Zur Beweisbarkeit von P $\stackrel{?}{=}$ NP

Christian Soltenborn

29. Januar 2003*

Zusammenfassung

Orakel erlauben die Erweiterung von Komplexitätsklassen. Manche Beweistechniken funktionieren unabhängig davon, ob darin Orakel verwendet werden oder nicht; diese heissen dann *relativierende Beweistechniken*. Diese Arbeit führt zunächst in das Zusammenspiel von Orakeln und Turingmaschinen ein. Auf dieser Grundlage werden zwei relativierende Beweise — der Zeithierarchiesatz sowie der Satz von Savitch — vorgestellt und deren Gültigkeit unter Verwendung eines *beliebigen* Orakels gezeigt. Kern der Arbeit ist die Präsentation zweier Orakel, die widersprüchliche Ergebnisse zu P $\stackrel{?}{=}$ NP bringen und damit relativierende Beweistechniken für diese Fragestellung ausschließen. Den Abschluss bildet eine kurze Übersicht über verbleibende, vielleicht erfolgversprechendere Beweisansätze.

1 Orakel und Turingmaschinen

Wie viele gute Ideen ist auch das Konzept der Orakel von bestechender Schlichtheit. Es basiert auf einer seit Urzeiten erfolgreichen Strategie zur Lösung von schwierigen Problemen: wenn man ein Problem nicht lösen kann, versuche man, es unter anderen Voraussetzungen zu betrachten. Oft wird die (in einem anderen „Universum“ angesiedelte) Problemlösung Hinweise auf Lösungsansätze im herkömmlichen „Universum“ liefern.

Was bedeutet das jedoch im Rahmen der Komplexitätstheorie? Die grundlegendste Voraussetzung ist wohl, dass jede Berechnung gewisse Ressourcen benötigt. Man stelle sich nun ein Universum vor, in dem bestimmte Berechnungen in konstanter Zeit erfolgen! Es wäre z.B. ein „SAT-Universum[1]“ denkbar, in dem ein Algorithmus im Verlauf seiner Berechnungen Fragen der Art „*Ist diese Bool'sche Formel erfüllbar?*“ stellt. Die Antwort erfolgt sofort und beeinflusst natürlich den weiteren Verlauf der Berechnungen. Zugeben: ein nicht sehr realistisches Szenario, aber äusserst hilfreich...

Wie muss nun eine Turingmaschine beschaffen sein, um ein Orakel nutzen zu können?

*Enstanden im Rahmen des Seminars *Perlen der theoretischen Informatik* von Prof. Friedhelm Meyer auf der Heide, Universität Paderborn, WS 02/03. Dank an Dr. Martin Ziegler für die ausgezeichnete und engagierte Betreuung!

[1] Papadimitriou hat das sehr schön ausgedrückt: [Pap94], S.339: „This is *the world of* SAT, where a friendly “oracle” answers all our SAT queries for free“

Definition 1 Eine *Turingmaschine $M^?$ mit Orakel* ist eine Mehrband-Turingmaschine, die ein ausgezeichnetes Band, das Frageband, sowie drei besondere Zustände $q_?, q_{yes}$ und q_{no} besitzt.

Die Turingmaschine $M^?$ ist unabhängig vom verwendeten Orakel. Das $^?$ zeigt an, dass jede Sprache als Orakel dienen kann. Sei also L eine beliebige Sprache aus $\{0,1\}^*$. Benötigt die Turingmaschine M^L im Verlauf ihrer Berechnungen die Antwort auf eine Frage der Form „$x \in L\,?$“, so schreibt sie zunächst das Wort x auf das Frageband. Danach geht sie in den Zustand $q_?$, aus dem sie nach *einem* weiteren Rechenschritt in den Zustand q_{yes} (q_{no}), gelangt, der anzeigt, dass x (nicht) in L liegt.

Die Zeitkomplexität von Turingmaschinen mit Orakeln ist genauso definiert wie bei normalen Turingmaschinen. Deshalb sind solche Maschinen auch so unrealistisch!

Hat man erst einmal Turingmaschinen mit Orakel definiert, so sind Komplexitätsklassen mit Orakel leicht zu beschreiben:

Definition 2 Sei $\mathcal{C}$ eine Komplexitätsklasse, die über Turingmaschinen und deren Zeit- bzw. Platzbedarf definiert ist. Dann ist $\mathcal{C}^L$ die Klasse aller Sprachen, die von Turingmaschinen mit Orakel L sowie dem gleichen Zeit- bzw. Platzbedarf wie in $\mathcal{C}$ entschieden werden.

2 Relativierende Beweistechniken

Funktionieren Beweistechniken unabhängig davon, ob darin Orakel verwendet werden oder nicht, so nennt man diese *relativierende Beweistechniken*. Ein triviales Beispiel hierfür ist, dass für beliebiges Orakel L $\mathbf{P}^L \subseteq \mathbf{NP}^L$ gilt: jede deterministische Turingmaschine mit Orakel ist zugleich eine nichtdeterministische Maschine mit Orakel. Dieses Kapitel stellt zwei auf relativierenden Techniken basierende Beweise vor: den Zeithierarchiesatz und den Satz von Savitch.

