes}, q_{no} (Antwort).

Ist $A \subseteq \Sigma^*$, dann definiert man Berechnungen $T^A(x)$ von T auf x mit Orakel A wie folgt:

- Wie üblich mit der zusätzlichen Regel:
  Falls T in q_Q so wird T in Zustand q_{yes}, q_{no} versetzt und zwar in einem Schritt, je nach dem, ob die aktuelle Beschriftung $z \in \Sigma^*$ des Orakelbands (Anfrage/Query) in A ist (q_{yes}) oder nicht (q_{no}).

Man schreibt $L^A(T)$ oder $L(T^A)$ für die von T akzeptierte Sprache, falls Anfragen gemäß A beantwortet werden.

Beispiel

Sei $STCONN = \{(G, s, t) \mid \exists \text{ Pfad von } s \text{ nach } t \ \in G\}$

[1] zum Beispiel benutzt für $NP \subseteq EXP$
[2] zum Beispiel benutzt für $P \subset EXP$ (Ladner)

Feststellen, ob ein Graph G einen nichttrivialen Zyklus enthält. Zähle alle Paare (u,v) auf und frage jeweils (G,u,v) und (G,v,u) ab.

Damit ist eine Maschine T beschrieben, sodass $CYCLE = L^{STCONN}(T)$.
Die Laufzeit von T ist $|G|^2$ (insbesondere polynomiell). Es ist also $CYCLE \in P^{STCONN}$. (Definition unten)
Nachdem nun $STCONN \in P$ folgt $CYCLE \in P$.

Definition

Sei $A \subseteq \Sigma^*$.

Man definiert P^A als die Menge aller Sprachen L sodass eine deterministische Turingmaschine T existiert mit $L = L^A(T)$ und $DTIME(T^A(x)) \leq p(|x|)$ für ein Polynom p.

Analog NP^A.

Beobachtung: $A \in P \Rightarrow P^A = P \wedge NP^A = NP$.

Beispiel

$SAT \in P^{SAT}$

$NODE-COVER \in P^{SAT}$

$NP \in P^{SAT}$

$TAUT = \{\phi \mid \phi \text{ allgemeingültig}\} \in P^{NP}$

$IMPL = \{(\phi,\psi) \mid \phi \text{ allgemeingültig} \Rightarrow \psi \text{ allgemeingültig}\} \in P^{NP}$

$CIRCUIT - MIN = \{(\text{Schaltkreis } C, A) \mid \text{Schaltkreis } C' \text{ der Größe } \leq k \text{ und } C \equiv C'\} \in NP^{SAT}$

Wr zeigen jetzt die folgenden zwei Sätze, aufgrund derer kein Beweis für $P = NP$ oder $P \neq NP$ existieren kann, welcher in Gegenwart von Orakeln auch funktioniert:

Satz

$$\exists A \in \Sigma^* : P^A = NP^A$$

Beweis

$A = \{(T, x, 0^k) \mid \text{ deterministische Turingmaschine } T \text{ akzeptiert } x \text{ und benutt dabei } \leq k \text{ Bandzellen}\}$

A ist offensichtlich entscheidbar.

Sei T eine nichtdeterministische Orakel-Turingmaschine und $p(n)$ ein Polynom, das die Laufzeit von T beschränkt und somit auch die Größe aller Orakelanfragen.
Wir müssen eine deterministische polynomiell zeitbeschränkte Turingmaschine T' mit Orakel A bauen, sodass $L^A(T') = L^A(T)$.

Zunächst konstruieren wir eine deterministische Turingmaschine T_{HILF}, die ohne Orakelbenutzung die Sprache $L^A(T)$ entscheidet, indem alle Orakelanfragen "mit Bordmitteln" (also selbst) beantwortet werden unter Verwendung der Entscheidbarkeit von A.

Vollständige Berechnungssequenzen einer Berechnung, deren Bandplatz $\leq k$ ist und t Schritte lang ist, benötigen Platz $\mathcal{O}(k * t)$.
Orakelanfragen einer Berechnung von T auf x (Eingabe) haben die Form $(S, y, 0^k)$ wobei $|S|, |y|, k \leq p(|x|)$.

Wir verwenden also jetzt 3 Hilfsbänder, eines für die nichtdeterministische Berechnung von T auf x, die wir der Reihe nach alle simulieren, eines für Orakelanfragen, eines für die Beantwortung der Orakelanfragen.
Diese Maschine ist deterministisch und benötigt auf ihren Bändern höchstens $q(|x|)$ Platz, wobei q ein von p abgeleitetes Polynom ist (in etwa $q(n) = \mathcal{O}(p(n)^2)$.

Die eigentlichte Maschine T' arbeitet jetzt wie folgt:
Gegeben Eingabe x, schreibe $(T_{HILF}, x, O^{q(|x|)})$ auf das Orakelband. Falls q_{yes}, dann akzeptiere. Falls q_{nein}, dann verwerfe.

q.e.d.

<u>Satz</u>

$$\exists B \in \Sigma^* : P^B \neq NP^B$$

<u>Beweis</u>

Falls $B \subseteq \Sigma^*$, definiere $L_B = \{0^k \mid \exists x \in \Sigma^* : |x| = k \wedge x \in B\}$
Offensichtlich ist $L_B \in NP^B$, egal was B ist.

Es gilt jetzt, B so zu wählen, dass für jede polynomiell zeitbeschränkte deterministische Orakel-Turingmaschine gitl: $L_B \neq L^B(T)$.

Sei $i \mapsto T_i$ eine effektive Aufzählung von Orakel-Turingmaschinen, sodass $DTIME(T_i^x(x)) \leq |x|^i + i$ (alternativ $(|x| + 2)^i$ wie letztes Mal).
Für alle deterministischen Orakel-Turingmaschinen S und alle Orakel x muss i existieren, sodass $L(S^x) = L(T_i^x)$. T_i ist die durch i beschriebene Orakel-Turingmaschine künstlich auf Laufzeit $n^i + i$ beschränkt. Jetzt muss also für jedes i ein n_i existieren, sodass $T_i^B(0^{n_i})$ akzeptiert und B enthält kein Wort der Länge n_i (dann ist nämlich

$0^{n_i} \notin L_B$), oder aber $T_i^B(0^{n_i})$ verwirft und B enthält ein Wort x_i mit $|x_i| = n_i$, denn dann ist $0^{n_i} \in L_B$.
Dann ist in der Tat $L_B \notin P^B$.

