Proseminar Theoretische Informatik
Seminarleiter: Prof. Dr. Helmut Alt

NP-Vollständigkeit *(NP-Completeness)*

von Tobias Laatsch

Wintersemester 2010/11

1 Voraussetzungen

1.1 P und NP

Die beiden Sprach- bzw. Komplexitätsklassen werden an dieser Stelle nur kurz angerissen, da sie zum einen explizit Thema anderer Vorträge sind und es zum anderen vollkommen ausreichend ist, P und NP nur grob zu kennen, um diesen Vortrag zu verstehen.

Definition: Sei M_d eine beliebige *deterministische* und M_nd dementsprechend eine beliebige *nichtdeterministische* Turingmaschine. Sei $n := |w|$. Definieren die Sprachklassen
$TIME(t(n)) := \{L|\ \exists M_\mathrm{d} :\ L(M_\mathrm{d}) = L \wedge time_{M_\mathrm{d}}(w) \in O(t(n))\}$[1]
$NTIME(t(n)) := \{L|\ \exists M_\mathrm{nd} :\ L(M_\mathrm{nd}) = L \wedge time_{M_\mathrm{nd}}(w) \in O(t(n))\}$
$P := \bigcup_n TIME(t(n))$ und $NP := \bigcup_n NTIME(t(n))$

Es beschreibt also P die Menge aller Sprachen, welche von einer deterministischen Turingmaschine in polynomieller Zeit erkannt werden können[2]. In NP liegen demnach alle Sprachen, die von einer nichtdeterministischen Turingmaschine in polynomieller Zeit erkannt werden können.

Korollar: Da für jede deterministische Turingmaschine eine äquivalente nichtdeterministische Turingmaschine gefunden werden kann, gilt für *jede* Funktion $t(n)$:
$TIME(t(n)) \subseteq NTIME(t(n))$
Insbesondere gilt dann auch $P \subseteq NP$. Ob aber P eine *echte* Teilmenge von NP ist, ist bis heute nicht bekannt.

1.2 Reduzierbarkeit (Wiederholung)

Den Begriff der Reduzierbarkeit haben wir bereits in der Vorlesung "Grundlagen der Theoretischen Informatik" kennengelernt. Da im folgenden dieser Begriff benutzt wird, führen wir uns diesen noch einmal kurz zu Gemüte.

Definition: Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen. Dann ist L_1 auf L_2 reduzierbar, geschrieben $L_1 \preceq L_2$, wenn eine berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ existiert, so dass folgende Aussage gilt: $\forall x \in \Sigma^* :\ x \in L_1 \Leftrightarrow f(x) \in L_2$

Bei der Reduzierbarkeit handelt es sich um eine Quasiordnungsrelation, d.h. sie ist **reflexiv** und **transitiv**. Die Transitivität wird für uns später eine entscheidende Rolle spielen.

Lemma: Bezeichne $AUFZ$ die Menge der aufzählbaren und ENT die Menge der entscheidbaren Sprachen. Folgende Aussagen sind wahr:

[1] Genaueres zu $time_M$ folgt in Punkt 1.3

[2] Schöning z.B. spricht explizit von Mehrbandturingmaschinen – entscheidend ist aber nur die Unterscheidung deterministisch/nichtdeterministisch

$L_1 \preceq L_2 \wedge L_2 \in ENT \rightarrow L_1 \in ENT$ sowie $L_1 \preceq L_2 \wedge L_2 \in AUFZ \rightarrow L_1 \in AUFZ$

Wenn also eine Sprache L_1 auf eine andere Sprache L_2 reduziert werden kann (L_2 also in gewisser Weise "schwerer" als L_1 ist), befinden sich L_1 und L_2 in derselben Sprachklasse (bezogen auf entscheidbar/rekursiv aufzählbar). Auch die Kontraposition ist interessant: $L_1 \notin ENT \rightarrow L_1 \not\preceq L_2 \vee L_2 \notin ENT$
Wenn also L_1 auf L_2 reduziert werden kann **und** L_1 keine entscheidbare Sprache ist, gilt dies ebenso für L_2. [3]

1.3 Polynomielle Reduzierbarkeit

Wir erweitern nun den Begriff der Reduzierbarkeit, indem wir auch Betrachtungen zur Laufzeit anstellen. Dies ist elementar wichtig für den Beweis des später diskutierten *Cook-Levin-Theorems*.