2.1 Der Zeithierarchiesatz

Eine wichtige Frage in der Komplexitätstheorie ist, ob eine Turingmaschine durch das Hinzufügen neuer Eigenschaften (z.B. Nichtdeterminismus oder die Fähigkeit, die Eingabe zu überschreiben) eine neue Mächtigkeit erlangt, d.h. Sprachen entscheiden kann, die ohne diese Eigenschaft nicht zu entscheiden wären. Der Zeithierarchiesatz beschäftigt sich mit einer solchen Frage: kann eine Turingmaschine M_1, der mehr Berechnungszeit zur Verfügung steht als einer Maschine M_2, tatsächlich mehr Sprachen entscheiden? Dies ist in der Tat der Fall — als „Abfallprodukt“ erhält man das Ergebnis $\mathbf{P} \subsetneq \mathbf{EXP}$.

Sei also $t(n) \geq n$ eine konstruierbare Komplexitätsfunktion, sei M_t die Turingmaschine, die $t(n)$ berechnet. Weiter sei folgende zeitbeschränkte Version des Halteproblemes gegeben:

$$H_t = \{\, (M,x) \mid M \text{ akzeptiert Eingabe } x \text{ nach höchstens } t(|x|) \text{ Schritten}\}$$

Dabei stellt M die Kodierung einer beliebigen deterministischen Turingmaschine dar.

Lemma 3 $H_t \in \boldsymbol{TIME}(t(n)^3)$

Beweisskizze[2]: Es wird eine Turingmaschine M_{H_t} beschrieben, die H_t in Zeit $t(n)^3$ entscheidet. Die Berechnung teilt sich in zwei Phasen auf. Zunächst wird die Turingmaschine M_t mit Eingabe x gestartet: das Ergebnis (die Zeitschranke für dieses Halteproblem) wird auf einem Band gespeichert. Hierzu ist Zeit $\mathcal{O}(t(|x|))$ nötig. Im Rahmen dieser Initialisierungsphase wird auch geprüft, ob die Eingabe die richtige Form hat: handelt es sich wirklich um die Kodierung einer Turingmaschine mit einer beliebigen Eingabe? Dies geht in Zeit $\mathcal{O}(n)$ $(n = |M, x|)$. Damit ergibt sich bis zu diesem Zeitpunkt eine Gesamtlaufzeit von $\mathcal{O}(t(|x|) + n) = \mathcal{O}(t(n))$.

In der nun beginnenden Phase simuliert M_{H_t} Schritt für Schritt die Turingmaschine M mit Eingabe x. Dabei muss zweimal pro Rechenschritt von M über die Eingabe gelaufen werden: zunächst werden alle für den aktuellen Rechenschritt nötigen Informationen gesammelt und auf einem Band zwischengespeichert, welches auch eine Kodierung des aktuellen Zustandes von M enthält. Im zweiten Rechenschritt wird der aktuelle Konfigurationsübergang von M identifiziert. Danach werden die entsprechenden Modifikationen vorgenommen, und der in der ersten Phase initialisierte Zähler wird um eins dekrementiert.

Die Überlegungen zur Laufzeit der zweiten Phase erfordern etwas mehr Aufwand. Zunächst überlege man sich, dass die Simulation eines Rechenschrittes in Zeit $\mathcal{O}(l_M k_M^2 t(|x|))$ erledigt werden kann, wobei k_M die Anzahl der Bänder von M bezeichnet und l_M die Länge der Kodierung eines Symboles bzw. Zustandes von M. Diese Werte sind jedoch nach oben beschränkt durch den Logarithmus der Länge von M! Somit ergibt sich pro simuliertem Rechenschritt von M eine Laufzeit von $\mathcal{O}(\log^3(|M|)t(|x|))$, was, großzügig nach oben abgeschätzt, $\mathcal{O}(t(n)^2)$ ergibt. Da höchstens $t(n)$ Rechenschritte simuliert werden, ist die gesamte Laufzeit der zweiten Phase $\mathcal{O}(t(n)^3)$.

Es können nun zwei Fälle eintreten. Entweder wird M die Eingabe x nach höchstens $t(|x|)$ Schritten akzeptieren – dann akzeptiert auch M_{H_t}. Andernfalls — M hat die Eingabe x abgelehnt oder die Zeitschranke ist überschritten — lehnt M_{H_t} die Eingabe ab. Die Turingmaschine M_{H_t} entscheidet H_t in Zeit $\mathcal{O}(t(n)^3)$, was durch die Zusammenfassung mehrerer Symbole zu einem[3] auf $\leq t(n)^3$ gebracht werden kann. □

Lemma 4 $H_t \notin \mathbf{TIME}(t(\lfloor \frac{n}{2} \rfloor))$

Beweis: Angenommen, es gibt eine Turingmaschine M_{H_t}, die H_t in Zeit $t(\lfloor \frac{n}{2} \rfloor)$ entscheidet. Dann kann man auch die *diagonalisierende* Turingmaschine D_t betrachten, die bei Eingabe (M) genau dann akzeptiert, wenn die Turingmaschine M_{H_t} die Eingabe (M, M) ablehnt:

$$D_t(M): \textbf{ if } M_{H_t}(M, M) = \text{„yes“ } \textbf{then } \text{„no“ } \textbf{else } \text{„yes“}$$

Offensichtlich hängt die Laufzeit von D_t von der Laufzeit von M_{H_t} ab: D_t läuft bei Eingabe (M) genauso lang wie M_{H_t} bei Eingabe (M, M). Mit $n = |M|$ erhält man somit für D_t eine Laufzeit $t(\lfloor \frac{2n+1}{2} \rfloor) = t(n)$.

