Komplexitätstheorie

Arfst Nickelsen
Universität zu Lübeck
Institut für Theoretische Informatik
Wintersemester 2006/07

Stand 8. Februar 2007

Inhaltsverzeichnis

Vorbemerkung

Die Lehrveranstaltung „Komplexitätstheorie" fand (in dieser Form zum ersten Mal) im WS 2006/07 an der Universität zu Lübeck statt. Sie wandte sich an Studierende des Diplomstudiengangs und des Masterstudiengangs Informatik, die sich im Hauptstudium befanden. Die Veranstaltung wurde als zweistündige Vorlesung mit einstündiger Übung durchgeführt. Im Laufe des Semesters wurden 11 Aufgabenblätter ausgegeben.

Bei der Durchführung der Veranstaltung wurden Kenntnisse, wie sie im Bereich Algorithmen und Komplexität üblicherweise im Grundstudium bzw. Bachelorstudium erworben werden, vorausgesetzt. Dieses Skript beruht in Teilen auf einem Skript zu einer sogenannten Basisveranstaltung am Fachbereich Informatik der Technischen Universität Berlin mit dem Titel *Berechnungsmodelle und Komplexität* [NT04].

Definitionen und Beweise einzelner Sätze sind oft aus bewährten Lehrbüchern übernommen. Insbesondere möchte ich die Bücher von Papadimitriou *Computational Complexity* [Pap94], Balcázar, Díaz und Gabarró *Structural Complexity I* [BDG93], Wagner *Einführung in die Theoretische Informatik* [Wag94] und Schöning *Complexity and Structure* [Sch85] nennen. Auf weitere benutzte Literatur wird in den einzelnen Kapiteln verwiesen.

Ich danke Jan Arpe für die Unterstützung bei der Durchführung der Veranstaltung.

Web-Site der Vorlesung: `http://www.tcs.uni-luebeck.de/Lehre/2006-WS/Complexity/index.html`.
Web-Site des Autors: `http://www.tcs.uni-luebeck.de/pages/arfst`.
Email-Adresse des Autors: `Arfst.Nickelsen@tcs.uni-luebeck.de`.

Lübeck, Februar 2007 Arfst Nickelsen

1 Probleme, Ressourcen, Klassen

1.1 Probleme, Maschinen, Ressourcen

In der Komplexitätstheorie geht es darum, Berechnungsprobleme nach ihrer Komplexität zu klassifizieren. Mit Komplexität ist dabei der notwendige „Aufwand“ oder unvermeidbare „Ressourcenverbrauch“ gemeint. Der Ressourcenverbrauch kann dabei stark vom betrachteten Berechnungsmodell abhängen.

Wir unterteilen nach *Art des Berechnungsproblems*:

- *Entscheidung von Sprachen*

  Für eine Eingabe x soll die Zugehörigkeit zu einer vorgegebenen Menge A, auch *Sprache* genannt, entschieden werden. Ist $x \in A$, so wird x *akzeptiert*; ist hingegen $x \notin A$, so wird x *verworfen*.

- *Funktionsberechnung*

  Eine Funktion f soll berechnet werden. Bei Eingabe x soll die Ausgabe $f(x)$ produziert werden.

- *Konstruktionsprobleme*

  Zur Eingabe x, hier *Probleminstanz* genannt, soll eine *Lösung* ausgegeben werden. Die Menge der zulässigen Lösungen zu x besteht nicht notwendigerweise nur aus einem Element.

- *Optimierungsprobleme*

  Optimierungsprobleme sind Konstruktionsprobleme, bei denen Lösungen zusätzlich ein Wert zugeordnet ist. Zu einer Probleminstanz x soll eine Lösung mit optimalem Wert oder zumindest möglichst hohem/niedrigem Wert ausgegeben werden.

Funktionsberechnungen sind spezielle Konstruktionsaufgaben, Sprachentscheidung kann als Berechnung der charakteristischen Funktion angesehen werden. Die charakteristische Funktion zu einer Sprache A ist definiert als

$$\chi_A(x) := \begin{cases} 1, & \text{falls } x \in A, \\ 0 & \text{sonst.} \end{cases}$$

Weiterhin unterteilen wir nach dem *Algorithmentyp* und nach der *Art des Berechnungsmodells*, neudeutsch *computing device*. Wir werden betrachten:

- *deterministisch, sequenziell* (deterministische Turingmaschinen)
- *nichtdeterministisch* (nichtdeterministische Turingmaschinen)
- *parallel* (Schaltkreise)
- *probabilistisch* (Turingmaschinen mit Zugriff auf Zufallsbits)

Außerdem unterteilen wir nach *Art der betrachteten Ressource*, insbesondere nach:

- *Rechenzeit*
  Anzahl der elementaren Schritte, die zur Berechnung benötigt werden.

- *Speicherplatz*
  Anzahl der elementaren Speicherplätze wie Band- oder Speicherzellen oder Register, die während der Berechnung benutzt werden.

Andere Parameter von Algorithmen oder Berechnungsmodellen kann man oft auch als Ressource auffassen. Etwa die Schaltkreistiefe, die bei Schaltkreisen die Rechenzeit ersetzt, die Schaltkreisgröße, die Anzahl der nichtdeterministischen Verzweigungen (bei nichtdeterministischen Algorithmen), die Anzahl der „Zufallsbits“ (bei probabilistischen Algorithmen), die Güte der Näherung (bei Approximationsalgorithmen) oder auch die Programmgröße.

In der Komplexitätstheorie liegt der Schwerpunkt nicht auf dem *Algorithmenentwurf* für spezielle Probleme, sondern man sucht nach prinzipiellen oberen und unteren Schranken für den Ressourcenverbrauch für bestimmte Probleme oder Problemgruppen und fragt nach den Beziehungen zwischen den verschiedenen Berechnungsmodellen und Ressourcenmaßen.

1.2 Beispiele von Berechnungsproblemen

Um die Unterscheidung in Sprach-, Funktions- und Konstruktionsprobleme zu illustrieren, stellen wir einige Berechnungsprobleme vor. Sie stammen aus drei typischen Themengebieten. Erstens aus dem Bereich der Auswertung aussagenlogischer Formeln, zweitens aus der Graphentheorie und drittens aus dem Gebiet Arithmetik/Zahlentheorie.

Die Probleme sind hier als Paare von Eingabebeschreibung und zu lösender Fragestellung angegeben. Eine andere Art, beispielsweise ein Entscheidungsproblem anzugeben, ist die Angabe der Menge, deren Elemente der Algorithmus akzeptieren soll. Solche Mengen sind dann Teilmengen der Menge der Worte über einem geeigneten Alphabet Σ. Entsprechend sind Funktionsprobleme „eigentlich" Funktionen von Σ^* nach Γ^* und Konstruktionsprobleme Teilmengen von $\Sigma^* \times \Gamma^*$, also zweistellige Relationen, für geeignete Alphabete Σ und Γ.

Man kann sich in der Regel auf das Eingabealphabet (und Ausgabealphabet) $\Sigma\ (= \Gamma) = \{0,1\}$ beschränken, da sich Eingaben über anderen Alphabeten leicht in Eingaben über $\{0,1\}$ umkodieren lassen.

Zur Problembeschreibung gehört also auch immer die Angabe einer Kodierung der Eingabeobjekte (Formeln, Graphen, Zahlen, etc.) als Zeichenkette. Das Problem SAT (siehe unten) wäre, bei vorgegebener Kodierung für Formeln, dann

$$\textsc{Sat} = \{w \in \Sigma^* \mid w \text{ kodiert eine aussagenlogische Formel } \varphi;\ \varphi \text{ ist erfüllbar}\}\,.$$

Ist die Kodierung festgelegt, oder sind die Details der Kodierung für den Berechnungsaufwand nicht erheblich, so schreibt man auch

$$\textsc{Sat} = \{\varphi \mid \varphi \text{ ist erfüllbare aussagenlogische Formel}\}\,.$$

Aussagenlogische Formeln

- SAT als Entscheidungsproblem

  Eingabe: Eine aussagenlogische Formel φ
  Frage: Existiert eine erfüllende Belegung für φ?

- SAT als Konstruktionsproblem

  Eingabe: Eine aussagenlogische Formel φ
  Ausgabe: Eine erfüllende Belegung für φ oder „φ ist nicht erfüllbar"

Das **Erfüllbarkeitsproblem** SAT spielt in der Komplexitätstheorie eine wichtige Rolle, weil es *das* kanonische NP-vollständige Problem ist. Der Beweis von Cook der NP-Vollständigkeit von SAT von 1971 [Coo71] ist ein früher Meilenstein in der Geschichte der Komplexitätstheorie.

Das Verhältnis der beiden Varianten von SAT zueinander ist typisch. Beim Entscheidungsproblem wird nach der Existenz einer Lösung für eine Problemstellung gefragt; beim Konstruktionsproblem soll eine solche Lösung tatsächlich angegeben werden. Bei SAT und vielen anderen Problemen kann man aus Algorithmen, die das Entscheidungsproblem effizient lösen, Algorithmen bauen, die auch das Konstruktionsproblem effizient lösen.

Graphen

- PATH als Entscheidungsproblem

  Eingabe: Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
  Frage: Existiert ein gerichteter Pfad von s nach t?

- PATH als Konstruktionsproblem

  Eingabe: Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
  Ausgabe: Ein gerichteter Pfad von s nach t oder „kein Pfad von s nach t"

- PATH als Optimierungsproblem

  Eingabe: Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
  Ausgabe: Ein gerichteter Pfad von s nach t
  Ziel: Der Pfad von s nach t soll möglichst kurz sein.

Viele algorithmische Probleme sind Graphen-Probleme oder lassen sich in solche umformulieren. Das **Erreichbarkeitsproblem** PATH ist ein Basisproblem in diesem Bereich. Es ist auch unter den Bezeichnungen s-t-CON (für *s-t*-connectivity) und GAP (für graph accessibility problem) bekannt.

Arithmetik/Zahlentheorie

- PRIMES (Entscheidungsproblem)

  Eingabe: Eine natürliche Zahl n (binär kodiert)
  Frage: Ist n eine Primzahl?

- FACTORING (Konstruktionsproblem)

  Eingabe: Eine natürliche Zahl n (binär kodiert)
  Ausgabe: Zwei Zahlen $a, b < n$ mit $ab = n$ oder „n ist prim"

Für das Problem PRIMES gibt es einen deterministischen Algorithmus mit polynomieller Laufzeit [AKS02] und einige relativ schnelle probabilistische Polynomialzeit-Algorithmen [MR95]. Für das Problem FACTORING sind keine deterministischen oder probabilistischen Polynomialzeit-Algorithmen bekannt.

1.3 Turingmaschinen

Als grundlegendes Berechnungsmodell verwenden wir Turingmaschinen.