Definition: Sei f die von der Turingmaschine M_f berechnete Funktion. Beschreibe $time_{M_f}$ die Schrittanzahl von M_f in Abhängigkeit des Eingabewortes w. Dann ist f in **polynomieller** Zeit berechenbar genau dann, wenn $\exists k \in \mathbb{N} : \ time_{M_f}(w) \in O(|w|^k)$.

Insbesondere sind so auch Funktionen mit "besserer" Laufzeit (logarithmisch, $\sqrt{|w|}$, etc.) in polynomieller Zeit berechenbar.

Definition: Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen. Dann ist L_1 auf L_2 **polynomial** reduzierbar (oft auch Polynomialzeitreduktion genannt), geschrieben $L_1 \preceq_p L_2$, wenn eine **mit polynomiellem Zeitaufwand** berechenbare Funktion $f : \Sigma^* \rightarrow \Sigma^*$ existiert, so dass folgende Aussage gilt: $\forall x \in \Sigma^* : \ x \in L_1 \Leftrightarrow f(x) \in L_2$

Auch die Polynomialzeitreduktion ist eine Quasiordnung.

Lemma: Folgende Aussagen sind wahr:
$L_1 \preceq_p L_2 \wedge L_2 \in P \rightarrow L_1 \in P$ sowie $L_1 \preceq_p L_2 \wedge L_2 \in NP \rightarrow L_1 \in NP$

Beweis: Sei $L_2 \in P$ und M_2 deterministische Turingmaschine mit $L(M_2) = L_2$. Sei weiterhin L_1 eine mittels f auf L_2 polynomiell reduzierbare Sprache.
Betrachten die deterministische Turingmaschine M_f, welche f berechnet. Deren Schrittanzahl kann per Definition durch ein Polynom abgeschätzt werden. Sei n^k dieses Polynom, wobei $n := |w|$ die Länge des Eingabewortes darstellt. Da M_f maximal $c_1 \cdot n^k$, $c_1 \in \mathbb{N}$ Schritte durchführt, ist auch $|f(w)|$ maximal $c \cdot n^k$.
Betrachten nun M_2. Da $L(M_2) \in P$ gibt es ein Polynom, welches die Schrittanzahl von M_2 abschätzt. Sei dieses Polynom m^l. Dann führt M_2 bei einer Eingabe der Länge m maximal $c_2 \cdot m^l$, $c_2 \in \mathbb{N}$ Schritte durch.
Schalten nun M_f und M_2 hintereinander und erhalten eine neue Turingmaschine M_1, welche genau die Sprache L_1 erkennt. Diese führt bei einer Eingabe der Länge n zunächst maximal $c_1 \cdot n^k$ Schritte durch und erhält ein Wort, welches ebenfalls maximal so lang ist. Der von M_2 berechnete Teil führt dann maximal $c_2 \cdot (c_1 \cdot n^k)^l = c_2 \cdot c_1^l \cdot n^{kl}$ Schritte durch. Das bedeutet aber, dass die Laufzeit von M_1 durch das Polynom n^{kl} abgeschätzt werden kann – und $L(M_1) = L_1$ dementsprechend in P liegt. Der Beweis lässt sich analog für nichtdeterministische Turingmaschinen wiederholen.[4] $\square$

[3] Genaueres (auch der Beweis des Lemmas) findet sich bei Schöning, Kapitel 2.6.
[4] Etwas formaler ist der Beweis etwa bei Schöning, Kapitel 3.2, oder bei Sipser, Theorem 7.25

2 NP-Vollständigkeit

2.1 NP-Schwere

Wir bezeichnen ein Problem (eine Sprache) als NP-schwer, falls es mindestens so schwer ist wie alle anderen Probleme in NP. Dies klingt zunächst etwas schwammig und willkürlich definiert. Wir haben aber bereits eine Quasiordnungsrelation definiert, mit welcher wir die Schwere von Problemen in NP "ordnen" können, nämlich die **Polynomialzeitreduktion**.