[2]Die exakte Beschreibung der Turingmaschine kann in [Pap94], S. 143f nachgelesen werden

[3]Papadimitriou bezeichnet diese Technik als „linear speedup“ ([Pap94], Theorem 2.1)

Akzeptiert nun D_t sich selbst als Eingabe? Es müssen zwei Fälle unterschieden werden:

- $D_t(D_t) =$ „yes“. Dann hat die Turingmaschine M_{H_t} die Eingabe (D_t, D_t) abgelehnt, was gleichbedeutend damit ist, dass $(D_t, D_t) \notin H_t$ gilt. Aus der Definition von H_t folgt demnach, dass D_t sich selbst als Eingabe nicht innerhalb von $t(n)$ Schritten akzeptiert. Oben wurde jedoch gezeigt, dass die Laufzeit von D_t gerade $t(n)$ ist! Demnach lehnt D_t sich selbst als Eingabe ab, was äquivalent dazu ist, dass $D_t(D_t) =$ „no“ gilt.
- $D_t(D_t) =$ „no“. Die Argumentation ist die gleiche wie im ersten Fall: M_{H_t} akzeptiert die Eingabe (D_t, D_t), woraus folgt, dass $(D_t, D_t) \in H_t$ gilt. Somit akzeptiert D_t seine eigene Kodierung: $D_t(D_t) =$ „yes“.

Die Annahme, dass es eine Turingmaschine gibt, die H_t in Zeit $t(\lfloor \frac{n}{2} \rfloor)$ entscheidet, wurde demnach zu einem Widerspruch geführt. □

Kombiniert man nun die Lemmata 3 und 4, so erhält man:

Satz 5 Sei $t(n) \geq n$ eine konstruierbare Komplexitätsfunktion. Dann gilt:

$$\mathbf{TIME}(t(n)) \subsetneq \mathbf{TIME}(t(2n+1)^3)$$

Tatsächlich kann man zeigen, dass die Hierarchie innerhalb der Klassen noch viel feiner aufgelöst ist — die Funktion $t(2n+1)^3$ lässt sich durch eine beliebige Funktion $T(n)$ mit $T(n) = \omega(t(n))$ ersetzen[4]. Wichtig ist jedoch, dass die Funktion polynomiell ist, wenn nur $t(n)$ polynomiell ist! Dies führt zu folgendem Resultat:

Korollar 6 $\mathbf{P} \subsetneq \mathbf{EXP}$

Beweis: Jede polynomielle Funktion wird irgendwann kleiner als 2^n sein, demnach gilt

$$\mathbf{P} \subseteq \mathbf{TIME}(2^n) \subseteq \mathbf{EXP}$$

Nach Satz 5 gilt jedoch

$$\mathbf{TIME}(2^n) \subsetneq \mathbf{TIME}(2^{(2n+1)^3}) \subseteq \mathbf{EXP}$$

. □

2.2 Satz von Savitch

Der Zeithierarchiesatz erwies sich als nützlich dazu, zwei Komplexitätsklassen zu separieren, die auf dem deterministischen Rechenmodell basieren: $\mathbf{P} \subsetneq \mathbf{EXP}$. Savitchs Theorem beruht auf einer anderen Auffassung des Wortproblems, die sich auch über die Grenzen des Rechenmodells hinaus anwenden lässt: eine platzbeschränkte Turingmaschine hat, deterministisch oder nicht, nur endlich viele mögliche Konfigurationen. Dadurch kann sie mitsamt ihrer Eingabe als Graph aufgefasst werden, die Frage „$x \in L$?“ ist dann äquivalent dazu, ob es einen Weg von der Startkonfiguration zu einer akzeptierenden Konfiguration gibt, was im folgenden Wegproblem genannt wird.

[4] genaueres unter [Für82]

Definition 7 Ein *Konfigurationengraph* $G(M, x)$ zu einer platzbeschränkten Turingmaschine M mit Eingabe x hat als Menge der Knoten die Menge aller möglichen Konfigurationen. Zwischen zwei Knoten v_1 und v_2 existiert genau dann eine Kante, wenn es einen Konfigurationsübergang $v_1 \vdash v_2$ gibt.

Oben wurde bereits erwähnt, dass ein Konfigurationengraph endlich viele Knoten hat. Um Aussagen über die Komplexität von Algorithmen mit einem solchen Graphen als Eingabe zu treffen, reicht dies natürlich nicht aus. Jedoch lässt sich eine obere Schranke für die Anzahl der Konfigurationen angeben:

Lemma 8 Sei $G(M, x) = (V, E)$ der Konfigurationengraph einer $s(n)$-platzbeschränkten Turingmaschine. Dann gibt es eine nur von M abhängende Konstante c, so dass gilt:

$$|V| \leq c^{s(n)}$$

Beweis: Eine Konfiguration einer k-Band-Turingmaschine M mit Eingabe x enthält Informationen über den aktuellen Zustand, den Inhalt der k Bänder sowie die Positionen der k Schreib-Lese-Köpfe. Für den Zustand gibt es $|Q|$ Möglichkeiten. Da M $s(n)$-platzbeschränkt ist, gibt es pro Band $|\Sigma|^{s(n)}$ Möglichkeiten für den Bandinhalt und $s(n)$ Möglichkeiten für die Position des Kopfes. Insgesamt sind dies $|Q| \cdot |\Sigma|^{ks(n)} \cdot s(n)^k$ Möglichkeiten. Es folgt:

$$\begin{aligned}
& |Q| \cdot |\Sigma|^{ks(n)} \cdot s(n)^k \\
= \; & |Q| \cdot \left(|\Sigma|^{s(n)} \cdot s(n)\right)^k \\
= \; & \left(\sqrt[k]{|Q|}\right)^k \cdot \left(|\Sigma|^{s(n) \log_{|\Sigma|} s(n)}\right)^k \\
\leq \; & \left(\left(\sqrt[k]{|Q|} \cdot |\Sigma|\right)^{2k}\right)^{s(n)}
\end{aligned}$$

Mit $c = (\sqrt[k]{|Q|} \cdot |\Sigma|)^{2k}$ ist das gewünschte Ergebnis gezeigt. □

Nach diesen Vorüberlegungen kann nun der Satz von Savitch in Angriff genommen werden. Savitch hat einen Algorithmus entwickelt, der das Wegproblem auf sehr platzeffiziente Weise löst. Dies geht natürlich auf Kosten der Berechnungszeit, die äußerst „verschwenderisch“ benutzt wird...

Satz 9 Das Wegproblem zu einem Graphen G mit N Knoten ist auf Platz $\mathcal{O}(\log^2 N)$ lösbar.

Beweis: Sei G ein Graph mit N Knoten, seien v_1, v_2 Knoten des Graphen und $i \geq 0$. Der Ausdruck $\mathbf{PATH}(v_1, v_2, i)$ sei dann wahr, wenn es einen Weg von v_1 nach v_2 der Länge höchstens 2^i gibt. Da jeder Weg in G höchstens Länge N hat, kann das Wegproblem in G gelöst werden, wenn man den Wahrheitsgehalt von $\mathbf{PATH}(v_1, v_2, \lceil \log N \rceil)$ berechnet. Genau dies soll Aufgabe der im folgenden beschriebenen Turingmaschine sein.

Doch zunächst die sehr elegante Idee des Algorithmus: wenn es zwischen v_1 und v_2 einen Weg der Länge höchstens 2^i gibt, so muss es auf der Mitte dieses Weges einen Knoten v geben, dessen Abstand sowohl von v_1 als auch von v_2 höchstens 2^{i-1} ist. $\mathbf{PATH}(v_1, v_2, i)$ kann also auch berechnet werden, in dem man für alle Knoten v von G prüft, ob $\mathbf{PATH}(v_1, v, i-1)$ *und* $\mathbf{PATH}(v, v_2, i-1)$

gilt. Der Fall $i = 0$ ist dabei leicht zu behandeln: es muß lediglich geprüft werden, ob $v_1 = v_2$ gilt oder ob es eine Kante (v_1, v_2) gibt.

Nun zur Beschreibung der Turingmaschine. Es wird eine deterministische 3-Band-Turingmaschine benutzt, die als Eingabe die Adjazenzmatrix von G sowie geeignete Kodierungen der Knoten v_1 und v_2 sowie der Zahl i enthält. Zunächst wird das Tripel (v_1, v_2, i) auf Band 2 kopiert (dieses Band wird im folgenden immer eine Anzahl solcher Tripel enthalten). Ist nun $i = 0$, so wird anhand der Eingabe geprüft, ob v_1 und v_2 gleich bzw. adjazent sind. Ist hingegen $i \geq 1$, so wird der rekursive Algorithmus gestartet:

Als erstes wird ein Knoten v auf dem dritten Band gemerkt: dieser ist der aktuelle Kandidat für den Mittelpunkt zwischen v_1 und v_2. Nun wird das Tripel $(v_1, v, i-1)$ auf Band 2 geschrieben, die Berechnung von $\mathbf{PATH}(v_1, v, i-1)$ beginnt. Es sind zwei Fälle zu unterscheiden:

- $\mathbf{PATH}(v_1, v, i-1) =$ „no“.
  Dann wird das Tripel $(v_1, v, i-1)$ vom zweiten Band gelöscht und das lexikographisch nächste v auf Band 3 geschrieben, die Berechnung geht mit dem nächsten Knoten weiter.

- $\mathbf{PATH}(v_1, v, i-1) =$ „yes“.
  Wiederum wird $(v_1, v, i-1)$ vom zweiten Band gelöscht, nun aber durch $(v, v_2, i-1)$ ersetzt (das aktuelle v_2 kann im linken Tripel nachgeschaut werden). Wieder sind zwei Fälle zu unterscheiden:

  - $\mathbf{PATH}(v, v_2, i-1) =$ „no“.
    $(v, v_2, i-1)$ wird vom Band gelöscht. Ist noch ein zu betrachtender Knoten v vorhanden, wird die Berechnung mit dem Tripel $(v_1, v, i-1)$ fortgesetzt.
  - $\mathbf{PATH}(v, v_2, i-1) =$ „yes“.
    Zunächst wird durch Vergleichen mit dem linken Tripel festgestellt, ob es sich um den zweiten Rekursionsaufruf handelt. Ist dies der Fall, so wird $\mathbf{PATH}(v_1, v_2, i)$ mit „yes“ beantwortet.