Eine *Turingmaschine* hat folgende Bestandteile:

- Ein *Eingabeband* und $k \geqslant 0$ viele zusätzliche *Arbeitsbänder*. Die Bänder sind in (beidseitig unendlich viele) Felder unterteilt. In jedem Feld steht ein Symbol aus einem endlichen *Bandalphabet* Γ. Das *Leerzeichen* $\Box \in \Gamma$ deutet an, dass in einem Feld, wo es steht, eigentlich nichts steht.
- Ein *Lese- und Schreibkopf* für jedes Band.
- Eine Steuereinheit, die sich in einem der Zustände aus einer endlichen Zustandsmenge Q befindet, Informationen über die von den Köpfen gelesenen Symbole bekommt und deren Aktivitäten steuert.

Eine Turingmaschine arbeitet taktweise. In einem Arbeitstakt kann sie

- in Abhängigkeit vom gegenwärtigen Zustand und von den gelesenen Bandsymbolen
- einen neuen Zustand annehmen, die gelesenen Bandsymbole verändern und jeden der Köpfe um maximal ein Feld bewegen.

Definition 1.1. Eine Turingmaschine ist vollständig bestimmt durch ein Tupel $(Q, \Sigma, \Gamma, k, q_A, Q_{\text{akz}}, \Delta)$, wobei die Bedeutungen der einzelnen Komponenten in der folgenden Tabelle angegeben sind.

Symbol	Bedeutung
Q	endliche Menge von Zuständen
Σ	endliches Eingabealphabet
Γ	endliches Arbeitsalphabet, $\Sigma \subseteq \Gamma$, $\Box \in \Gamma$
k	Anzahl der Arbeitsbänder
q_A	ein spezieller Anfangszustand aus Q
Q_{akz}	Menge akzeptierender Zustände, $Q_{\text{akz}} \subseteq Q$
Δ	Übergangsrelation, $\Delta \subseteq \big(Q \times \Gamma^{k+1}\big) \times \big(Q \times \Gamma^{k+1} \times \{L, R, N\}^{k+1}\big)$

Eine Turingmaschine heißt *deterministisch*, falls Δ eine partielle *Funktion* ist, falls es also zu jedem $l \in Q \times \Gamma^{k+1}$ höchstens ein $r \in Q \times \Gamma^{k+1} \times \{L, R, N\}^{k+1}$ gibt, so dass $(l, r) \in \Delta$.

Im Folgenden ist, solange nichts anderes gesagt wird, immer $\Sigma = \{0, 1\}$.

Bei *offline-Maschinen* ist das Beschreiben des Eingabebandes nicht zugelassen. Der Kopf auf dem Eingabeband darf sich auch nicht über das Eingabewort und die zwei Felder direkt links und direkt rechts von dem Eingabewort hinausbewegen. Solche Maschinen zu betrachten, ist insbesondere angebracht, wenn man den Platzverbrauch von Algorithmen untersuchen will. Maschinen mit gesondertem nicht lesbaren *Ausgabeband* werden für Konstruktionsprobleme und Funktionsberechnungen verwendet.

Definition 1.2 (Konfiguration)**.** Eine *Konfiguration* ist eine Beschreibung einer momentanen Gesamtsituation der Turingmaschine. Die Menge der Konfigurationen $\mathrm{Konf} \subseteq Q \times \left(\Gamma^* \circ \{\triangleright\} \circ \Gamma^+\right)^{k+1}$ enthält also Tupel der Form $(q, w_0, \ldots, w_k)$, wobei jedes w_i genau einmal „$\triangleright$“ enthält und rechts von „$\triangleright$“ mindestens ein Zeichen steht. „$\triangleright$“ markiert die jeweilige Kopfposition.

Definitionen 1.3 (Berechnungen)**.**

- Die Konfiguration $c_2 \in \mathrm{Konf}$ ist *Nachfolgekonfiguration* von $c_1 \in \mathrm{Konf}$, falls sich c_2 durch eine Anwendung einer Instanz von Δ aus c_1 ergibt. Notation: $c_1 \vdash c_2$; und $c_1 \vdash^* c_2$ für den reflexiven, transitiven Abschluss (c_2 ist von c_1 aus erreichbar). Die *Anfangskonfiguration* $(q_A, \triangleright w, \triangleright\square, \ldots, \triangleright\square)$ gibt die Situation der Maschine bei Beginn der Berechnung wieder.
- Eine *Berechnung* ist eine Folge $c_0, c_1, c_2, \ldots$ von Konfigurationen, so dass c_0 Anfangskonfiguration ist und $c_i \vdash c_{i+1}$ für alle in Frage kommenden i gilt. Unendliche Berechnungen sind zunächst zugelassen. Eine *stoppende Berechnung* ist eine nicht verlängerbare endliche Berechnung. Eine *akzeptierende Berechnung* ist eine stoppende Berechnung, bei der in der letzten Konfiguration der Zustand in Q_{akz} ist. Eine *verwerfende Berechnung* ist eine stoppende, nicht akzeptierende Berechnung.
- *Akzeptiertes Wort*: M akzeptiert w gdw es eine akzeptierende Berechnung von M bei Eingabe w gibt.
- Sei $A \subseteq \Sigma^*$ und M eine (Akzeptor-)Turingmaschine. Dann ist A ist *die von M akzeptierte Sprache*, geschrieben $A = L(M)$, falls
  $$w \in A \iff (M \text{ akzeptiert } w).$$

1.4 Zeit- und Platzklassen

Die zwei entscheidenden Aufwandsmaße bei Turingmaschinen sind Zeit und Platz.

Definition 1.4 (Zeit- und Platzaufwand)**.**
Der *Zeitaufwand von M bei Eingabe w* ist

$$t_M(w) := \text{Länge der längsten Berechnung von } M \text{ bei Eingabe } w.$$

Gezählt wird die Anzahl der Schritte.

Der Platzaufwand von M bei Eingabe w bei Berechnung b ist die Anzahl der während der Berechnung von den Köpfen auf den Arbeitsbändern besuchten Felder.

Der *Platzaufwand von M bei Eingabe w* ist

$$s_M(w) := \text{Platzaufwand der platzintensivsten Berechnung von } M \text{ bei Eingabe } w.$$

Der *Zeitaufwand* $T_M \colon \mathbb{N} \to \mathbb{N}$ von M ist definiert durch

$$T_M(n) := \max\{\, t_M(w) \mid |w| = n \,\}.$$

Der *Platzaufwand* $S_M \colon \mathbb{N} \to \mathbb{N}$ von M ist definiert durch

$$S_M(n) := \max\{\, s_M(w) \mid |w| = n \,\}.$$

Sprachen mit gleicher bzw. ähnlicher Oberschranke für die Aufwandsfunktion werden zu *Komplexitätsklassen* zusammengefasst.

Definition 1.5 (TIME- und SPACE-Sprachklassen)**.**
Es sei $A \subseteq \Sigma^*$ eine Sprache, $t\colon \mathbb{N} \to \mathbb{N}$ eine Zeitschranke und $s\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke.

- Es ist $A \in \mathrm{DTIME}(t)$, falls eine DTM M existiert mit $L(M) = A$ und $T_M \in O(t)$.
- Es ist $A \in \mathrm{DSPACE}(s)$, falls eine offline-DTM M existiert mit $L(M) = A$ und $S_M \in O(s)$.
- Es ist $A \in \mathrm{NTIME}(t)$, falls eine NTM M existiert mit $L(M) = A$ und $T_M \in O(t)$.
- Es ist $A \in \mathrm{NSPACE}(s)$, falls eine offline-NTM M existiert mit $L(M) = A$ und $S_M \in O(s)$.

Definition 1.6 (Grundlegende Komplexitätsklassen)**.**

$$\begin{aligned}
\mathrm{P} &= \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(n^k) & \mathrm{NP} &= \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}(n^k)\\
\mathrm{PSPACE} &= \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}(n^k) & \mathrm{NPSPACE} &= \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}(n^k)\\
\mathrm{L} &= \mathrm{DSPACE}(\log n) & \mathrm{NL} &= \mathrm{NSPACE}(\log n)\\
\mathrm{E} &= \bigcup_{c\in\mathbb{N}} \mathrm{DTIME}(2^{cn}) & \mathrm{NE} &= \bigcup_{c\in\mathbb{N}} \mathrm{NTIME}(2^{cn})\\
\mathrm{EXP} &= \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(2^{n^k}) & \mathrm{NEXP} &= \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}(2^{n^k})
\end{aligned}$$

Beispiele (Die Definition der Entscheidungsprobleme findet sich im Anhang A.):
PALINDROME ist in L, PATH (als Entscheidungsproblem) ist in NL, CVP und GENERABILITY in P, SAT und CLIQUE sind in NP, MAJSAT ist in PSPACE.

1.5 Ressourcenkompression

Bei den obigen Definitionen von Platz- und Zeitklassen wird keine Einschränkung gemacht bezüglich der Anzahl der Bänder der zugrundeliegenden Turingmaschinen. Dies ist gerechtfertigt, da bei Beschränkung der Bänderzahl der Verbrauch an Platz sich nicht ändert und der Verbrauch an Zeit zur Lösung eines Problems nicht zu stark steigt.

Außerdem werden dadurch, dass für den Ressourcenverbrauch nur Zugehörigkeit zu O-Klassen verlangt wird, konstante Faktoren beim Platz- und Zeitverbrauch ignoriert. Die Rechtfertigung hierfür liefern die folgenden Kompression- und Beschleunigungssätze.

Die Sätze sind für Akzeptormaschinen formuliert, aber entsprechende Sätze gelten auch für DTM mit Ausgabeband, die Funktionen berechnen.

Beweise dieser Sätze finden sich zum Beispiel in [HU79] oder [BDG93]. Hier geben wir keine Beweise an, da die Ergebnisse für das bessere Verständnis der obigen Klassendefinitionen hilfreich sind, aber nicht im Mittelpunkt unseres Interesses stehen.

Satz 1.7 (Bänder sparen bei platzbeschränkten TM)**.**
Sei M eine offline-NTM mit $k > 1$ Arbeitsbändern, die die Platzschranke $s\colon \mathbb{N} \to \mathbb{N}$ einhält. Dann existiert eine offline-NTM M' mit einem *Arbeitsband, die ebenfalls die Platzschranke s einhält, so dass $L(M) = L(M')$.*
Ist M deterministisch, so ist auch M' deterministisch.

Satz 1.8 (Platzkompression)**.** *Sei M eine offline-NTM mit k Arbeitsbändern, die die Platzschranke $s\colon \mathbb{N} \to \mathbb{N}$ einhält. Sei c eine Konstante mit $0 < c < 1$. Dann existiert eine offline-NTM M' mit k Arbeitsbändern, die die Platzschranke $c \cdot s(n)$ einhält, so dass $L(M) = L(M')$.*
Ist M deterministisch, so ist auch M' deterministisch.