Definition: Sei L_1 eine beliebige Sprache. L_1 ist NP-schwer, falls $\forall L \in NP: \ L \preceq_p L_1$

In manchen Büchern ist von NP-Härte bzw. NP-harten Problemen/Sprachen die Rede. Dies wird z.T. als Übersetzungsfehler (engl. NP-hard) angesehen. Auch in diesem Vortrag wird NP-Schwere als passendere Bezeichnung vorgezogen.

2.2 NP-Vollständigkeit

Definition: Eine Sprache L bezeichnet man als NP-vollständig, falls $L \in NP$ und L NP-schwer ist.

Satz: Sei L_1 eine NP-vollständige Sprache. Es gilt $L_1 \in P \Leftrightarrow P = NP$

Beweis: Es gilt $\forall L \in NP: \ L \preceq_p L_1$ sowie $L_2 \preceq_p L_1 \wedge L_1 \in P \rightarrow L_2 \in P$. Daraus folgt direkt $\forall L \in NP: \ L \in P$. Sei umgekehrt $P = NP$. Dann folgt aus $L_1 \in NP$ direkt $L_1 \in P$ $\square$

Damit haben wir schon eine ganze Menge gewonnen, denn es genügt zum Beweis der Gleichheit bzw. Ungleichheit von P und NP zu zeigen, ob ein beliebiges, NP-vollständiges Problem in P liegt – oder eben nicht. Wir können auch neue Probleme durch Polynomialzeitreduktion auf bekannte, NP-vollständige Probleme zurückführen und uns hierdurch eine womöglich sinnlose Suche nach einem effizienten Algorithmus hierfür ersparen.
Problem: Wir kennen bisher kein einziges NP-vollständiges Problem! Dies liefert uns das Cook-Levin-Theorem.

3 Das Cook-Levin-Theorem

Das Cook-Levin-Theorem ist benannt nach dem Amerikaner Steven Cook und dem Ukrainer Leonid Levin, welche heute an der *University of Toronto* bzw. der *Boston University* lehren. Beide fanden unabhängig voneinander in den 1970ern heraus, dass das **Erfüllbarkeitsproblem der Aussagenlogik** (engl. Satisfiability Problem, SAT) NP-vollständig ist.
Ein boolescher Term ϕ wird *erfüllbar* genannt, falls es irgendeine Belegung seiner Variablen mit Wahrheitswerten gibt, so dass dieser zu 1 ausgewertet wird. Dementsprechend ist $SAT := \{\phi |\ \phi \text{ ist erfüllbar}\}$. Um nachzuweisen, dass SAT NP-vollständig ist, müssen wir zwei Dinge beweisen.

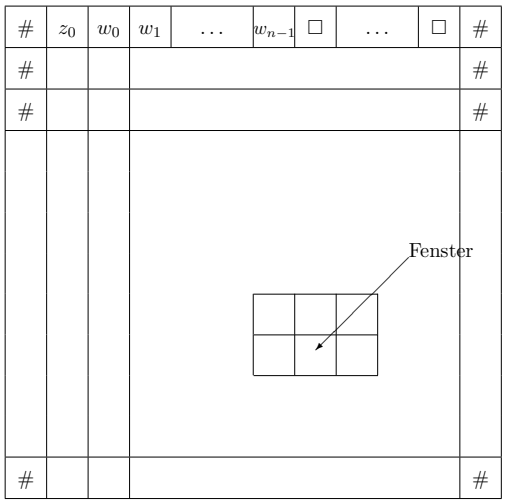
Lemma: $SAT \in NP$.