Beispiel Die folgende Zeichnung illustriert eine Konfiguration der Turingmaschine:

Band 1	$G(M, x), v_1, v_2, \lceil \log N \rceil$
Band 2	$(v_1, v_2, \lceil \log N \rceil), (v, v_2, \lceil \log N \rceil - 1), (v, v', \lceil \log N \rceil - 2)$
Band 3	v'

In der vorliegenden Situatuion wurde ein Knoten v gefunden, der von v_1 aus über einen Weg der Länge $\leq 2^{\lceil \log N \rceil - 1}$ erreichbar ist ($\mathbf{PATH}(v_1, v, \lceil \log N \rceil - 1)$ ist wahr). Die Turingmaschine überprüft nun, ob auch $\mathbf{PATH}(v, v_2, \lceil \log N \rceil - 1)$ wahr ist. Dabei betrachtet sie einen Knoten v', um $\mathbf{PATH}(v, v', \lceil \log N \rceil - 2)$ zu prüfen.

Offensichtlich berechnet die beschriebene Turingmaschine den Wahrheitsgehalt von $\mathbf{PATH}(v_1, v_2, \lceil \log N \rceil)$ korrekt. Auf dem dritten Band befindet sich dabei stets die Kodierung eines Knoten von G, wofür Platz $\lceil \log N \rceil$ ausreicht. Auf dem zweiten Band befinden sich im schlimmsten Fall $\lceil \log N \rceil$ Tripel, von denen jedes Länge $3\lceil \log N \rceil$ hat. Insgesamt benötigt die Turingmaschine Platz $\mathcal{O}(\log^2 N)$. □

Satz 9 in Verbindung mit Lemma 8 liefert sofort das folgende wichtige Ergebnis:

Korollar 10 Sei $s(n) \geq \log n$ eine konstruierbare Komplexitätsfunktion. Dann gilt:

$$\mathbf{NSPACE}(s(n)) \subseteq \mathbf{SPACE}(s(n)^2)$$

Beweis: Um eine $s(n)$-platzbeschränkte, nichtdeterministische Turingmaschine M mit Eingabe x deterministisch zu simulieren, kann der Algorithmus aus Satz 9 zur Entscheidung des Wegproblems benutzt werden. Allerdings ist nun kein Konfigurationengraph gegeben, sondern eine Kodierung von M mit Eingabe x. Da der Algorithmus nur überprüft, ob zwei Knoten durch eine Kante verbunden sind, wird nun eine implizite Darstellung des Konfigurationengraphen benutzt: ein Ausdruck der Form $\mathbf{PATH}(v_1, v_2, 0)$ wird bewertet, indem in der Übergangsfunktion von M nachgeschaut wird, ob $v_1 \vdash v_2$ ein zulässiger Konfigurationsübergang ist.

Woher weiss die simulierende Turingmaschine, welche akzeptierende Endkonfiguration von der Startkonfiguration aus erreicht werden könnte? Hier gibt es zwei Möglichkeiten: zum einen kann man (da es auf den Zeitbedarf der Turingmaschine ja nicht ankommt) den Algorithmus mit allen akzeptierenden Endkonfigurationen als Endknoten ausführen. Dabei ist lediglich ein zusätzliches Band mit Platzbedarf $\lceil \log n \rceil$ nötig. Zum anderen kann jede Turingmaschine so modifiziert werden, dass sie nur eine akzeptierende Endkonfiguration hat. Dazu wird beim Erreichen einer akzeptierenden Konfiguration in ein (konstant langes) Unterprogramm gesprungen, welches den Inhalt aller Bänder löscht und erst dann akzeptiert.

Da der Konfigurationengraph $G(M, x)$ nach Lemma 8 $c^{s(n)}$ Knoten hat, reicht für die Simulation Platz $\mathcal{O}(s(n)^2)$. □

Korollar 11

$$\mathbf{PSPACE} = \mathbf{NPSPACE}$$

2.3 Relativierende Beweistechniken

Für die in diesem Kapitel gezeigten Beweise ist charakteristisch, dass sie auf der Simulation von Turingmaschinen basieren. Dadurch sind sie unabhängig von eventuell zur Verfügung stehenden Orakeln. Solche Beweise nennt man *relativierende Beweise*. Sei L also ein beliebiges Orakel. Im folgenden werden die einzelnen Beweise auf das L zugehörige „Universum“ übertragen.

Zeithierarchiesatz: es wurde für die Sprache H_t gezeigt, dass sie zwar in $\mathbf{TIME}(t(n)^3)$, aber nicht in $\mathbf{TIME}(t(\lfloor \frac{n}{2} \rfloor))$ liegt. Diese Sprache war aber über eine Turingmaschine definiert, die in diesem „Universum“ über das Orakel L verfügt. Die zu betrachtende Sprache ist also nun

$$H_t^L = \{ (M^L, x) \mid M^L \text{ akzeptiert Eingabe } x \text{ nach höchstens } t(|x|) \text{ Schritten} \}$$

In Lemma 3 wurde die Turingmaschine M von der Turingmaschine M_{H_t} für $t(|x|)$ Schritte simuliert. Nun wird die Turingmaschine M^L für $t(|x|)$ Schritte simuliert. Die simulierende Turingmaschine ist $M_{H_t}^L$, welcher ebenfalls das Orakel L zur Verfügung steht.