Beobachtung: Wenn $T_i^X(x)$ nach t Schritten hält und U aus Wörtern y mit $|y| > t$ besteht, dann gilt $T_i^{X \cup U}(x)$ akzeptiert $\Leftrightarrow$ $T_i^X(x)$ akzeptiert.

Vorlesung vom 19.11.15

Wir definieren rekursiv $n_i \in \mathbb{N}, B(i) \subseteq \Sigma^*, \Sigma = \{0,1\}, n_0 = 0, B(0) = \emptyset$
Falls $B(0), \dots, B(i-1)$ schon definiert, definiere $n_i, B(i)$ so, dass gilt $\exists x : |x| = n_i \in B(i) \Leftrightarrow T_i^{B(i)}(0^{n_i}$ akzeptiert nicht.
Außerdem sollte $T_i(0^{n_i})$ keine Elemente von $B(j) j > i$ anfragen.
$B = \bigcup_{i \geq 0} B(i)$

Es gilt dann $T_i^B(0^{n_i})$ akzeptiert nicht $\Leftrightarrow T_i^{B(i)}(0^{n_i})$ akzeptiert nicht $\Leftrightarrow 0^{n_i} \in L_{B(i)} \Leftrightarrow 0^{n_i} \in L_b$ (Falls wir zusätzlich dafür sorgen, dass $B(j)$ mit $j \geq i$ keine Elemente der Länge n_i enthält.)

Allgemein: $B(i) = \begin{cases} B(i-1) & , falls \dots \\ B(i-1) \cup \{x\} & , falls \dots \end{cases}$

$n_i B(i) \in n_{i-1}(i-1)\ n_i$ sodass

- $n_i > n_{i-1}^{i-1} + i - 1$ (Größer als die Laufzeit von $T_{i-1}(0^{n_{i-1}})$ und $T_j(0^{n_j})$ für $j < i$.)

- $2^{n_i} > n_i^i + i$ (da 2^x schneller wächst als $x^i + i$.)

Es gibt also mehr Wörter x mit $|x| = n_i$ als die Laufzeit von $T_i(0^{n_i})$.

Halbkonkrete Ausführung dieser Aufzählung für die ersten drei Schritte:
$n_0 = 0, B(0) = \emptyset, n_1 > 0^0 + 0, 2^{n_1} > n_1 + 1$
Rechne $T_1^{B(0)}(000)$. Wir nehmen an, dass nicht akzeptiert wird. Also $B(1) = B(0) = \emptyset$
$n_2 > n_1^1 = 4, 2^{n_2} > n_2^2 + 2, \rightsquigarrow n_2 = 5$ Rechne $T_2^{B(1)}(00000)$. Wir nehmen an, dass akzeptiert wird. Diese Rechnung dauerte $\leq 5^2 + 2 = 27$ Schritte. Es gibt 32

Wörter der Länge 5. Sei x eines, das nicht abgefragt wurde. $x = 10110$. $B(2) = B(1) \cup \{10110\} = \{10110\}$
$n_3 > 27, 2^{n_3} > n_3^3 + 3, \rightsquigarrow n_3 = 28$ Rechne $T_3^{B(2)}(0^{28})$. Wir nehmen an, dass nicht akzeptiert wird. Also $B(1) = B(0) = \emptyset$
...

Invariante $B \cap \{x \mid |x| \leq n_i\} = B(i)$
Rechenzeit von $T_i(0^{n_i}) < n_{i+1}$ $(> n_i^i + i)$

B unterscheidet sich von $B(i-1)$ nur durch Wörter die von $T_i^{B(i-1)}(0^{n_i})$ nicht angefragt werden. Entweder, da sie länger sind als die Rechenzeit oder von der Länge n_i sind, aber so gewählt wurden, dass sie nicht gefragt werden.

$T_i^B(0^{n_i})$ akzeptiert
$\Leftrightarrow T_i^{B(i-1)}(0^{n_i})$ akzeptiert
$\Leftrightarrow \neg\exists x \in B(i) : |x| = n_i$
$\Leftrightarrow 0^{n_i} \notin L_B$

Das heißt $L(T_i^B) \neq L_B$

$L_B \notin P^B$

q.e.d.

3.8 Polynomielle Hierarchie

Definition

$co\text{–}\phi = \{L \subseteq \Sigma^* \mid \Sigma^* \setminus L \in \phi\}$

$co\text{–}P = P$

$co\text{–}NP$ ist nicht offensichtlich gleich NP wegen der asymmetrischen Akzeptanzbedingungen bei Nichtdeterminismus.

<u>**Definition**</u>

Die **polynielle Hierarchie** (PH):

$\Delta_0^P = P$

$\Sigma_0^P = \Pi_0^P = P$

$\Sigma_1^P = NP$

$\Pi^P = co\text{–}NP$

Das soll andeuten, dass von der PH die Rede ist. Es hat nicht mit einem Exponenten oder Orakel zu tun.

<u>**Definition**</u>

relativierte Quantoren

Notation: $\exists_n x.A(x) \equiv \exists x \in \Sigma^*.|x| \leq n \land A(x)$

Ebenso: $\forall_n x.A(x) \equiv \forall x \in \Sigma^*.|x| \leq n \Rightarrow A(x)$

Außerdem: $\neg\exists_n x.A(x) \Leftrightarrow \forall_n x.\neg A(x)$

und: $\neg\forall_n x.A(x) \Leftrightarrow \exists_n x.\neg A(x)$

Definition

Die weitere **polynielle Hierarchie** (PH):

$\Sigma_2^P = NP^{SAT} = NP^{NP}$

$\Pi_2^P = co\text{–}NP^{SAT} = co\text{–}NP^{NP}$

$\Delta_2^P = P^{SAT} = P^{NP}$

$\Sigma_{n+1}^P = NP^{\Sigma_n^P}$

$\Pi_{n+1}^P = co\text{–}NP^{\Sigma_n^P}$

$\Delta_{n+1}^P = P^{\Sigma_n^P}$

Anmerkung:
$NP^{\phi} = \bigcup_{X \in \phi} NP^X$

3.9 Kollabieren der polynomiellen Hierarchie

Falls $P = NP$, so folgt $\Sigma_n^P = P$.