Beweis: Dies nachzuweisen ist einfach. Wir konstruieren eine nichtdeterministische Turingmaschine M, welche genau SAT erkennt. Diese überprüft in einem ersten Durchlauf zunächst, ob die Eingabe ein gültiger boolescher Term ist. Danach kopiert der Lese-/Schreibkopf alle Variablen an das Ende der Eingabe/auf ein zweites Band. Dies ist notwendig, da weder die

Anzahl der Variablen noch deren jeweilige Vorkommen bekannt sind – und M hier deswegen nicht mit Zuständen arbeiten kann.
Diese "Liste" der Variablen benutzt M nun, um *nichtdeterministisch* deren Vorkommen mit entweder 1 oder 0 zu ersetzten. Den so entstehenden Ausdruck wertet M aus und wechselt in einen akzeptierenden Zustand, falls am Ende der Rechnung 1 auf dem/einem Band steht. Falls es also eine erfüllende Belegung des Eingabeterms gibt, wird M in einer der Rechnungen einen akzeptierenden Zustand erreichen und $L(M) = SAT$. Da außerdem für die Auswertung boolescher Ausdrücke Algorithmen mit polynomiellem Zeitaufwand existieren, ist $SAT \in NP$. □

Lemma: SAT ist NP-schwer.

Beweis: Hierzu muss ein beliebiges Problem $L_1 \in NP$ polynomiell auf SAT reduziert werden können. Dazu konstruieren wir boolesche Terme, deren Variablen so gewählt werden, dass sie alle Rechnungen der L_1 akzeptierenden Turingmaschine "simulieren".
Sei $M_1 = \{Z, \Sigma, \Gamma, \delta, z_0, \square, E\}$ mit $Z = \{z_0, \ldots, z_k\}$, $\Gamma = \{\gamma_0, \ldots, \gamma_l\}$ und $E = \{z_e\}$ die L_1 erkennende Turingmaschine. Eine Rechnung von M_1 kann *eindeutig* durch eine Folge von Konfigurationen dargestellt werden. Wir erweitern dies auf eine tabellarische Darstellung in sogenannten *Tableaus*, in denen jede Zeile einer Konfiguration entspricht.
Nehmen wir nun OBdA an, dass jede Rechnung von M_1 maximal $p(n) \in \mathbb{N}[n]$ Schritte benötigt. Auch hierbei beschreibt n wieder die Länge des Eingabeworts. Dann passt jede *akzeptierende* Rechnung von M_1 in ein $(p(n)+3) \times (p(n)+3)$-Tableau. [5]



Die erste und letzte Spalte füllen wir mit #-Symbolen, weil es im späteren Vorgehen wichtig sein wird, Anfang und Ende der Konfigurationen zu kennen. Wir könnten die Spaltenanzahl des Tableaus auch $2p(n) + 3$ wählen für Turingmaschinen, deren Band auf beiden Seiten unendlich ist, belassen es aber bei der quadratischen Form. Diese bietet ausreichend Platz für alle Konfigurationen, da nach $p(n)$ Schritten höchstens $p(n)$ Zeichen auf dem Band stehen können; zusammen mit den begrenzenden #-Symbolen und dem aktuellen Zustand ergibt das eine Konfiguration mit $p(n) + 3$ Zeichen.
Anhand dieses Schemas erstellen wir nun einen booleschen Term ϕ mit folgenden Bestandteilen: $\phi = \phi_{\text{start}} \wedge \phi_{cell} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$. Alle Variablen haben die Form $x_{i,j,c}$ mit $i, j \in \{0, \ldots p(n)+2\}$ und $c \in C = Z \cup \Gamma \cup \{\#\}$. Wenn $x_{i,j,c}$ auf 1 gesetzt wird, bedeutet dies, dass in der i-ten Zeile und j-ten Spalte des Tableaus das Symbol c steht.
Die Teilformel ϕ_{start} stellt sicher, dass die Startkonfiguration in der ersten Zeile steht.

$$\phi_{start} = x_{0,0,\#} \wedge x_{0,1,z_0} \wedge x_{0,2,w_0} \wedge \ldots \wedge x_{0,n+1,w_{n-1}} \wedge x_{0,n+2,\square} \wedge \ldots \wedge x_{0,p(n)+1,\square} \wedge x_{0,p(n)+2,\#}$$