Auch Lemma 4 beruht auf der Simulation der Maschine M. Wieder ändert sich am Beweis selbst nichts, es muss nur an den entsprechenden Stellen die Befragung des Orakels mit simuliert werden. Der Widerspruch bleibt bestehen.

Insgesamt ist für ein beliebiges Orakel L gezeigt, dass gilt:

$$\mathbf{P}^L \subsetneq \mathbf{EXP}^L$$

Satz von Savitch: der Konfigurationengraph zu einer Turingmaschine M^L mit Eingabe x lässt sich genauso bilden wie der zu einer „herkömmlichen“ Turingmaschine. Die Zahl der möglichen Konfigurationen bleibt endlich, da auch das Frageband von M^L platzbeschränkt ist[5]. Demnach läßt sich auch der Beweis des Satzes von Savitch in das mit L versehene Universum übertragen, es gilt:

$$\mathbf{PSPACE}^L = \mathbf{NPSPACE}^L$$

3 P $\stackrel{?}{=}$ NP in anderen Universen

In diesem Kapitel wird das Verhältnis von **P** und **NP** im Zusammenspiel mit Orakeln untersucht. Dabei werden Erkenntnisse gewonnen, die eine ganze Klasse von Beweistechniken vom Angriff auf P $\stackrel{?}{=}$ NP ausschließen.

Die folgende Darstellung orientiert sich stark an dem ausgezeichneten Buch *Computional Complexity* von Christos Papadimitriou[6]. Die Entdeckung der weiter unten beschriebenen Orakel geht auf Theodore Baker, John Gill und Robert Solovay zurück[7].

3.1 $\mathbf{P}^A = \mathbf{NP}^A$

Gesucht wird eine Sprache, die den Unterschied zwischen Determinismus und Nichtdeterminismus aufhebt. Die Idee ist einfach: der Determinismus wird durch das Orakel so mächtig gemacht, dass Nichtdeterminismus den Turingmaschinen keine neue Fähigkeit hinzufügt. Hier kommt der Satz von Savitch ins Spiel: Savitch hat ja gerade gezeigt, dass **PSPACE** = **NPSPACE** gilt! Dies gilt es zu nutzen.

Satz 12 Sei A eine beliebige **PSPACE**-vollständige Sprache. Dann gilt:

$$\mathbf{P}^A = \mathbf{NP}^A$$

Beweis: Zu zeigen ist:

$$\mathbf{PSPACE} \subseteq \mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{NPSPACE} \subseteq \mathbf{PSPACE}$$

Da A **PSPACE**-vollständig ist, kann jede Sprache L aus **PSPACE** durch eine deterministische, polynomielle Turingmaschine entschieden werden, die zunächst die Reduktion von L auf A durchführt und dann einmal das Orakel benutzt, womit die erste Inklusion gezeigt ist. Die zweite ist klar. Um die dritte Inklusion zu zeigen, überlege man sich, dass trivialerweise **PSPACE** $\subseteq$ **NPSPACE** gilt. Das bedeutet, dass eine nichtdeterministische, polynomiell platzbeschränkte Turingmaschine Orakel-Anfragen aus eigener Mächtigkeit heraus beantworten kann. Die letzte Inklusion ist der Satz von Savitch. □

[5] Tatsächlich ist die Definition von Platzkomplexität bei Turingmaschinen mit Orakeln etwas subtiler; genaueres entnehme man z.B. [Pap94], Anmerkung 14.5.8

[6] [Pap94]

[7] [BGS75]

3.2 $\mathbf{P}^B \neq \mathbf{NP}^B$

Gesucht wird eine Sprache L und ein Orakel B, so dass $L \in \mathbf{NP}^B \setminus \mathbf{P}^B$ gilt. Der Beweis des folgenden Satzes liefert die Konstruktion des Orakels B.

Satz 13 Es existiert ein Orakel B, so dass gilt:

$$\mathbf{P}^B \neq \mathbf{NP}^B$$

Beweis: Zunächst sei die Sprache L folgendermaßen definiert:

$$L := \{0^n \mid \exists x \in B : |x| = n\}$$

Sofort ist klar, dass L in $\mathbf{NP}^B$ liegt: eine nichtdeterministische Turingmaschine rät bei Eingabe 0^n ein x mit $|x| = n$ und benutzt dann einmal das Orakel, um zu überprüfen, ob x in B liegt.

Bevor nun das Orakel B definiert wird, ist noch eine Überlegung vonnöten: man mache sich klar, dass es eine Aufzählung aller deterministischen, polynomiell zeitgebundenen Turingmaschinen mit Orakel $M_1^?, M_2^?, \ldots$ gibt. Zunächst ist klar, dass Turingmaschinen sich aufzählen lassen: die aus EBFS bekannte Sprache GÖDEL ist gerade eine solche Aufzählung. Nun kann man Turingmaschinen mit einem „Wecker“ versehen, der die Berechnung abbricht, wenn eine polynomielle Zeitschranke n^k überschritten wird. Eine gesuchte Aufzählung ist nun wie folgt: zunächst kommt $M_1^?$ mit Zeitschranke n^1, dann $M_1^?$ und $M_2^?$ mit Schranke n^2, es folgen $M_1^?, M_2^?, M_3^?$ mit Schranke n^3 ... Wichtig: in dieser Aufzählung kommen zu jeder Turingmaschine $M^?$ unendlich viele äquivalente (d.h. die gleiche Sprache in der gleichen Zeit wie $M^?$ entscheidende) Turingmaschinen vor. Die Aufzählung enthält nämlich alle gültigen Kodierungen von Turingmaschinen, also auch solche, die z.B. „unnütze“, d.h. nie verwendete Zustände enthalten. Diese Eigenschaft wird später von großem Nutzen sein.