Satz 1.9 (Bänder sparen bei zeitbeschränkten TM). *Sei M eine NTM mit $k > 1$ Arbeitsbändern, die die Zeitschranke $t\colon \mathbb{N} \to \mathbb{N}$ einhält. Dann existiert eine NTM M' mit* einem *Arbeitsband mit Zeitschranke $t(n) \log t(n)$, so dass $L(M) = L(M')$.*

Ist M deterministisch, so ist auch M' deterministisch.

Satz 1.10 (Alle Arbeitsbänder sparen bei zeitbeschränkten TM). *Sei M eine NTM mit $k > 0$ Arbeitsbändern, die die Zeitschranke $t\colon \mathbb{N} \to \mathbb{N}$ einhält. Dann existiert eine NTM M' ohne gesondertes Arbeitsband mit Zeitschranke $t(n)^2$, so dass $L(M) = L(M')$.*

Ist M deterministisch, so ist auch M' deterministisch.

Satz 1.11 (Zeitkompression).

Sei M eine NTM mit $k > 0$ Arbeitsbändern, die die Zeitschranke $t\colon \mathbb{N} \to \mathbb{N}$ mit $\lim_{n\to\infty} n/t(n) = 0$ einhält. Sei c eine Konstante mit $0 < c < 1$.Dann existiert eine NTM M' mit k Arbeitsbändern, die die Zeitschranke $c \cdot t(n)$ einhält, so dass $L(M) = L(M')$.

Ist M deterministisch, so ist auch M' deterministisch.

Satz 1.12 (Zeitkompression bei linearen Funktionen).

Sei M eine NTM mit $k > 0$ Arbeitsbändern, die die Zeitschranke $t\colon \mathbb{N} \to \mathbb{N}$ mit $t \in O(n)$ einhält. Sei $\epsilon > 0$ eine Konstante.Dann existiert eine NTM M' mit k Arbeitsbändern, die die Zeitschranke $(1+\epsilon)n$ einhält, so dass $L(M) = L(M')$.

Ist M deterministisch, so ist auch M' deterministisch.

1.6 Funktionenklassen

Ebenso wie man Sprachklassen definieren kann über den zum Entscheiden nötigen Platz- oder Zeitaufwand, so kann man auch Funktionenklassen einführen. Hierbei beschränken wir uns aber auf deterministische Maschinen, da sich keine Art allgemein durchgesetzt hat, wie man die von einer nichtdeterministischen Maschine berechnete Funktion definiert.

Definition 1.13 (TIME- und SPACE-Funktionenklassen).

Sei $f\colon \Sigma^* \to \Sigma^*$ eine Funktion, $t\colon \mathbb{N} \to \mathbb{N}$ eine Zeitschranke und $s\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke.

- Es ist $f \in \text{FDTIME}(t)$, falls eine DTM M mit Ausgabeband existiert, die f berechnet, und für die $T_M \in O(t)$ gilt.
- Es ist $A \in \text{FDSPACE}(s)$, falls eine offline-DTM M existiert, die f berechnet, und für die $S_M \in O(s)$ gilt. Die auf dem Ausgabeband beschriebenen Bandfelder werden beim Platzverbrauch nicht mitgerechnet.

Von besonderem Interesse sind Funktionen, die sich in polynomieller Zeit oder auf logarithmischem Platz berechnen lassen.

Definition 1.14 (Grundlegende Funktionenklassen).

$$\text{FP} = \bigcup_{k\in\mathbb{N}} \text{FDTIME}(n^k) \qquad\qquad \text{FL} = \text{FDSPACE}(\log n)$$

Eine Funktionenklasse ist *unter Komposition abgeschlossen*, wenn für zwei Funktionen f, g aus der Klasse auch immer die Hintereinanderausführung $g \circ f$ in der Klasse ist. FP und FL sind unter Komposition abgeschlossen. Für FP ist dies leicht zu sehen; das Ergebnis für FL wird im nächsten Kapitel behandelt.

Beispiele: Path als Konstruktionsproblem ist in FP (Eigentlich: Eine Funktion, die das Konstruktionsproblem löst, ist in FP). Ob Path auch in FL ist, ist bisher unbekannt. Addition und Multiplikation von zwei Binärzahlen ist in FL (und damit auch in FP).

2 Inklusionen, Hierarchiesätze, Diagonalisierung

2.1 Einfache Inklusionsbeziehungen

Wie liegen die Zeit- und Platzklassen zueinander? Wir beginnen mit einigen einfachen Inklusionsbeziehungen, die sich (fast) direkt aus den jeweiligen Definitionen ergeben.

Lemma 2.1 (Einfache Inklusionsbeziehungen)**.**

1. *Deterministische Maschinen sind spezielle nichtdeterministische Maschinen. Deshalb gilt insbesondere*

   $\mathrm{P} \subseteq \mathrm{NP}$, $\mathrm{E} \subseteq \mathrm{NE}$ *und* $\mathrm{EXP} \subseteq \mathrm{NEXP}$*; und auch* $\mathrm{L} \subseteq \mathrm{NL}$ *und* $\mathrm{PSPACE} \subseteq \mathrm{NPSPACE}$.

2. *Erhöht man den zulässigen Ressourcenverbrauch, so kann die dadurch definierte Klasse nur größer werden. Deshalb gilt insbesondere*

   $\mathrm{P} \subseteq \mathrm{E} \subseteq \mathrm{EXP}$ *und* $\mathrm{NP} \subseteq \mathrm{NE} \subseteq \mathrm{NEXP}$*; und auch* $\mathrm{L} \subseteq \mathrm{PSPACE}$ *und* $\mathrm{NL} \subseteq \mathrm{NPSPACE}$.

3. *Ein Schreib-Lese-Kopf einer Maschine, die höchstens $t(n)$ Schritte macht, kann hhöchstens $t(n)$ Bandfelder besuchen. Deshalb gilt*

   $\mathrm{P} \subseteq \mathrm{PSPACE}$ *und* $\mathrm{NP} \subseteq \mathrm{NPSPACE}$.

2.2 Beziehung zu formalsprachlichen Klassen

Man kann Sprachen nicht nur ordnen nach dem Aufwand, den Maschinen zum Entscheiden von Sprachen benötigen. Man kann auch untersuchen, welche Art von Grammatikregeln ausreichen, damit eine Grammatik genau die Wörter aus der gewünschten Sprache erzeugt. Dies führt in einer ersten Grobeinteilung zu den Stufen der sogenannten Chomsky-Hierarchie. Die unterste Stufe besteht aus den regulären Sprachen. Dann folgen die kontextfreien und kontextsensitiven Sprachen und schließlich die Stufe der überhaupt mit Grammatiken erzeugbaren Sprachen.

Die Stufen der Chomsky-Hierarchie stehen nicht beziehungslos neben den Zeit- und Platzklassen. Wir halten deshalb diese Beziehungen hier fest, ohne auf Einzelheiten einzugehen.

REG bezeichne die Menge der *regulären Sprachen*. CFL bezeichne die Menge der *kontextfreien Sprachen*. CSL bezeichne die Menge der *kontextsensitiven Sprachen*. REC bezeichne die Menge der *entscheidbaren* (auch *rekursiv* genannten) Sprachen. RE bezeichne die Menge der *von allgemeinen Grammatiken erzeugbaren* (auch *rekursiv aufzählbar* genannten) Sprachen. Es gilt $\mathrm{REG} \subsetneq \mathrm{CFL} \subsetneq \mathrm{CSL} \subsetneq \mathrm{REC} \subsetneq \mathrm{RE}$.

Satz 2.2.

1. $\mathrm{REG} \subseteq \mathrm{DSPACE}(\mathit{const})$, $\mathrm{REG} \subseteq \mathrm{DTIME}(n)$, *und* $\mathrm{REG} \subsetneq \mathrm{L}$.
2. $\mathrm{CFL} \subsetneq \mathrm{P}$ *und* $\mathrm{L} \not\subseteq \mathrm{CFL}$.
3. $\mathrm{CSL} = \mathrm{NSPACE}(n)$.
4. $\mathrm{DTIME}(t(n)) \subseteq \mathrm{REC}$ *für jede Schrankenfunktion t.*

2.3 Konstruierbare Funktionen, Hierarchiesätze, Diagonalisierung

Führt die Erhöhung von Ressourcenschranken dazu, dass die zugehörigen Klassen größer werden? Positive Ergebnisse dieser Art nennt man *Hierarchiesätze*. Hierarchiesätze kann man leichter beweisen, wenn man als Ressourcenschranken nur verhältnismäßig gutartige Funktionen zulässt. Zeit- bzw. Platzkonstruierbarkeit ist eine Art, „Gutartigkeit" formal zu fassen:

Definition 2.3 (zeitkonstruierbar, platzkonstruierbar)**.**

1. Eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ ist *zeitkonstruierbar*, falls eine deterministische Turingmaschine existiert, die für jede Eingabe der Länge n nach genau $f(n)$ Schritten stoppt.

2. Eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ ist *platzkonstruierbar*, falls es eine deterministische Offline-Turingmaschine M mit einem Arbeitsband gibt, die sich wie folgt verhält: Für jede Eingabe der Länge n stoppt M in einer Konfiguration, bei der auf dem Arbeitsband genau $f(n)$ Felder markiert sind; und M besucht während der Rechnung keine weiteren Felder auf dem Arbeitsband.

Sehr viele Funktionen sind zeit- bzw. platzkonstruierbar. Platzkonstruierbar sind zum Beispiel $\lceil \log n \rceil$, n^k, 2^n, $n!$. Die Summe und das Produkt zweier platzkonstruierbarer Funktionen sind wiederum platzkonstruierbar. Zeitkonstruierbar sind zum Beispiel cn^k, $n\lceil \log n \rceil^k$, $n \log^* n$, 2^{cn^k} für $c, k \geqslant 1$. Die Summe zweier zeitkonstruierbarer Funktionen ist wiederum zeitkonstruierbar. Wir verzichten hier auf den Beweis dieser Eigenschaften. Mehr dazu findet sich z.B. in [BDG93]

Solche Funktionen lassen sich auch verwenden, um Maschinen mit expliziten Zeitoberschranken (Stoppuhren) und expliziten Platzoberschranken zu versehen.

Der folgende Satz (ohne Beweis) zeigt beispielhaft, wie solche Stoppuhren eingesetzt werden können.

Satz 2.4 (Abbruch bei Zeitüberschreitung).
Sei M eine nichtdeterministische Turingmaschine, sei $f\colon \mathbb{N} \to \mathbb{N}$ eine zeitkonstruierbare Funktion. Für jedes $w \in L(M)$ gebe es einen akzeptierenden Berechnungspfad der Länge höchstens $f(|w|)$.