Im allgemeinen gilt: Falls $\Sigma_{n+1}^P = \Sigma_n^P$ für ein bestimmtes n, dann auch $\Sigma_{n'}^P = \Sigma_n^P$ für alle $n' \geq n$. Man sagt dann PH kollabiere auf der n. Stufe.

Vorlesung vom 23.11.15

Satz

$L \in \Sigma_2^P \Leftrightarrow$
$\exists L' \in P,$ Polynom $p : x \in L \Leftrightarrow \exists_{p(|x|)} y . \forall_{p(|x|)} z . (x, y, z) \in L'$

Beweis

- "$\Leftarrow$":

  Seien $L' \in P$, p Polynom vorgegeben.

  Gesucht: nichtdeterministische Polynomialzeit-Orakel-Turingmaschine M sodass M mit geeignetem NP-Orakel die Sprache L entscheide.

  $L'' = \{(x, y) \mid \exists_{p(|x|)} z . (x, y, z) \notin L'\}$

  $L'' \in NP$ (Rate z und prüfe $(x, y, z) \notin L'$)

  $x \in L \Leftrightarrow \exists_{p(|x|)} y . (x, y) \notin L''$

  Die Maschine M rät also y und prüft $(x, y) \notin L''$ mit Orakel für L''.

- "$\Rightarrow$":

  Sei M eine nichtdeterministische durch $p(n)$ laufzeitbeschränkte Turingmaschine mit Orakel $X \in NP$, z.B. $X = SAT$, $L = L(M)$

Es ist $x \in L = L(M)$ genau dann, wenn
$\exists$ Lauf y von M auf $: M$ akzeptiert $x \wedge |y| \leq q(|x|)$
wobei q ein Polynom ist mit $q(n) = \mathcal{O}(p(n)^2)$

Um zu prüfen, ob y tatsächlich ein LAuf ist und noch dazu akzeptierend, muss neben allem möglichen, was in polynomieller Zeit geht, z.B.

– Folgefkonfigurationen jeweils gemäß der Maschinentafel (δ_M) aus Vorgängerkonfigurationen enthalten,

– Am Anfang Startzustand, am Ende akzeptierender Zustand,

– Input okay in Startkonfiguration kopiert,

auch geprüft werden, dass alle Orakelanfragen richtig beantwortet wurden.

...

und

$\exists \eta . \eta_i$ erfüllt die i. positiv beantwortete Orakelanfrage $q_i \in SAT$.

und

$\forall \rho . \rho_j$ erfüllt die j. negativ beantwortete Orakelanfrage $\rho_j \in SAT$ nicht.

Es gibt also ein Polynom $q(n)$ sodass $x \in L(M) \Leftrightarrow \exists_{q(|x|)} y . L_1(x, y) \wedge \exists_{p(|x|)} \eta . L_2(y, \eta) \wedge \forall_{q(|x|)} \rho . L_3(y, \rho)$
wobei $L_1, L_2, L_3 \in P$

– $L_1(x, y) \Longleftrightarrow y$ ist akzeptierender Lauf von M auf x bis auf Orakelanfragen

- $L_2(y,\eta) \iff$ Die in y positiv beantworteten Orakelanfragen sind $\rho_1 \dots \rho_k, \eta = |y_1 \dots y_k$ und $\eta_i \vDash \psi_i$ für $i = 1 \dots k$

- $L_3(y,\rho) \iff$ Die in y negativ beantworteten Orakelanfragen sind $\psi_1 \dots \psi_k, \rho = \rho_1 \dots \rho_k$ und $\rho \mp \psi_i$ für $i = 1 \dots k$

q.e.d.

<u>Korollar</u>

$L \in \Pi_2^P \iff$
$\exists$ Polynom $p \wedge L' \in P : x \in \Leftrightarrow \forall_{p(|x|)} y . \exists_{p(|x|)} z.(x, y, z) \in L'$

<u>Satz</u>

Ist $L \in \Sigma_n^P$, so existiert ein Polynom $p(n)$ und $L' \in P$ sodass
$x \in L \Leftrightarrow y_1 \forall_{p(|x|)} y_2 \exists_{p(|x|)} y_3 \dots Q_{p(|x|)} y_n.(x, y_1, \dots y_n) \in L'$
n gerade: $Q = \forall$
n ungerade: $Q = \exists$

Beweis durch Induktion über n.

<u>Satz</u>

Satz von Kamp-Lipton

Falls SAT polynomiell große Schaltkreise hat, so kollabiert die PH auf der zweiten Stufe.
Das heißt $\forall n \geq 2 : \Sigma_n^P = \Sigma_2^P$.

Dass SAT polynomielle Schaltkreise hat soll heißen: Es gibt ein Polynom $p(n)$ und für jedes $n \in \mathbb{N}$ einen boolschen Schaltkreis C_n mit n Inputs und Größe $\leq p(n)$

$(|C_n| \leq p(n))$.
Für alle aussagenlogische Formeln ϕ mit $|\phi| = n$ gilt $\phi \in SAT \Leftrightarrow C(\phi) = TRUE$, $C(\phi)$ die Bitkodierung von ϕ an die n Inputs von C anlegen.

Kurznotation: $SAT \in P/poly$

Der Satz von Kamp-Lipton sagt also $SAT \in P/poly \Rightarrow$ PH $= \Sigma_2^P$

Hilfsmittel: Selbstreduzierbarkeit von SAT
Ein Schaltkreis C_n wie oben beschrieben kann so umgebaut werden in einem Schaltkreis D_n polynomielle Größe, dass bei Antwort TRUE eine erfüllende Belegung zurückgeliefert wird.

Das heißt D_n hat n Ausgänge, die eine Belegung kodieren sollen. Spezifiere

- $\phi \in SAT. D_n(\phi)(\eta, TRUE)$ mit $\eta \models \phi$
- $\phi \notin SAT. D_n(\phi)(\_, FALSE)$

D_n ruft C_n insgesamt $m \leq n$ mal auf, wobei m die Zahl der Variablen plus eins ist.

Vorlesung vom 26.11.15

4 Platzkomplexität

4.1 Platzkonstruierbare Funktionen

<u>**Definition**</u>

Eine Funktion $s(n)$ heißt **platzkonstruierbar** genau dann, wenn eine deterministische Turingmaschine existiert, die bei Eingabe 0^n genau $s(n)$ Bandfelder beschreibt und dann hält.