Jetzt geht es an die Definition von B. Diese geschieht schrittweise: am Anfang des i-ten Schrittes ist B_{i-1} bereits berechnet und enthält alle Worte x aus B mit $|x| < i$. Als Hilfsmittel ist noch eine Menge X nötig, in der die Worte gemerkt werden, die nicht zu B hinzugefügt werden dürfen.

Am Anfang gilt $B_0 = X = \emptyset$. Um nun B_i zu definieren, wird die Turingmaschine M_i^B mit Eingabe 0^i für $i^{\log i}$ Schritte simuliert. Wichtig ist dabei die Anzahl der simulierten Schritte: $i^{\log i}$ wächst langsamer als jede exponentielle, aber schneller als jede polynomielle Funktion.

Nun kann M_i^B im Verlaufe der Simulation natürlich das (noch nicht komplett definierte) Orakel B befragen. Wie wird eine solche Frage „$x \in B$?“ beantwortet? Es sind zwei Fälle zu unterscheiden:

- $|x| < i$. Dies läßt sich schon in B_{i-1} nachschauen, bereitet also keine Probleme.
- $|x| \geq i$. An dieser Stelle kommt die Menge X ins Spiel: die Frage „$x \in B$?“ wird mit „no“ beantwortet, und x wird in X gemerkt.

Wie wird die Turingmaschine M_i^B die Eingabe 0^i bewerten? Hier sind drei Fälle zu unterscheiden:

- M_i^B lehnt Eingabe 0^i innerhalb von $i^{\log i}$ Schritten ab.
  Hier wird B_i definiert als $B_{i-1} \cup \{x \in \{0,1\}^* \mid |x| = i, x \notin X\}$. Daraus folgt

nach Definition von L sofort, dass $0^i \in L$, da B nun mindestens ein Wort der Länge i enthält. Denn die Menge $\{x \in \{0,1\}^* \mid |x| = i, x \notin X\}$ kann nicht leer sein: alle bisher simulierten Turingmaschinen haben zusammen nicht mehr als $\sum_{j=1}^{i} j^{\log j}$ Schritte gemacht, demnach kann X nicht mehr als $\sum_{j=1}^{i} j^{\log j}$ Worte enthalten. Es gilt jedoch $\sum_{j=1}^{i} j^{\log j} < 2^i = |\{0,1\}^i|$. Da $0^i \in L$ gilt, M_i^B die Eingabe 0^i jedoch abgelehnt hat, kann L nicht die von M_i^B entschiedene Sprache sein.

- M_i^B akzeptiert Eingabe 0^i innerhalb von $i^{\log i}$ Schritten.
  Dann wird $B_i = B_{i-1}$ gesetzt. B enthält also keine Worte der Länge i und wird auch nie welche enthalten, da nur in der i-ten Runde Worte der Länge i zu B hinzugefügt werden können! Nach Definition von L gilt also $0^i \notin L$, womit die Maschine M_i^B, die 0^i ja akzeptiert hat, nicht die Sprache L entscheidet.

- M_i^B hält nicht innerhalb von $i^{\log i}$ Schritten.
  In diesem Fall ist die polynomielle Schranke $p(n)$ von M_i^b so groß und i so klein, dass $i^{\log i} < p(i)$ gilt. Es wird genauso wie im zweiten Fall verfahren: $B_i = B_{i-1}$. Oben wurde jedoch gezeigt, dass Turingmaschinen, die äquivalent zu M_i^B sind, unendlich oft in der Aufzählung der Maschinen auftauchen. Demnach gibt es eine Turingmaschine M_I^B mit $L(M_I^B) = L(M_i^B)$ und der gleichen Laufzeit wie M_i^B, bei der $I^{\log I} \geq p(I)$ gilt. Für diese Turingmaschine trifft jedoch einer der Fälle eins und zwei zu, womit gezeigt ist, dass auch M_i^B nicht die Sprache L entscheidet.

Insgesamt wurde für *alle* polynomiell zeitbeschränkten Turingmaschinen nachgewiesen, dass sie nicht die Sprache L entscheiden. Es wurde also $L \notin \mathbf{P}^B$ gezeigt. □

3.3 Bedeutung für P $\stackrel{?}{=}$ NP

Die Existenz von widersprüchlichen Orakeln zu P $\stackrel{?}{=}$ NP schliesst alle relativierenden Beweisverfahren aus. Interessanterweise tut sich die Gemeinde der Komplexitätstheorie-Forscher jedoch schwer darin, diese Aussage zu formalisieren. Klar ist, dass alle auf der Simulation von Turingmaschinen basierenden Beweise in diese Kategorie fallen: wenn die eine Turingmaschine in einem Rechenschritt das Orakel benutzt, tut die andere es eben auch.