Dann existiert eine nichtdeterministische Turingmaschine M' mit $L(M) = L(M')$, so dass für alle Eingaben der Länge n jeder Berechnungspfad von M höchstens die Länge $2n + f(n)$ hat.

Ein analoger Satz (hier ebenfalls ohne Beweis) gilt für platzkonstruierbare Platzschranken:

Satz 2.5 (Abbruch bei Platzüberschreitung).
Sei M eine nichtdeterministische Turingmaschine, sei $s\colon \mathbb{N} \to \mathbb{N}$ eine platzkonstruierbare Funktion. Für jedes $w \in L(M)$ gebe es einen akzeptierenden Berechnungspfad, der höchstens $s(|w|)$ Platz verwendet.

Dann existiert eine nichtdeterministische Turingmaschine M' mit $L(M) = L(M')$, so dass für alle Eingaben der Länge n jeder Berechnungspfad von M höchstens $s(n) + 1$ Platz benötigt.

Als Nächstes wenden wir uns den Hierarchiesätzen zu. Man möchte zeigen, dass es für zwei Klassen $\mathcal{C}$ und $\mathcal{C}'$, die über den gleichen Ressourcen- und Maschinentyp definiert sind, und für die $\mathcal{C} \subseteq \mathcal{C}'$ gilt, weil für $\mathcal{C}'$ höhere Ressourcenschranken zugelassen sind, eine Sprache $A \in \mathcal{C}' \setminus \mathcal{C}$ gibt.

Die Technik, mit der eine solche Sprache konstruiert wird, heißt *Diagonalisierung*. Diagonalisierung ist ein sehr allgemeines Beweisprinzip. Eingeführt wurde es durch Cantors Beweis, dass die Potenzmenge einer Menge A immer mächtiger (größer) ist als A selbst (nachlesbar z. B. in [Hal73]). Auch der Beweis, dass das Halteproblem nicht entscheidbar ist, ist ein Diagonalisierungsbeweis (nachlesbar z. B. in [Wag94, Odi89]). Wir führen die Technik vor am Maschinentyp „deterministische Turingmaschinen“ und für die Ressource „Platz“. Dieser Hierarchiesatz wurde 1965 von Hartmanis und Stearns bewiesen:

Satz 2.6 (Platzhierarchiesatz für deterministische Maschinen).
Seien $s(n)$ und $s'(n)$ Platzschranken, s' sei platzkonstruierbar. Weiterhin sei

$$s \in o(s'), \text{ d.h. } \lim_{n\to\infty} \frac{s(n)}{s'(n)} = 0 \ .$$

Dann enthält DSPACE(s') *eine Sprache, die nicht in* DSPACE(s) *liegt.*

Beweis. Es wird eine Sprache $A \in \mathrm{DSPACE}(s') \setminus \mathrm{DSPACE}(s)$ angegeben, indem man eine deterministische TM M angibt, die auf Platz $O(s')$ arbeitet. M verfügt über zwei Arbeitsbänder. Wir setzen $A := L(M)$.

Die Maschine M arbeitet wie folgt:

1. Die Eingabe sei w mit $|w| = n$.
   Markiere $s'(n)$ nach links und nach rechts auf jedem der zwei Arbeitsbänder. Bewege die Köpfe auf den Arbeitsbändern auf die Startposition zurück. Immer wenn im Folgenden die Maschine im Begriff ist, den markierten Bereich zu verlassen, bricht sie die Berechnung ab und verwirft.

2. Prüfe, ob sich w in $1^*0w'$ zerlegen lässt. Wenn nein, verwerfe.
3. Prüfe, ob w' eine deterministische Offline-TM mit einem Arbeitsband und Arbeitsalphabet $\{0,1,\Box\}$ kodiert. Wenn nein, verwerfe. Wenn ja, sei $M_{w'}$ die kodierte TM.
4. Kopiere w' auf das zweite Arbeitsband.
5. Simuliere die Berechnung von $M_{w'}$ bei Eingabe w. Dabei werden die Kopfbewegungen des Eingabekopfes von $M_{w'}$ auf dem Eingabeband von M ausgeführt. Das erste Arbeitsband von M übernimmt dabei die Rolle des Arbeitsbandes von $M_{w'}$. Auf dem zweiten Arbeitsband, auf dem der Code von $M_{w'}$ wird der jeweils aktuelle Zustand von $M_{w'}$ gemerkt und der nächste Zustand in der Berechnung bestimmt.
6. Falls die Simulation beendet wird und $M_{w'}$ akzeptiert w, so verwerfe. Falls $M_{w'}$ hingegen w verwirft, so akzeptiere.

Die von M akzeptierte Sprache A ist offenbar in DSPACE(s').

Wäre A auch in DSPACE(s), so gäbe es eine Konstante c und eine durch $c \cdot s(n)$ platzbeschränkte deterministische Offline-TM X, die A akzeptiert. O.B.d.A. können wir annehmen, dass X ein Arbeitsband hat, das Arbeitsalphabet $\{0,1,\Box\}$, verwendet und auf allen Eingaben hält.

Betrachten wir eine solche Maschine X.

Sei w_X der Code für X. Da $s'(n)$ schneller wächst als $s(n)$, genügt für ein genügend großes m der Platz $s'\big(|1^m0w_X|\big)$, um w_X auf dem zweiten Arbeitsband zu speichern und um X auf der Eingabe 1^m0w_X zu simulieren. Dann ist das Wort 1^m0w_X genau dann in A, wenn X es verwirft. Also ist $A \neq L(X)$. $\Box$

Ein ähnlicher Satz gilt für Zeitschranken, analoge Ergebnisse gelten auch für nichtdeterministische Klassen (schwerer zu beweisen). Wir zitieren diese Sätze im Folgenden ohne Beweis. Für detaillierte Ergebnisse und Verweise auf die Originalliteratur konsultiere man [WW86] oder [HU79].

Satz 2.7 (Platzhierarchiesatz für nichtdeterministische Maschinen)**.**
Seien s und s' Platzschranken, s' sei platzkonstruierbar. Weiterhin sei

$$s \in o(s').$$

Dann enthält NSPACE(s') *eine Sprache, die nicht in* NSPACE(s) *liegt.*

Satz 2.8 (Zeithierarchiesatz für deterministische Maschinen)**.**
Seien t und t' Zeitschranken, t' sei zeitkonstruierbar. Weiterhin sei

$$t \log t \in o(t').$$

Dann enthält DTIME(t') *eine Sprache, die nicht in* DTIME(t) *liegt.*

Satz 2.9 (Zeithierarchiesatz für nichtdeterministische Maschinen)**.**
Seien t und t' Zeitschranken, t' sei zeitkonstruierbar. Weiterhin sei

$$t \in o(t').$$

Dann enthält NTIME(t') *eine Sprache, die nicht in* NTIME(t) *liegt.*

Die Diagonalisierungsergebnisse dienen dazu, Komplexitätsklassen zu trennen:

Satz 2.10. *Es gilt*

$$\begin{aligned}
\mathrm{P} &\subsetneq \mathrm{E} \subsetneq \mathrm{EXP},\\
\mathrm{NP} &\subsetneq \mathrm{NE} \subsetneq \mathrm{NEXP},\\
\mathrm{L} &\subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE},\\
\mathrm{NL} &\subsetneq \mathrm{NPSPACE} \subsetneq \mathrm{NEXPSPACE}\,.
\end{aligned}$$

3 Einiges über Platzklassen

3.1 Konfigurationen

Bei Beweisen von Eigenschaften platzbeschränkter Turingmaschinen spielt oft die Aufzählung aller möglichen Konfigurationen (oder zumindest ihre Anzahl) eine Rolle. Offline-Turingmaschinen verändern ihre Eingabe nicht. Deshalb ist die Konfiguration einer offline-TM mit einem Arbeitsband bei Eingabe x, $|x| = n$, komplett bestimmt durch den Zustand q, die Position h_{Ein} des Kopfes auf dem Eingabeband, die Position h_{Arb} des Kopfes auf dem Arbeitsband und den Inhalt des Arbeitsbandes b.

Zum Speichern der Eingabe-Kopfposition genügt ein Binärzähler mit $\log n$ Bits. Die Darstellung einer Konfiguration hat deshalb Länge $\log n + O(s(n))$. Ist $s(n) \geqslant \log n$, liegt die Konfigurationslänge in $O(s(n))$ und es gibt höchstens $2^{cs(n)}$ verschiedene Konfigurationen (für passende Konstante c). Ist $s(n) = O(\log n)$, ist die Anzahl der Konfiguration polynomiell beschränkt.

Man kann die Konfigurationsdarstellung bei platzkonstruierbaren Funktionen auf eine einheitliche Länge bringen, die nur von n abhängt, und dann in Algorithmen alle Konfigurationen systematisch aufzählen (ohne die Eingabe, die ja unveränderbar auf dem Eingabeband steht).

3.2 Verhältnis von Platz und Zeit

Wie verhalten sich Platz- und Zeitklassen zueinander? Die eine Richtung wurde bereits im vorigen Kapitel besprochen: Es gilt $\text{DTIME}(t) \subseteq \text{DSPACE}(t)$ und $\text{NTIME}(t) \subseteq \text{NSPACE}(t)$. Das folgende Resultat verbessert diese zwei Inklusionen noch:

Satz 3.1 (Nichtdeterministische Zeit versus deterministischer Platz)**.**
Sei $t(n)$ eine Zeitschranke. Dann ist

$$\text{NTIME}(t) \subseteq \text{DSPACE}(t).$$

Beweis Sei $A = L(M)$ für eine durch $t(n)$ zeitbeschränkte NTM M. O.B.d.A. sei die Übergangsrelation von M so, dass jede Konfiguration höchstens zwei Nachfolgekonfigurationen hat. Bei Eingabe eines Wortes w kann ein deterministischer Algorithmus nacheinander jede Berechnung von M simulieren. Die Folge von nichtdeterministischen Entscheidungen der aktuellen Berechnung kann man sich dabei durch einen 0/1-String der Länge höchstens $t(n)$ merken. Der Algorithmus akzeptiert w, falls eine akzeptierende Berechnung von M bei Eingabe w gefunden wird. □

Nun betrachten wir die andere Richtung:

Satz 3.2 (Nichtdeterministischer Platz versus deterministische Zeit)**.**
Sei $s\colon \mathbb{N} \to \mathbb{N}$ platzkonstruierbar und $s(n) \geqslant \log n$. Dann ist

$$\text{NSPACE}(s) \subseteq \text{DTIME}\big(2^{O(s)}\big).$$

Beweis Sei M eine durch s platzbeschränkte Offline-NTM. O.B.d.A. hat M genau eine akzeptierende Konfiguration (leere Arbeitsbänder, Eingabeband-Kopf auf erstem Zeichen). Gibt es überhaupt bei Eingabe w eine akzeptierende Berechnung, so gibt es auch eine akzeptierende Berechnung ohne Wiederholung von Konfigurationen. Die Anzahl verschiedener Konfigurationen von M bei Eingabe w mit $|w| = n$ ist $O\big(2^{O(s)}\big)$.