[5] Das Schema (sowie der gesamte Beweis) orientieren sich stark an Sipser, Theorem 7.30

Die Teilformel ϕ_{cell} stellt sicher, dass in jeder Zelle des Tableaus nur ein Zeichen steht.

$$\phi_{\text{cell}} = \bigwedge_{i=0}^{p(n)+2} \left[\bigwedge_{j=0}^{p(n)+2} \Big(\bigvee_{c \in C} x_{i,j,c} \Big) \wedge \Big(\bigwedge_{\substack{c_1, c_2 \in C \\ c_1 \neq c_2}} \overline{(x_{i,j,c_1} \wedge x_{i,j,c_2})} \Big) \right]$$

Die Teilformel ϕ_{move} stellt sicher, dass jedes 2×3-*Fenster* korrekt ist – und damit implizit, dass das dargestellte Tableau insgesamt aus einer korrekten Folge von Konfigurationen besteht. Hierzu sei $W_M = \{(c_0, c_1, c_2, c_3, c_4, c_5) |\ c_0 \dots c_5 \text{ ist korrektes Fenster eines Tableaus von } M\}$

$$\phi_{\text{move}} = \bigwedge_{i=0}^{p(n)+1} \left[\bigwedge_{j=0}^{p(n)} \Big(\bigvee_{a \in W_{M_1}} \left(x_{i,j,a_0} \wedge x_{i,j+1,a_1} \wedge x_{i,j+2,a_2} \wedge x_{i+1,j,a_3} \wedge x_{i+1,j+1,a_4} \wedge x_{i+1,j+2,a_5} \right) \Big) \right]$$

Die Teilformel ϕ_{accept} stellt sicher, dass der Endzustand erreicht wurde. Wir können voraussetzen, dass M_1 den Endzustand nicht mehr verlässt. Dann reicht es aus, nur die letzte Konfiguration zu überprüfen.

$$\phi_{\text{accept}} = \bigvee_{j=1}^{p(n)+1} x_{p(n)+1,j,q_e}$$

Wir "zählen" nun die Anzahl der Variablen in ϕ. Für die Teilformeln ϕ_{start} und ϕ_{accept} ist dies sehr einfach – diese enthalten $2p(n) + 4$ Variablen.
Schwieriger wird es bei ϕ_{cell}. Zunächst erkennen wir, dass aufgrund der zwei $\bigwedge$ die Anzahl der Variablen $(p(n) + 2) \cdot (p(n) + 2) \cdot x$ ist, wobei x die Variablen der inneren Formel beschreibt; diese hängen von der Konstante $|C|$ ab, genauer: $x = |C| + |C|^2 - |C| = |C|^2 =: c_1$. Insgesamt gilt also $|\phi_{\text{cell}}| = c_1 \cdot (p(n) + 2)^2$.
Bei ϕ_{move} ist die Größe der inneren Formel $6 \cdot |W_{M_1}|$. Auch dies ist eine Konstante, denn es gibt für jeden Übergang in δ nur eine Kombination aus konstant vielen Zuständen, Zeichen und Bewegungen. Sei $c_2 := 6 \cdot |W_{M_1}|$. Dann gilt: $\phi_{\text{move}} < c_2 \cdot (p(n) + 2)^2$.
Die Gesamtgröße von ϕ lässt sich dann durch $(c_1 + c_2) \cdot (p(n) + 2)^2 + 2p(n) + 4 \in O(p^2(n))$ abschätzen und ist daher polynomiell. Wir gehen daher auch davon aus, dass eine entsprechende Turingmaschine existiert, welche in polynomieller Zeit eine solche Formel erzeugen kann. □

Satz: SAT ist NP-vollständig.

Beweis: Dies folgt direkt aus den bewiesenen Lemmata. □

Literaturverzeichnis

1. Michael Sipser: *Introduction to the Theory of Computation.*
   PWS Publishing Company, 1997.

2. Uwe Schöning: *Theoretische Informatik Kurz Gefasst.*
   Spektrum Akademischer Verlag, 2002.