Insgesamt wird die Existenz von widersprüchlichen Orakeln zu einem Problem als starker Hinweis darauf gesehen, dass das Problem sehr schwer lösbar ist. Eine Zeitlang gab es sogar die Vermutung, dass solche Probleme mit den heutigen Mitteln der Mathematik bzw. der theoretischen Informatik überhaupt nicht zu lösen sind! Dies konnte jedoch 1985 von Juris Hartmanis auf sehr elegante Weise widerlegt werden[8] – der Beweis verwendet wiederum Orakel und eine „doppelte Relativierung“.

Ende 1999 gelang es Adi Shamir gar, die Identität zweier Komplexitätsklassen zu zeigen, für die ein separierendes Orakel bekannt war: **IP** = **PSPACE**. Damit gab es zum ersten Mal eine nicht relativierende Beweismethode, die ein Orakel schlug.

[8] [Har85]

3.4 Verbleibende Beweisansätze

- Polynomielle Algorithmen
  Die Möglichkeit, einen solchen Algorithmus[9] für ein **NP**-vollständiges Problem zu finden, ist natürlich erst dann ausgeschlossen, wenn $\mathbf{P} \neq \mathbf{NP}$ gezeigt ist. Zwei Gründe sprechen aber dagegen: erstens ist dies noch für keines der zahlreichen **NP**-vollständiges Problem gelungen, und zweitens gibt es eine Menge auf der Annahme von $\mathbf{P} \neq \mathbf{NP}$ basierende Ergebnisse, die sich gut in die vorhandene Theorie integrieren. Wer sich trotzdem auf die Suche machen will, findet unter [Cre02] eine ganze Sammlung **NP**-vollständiger Probleme.

- Untere Schranken
  Das Zeigen von unteren Schranken hat sich selbst für (vermeintlich leichtere) Probleme aus **P** als ausserordentlich schwierig erwiesen.

- Algebraische Ansätze
  Valiant hat 1980 ein algebraisches Pendant zu **NP** vorgeschlagen; dem Rechenschritt einer Turingmaschine entspricht dort eine arithmetische Operation bzw. ein Vergleich zweier Werte aus einer Menge wie den reellen Zahlen, was den Einsatz anderer mathematischer Methoden erlaubt. Weitere Informationen liefert z.B. [Bür98].

- Die Endliche Modelltheorie
  Man kann zeigen, dass Aussagen über bestimmte endliche Strukturen (z.B. Graphen) äquivalent zu Problemen aus der theoretischen Informatik sind. Hat man diese erst einmal in die mathematische Sprache der Logik übersetzt, so bieten sich neue Techniken zur Untersuchung an. Eine Einführung in die Modelltheorie findet sich z.B. unter [Vää99].

- Interaktive Beweise
  Hier werden Turingmaschinen betrachtet, die über ein *Protokoll* kommunizieren, was zu erstaunlichen Ergebnissen führt. Der Beweis, dass $\mathbf{IP} = \mathbf{PSPACE}$ gilt, gehört in diese Kategorie. Ein erster Einstieg in die Thematik ist unter [Bau96] zu finden; [Cha01] stellt wichtige Aussagen dieses recht neuen Bereiches der theoretischen Informatik humorvoll und leicht verständlich dar.

Literatur

[Bau96] Sven Baumer: *Einführung in interaktives Beweisen*, Abteilung Theoretische Informatik der Universität Ulm, 1996
`http://theorie.informatik.uni-ulm.de/Seminare/IP/ip.ps`

[BGS75] Theodore Baker, John Gill, Robert Solovay: *Relativizations of the P=?NP question*, SIAM Journal of Computing 4, 1975

[9] In [Gas02] findet sich eine hübsche, wenn auch nicht ganz ernstzunehmende Bemerkung: „There will be an $O(n^{\log^*(n)})$ algorithmn for an NP-complete problem, rendering the whole P $\stackrel{?}{=}$ NP question essentially irrelevant :-)“

[Bür98] Peter Bürgisser: *Completeness and Reduction in Algebraic Complexity Theory*, Habilitationsschrift, Universität Zürich, 1998
`http://www.math.unizh.ch/~buerg/papers/habil.ps.gz`

[Cha01] Bernard Chazelle: *The PCP Theorem*, Bourbaki Seminar 895, 2001
`http://www.cs.princeton.edu/~chazelle/pubs/bourbaki.ps`

[Cre02] Pierluigi Crescenzi, Viggo Kann: *A compendium of NP optimization problems*, WWW, 2002
`http://www.nada.kth.se/~viggo/problemlist/compendium.html`

[Für82] Martin Fürer: *The tight deterministic time hierarchy*, Proceedings of the 14th annual ACM Symposium on Theory of Computing, 1982

[Gas02] William I. Gasarch: *The P=?NP Poll*, SIGACT News Complexity Theory Column 36, 2002
`http://www.cs.umd.edu/%7Egasarch/papers/poll.ps`

[Har85] Juris Hartmanis: *Solvable Problems with Conflicting Relativizations*, Bulletin of the European Association for Theoretical Computer Science 27, 1985

[Pap94] Christos H. Papadimitriou: *Computional Complexity*, Addison-Wesley Publishing Company, 1994

[Vää99] Jouko Väänänen: *A Short Course on Finite Model Theory*, Department of Mathematics, University of Helsinki, 1999
`http://www.math.helsinki.fi/~logic/people/jouko.vaananen/shortcourse.pdf`