Betrachte eine deterministische Turingmaschine M', die bei Eingabe w einen gerichteten Graphen erzeugt, der als Knoten die Konfigurationen von M hat. Zwei Konfigurationen c, c' sind durch eine Kante verbunden, wenn c' direkte Nachfolgekonfiguration von c bezüglich M ist. Um die Anzahl der Konfigurationen auszurechnen, benutzt M' die Platzkonstruierbarkeit von s. Nun prüft M', ob es in diesem Graphen einen Pfad von der Startkonfiguration bis zur akzeptierenden Konfiguration geht. Dies benötigt polynomiell viele Schritte gemessen in der Größe des Graphen. Ein Polynom in $O\big(2^{O(s)}\big)$ ist wieder $O\big(2^{O(s)}\big)$. Die Maschine M' akzeptiert die gleiche Sprache wie M. □

3.3 Komposition von FL-Funktionen

Komplexitätsklassen sollten möglichst gewisse Abschlusseigenschaften haben. Bei Klassen von Entscheidungsproblemen prüft man als erstes, ob Abgeschlossenheit unter Schnitt, Vereinigung und Komplement vorliegt. Bei Funktionenklassen ist die Abgeschlossenheit unter Komposition grundlegend. Dass FP unter Komposition abgeschlossen ist, ist leicht zu sehen. Auch FL ist abgeschlossen, aber dies ist nicht ganz so leicht zu sehen.

Der folgende Beweis benutzt, dass für eine Funktion aus FL die Funktionswerte nur polynomiell groß bezüglich der Eingabelänge werden können. Das liegt daran, dass eine $\log n$ platzbeschränkte Maschine nur polynomiell viele verschiedene Konfigurationen hat.

Satz 3.3. *Sind $f, g \in \mathrm{FL}$, so ist auch $f \circ g \in \mathrm{FL}$.*

Beweis M_f berechne f und M_g berechne g, beide seien durch $\log n$ platzbeschränkte Offline-DTM. Wir konstruieren eine Offline-DTM M, die bei Eingabe w den Wert $f\big(g(w)\big)$ berechnet, wie folgt:

Die Maschine M simuliert M_f bei Eingabe $g(w)$. Dabei merkt sich M die Position des Eingabekopfes von M_f in einem Binärzähler i. Immer wenn M_f den Eingabekopf nach rechts (beziehungsweise nach links) bewegt, wird i um 1 erhöht (beziehungsweise erniedrigt). Jedesmal, wenn der Wert von i sich ändert, wird eine Simulation von M_g eingeschoben. In einem Zähler j wird registriert, das wievielte Zeichen M_g als nächstes auf das Ausgabeband von M_g schreiben würde. Für $j < i$ werden diese Zeichen nicht gemerkt. Deshalb geht die Länge der Ausgabe von M_g nicht in den Platzverbrauch von M ein. Gibt M_g das i-te Zeichen aus, wird die Simulation von M_g abgebrochen und die Simulation von M_f fortgesetzt.

Ist $|w|^k$ eine Oberschranke für $|g(w)|$, so wird für den Zähler i höchstens $k \log |w|$ Platz gebraucht. Die Simulationen von M_g brauchen jedesmal höchstens $\log |w|$ Platz (der gleiche Platz kann bei jeder Simulation von neuem gebraucht werden). Die Simulation von M_f braucht höchstens Platz $\log |w|^k = k \log |w|$. Insgesamt wird also nur $O(\log |w|)$ Platz verbraucht. □

3.4 Nichtdeterminismus versus Determinismus

Wie verhalten sich nichtdeterministische und deterministische Platzklassen zueinander? Bei zeitbeschränkten Maschinen steigt beim Simulieren einer NTM durch eine DTM die Rechenzeit exponentiell (zumindest ist nichts besseres bekannt). Bei platzbeschränkten Maschinen muss man beim Übergang zu deterministischen Algorithmen nur eine Quadrierung des Aufwandes in Kauf nehmen:

Satz 3.4 (Savitch 1970)**.** *Sei $s\colon \mathbb{N} \to \mathbb{N}$ platzkonstruierbar und $s(n) \geqslant \log(n)$. Dann ist*

$$\mathrm{NSPACE}(s) \subseteq \mathrm{DSPACE}(s^2).$$

Beweis Sei $L = L(M_1)$; dabei sei M_1 eine NTM mit Platzschranke s. Die Maschine M_1 habe nur ein Arbeitsband und nur eine akzeptierende Endkonfiguration. Gesucht ist eine DTM M_2 mit Platzschranke s^2, so dass $L(M_1) = L = L(M_2)$. Für die Konstruktion benutzen wir ein Prädikat Reach_x. Für Konfigurationen K_1, K_2 und $j \in \mathbb{N}$ definieren wir:

$$\mathrm{Reach}_x(K_1, K_2, j) \iff \text{von } K_1 \text{ aus ist } K_2 \text{ mit } M_1 \text{ erreichbar in höchstens } 2^j \text{ Schritten.}$$

M_2 arbeitet nun wie folgt:

Eingabe x
  if $\mathrm{Reach}_x(K_{\mathrm{Start}}, K_{\mathrm{akz}}, cs(n))$
  then akzeptiere x
  else verwerfe x.

Dabei wird das Prädikat Reach_x wie folgt ausgewertet:

Reach$_x(K_1, K_2, j)$:
  if $j = 0$
  then ***if*** $K_1 = K_2$ ***or*** K_2 ist direkter Nachfolger von K_1
    then gib *true* aus
    else gib *false* aus
  else forall Konfigurationen K:
    if Reach$_x(K_1, K, j-1)$ ***and*** Reach$_x(K, K_2, j-1)$
    then gib *true* aus
    else (für keine Konfiguration *true* ausgegeben) gib *false* aus.

Zum Platzbedarf von M_2: Die Tiefe der Rekursion ist $cs(n)$. Für jeden Aufruf der Prozedur muss man sich die Konfigurationen der Größe $O(s)$ und den Zähler j (als Binärzahl) merken. Der Gesamtplatzbedarf ist also $O(s^2)$. □

Wenden wir diesen Satz an auf Polynome als Platzschranken, so erhalten wir:

Korollar 3.5.

$$\text{PSPACE} = \text{NPSPACE}$$

3.5 Komplementabschluss

Wird eine Sprache A von einer NTM M entschieden, gibt es dann auch eine NTM M', die das Komplement $\bar{A}$ entscheidet und nicht mehr Ressourcen verbraucht als M? Für die Ressource Zeit ist dies unbekannt. Man weiß beispielsweise nicht, ob NP = co-NP.

Die analoge Frage für Platzklassen wurde 1964 von Kuroda, der bewies, dass CSL = NSPACE(n) ist, aufgeworfen. 1987 wurde sie unabhängig voneinander von dem US-Amerikaner Immerman [Imm88] und dem slowakischen Studenten Szelepcsényi [Sze88] beantwortet. Ihr Satz besagt, dass alle üblicherweise betrachteten Platzklassen unter Komplementbildung abgeschlossen sind. Den Beweis kann man beispielsweise in [BDG90] oder [Pap94] nachlesen.

Satz 3.6 (Immerman, Szelepcsényi 1987)**.**
Sei $s\colon \mathbb{N} \to \mathbb{N}$ platzkonstruierbar und $s(n) \geqslant \log n$. Dann ist

$$\text{NSPACE}(s) = \text{co-NSPACE}(s).$$

Im folgenden Beweis berechnet eine nichtdeterministische Turingmaschine einen Funktionswert. Da es grundsätzlich interessant ist, auf welche Weise man von NTMs berechnete Funktionen definieren kann, geben wir eine solche Möglichkeit an:

Definition 3.7 (von NTM berechnete Funktion)**.** Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ wird von einer NTM M_f (mit Ausgabeband und mit akzeptierenden und verwerfenden Zuständen) berechnet, falls für alle x am Ende aller akzeptierender Berechnungspfade von M_f bei Eingabe x der Wert $f(x)$ auf dem Ausgabeband steht.

Beweis des Satzes. Es genügt zu zeigen, aus $A \in \text{NSPACE}(s)$ folgt, dass auch $\overline{A} \in \text{NSPACE}(s)$ Sei $A \in \text{NSPACE}(s)$. Also ist $A = L(M)$ für eine durch $s(n)$ platzbeschränkte Offline-NTM M. O.B.d.A. habe M ein Arbeitsband, und akzeptierende Zustände werden nur in Endkonfigurationen angenommen. Gesucht ist eine NTM $\overline{M}$ mit $L(\overline{M}) = \overline{A}$. $\overline{M}$ soll also die eine Eingabe x akzeptieren, falls $x \notin L(M)$, also im Konfigurationsgraphen von M bei Eingabe x von der Startkonfiguration aus keine akzeptierende Endkonfiguration erreichbar ist.

Betrachte eine Eingabe x mit $|x| = n$. Sei Konf(x) die Menge der Konfigurationen von M bei Eingabe x. Die Elemente von Konf(x) lassen sich z.B. durch Bitstrings (0-1-strings), die alle die gleiche Länge haben, darstellen. Dadurch lassen sich die Konfigurationen linear ordnen; damit ist klar, was $\alpha < \beta$ für $\alpha, \beta \in \text{Konf}(x)$ bedeutet. Die Darstellung einer Konfiguration habe die Größe $c \cdot s(n)$ für eine passende Konstante $c \in \mathbb{N}$. (Hier wird $s(n) \geqslant \log(n)$ benutzt.) Sei $k_{\text{start}}(x)$ die Startkonfiguration bei Eingabe x.

Definiere ein Prädikat Reach durch

Reach(x, β, i) falls von $k_{\text{start}}(x)$ aus die Konfiguration $\beta \in \text{Konf}(x)$ in $\leqslant i$ Schritten erreichbar ist.

Eine NTM M_1, die Reach entscheidet, arbeitet wie folgt: M_1 rät einen Pfad von $k_{\text{start}}(x)$ nach β, merkt sich in einem Binärzähler die Anzahl der bereits gemachten Schritte. M_1 arbeitet so, wie die NTM, die belegt, dass das Problem s-t-Con in NL liegt. Der Platzverbrauch von M_1 bei gegebenem x ist linear in $|\beta| + |\operatorname{bin}(i)|$; bleibt also in $O(s(n) + \log(n)) = O(s(n))$, da nur $i \leqslant |\text{Konf}(x)| \leqslant 2^{c \cdot s(n)}$ betrachtet werden brauchen.