<u>**Beispiel**</u>

Alle Polynome mit Koeffizienten $\in \mathbb{Q}^+$, die Wurzelfunktion, die Logarithmus-Funktion, die Potzenfunktion usw. sind platzkonstruierbar.

4.2 Platzverbrauch einer Turingmaschine

<u>**Definition**</u>

Der **Platzverbrauch einer Turingmaschine** (deterministisch oder nichtdeterministisch) bei Eingabe x ist

- **erste Definition**
  die Größe des beschriebenen Teils aller Bänder am Ende der Berechnung. (Mit dieser Definition ist der Platzverbrauch stets $\geq |x|$).

- **zweite Definition**
  die Endgröße aller anderen Bänder, wobei das Eingabeband nicht überschrieben werden darf.

Die zweite Definition ist Standard,w enn sublineare Platzschranken betrachtet werden, zum Beispiel $\log(n)$. Oberhalb von $\mathcal{O}(n)$ sind die beiden Definitionen äquivalent.

Notation: $DSPACE_M(x)$ und $NSPACE_M(x)$

$DSPACE_M(s(n)) = \{L \mid \exists DTMM : L = L(M) \wedge DSPACE_M(x) = \mathcal{O}(s(|x|))\}$

$NSPACE_M(s(n)) = \{L \mid \exists DTMM : L = L(M) \wedge NSPACE_M(x) = \mathcal{O}(s(|x|))\}$

$PSPACE = \bigcup_{k \geq 0} DSPACE(n^k)$ (polynomieller Platz)

$LINSPACE = DSPACE(n)$

$LOGSPACE = DSPACE(\log n)$ (auch als L bezeichnet)

$NLOGSPACE = NSPACE(\log n)$ (auch als NL bezeichnet)

<u>**Beispiel**</u>

STCONN (Erreichbarkeit in gerichteten Graphen) ist $\in NLOGSPACE$ (rate Pfad) und $\in LINSPACE$ (Tiefensuche/Breitensuche)

Es gibt eine triviale, aber wissenswerte Beziehung zwischen Zeit- und Platzkomplexität:

$$DSPACE(s(n)) \subseteq DTIME(2^{\mathcal{O}(s(n))})$$

Hat die Berechnung nach $2^{c*s(n)}$ Schritten nicht geendet, so kann abgebrochen werden wegen Wiederholung einer globalen Konfiguration. (Tatsächlicher Platzverbrauch $\leq c * s(n)$)

$$DTIME(t(n)) \subseteq DSPACE(t(n))$$

Mehr Platz als Laufzeit kann nicht angefordert werden.

Satz

Für deterministische Einband-Turingmaschinen T gilt:

$DTIME_T(x) = \mathcal{O}(t(|x|) \Rightarrow L(T) \in DSPACE(\sqrt{t(n)})$

Für Mehrband-Turingmaschinen gibt es einen ähnliches Satz, bei dem der Platz allerdings etwas größer ist. Dass er für Einband-Turingmaschinen so gut ist, ist gewissermaßen kurios.

4.3 Platzhierarchiesatz

Satz

"Echt mehr Platz hilft auch mehr."

Für genaue Aussage und Beweis siehe z.B. Papadimitrion

Wichtige Konsequenz:

$$LOGSPACE \subset PSPACE$$

$$LOGSPACE \subseteq NLOGSPACE \subseteq P \subseteq NP \subseteq PH \subseteq PSPACE \subseteq EXP$$

Von jeder dieser Inklusionen ist unbekannt, ob sie echt sind. Mindestens eine muss aber echt sein.

4.4 Zusammenhänge von Platzkomplexitätsklassen

Satz

Satz von Savitch

Für eine platzkonstruierbare Funktion $s(n) \geq \log(n)$ ist
$NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$

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

Beweis

Sei eine nichtdeterministische Turingmaschine T gegeben mit Platzbedarf $S = c * s(|x|)$ bei Eingabe x. Wir betrachten eine Kodierung der globalen Konfigurationen von $T(x)$ durch Wörter der Länge S und o. B. d. A. gebe es exakt eine akzeptierende Endkonfiguration s_{ACC}. (Alle Bänder am Ende löschen, d.h. mit 0 überschreiben.)

$x \in L(T) \Longleftrightarrow s_{ACC}$ von s_{INI} aus in $\leq 2^S$ Schritten erreichbar

Hier steht s_{INI} für die Startkonfiguration bei Eingabe x. 2^S ist die Gesamtzahl der Konfigurationen.

Das heißt s_{ACC} ist von s_{INI} aus im Graphen der Konfigurationen erreichbar (Spezialfall von STCONN).

q.e.d.

Notation:

$s \rightarrow_T s'$: s' ist 1-Schritt-Folgekonfiguration von s in T und kann in $LOGSPACE$ entschieden werden.

$REACH(s,s')$: $s \rightarrow^* s'$ (s' ist von s erreichbar)

$REACH(s,s',i)$: $s \rightarrow^{\leq 2^i} s'$ (s' ist von s in weniger als 2^i Schritten erreichbar)

$x \in L(T) \Longleftrightarrow REACH(s_{INI}, s_{ACC}, S)$

Es gilt

$$REACH(s,s',0) \Longleftrightarrow s = s' \vee s \rightarrow_T s'$$

$$(2^0 = 1)$$

$$REACH(s,s',i+1) \Longleftrightarrow \exists \check{s} : REACH(s,\check{s},i) \wedge REACH(\check{s},s',i)$$

$$(2^{i+1} = 2 * 2^i)$$

Dies liefert eine rekursive Implementierung von $REACH(s,s',i)$
($\exists \check{s} \rightsquigarrow$ for $\check{s} \in$ globale Konfigurationen)

Der Rekursionsstack hat Tiefe S (Toplevel-Aufruf $REACH(s_{INI}, s_{ACC}, S)$).

Jeder Activationrecord hat Größe $\mathcal{O}(S)$ genauer gesagt $2S$ für die beiden Parameter $s, s', \log(S)$ für die Parameter i. Wenn gewünscht noch ein weiteres S für die for-Schleife.