Wir betrachten nun die *Anzahl* der von $k_{\text{start}}(x)$ aus erreichbaren Konfigurationen. Gesucht ist eine NTM M_2, die bei Eingabe α die Anzahl $N(\alpha)$ berechnet (im Sinne von Definition 3.7). Definiere

$$N_k(\alpha) := |\{\beta \mid \beta \in \text{Konf}(x),\ \alpha \overset{*}{\underset{M}{\vdash}} \beta \text{ in höchstens } k \text{ Schritten }\}| \ .$$

Die Anzahl der überhaupt erreichbaren Konfigurationen ist dann $N_{|\text{Konf}(x)|}$ bzw. $N_{2^{c \cdot s(n)}}$. Gesucht ist eine NTM M_2, die für x die Anzahl $N_{2^{cs(n)}}$ berechnet (im Sinne von Definition 3.7).

M_2 bestimmt nacheinander (induktiv) N_0, N_1, N_2 usw., bis $N_{2^{cs(n)}}$. (Alternativ kann man auch festlegen, dass M_2 die Werte N_k berechnet, bis der Fall $N_k = N_{k-1}$ eintritt.)

Arbeitsweise von M_2:

$k = 0$ Setze $N_0 = 1$.

Schritt von k auf $k+1$ Der Wert N_k liegt vor.
zähler := 0.
forall $\beta \in \text{Konf}(x)$:
Es wird geprüft, ob β in $k+1$ Schritten erreichbar ist.
Setze reachable(β) := FALSE.
Rate Folge von Konfigurationen $\gamma_1 < \cdots < \gamma_{N_k}$.
Nacheinander, jeweils nur γ_{i-1}, γ_i speichern;
falls keine solche Folge entsteht, abbrechen und verwerfen.
forall γ_i
Prüfe jeweils mit M_1, ob Reach(x, γ_i, k) zutrifft.
if M_1 verwirft ***then*** breche ab und verwerfe.
if $\gamma_i = \beta$ oder $\gamma_i \underset{M}{\vdash} \beta$ ***then*** reachable(β) := TRUE.
end forall
if reachable(β) ***then*** zähler := zähler + 1.
end forall
N_{k+1} := zähler.

Die NTM M_2 kann die Konfigurationen systematisch aufzählen, da $s(n)$ platzkonstruierbar ist. M_2 benötigt nur $s(n)$ Platz, da nur k, N_k, i, γ_{i-1}, γ_i, zähler, reachable(β) und Bandinhalte von M_1 gespeichert werden müssen.

M_2 berechnet N_k für jedes k, und damit auch $N_{2^{cs(n)}}$, korrekt.

Beweis durch Induktion. Sei N_k korrekt berechnet. Das heißt, auf allen bisher nicht-verwerfenden Pfaden ist N_k berechnet. Für jedes β wird die Berechnung nur dann nicht abgebrochen, wenn genau die richtigen γ_i, die in höchstens k Schritten erreichbar sind, geraten werden, und wenn ihre Erreichbarkeit bestätigt wird. Auf diesem Pfad wird der Zähler zähler erhöht gdw β in $k+1$ Schritten erreichbar ist.

Nun wird die Arbeitsweise von $\overline{M}$ bei Eingabe x angegeben.

1. Bilde $k_{\text{start}}(x)$.
2. Berechne mithilfe von M_2 den Wert $N_{2^{cs(n)}}$.
3. Berechne eine aufsteigende Folge von Konfigurationen $\gamma_1 < \cdots < \gamma_N$. Dabei werden immer nur zwei direkt aufeinanderfolgende Konfigurationen γ_{i-1}, γ_i gespeichert. Für jedes γ_i teste,
   (a) ob Reach$(x, \gamma_i, 2^{cs(n)})$ (mit M_1), und
   (b) ob γ_i verwerfend.
4. Falls jedes geratene γ_i die zwei Tests besteht, akzeptiere x.

Falls $x \notin A$, so existieren $N_{2^{cs(n)}}$ verschiedene γ_i, die den Test bestehen, und x wird akzeptiert. Falls $x \in A$, so ist eine der N verschiedenen erreichbaren Konfigurationen akzeptierend, und x wird auch bei richtig geratenen γ_i verworfen. $\square$

Wenden wir diesen Satz auf uns besonders interessierende Klassen an, so erhalten wir:

Korollar 3.8.

1. NL = co-NL
2. NSPACE(n) = co-NSPACE(n)
3. CSL *ist unter Komplementbildung abgeschlossen, da* CSL = NSPACE(n).

4 Reduktionen

Reduktionen werden benutzt, um Berechnungsprobleme in ihrer Komplexität zu vergleichen. Dies erlaubt uns dann beispielsweise, für zwei Sprachen A und B zu sagen, dass A *nicht wesentlich schwerer als* B zu lösen ist. Wir können dann auch von den schwersten Problemen in einer Komplexitätsklasse sprechen. Und wir können eventuell für ein neues Problem eine obere Schranke für dessen Komplexitätbestimmen, indem wir es auf ein bekanntes Problem (mit bekanntem Aufwand) reduzieren.

A ist reduzierbar auf B soll also die Idee, dass ein „guter "Algorithmus für B einen „ebenso guten"Algorithmus für A liefert, formal fassen. Eine Reduzierbarkeitsrelation sollte daher auf jeden Fall reflexiv und möglichst auch transitiv sein. Als Notation werden daher Varianten des „Kleiner-Gleich"-Zeichens verwendet.

Einige Sprechweisen sind für alle Reduktionen üblich:

Definition 4.1 (Allgemeines zu Reduktionen)**.** Sei $\leqslant_r$ eine Reduktionsrelation.

1. Wir schreiben $A \equiv_r B$, falls $A \leqslant_r B$ und $B \leqslant_r A$. Man sagt dann, dass *A und B äquivalent sind.*
2. Für eine Sprache B sei $R_r(B) := \{ A \mid A \leqslant_r B \}$ der *Reduktionsabschluss von B.*
3. Für eine Sprachklasse $\mathcal{C}$ sei $\mathrm{R_r}(\mathcal{C}) := \{ A \mid \text{ es existiert ein } B \in \mathcal{C} \text{ mit } A \leqslant_r B \}$ der *Reduktionsabschluss von $\mathcal{C}$.*
4. Sei $\mathcal{C}$ eine Komplexitätsklasse. $\mathcal{C}$ heißt *abgeschlossen unter $\leqslant_r$-Reduktion*, falls $\mathrm{R_r}(\mathcal{C}) = \mathcal{C}$.

4.1 Many-One-Reduktion

Die einfachste Reduktionsart ist die Übersetzung von Instanzen des einen Problems in Instanzen des anderen Problems. Der eigentliche Übersetzungsprozess sollte möglichst einfach sein; weshalb wir verlangen werden, dass die Übersetzung auf logarithmischem Platz oder zumindest in polynomieller Zeit durchgeführt werden kann.

Definition 4.2 (Many-One-Reduktion)**.** Seien A und B Sprachen.

1. Wir schreiben $A \leqslant_{\mathrm{m}}^{\log} B$, falls ein $f \in \mathrm{FL}$ existiert, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \iff f(x) \in B$.
2. Wir schreiben $A \leqslant_{\mathrm{m}}^{\mathrm{p}} B$, falls ein $f \in \mathrm{FP}$ existiert, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \iff f(x) \in B$.

Lemma 4.3.

1. *Die Relationen $\leqslant_m^{\log}$ und $\leqslant_m^p$ sind reflexiv und transitiv.*
2. *Die Relationen $\equiv_m^{\log}$ sind Äquivalenzrelationen.*
3. *Es gilt $A_1 \leqslant_m^{\log} A_2 \iff \bar{A}_1 \leqslant_m^{\log} \bar{A}_2$.*
4. *Es ist $\leqslant_m^{\log} \subseteq \leqslant_m^p$, d.h. aus $A_1 \leqslant_m^{\log} A_2$ folgt stets auch $A_1 \leqslant_m^p A_2$.*

Beweis. Zum Beweis der Transitivität: Sei $A \leqslant_{\mathrm{m}}^{\log} B$ via f und $B \leqslant_{\mathrm{m}}^{\log} C$ via g. Dann ist $A \leqslant_{\mathrm{m}}^{\log} C$ via $g \circ f$, denn FL ist nach Satz 3.3 unter Komposition abgeschlossen. Die nächsten zwei Aussagen folgen direkt aus der Definition. Die letzte Aussage gilt, da logarithmisch platzbeschränkte Turingmaschinen polynomiell zeitbeschränkt sind. □

4.2 Turing- und Truth-Table-Reduktion

Andere Reduktionen formalisieren die Idee, dass ein Algorithmus für ein Problem A Zugriff auf die Ergebnisse eines Algorithmus für B hat.

Definition 4.4 (Orakel-Turingmaschine)**.** Eine Orakel-Turingmaschine (kurz: OTM) ist eine Turingmaschine mit einem speziellen Band, dem sogenannten Orakel- oder Fragenband. Die OTM hat drei spezielle Zustände q_{frage}, q_{ja} und q_{nein}.

Zur Definition von Berechnungen einer OTM M bei Eingabe x mit einem Orakel (einer Sprache) B: Wenn M im Zustand q_{frage} ist und ein Wort q auf dem Orakelband steht, so ist in der nächsten Konfiguration das Orakelband leer und M im Zustand q_{ja} oder q_{nein}. Und zwar:

$$\begin{aligned} q_{\text{ja}} &\text{ , falls } q \in B \text{ und} \\ q_{\text{nein}} &\text{ , falls } q \notin B. \end{aligned}$$

Für eine OTM M und ein Orakel B sei $L(M, B) := \{x \mid \ M \text{ akzeptiert } x \text{ mit Orakel } B\}$.

Definition 4.5 (Turing-Reduktion)**.** Für Sprachen A, B ist

1. $A \leqslant_{\mathrm{T}}^{\mathrm{p}} B$ gdw eine polynomiell zeitbeschränkte deterministische OTM M existiert mit $A = L(M, B)$.
2. $A \leqslant_{k(n)\text{-T}}^{p} B$ gdw eine polynomiell zeitbeschränkte deterministische OTM M existiert mit $A = L(M, B)$, die für jede Eingabe der Länge n höchstens $k(n)$ Fragen stellt.

Man kann zwar auch Logspace-Turing-Reduktion definieren; man muss dann aber darauf achten, ob das Fragenband beim Platzverbrauch der Maschine mitzählt oder nicht. Das Fragenband sollte nicht als versteckter Arbeitsspeicher missbraucht werden können.

Bei der Turing-Reduktion kann die Entscheidung, welche Fragen als nächstes gestellt werden, von der Antwort auf bisherige Fragen abhängen. Eine solche Art der Fragen-Erzeugung nennt man *adaptiv.* Als nächstes wird eine *nicht-adaptive* Variante der Turing-Reduktion definiert. Hier werden die Fragen *parallel* (alle auf einmal) gestellt. Diese Reduktion heißt Truth-Table-Reduktion.