Die Gesamtgröße des Stacks ist beschränkt durch $S * \mathcal{O}(S) = \mathcal{O}(S^2)$.

Historisch wurde zunächst gezeigt, dass STCONN in $DSPACE(\log(n)^2)$ liegt. Der Satz von Savitch kann auch hieraus abgeleitet werden.

Vorlesung vom 30.11.15

Satz

Satz von Immerman-Szelepcsényi

Sei $s(n) \geq \log(n)$.

Dann ist $NSPACE(s(n)) = co\text{–}NSPACE(s(n))$

Wichtiger Spezialfall: $co\text{–}STCONN \in NSPACE(\log(n)) = NL = NLOGSPACE$
Die allgemeine Behauptung kann aus diesem Spezialfall leicht gefolgert werden (durch den Graph der globalen Konfiguration).

Beweis

Es sei eine nichtdeterministische Turingmaschine T vorgelegt und $NSPACE_T(x) \leq c * s(|x|)$. Wir müssen eine nichtdeterministische Turingmaschine T' konstruieren, sodass $NSPACE_{T'}(x) = \mathcal{O}(s(|x|))$ und $x \in L(T') \Leftrightarrow x \notin L(T)$.
Es existiert akzeptierende Berechnung von T' auf x genau dann, wenn alle Berechnungen von T auf x verwerfen.

Sei x fixiert und o. B. d. A. $\Sigma = \{0, 1\}$.
Schreibe S_{INI} für die globale Startkonfiguration von T auf x und S_{ACC} für die (o. B. d. A.. einzige) akzeptierende globale Konfiguration. Weiter sei $S = c' * s(|x|)$ so gewählt, dass alle globalen Konfigurationen durch $0/1$-Strings der Länge S kodiert werden.
$s \rightarrow_T s'$ bedeute, dass s' in einem Schritt aus s hervorgehen kann (Das kann in Platz $\log(S)$ entschieden werden).
T' soll nun x akzeptieren genau dann, wenn kein Pfad (der Länge 2^S) von s_{INI} zu s_{ACC} existiert. (Anzahl der globalen Konfigurationen ist kleiner als 2^S. Ein einziger Pfad braucht, wenn er voll ausgeschrieben wird, schon Platz $S * 2^S \notin \mathcal{O}(s(|x|))$.)

Vorbemerkung:

Nehmen wir an, dass die Anzahl N der von s_{INI} aus erreichbaren globalen Konfigurationen bekannt ist bzw. berechnet werden kann.

Wir zählen der Reihe nach alle globalen Konfigurationen auf (geht mit Platz $\mathcal{O}(S)$) und raten für jede von denen einen Pfad von S_{INI} dorthin. Durch Mitführen eines Zählers haben wir am Ende der Aufzählung die Anzahl derjenigen Knoten, für die das gelungen ist.

```
function A(...) {
    cnt = 0;
    for (s in globalConfigs) {
        pfad = guessPath();
        if (pfad.endsAt(s))
            cnt++;
        if (s = s_acc)
            return "reject";
    }
    if (cnt == N)
        return "accept";
    else
        return "reject";
}

function guessPath() {
    s = s_ini;
    for (i = 1; i <= 2^S; i++) {
        sX = guessNonDet({0,1}^S);
        if (s2 -> sX) {
            s2 = sX;
        } else {
            return "reject";
        }
        b = guessNonDet({0,1});
        if (b) {
            break;
        }
    }
}
```

Falls N die Anzahl der von s_{INI} aus erreichbaren globalen Konfigurationen ist, so kann A akzeptieren genau dann, wenn s_{ACC} von s_{INI} unerreichbar ist.
Begründung "$\Rightarrow$": Falls A akzeptiert, dann ist s_{ACC} tatsächlich unerreichbar, weil alle

N erreichbaren Konfigurationen in der for-Schleife als solche erkannt wurden und s_{ACC} nicht unter ihnen war.
Begründung "$\Leftarrow$": Falls s_{ACC} unerreichbar ist, so kann A akzeptieren, indem bei jedem der von s_{INI} aus erreichbaren s tatsächlich ein entsprechender Pfad geraten wird.

Grobe Struktur des Algorithmus für T':

```
N = 1;
for (i = 1 ... 2^S) {
    // Invariante: Anzahl der von s_ini aus in weniger als i-1 Schritten erreichbaren Konfigurationen ist gleich N
    updateN();
}
A();
```

Der Block `updateN()` wird selbst Nichtdeterminismus enthalten, in dem Sinne, dass die gesamte Berechnung verwerfend abgebrochen werden kann. Passiert das nicht, dann ist N korrekt aktualisiert und bei passender Wahl der nichtdeterministischen Entscheidungen, passiert das auch.

```
function updateN() {
    cnt = 0;
    for (s in globalConfigs) {
        reachable = false;
        cnt2 = 0;
        // alle N Stück die von s_ini aus in weniger als i-1 Schritten erreichbar sind, aufzählen
        for (sCheck in globalConfigs) {
            pfad = guessPath();
            if (pfad.endsAt(sCheck)) {
                cnt2++;
                if (sCheck == s || sCheck -> s)
                    reachable = true;
            }
        }
        if (cnt2 != N)
            return "reject";
        else if (reachable)
            cnt++;
    }
    N = cnt;
}
```

Am Ende von `updateN` hat entweder N den korrekten Wert oder es wurde verworfen. Es ist möglich, die nichtdeterministischen Entscheidungen so zu treffen, dass der korrekte Wert geliefert wird, sodass nicht verworfen wird.

q.e.d.

Vorlesung vom 03.12.15

4.5 Noch ein Satz von Cook

Definition

$NSPACE(s(n)) + STACK$

Intuitiv ist das alles, was mit $\mathcal{O}(s(n))$ platzbeschränkten Arbeitsbändern und einem unbeschränkten Stack berechnet werden kann.

Eine mögliche Formalisierung: Turingmaschine mit Stack hat zusätzlich ein Stackalphabet Γ und ein besonderes Symbol $A \in \Gamma$. Jedes "Quintupel" enthält eine zusätzliche Komponente, die Stack-Aktion, eine der folgenden drei:

- IDLE (Keller bleibt unverändert)
- POP (oberstes Kellersymbol wird entfernt)
- PUSH ($x \in \Gamma$ auf den Keller legen)

Außerdem ist das jeweils oberste Kellersymbol lesbar.
Die Maschine wird mit Kellerinhalt $\boxed{A}$ gestartet und hält per Definition, wenn der Keller leer wird, d.h. wenn A gePOPt wird.