Definition 4.6 (Orakel-Turingmaschine der Truth-Table-Art)**.** Man betrachtet eine Variante der Orakel-Turingmaschinen: Auf das Orakelband werden, durch Sonderzeichen getrennt, Fragen $q_1, \ldots, q_k$ geschrieben. Ist die OTM M im Zustand q_{frage}, so ist in der nächsten Konfiguration der Inhalt des Orakelbandes ersetzt durch den Bitstring $\chi_B(q_1) \ldots \chi_B(q_k)$, wobei B das verwendete Orakel ist. M darf während einer Berechnung nur höchstens einmal im Zustand q_{frage} sein.

Definition 4.7 (Truth-Table-Reduktion)**.** Für Sprachen A, B und $k\colon \mathbb{N} \to \mathbb{N}$ ist

1. $A \leqslant_{\mathrm{tt}}^{\mathrm{p}} B$ gdw eine polynomiell zeitbeschränkte deterministische OTM M der Truth-Table-Art existiert, so dass $A = L(M, B)$.
2. $A \leqslant_{k(n)\text{-tt}}^{p} B$ gdw eine polynomiell zeitbeschränkte deterministische OTM M der Truth-Table-Art existiert mit $A = L(M, B)$, die für jede Eingabe der Länge n höchstens $k(n)$ Fragen stellt.

Bei Beschränkung der Fragenzahl sind besonders die Fälle interessant, in denen k eine konstante oder eine logarithmische Funktion ist. – Das Stellen von Fragen und das Auswerten der Antworten kann man auch stärker trennen. Bei konstanter Fragenzahl ergibt sich dann die folgende alternative Definition. An ihr erkennt man, warum die Reduktion Truth-Table-, also Wahrheitstafel-Reduktion heißt.

Definition 4.8 (k-tt-Reduktion – Alternative)**.** $A \leqslant_{k\text{-tt}}^{\mathrm{p}} B$ gdw es eine polynomiell zeitbeschränkte deterministische Turingmaschine M gibt, die bei Eingabe x eine Liste von k Fragen $q_1, \ldots, q_k$ und eine k-stellige Funktion φ in Form einer Wahrheitstabelle ausgibt, derart dass

$$\chi_A(x) = \varphi(\chi_B(q_1), \ldots, \chi_B(q_k)).$$

Lemma 4.9 (Äquivalente Definitionen für k-tt)**.** *Die zwei Definitionen von polynomieller k-tt-Reduktion sind äquivalent.*

Lemma 4.10 (Inklusionen für Reduktionen)**.** *Es gilt*

1. $\leqslant_m^p \subseteq \leqslant_{\text{1-tt}}^{\text{p}} \subseteq \leqslant_{\text{2-tt}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{k\text{-tt}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{O(\log(n))\text{-tt}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{\text{tt}}^{\text{p}}$

2. $\leqslant_{\text{1-T}}^{\text{p}} \subseteq \leqslant_{\text{2-T}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{k\text{-T}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{O(\log(n))\text{-T}}^{\text{p}} \subseteq \ \ldots \ \subseteq \leqslant_{\text{T}}^{\text{p}}$

3. $\leqslant_{k(n)\text{-tt}}^{\text{p}} \subseteq \leqslant_{k(n)\text{-T}}^{\text{p}}$

4. *Für* $k(n) \in O(\log(n))$*:* $\leqslant_{k(n)\text{-T}}^{\text{p}} \subseteq \leqslant_{(2^{k(n)}-1)\text{-tt}}^{\text{p}}$

Lemma 4.11 (Komposition von k-tt)**.**
Wenn $A \leqslant_{k(n)\text{-}tt}^{p} B$ *und* $B \leqslant_{l(n)\text{-}tt}^{p} C$*, dann* $A \leqslant_{k(n)l(n)\text{-}tt}^{p} C$*.*

Satz 4.12 (Transitivität)**.**
Die Reduktionen $\leqslant_m^{\log}$*,* $\leqslant_m^p$*,* $\leqslant_{1\text{-}tt}^p$*,* $\leqslant_{btt}^p$*,* $\leqslant_{tt}^p$ *und* $\leqslant_T^p$ *sind reflexiv und transitiv.*

4.3 Trennung von Reduktionen

Lassen sich mehr Sprachen aufeinander reduzieren, wenn man mehr Fragen zulässt, von nicht-adaptiven zu adaptiven Fragen übergeht oder für die Auswertung der Orakelantworten mehr Möglichkeiten zulässt? Dies ist der Fall; die Inklusionen in Lemma 4.10, Teil *1.* und *2.* sind echt. Für eine der Inklusionen soll hier der Beweis geführt werden. Zum Beispiel gilt der folgende Satz:

Satz 4.13. *Es existieren Sprachen* A*,* B *in* EXP*, so dass* $A \leqslant_{\text{1-tt}}^{\text{p}} B$*, aber nicht* $A \leqslant_m^p B$*.*

Beweis. Es wird eine Sprache A durch Diagonalisierung konstruiert; und es wird $B := \overline{A}$ gewählt. Dann gilt sicher

$$A \leqslant_{\text{1-tt}}^{\text{p}} \overline{A} = B \ .$$

Die Sprache A wird schrittweise so konstruiert, so dass für jede Polynomialzeit-Maschine M, die eventuell als Reduktionsmaschine in Frage käme, gilt:

$$A \nleqslant_{\text{m}}^{\text{p}} \overline{A} \text{ via } M \ .$$

Sei $(M_i)_{i\in\mathbb{N}}$ eine Aufzählung von DTM, so dass

1. $T_{M_i}(n) \leqslant n^i + i$ für jedes i und n; und
2. für jedes f in FP existiert ein $i \in \mathbb{N}$, so dass M_i die Funktion f berechnet.

Eine solche Folge lässt sich leicht angeben, derart dass der Code von M_i nicht mehr als i Zeichen hat, und dieser Code bei Eingabe i in einer Zeit, die polynomiell in i ist, erzeugt werden kann. Insbesondere kann das Einhalten der Zeitschranke durch das Mitlaufen einer Stoppuhr-Turingmaschine garantiert werden.

Die Sprache A wird so konstruiert, dass sie nur Worte bestimmter, weit auseinanderliegender Wortlängen enthält. Setze $n_0 := 2$, und für alle $i \geqslant 0$ setze $n_{i+1} := 2^{n_i}$. Dann gilt (dies lässt sich durch Induktion zeigen):

$$(n_i)^i + i < n_{i+1} \ .$$

Die Sprache A wird so konstruiert, dass

$$A \subseteq \{1^{n_i} \mid i \in \mathbb{N}\} \ .$$

Sei $\chi_A(1^{n_j})$ für $j < i$ bereits festgelegt. Um zu bestimmen, ob 1^{n_i} in A ist oder nicht, simuliere M_i bei Eingabe 1^{n_i}. Sei f die von M_i berechnete Funktion.

Es gilt $|f(1^{n_i})| < n_{i+1}$, denn M_i macht weniger als n_{i+1} Schritte.

1. Falls $f(1^{n_i}) \neq 1^{n_i}$, ist also $\chi_A(f(1^{n_i}))$ bereits bekannt. Lege in diesem Fall fest:

$$1^{n_i} \in A \Leftrightarrow f(1^{n_i}) \in A\ .$$

2. Falls $f(1^{n_i}) = 1^{n_i}$, so haben wir die freie Wahl für den Wert $\chi_A(1^{n_i})$. Wir legen zum Beispiel fest:

$$1^{n_i} \in A\ .$$

Die Wahl des Wertes sichert in beiden Fällen, dass $1^{n_i} \in A \Leftrightarrow f(1^{n_i}) \in A$. Also ist die von M_i berechnete Funktion f keine Many-One-Reduktion von A auf $\overline{A}$. Da dies für jede der Maschinen M_i gilt, ist A nicht auf $\overline{A}$ many-one-reduzierbar.

Um zu zeigen, dass A, und damit auch B, in EXP sind, betrachte den Zeitaufwand des Algorithmus, der $\chi_A(x)$ zu einer Eingabe x berechnet:

- Der Algorithmus prüft, ob $x = 1^{n_i}$ für ein passendes i. Wenn nein, so ist $\chi_A(x) = 0$. Wenn $x = 1^{n_i}$, fahre fort.
- Für alle $j = 0, \dots, i$ bestimme nacheinander $\chi_A(1^{n_j})$ wie folgt:
- Für das gerade betrachtete j schreibe den Code von M_j hin. Simuliere M_j bei Eingabe 1^{n_j}. Bestimme so $\chi_A(1^{n_j})$; falls nötig, unter Verwendung der bereits berechneten Werte von χ_A.
- Gib $\chi_A(1^{n_i})$ aus.

Jede der Polynomialzeit-Maschinen M_j muss für weniger als 2^{n_i} Schritte simuliert werden. Es werden sublinear viele Maschinen simuliert. Also ist die gesamte Berechnung in exponentieller Zeit durchführbar. □

4.4 Abgeschlossenheit, Padding

Satz 4.14.

1. *Die Klassen* L, NL, P, NP, PSPACE *und* EXP *sind abgeschlossen unter* $\leqslant_{\mathrm{m}}^{\log}$*-Reduktion.*
2. *Die Klassen* P, NP, PSPACE *und* EXP *sind abgeschlossen unter* $\leqslant_m^p$*-Reduktion.*
3. *Die Klassen* P, PSPACE *und* EXP *sind abgeschlossen unter* $\leqslant_T^p$*-Reduktion.*

Beweis. Um beispielsweise die Abgeschlossenheit von NP unter $\leqslant_{\mathrm{m}}^{\mathrm{p}}$-Reduktion zu zeigen, argumentieren wir folgendermaßen: Sei $A \leqslant_{\mathrm{m}}^{\mathrm{p}} B$ via $f \in \mathrm{FP}$. Sei M_f eine DTM, die f berechnet mit Zeitschranke $p(n)$. Sei $B \in \mathrm{NP}$ via einer NTM M_B mit Zeitschranke $q(n)$. Eine NTM M_A, die A entscheidet, arbeitet bei Eingabe w wie folgt: Sie startet M_f bei Eingabe w und berechnet so $f(w)$. Dann startet sie M_B bei Eingabe $f(w)$. Sie akzeptiert, falls M_B das Wort $f(w)$ akzeptiert. Ihre Laufzeit ist in $O\big(p(n) + q\big(p(n)\big)\big)$, also polynomiell.