Formal:
$\delta = Q \times (\Sigma^k \times \Gamma) \times Q \times \Sigma^k \times S$
wobei S die Menge der Stack-Aktionen ist.

Beispiel

Der Beweis des Satzes von Savitch zeigt, dass STCONN $\in DSPACE(\log(n) + STACK)$ und $NSPACE(s(n)) \subseteq DSPACE(s(n) + STACK)$ ist.

Satz

$s(n) \geq \log(n) \Rightarrow NSPACE(s(n)) + STACK = DTIME(2^{\mathcal{O}(s(n))})$

Beweis

- "$\subseteq$":

  Sei T eine zunächst deterministische durch $s(n)$ platzbeschränkte Turingmaschine mit Stack.
  Wir fixieren eine Kodierung der "Bandkonfigurationen" von T. Diese Bandkonfigurationen beinhalten den Zustand und gesamten Inhalt aller Arbeitsbändern und Kopfpositionen. Bei Eingabe der Größe n haben die so kodierten Bandkonfigurationen die Größe $\mathcal{O}(s(n))$, also konkret $c * s(n)$ für ein festes c.
  Sei nun eine Eingabe fixiert. B sei die Menge der Bandkonfigurationen der Größe $c * s(|x|)$ und Γ das Stackalphabet.
  Wir definieren die partielle Funktion $f : B \times \Gamma \to B : f(b, x) = b'$
  b' ist die Bandkonfiguration, die erreicht wird, wenn man T in b und mit Kellerinhalt $\boxed{x}$ startet und so lange rechnet, bis X zum ersten Mal gePOPt wird.

  Bemerkung:
  $x \in L(T) \iff f(b_{INI}, A) = b'$ und b' akzeptierende Endkonfiguration ist. o. B. d. A. ist b' eindeutig.

  Bemerkung:
  Streng genommen sollte man schreiben $f_x(b, X)$ oder $f(x, b, X)$, da die Eingabe x in die Definition von f einfließt.

f gestattet folgende rekursive Definition:

$$f(b,X) = \begin{cases} b' & ,\text{POP} \\ f(b',X) & ,\text{IDLE} \\ f(f(b',Y),X) & ,\text{PUSH(Y)} \end{cases}$$

wobei b' die auf (b,X) unmittelbar folgende Bandkonfiguration ist.

Statt nun $f(b_{INI},A)$ durch Rewriting auszuwerten, was letztendlich der Berechnung von T entspräche, tabulieren wir die gesamten f-Werte systematisch mit dynamischer Programmierung. Die Anzahl der Bandkonfigurationen ist $2^{c*s(|x|)}$.
Damit gibt es ingesamt $\mathcal{O}(2^{c*s(|x|)})$, o. B. d. A. $2^{c*s(|x|)})$ viele f-Aufrufe. Eine Wertetabelle für f kann also in Zeit $2^{\mathcal{O}(s(|x|))}$ komplett ausgefüllt werden.
Gesamtrechenzeit: $\mathcal{O}((2^{c*s(|x|)})^2) = 2^{\mathcal{O}(s(|x|))}$

Wenn die Maschine nichtdeterministisch ist, dann nehmen wir statt f die Funktion $F : B \times \Gamma \times B \to bool$:

$$F = \begin{cases} true & \text{falls es Berechnung von } b \text{ nach } b' \text{ gibt, an deren Ende } X \text{ erstmals gePOPt wird} \\ false & \text{sonst} \end{cases}$$

F kann ganz genauso mit dynamischer Programmierung ausgewertet werden.

Sei umgekehrt T eine deterministische Turingmaschine mit Zeitschranke $2^{c*s(n)}$.
Die Kopfpositionen aller Köpfe könenn durch String der Länge $\mathcal{O}(s(n))$ kodiert werden (Binärkodierung der numerischen Positionen). a mit $|a| = c' * s(n)$ kodiere diese Positionen.
Ebenso können Zeitpunkte in der Berechnung durch String dieser Länge kodiert werden.

Wir definieren bei fester Eingabe x folgende rekursive Funktionen:

– $band(t,a) \in \Sigma^k$ als Bandinhalt zur Zeit t an den Positionen a.

$band(0,a) =$ Inhalt der initialisierten Bänder an den geforderten Positionen
$band(t+1,a) = \ldots zustand(t) \ldots band(t,a) \ldots kopf(t)$

– $zustand(t) \in Q$
$zustand(0) = q_0$

– $kopf(t) =$ Positionen aller Köpfe zur Zeit t
$kopf(t+1) = \ldots kopf(t) \ldots band(t,a) \ldots zustand(t)$

Die klassische rekursive Auswertung dieser Funktionen beziehungsweise des Toplevel-Aufrufs $zustand(2^{c*s(|x|)}) \in \{q_{ACC}, q_{REG}\}$ kann auf einer $DSPACE(s(n)) + STACK$ Maschine simuliert werden.

q.e.d.

Vorlesung vom 7.12.2015

4.6 Logarithmischer Platz

Eine Funktion $f : \Sigma^* \Rightarrow \Sigma^*$ ist in FL und heißt **in logarithmischem Platz berechenbar**, wenn es eine deterministische Turingmaschine gibt die bei Eingabe x:

- Den Funktionswert $f(x)$ auf ein gesondertes Ausgabeband schreibt.
- Das Ausgabeband nicht mehr ließt.
- Das Eingabeband (wie immer bei Space) nicht mehr beschreibt.
- $\mathcal{O}(log|x|)$ platzbeschränkte Arbeitsbänder besitzt.

Äquivalente Charakterisierung: Maschine schreibt das n-te Symbol auf ein besonderes Band (oder signalisiert es durch Einnehmen eines bestimmten Zustandes), wenn n in Binärkodierung auf ein besonderes Band geschrieben wird.

Bemerkung

- $2^{c*log(n)} = n^c$
- $f \in FP \Rightarrow |f(x)| = \mathcal{O}(n^c)$ für ein c, d. h. polynomiell beschränkt und auch $DTIME_T(x) \in \mathcal{O}(n^c)$ bei passenden T, welches f berechnet.

Bemerkung
$FL \subseteq FP$

Satz

FL ist unter Komposition abgeschlossen. $f, g \in FL \Rightarrow g \times f \in FL$

Gegeben Turingmaschinen T_f und T_g im zweiten Format (n-tes Symbol von $f(x)$ wird bei Eingabe von x und n "on demand" berechnet.

Turingmaschine $T_{g \times f}$ für $g \times f$ wird ebenfalls im 2. Format konstruiert. Sie simuliert auf der äußeren Ebene die Maschine T_g. Jede Leseanfrage der Eingabe wird in eine Berechnung von T_f auf x an der entsprechenden Ausgabeposition übersetzt.

4.7 Mehr Härte und Vollständigkeit

- X ist **NL-hart**, wenn für jedes $X' \in NL$ gilt $X' \leq_L X$
- X ist **P-hart**, wenn für jedes $X' \in P$ gilt $X' \leq_L X$
- X ist **NL- bzw. P-vollständig**, wenn X NL-hart bzw. P-hart ist und $X \in NL$ bzw. $X \in P$

Hierbei bedeutet $X' \leq_L X$, dass $f \in FL$ existiert, mit $x \in X' \Leftrightarrow f(x) \in X$

Bemerkung:
$X' \leq_L X \Rightarrow x \leq_P X$, bzw. $X \leq X'$

Satz

Das Problem STCONN (Erreichbarkeit im gerichteten Graphen) ist NL-vollständig.

Beweis

$STCON \in NL$ ✓

NL-hart: Sei T eine NL-Maschine. $x \in ((T)E)$ es existiert ein Pfad von a_{INI} nach a_{ACC} im Graphen, dessen Knoten die globale Konfiguration von T bei Eingabe x sind, a_{INI} die Startkonfiguration und a_{ACC} die (o. B. d. A. eindeutige) akzeptierende Endkonfiguration ist. Die Übersetzung von x in dem Graphen kann offensichtlich durch eine "Fl-Maschine" im 1. Format geschehen.

q.e.d.

4.8 Horn

Definition

Eine Klausel (Disjunktion von Literalen) mit höchstens einem positiven Literal heißt **Hornklausel**. Hornklauseln können logische äquivalent in den Formaten:

- $P_1, P_2, \ldots P_k \to q$
- d. h. $\neg p_1 \vee \neg p_2 \vee \ldots \neg p_k \vee q$
- oder $P_1, P_2, \ldots P_k \to \perp$
- d. h. $\neg p_1 \vee \neg p_2 \vee \ldots \neg p_k$

geschrieben werden.

Definition

Gegeben eine Menge von Hornklauseln H. Das Problem, ob H unerfüllbar ist, heißt

Hornproblem.

Definition

Eine Hornklausel der Form $\rightarrow q$ k = 0 heißt **Faktum**. Eine Hornklausel der Form $\rightarrow \bot$ k = 0 heißt **Goal**.

In der Regel entfällt H genau ein "Goal", $p \rightarrow \bot$.
H ist dann unerfüllbar, genau dann, wenn p aus den übrigen Klausen hergeleitet werden kann. $H \vdash p$ hierbei ist die Herleitbarkeit wie folgt definiert.

- $\rightarrow q \in H$ dann $H + g$ alle Fakten sind herleitbar
- $P_1, P_2, \dots P_k \rightarrow q \in H$ und $H + P_1, H + P_2, \dots H + P_k$ dann auch $H + g$

Bemerkung:
Eine Horn-Klausel Menge ohne Goals ist immer erfüllbar.

Bemerkung:
Eine Hornklauselmenge mit mehreren Goals ist unerfüllbar genau dann, wenn mindestens ein Goal herleibar ist.

Satz

$Horn \in P$ Wir benutzen eine dynamische Menge S, die am Ende alle herleitbaren Variablen enthalten soll. $S := \emptyset$

```
while(!done)
  s' = s
  for p_1, \dots, p_k  \rightarrow q \in H
    if{p1, \dots, p_k} \seteq S' then
      S' := S' \u {q}
    done := S' = S''; S := S'
return ''es existiert P \rightarrow \bot \in H mit q \subseteq S''
```

Dieses Vorgehen läuft in polynomieller Zeit, da die Zahl der Durchläufe durch die Zahl der Variablen beschränkt ist, aber nicht offensichtlich in lmgarithmischem Platz, da die dynamische Menge linearen Platz benötigt.

Beispiel

- $\to A$
- $A \to B$
- $F \to F$
- $B, C \to D$
- $A, B \to C$
- $A, D \to E$
- $E \to \bot$

$S : \emptyset, \{A\}, \{A, B\}, \{A, B, C\}, \{A, B, C, D\}, \{A, B, C, D, E\}$

Satz

Das Hornproblem ist P-vollständig.

Beweis

Genauo wie Satz Cock(SAT ist NP-vollständig), im Fall einer deterministschen Maschine entstehen bei der Übersetzung nur Horn-Klauseln.

Details Sei eine deterministsche Turingmaschine T gegeben, Eingabe $xDTIME_T(x) \subseteq p(|x|)$

Variablen

$zust(t,g)$ Zustand zu Zeit t ist g

$band(t,i,x)$ Bandinhalt zur Zeit t an Position i ist

$kopf(t,i)$ Kopf an Pos i zur Zeit t

Klauseln z.B.

- $band(t,i,x), kopf(t,i') \to band(t+1,i,x)$
- $band(t,i,x), kopf(t,i), zust(t,q) \to band(t+1,i,x')$
- $band(t,i,x), kopf(t,i)$
- $Zustand(t,q) \to zust(t+1,g') \ldots$

falls jeweils $S(g,x) = (g', x', m)$

Eingabe x wird akzeptiert genau dann, wenn $zust(P|x|), q_{ACC}$ herleibar ist um Goal $zust(P(|x|), q_{ACC} \to \perp$ hinzunehmen. Die Übersetzung von T, x in diese Klauselmenge kann mit FL-Maschine durchgeführt werden. Also $L(T) \leq_L$ HORN.

q.e.d.

Das bereits genannte offene Problem $NL \subsetneq P$ NL echt in P enthalten? kann also umformuliert werden, in die Frage, ob $HORN \in NL$.

Markieren von Klauseln
Auf ein Fakt darf jederzeit eine Marke gelegt werden. Liegen auf allen Prämissen $p_1, p_2, \ldots, p_k$ einer Klausel $p_1, p_2, \ldots, p_k \rightarrow$ schon Marken, dann darf q mit einer Marke beheftet werden.

Vorlesung vom 10.12.15

4.9 Quantifizierte Boolesche Formeln

Satz

QBF ist PSPACE-vollständig.

Quantifizierte Boolesche Formeln (QBF)

Gegeben ist eine boolesche Formel mit Quantoren, die über boolesche Werte rangieren.

$$\forall x \exists y . x \leftrightarrow \neg y$$

o. B. d. A. kann man diese Formeln als geschlossen voraussetzen (ggf. $\exists$, $\forall$ davorschreiben)
Gesucht ist die Antwort auf die Frage, ob diese Formel wahr ist.

Gegeben ist eine boolesche Formel ϕ.

$$\phi \text{ erfüllbar } \iff \exists x_1 \exists x_2 \ldots \exists x_n . \phi \in QBF$$

wobei $\{x_1 \ldots x_n\}$ die Variablen von ϕ sind.

$$TAUT \leq QBF$$

$$\phi \text{ Tautologie } \iff \forall x_1 \forall x_2 \ldots \forall x_n . \phi \in QBF$$

$AEQSAT = \{(\phi, \psi) \quad | \quad \phi$ erfüllbar $\Leftrightarrow \psi$ erfüllbar $\}$

$$(\phi, \psi) \in AEQSAT \iff (\exists x.\phi) \Leftrightarrow (\exists y.\psi) \in QBF$$

$$\exists x_1 \exists y_1 \forall x_2 \forall y_2.(\phi(x_2) \to \psi(y_1)) \land (\psi(y_2) \to \phi(x_1))$$

Spezialfall: QBF_n

Gegeben ist eine QBF-Formel in Pränexform (alle Quantoren ganz außen) mit maximal n Quantorenwechseln beginnend mit $\exists$.

Gegeben: $\exists x_1 \forall x_2 \exists x_3 \ldots \exists/\forall x_n.\phi$ wobei ϕ eine boolesche Formel ohne Quantoren ist.

QBF_n ist vollständig für Σ_n^P (polynomielle Hierarchie) falls $n \geq 1$.
(Lezteres ist unmittelbare Konsequenz aus der logischen Charakterisierung der PH und Satz von Cook.)

Rekursiver Algorithmus für QBF:

```
check(exists x.phi, eta) =
    check(phi, eta[x -> true]) or
    check(phi, eta[x -> false])
check(forall x.phi, eta) =
    check(phi, eta[x -> true]) and
    check(phi, eta[x -> false])
check(phi1 and phi2, eta) =
    check(phi1, eta) and
    check(phi2, eta)
check(x, eta) = eta(x)
    ...
```

Wi zeigen jetzt $L \leq QBF$ für beliebiges $L \in PSPACE$:

Beweis

Sei T eine deterministische Turingmaschine mit $L(T) = L$ und $DSPACE_T(x) \leq p(|x|)$ und $DTIME_T(x) \leq 2^{p(|x|)}$ wobei p ein Polynom ist. Außerdem habe T zu jeder Eingabe x genau eine Startkonfiguration a_{INI} und eine akzeptierende Endkonfiguration a_{ACC}. sodass $x \in L(T) \Leftrightarrow \exists a_1, a_2 \ldots a_{2^{p(|x|)}}$ mit $a_1 = a_{INI}, a_{2^{p(|x|)}} = a_{ACC}$ $a_i{}_T a_{i+1}$ wobei $_T$ ein Schritt der Auswertung durch T ist.
$REACH(a, a', k) :=$ a' ist in 2^k Schritten von a aus erreichbar.
$x \in L \Leftrightarrow REACH(a_{INI}, a_{ACC}, p(|x|))$
$REACH(a_{INI}, a', 0) \Leftrightarrow a \rightarrow_T a'$
$REACH(a_{INI}, a', k+1) \Leftrightarrow \exists \check{a}.REACH(a, \check{a}, k) \wedge REACH(\check{a}, a', k)$

Kodiere Konfiguration a durch $p(|x|)$ viele boolesche Variablen. $a \rightarrow_T a'$ durch quantorenfreie boolesche Formel.
Dies, zusammen mit Abwicklung der Rekursion, liefert eine QBF_1-Formel (nur Existenzquantoren), also wäre $PSPACE \subseteq NP$. Das ist natürlich falsch, denn die so erhaltene QBF_1-Formel hat die Größe $\Omega(2^{p(|x|)})$.
Die Aufblähung findet nicht statt, wenn man folgende Rekurenz verwendet:
$REACH(a, a') \Leftrightarrow a \rightarrow_T a'$
$REACH(a, a') \Leftrightarrow \exists \check{a} \forall b \forall b'.(b = a \wedge b' = \check{a} \vee b = \check{a} \wedge b' = a') \Rightarrow REACH(b, b', k)$
Die Größe von $REACH(a, a', k)$ ist $\mathcal{O}(k * p(|x|))$ lalalalala $\mathcal{O}(p(|x|)^2)$

q.e.d.

Die Charakterisierung von $PSPACE$ durch QBF liefert einen direkten Zusammenhang zu Gewinnstrategien in endlichen 2-Personen-Spielen. Die Existenz einer Gewinnstrategie kann in offensichtlicher Weise als QBF-Formeln können selbst als 2-Personen-Spiel verstanden werden.
Man kann ein entsprechendes Maschinenmodell definieren, die **alternierenden Turingmaschinen**.

Beispiel

Gegeben ein Graph $G = (V, E)$.
Knoten entsprechen beliebiger Belegung von boolesche Variablen. Kanten durch boolesche Formeln repräsentiert. Die Nachbarknoten eines Knoten können aufgezählt werden. $s, t \in V$ vorgegeben.

Es werden abwechselnd Marken auf den Graphen gesetzt. Spieler 1 hat weiße Marken, Spieler 2 hat schwarze Marken. Ein Spieler hat gewonnen, wenn es einen Pfad seiner Farbe von s nach t gibt.

Gefragt ist, ob Spieler 1 gewinnen kann. Man zeige, dass dieses Problem PSPACE-vollständig ist.