Beim Abschluss von L und NL unter $\leqslant_{\mathrm{m}}^{\log}$-Reduktion muss man die gleiche Technik zur Hintereinanderschaltung von Logspace-Algorithmen verwenden wie im Beweis von Satz 3.3 (Abschluss von FL unter Komposition). □

Der folgende Satz zeigt, dass nicht alle Sprachklassen unter $\leqslant_{\mathrm{m}}^{\log}$-Reduktion abgeschlossen sind. Die Beweistechnik, mit der dies bewiesen wird, heißt „Padding“(‘„Auspolstern“). Beim Padding werden Eingabeworte durch das Hinzufügen eines genügend langen Präfix oder Suffix aus 1en so lang gemacht, dass genügend Zeit (oder Platz) zur Verfügung steht, um eine gewisse Berechnung durchzuführen. Auch im Beweis der Platz- und Zeithierarchiesätze wurde schon Padding verwandt.

Satz 4.15. *Es gilt*

1. $\mathrm{R}_m^{\log}(\mathrm{E}) = \mathrm{EXP}$.
2. $\mathrm{R}_m^{\log}(\mathrm{DSPACE}(n)) = \mathrm{PSPACE}$.

Beweis. Wir beweisen nur Teil 1. Der Beweis für $\mathrm{R}_\mathrm{m}^{\log}(\mathrm{DSPACE}(n))$ geht analog.

Zunächst ist $\mathrm{E} \subseteq \mathrm{EXP}$. Also auch $\mathrm{R}_\mathrm{m}^{\log}(\mathrm{E}) \subseteq \mathrm{R}_\mathrm{m}^{\log}(\mathrm{EXP})$. Da EXP unter $\leqslant_\mathrm{m}^{\log}$-Reduktion abgeschlossen ist, ist also $\mathrm{R}_\mathrm{m}^{\log}(\mathrm{E}) \subseteq \mathrm{EXP}$.

Um die Inklusion $\mathrm{EXP} \subseteq \mathrm{R}_\mathrm{m}^{\log}(\mathrm{E})$ zu zeigen, betrachte man ein beliebiges $A \in \mathrm{EXP}$. Sei M_A eine DTM mit Zeitschranke $2^{p(n)}$, die A entscheidet; dabei sei zum Beispiel $p(n)$ ein Polynom der Form $n^k + k$. Definiere eine Sprache B als

$$B := \{w01^{p(|w|)} \mid w \in A\}.$$

Die Funktion f mit $f(w) = w01^{p(|w|)}$ ist eine many-one Reduktion von A auf B. Es gilt $f \in \mathrm{FL}$.

Die Sprache B wird von einer Maschine M_B entschieden, die wie folgt arbeitet: Bei Eingabe w' prüft M_B zunächst, ob w' die Form $w' = w01^{p(|w|)}$ für ein w hat. Wenn nein, verwirft M_B. Sonst führt sie die Berechnung durch, die M_A bei Eingabe w durchführen würde. Falls M_A w akzeptiert, so akzeptiert M_B w'; sonst verwirft M_B. Man sieht, dass $L(M_B) = B$, die Laufzeit von M_B bei Eingabe $w' = w01^{|w|}$ ist $q(n) + 2^{p(|w|)}$ für ein Polynom q und damit in $O(2^{|w'|})$. Also ist $B \in \mathrm{E}$. $\square$

5 Schaltkreise

5.1 Einführung

Einen Schaltkreis kann man einerseits als Beschreibung einer booleschen Funktion $f\colon \{0,1\}^n \to \{0,1\}^r$ auffassen, ganz ähnlich wie eine aussagenlogische Formel. Viele solche Funktionen kann man mit einem Schaltkreis knapper darstellen als mit einer Formel.

Andererseits kann man Schaltkreise als Modell für parallele Berechnungen auffassen. Jedes Gatter im Schaltkreis ist gewissermaßen ein kleiner Prozessor, dessen Aufgabe darin besteht, einmal im Laufe der Berechnung eine feste, ihm zugewiesene boolesche Operation auszuführen. Das Modell ist auch deshalb sehr restriktiv, weil die Kommunikationsstruktur „fest verdrahtet" ist und der Nachrichtentransport nur in eine Richtung verläuft.

Die Berechnungskraft von Schaltkreisen hängt davon ab, für welche elementaren Operationen Gatter zur Verfügung stehen. Wir beschränken uns im Folgenden auf elementare Funktionen der Stelligkeit höchstens 2, also auf Gatter mit Eingangsgrad (Fan-in) höchstens 2. Die zweistelligen Funktionen $\wedge$ („und") und $\vee$ („oder") und die einstellige Funktion $\neg$ („nicht") reichen aus, um alle booleschen Funktionen darzustellen. Der Bequemlichkeit halber erlauben wir auch konstante Gatter (mit Fan-in 0). Um leichter über Eingabe und Ausgabe des Schaltkreises sprechen zu können, fügen wir noch spezielle Eingabe- und Ausgabegatter hinzu.

Definition 5.1 (Schaltkreis)**.** Ein *Schaltkreis* ist ein gerichteter azyklischer Graph $G = (V, E)$. Die Knotenmenge V zerfällt in drei disjunkte Teilmengen. Die einzelnen Knotenmengen sind durchnummeriert.

$$V = \{e_1, \ldots, e_n, g_1, \ldots, g_s, a_1, \ldots, a_r\}\,.$$

Dabei heißen	$e_1, \ldots, e_n$	*Eingabeknoten*,
	$g_1, \ldots, g_s$	*Gatterknoten* oder einfach *Gatter* und
	$a_1, \ldots, a_r$	*Ausgabeknoten*.

Die Gatter g_i sind beschriftet: $\mathrm{label}(g_i) \in \{\neg, \vee, \wedge, 0, 1\}$.

Für die Ein- und Ausgrade der verschiedenen Knoten gilt folgendes:

e_i	Eingrad 0,	
a_i	Eingrad 1, Ausgrad 0,	
g_i	$\mathrm{label}(g_i) \in \{\vee, \wedge\}$	Eingrad 2,
	$\mathrm{label}(g_i) = \neg$	Eingrad 1,
	$\mathrm{label}(g_i) \in \{0, 1\}$	Eingrad 0.

Paaren von Schaltkreisen und Belegungen der Eingabegatter wird in kanonischer Weise durch die Schaltkreisauswertung eine Ausgabe aus $\{0,1\}^r$ zugeordnet.

Definition 5.2 (Schaltkreisgröße und Tiefe)**.**
Für einen Schaltkreis C mit $V = \{e_1, \ldots, e_n, g_1, \ldots, g_s, a_1, \ldots, a_r\}$ sei

- die *Größe* des Schaltkreises $\mathrm{Size}(C) := s$ die Anzahl der Gatter und
- die *Tiefe* des Schaltkreises $\mathrm{Depth}(C)$ die maximale Anzahl von Gattern auf einem Pfad von irgendeinem Eingabeknoten e_i zu irgendeinem Ausgabeknoten a_j.

Definition 5.3 (Schaltkreisfamilie, berechnete Funktion, akzeptierte Sprache)**.**

1. Ein Schaltkreis C mit n Eingabe- und r Ausgabeknoten *berechnet eine n-stellige Funktion* $\Phi_C\colon \{0,1\}^n \to \{0,1\}^r$. Sie ist definiert durch die rekursive Zuordnung von booleschen Werten zu Gattern bei gegebener Eingabe $x = x_1 \ldots x_n$. Für $\Phi_C(x)$ schreibt man oft einfach $C(x)$.

2. Eine *Schaltkreisfamilie* ist eine Folge $(C_n)_{n\in\mathbb{N}}$ von Schaltkreisen, so dass C_n eine n-stellige Funktion berechnet.

3. Eine Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ *berechnet eine Funktion* $f\colon \{0,1\}^* \to \{0,1\}^*$, falls für alle n und für alle $x \in \{0,1\}^n$ gilt: $f(x) = C_n(x)$.

4. Eine Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ *entscheidet eine Sprache* $A \subseteq \{0,1\}^*$, falls für alle n und für alle $x \in \{0,1\}^n$ gilt: $C_n(x) = \chi_A(x)$.

Definition 5.4 (Schranken für Größe und Tiefe)**.**

1. Eine Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ hat *Größenschranke* $S(n)$, falls $\mathrm{Size}(C_n) \leqslant S(n)$ für alle n.
2. Eine Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ hat *Tiefenschranke* $D(n)$, falls $\mathrm{Depth}(C_n) \leqslant D(n)$ für alle n.

Satz 5.5. *Für jede Funktion* $f\colon \{0,1\}^* \to \{0,1\}$ *existiert eine Schaltkreisfamilie* $(C_n)_{n\in\mathbb{N}}$ *exponentieller Größe und polynomieller Tiefe, die* f *berechnet.*

Beweisidee. Man benutze die Darstellung der einzelnen n-stelligen Funktionen durch Formeln in disjunktiver Normalform. □

Auch für nichtberechenbare Funktionen existieren also Schaltkreisfamilien. Zum Beispiel existiert für das unär kodierte diagonalisierte Halteproblem

$$\{\, 1^n \mid \text{die } n\text{-te Turingmaschine stoppt bei Eingabe } n \,\}$$

sogar eine Schaltkreisfamilie mit logarithmischer Tiefe. Um Schaltkreise als Modell für effiziente Berechnungen zu verwenden, müssen die Schaltkreise deshalb selber effizient konstruierbar sein. Es wird meistens verlangt, dass die Schaltkreise von einer logarithmisch platzbeschränkten Turingmaschine berechnet werden können. Damit Schaltkreise als Ausgabe einer Turingmaschine auftreten können, müssen wir eine Darstellung von Schaltkreisen als Zeichenketten vereinbaren. Dafür zählen wir die Knoten auf, geben zu jedem Knoten an, ob er Eingabe- oder Ausgabeknoten ist; Gatterknoten markieren wir dadurch, dass wir ihr Label und die Nummern ihrer Vorgängerknoten angeben. Ein Beispiel für eine Zeichenkette, die einen Schaltkreis kodiert, ist:

$$e\#e\#e\#\vee,1,2\#\neg,2\#\wedge,5,3\#\wedge,4,6\#a,7$$

In diesem Beispiel sind die Schaltkreisknoten sogar topologisch geordnet: Vorgängerknoten eines Knotens haben immer eine niedrigere Nummer als der Knoten selber.

Definition 5.6. Eine Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ heißt logspace-uniform, falls es eine logarithmisch platzbeschränkte DTM M gibt, die bei Eingabe 1^n den Schaltkreis C_n (als Zeichenkette kodiert) ausgibt.

Lemma 5.7. *Eine logspace-uniforme Schaltkreisfamilie ist polynomiell größenbeschränkt.*

5.2 Turingmaschinen werten Schaltkreise aus

Wie aufwändig ist die Auswertung eines Schaltkreises mittels einer Turingmaschine? Der folgende Satz besagt, dass sich Schaltkreise relativ schnell auswerten lassen; dabei wird *linear* viel Platz benötigt.

Satz 5.8. *Es gibt eine polynomiell zeitbeschränkte DTM* M*, die bei Eingabe eines Schaltkreises* C *mit* n *Eingabe- und* r *Ausgabeknoten und eines Bitstrings* $x \in \{0,1\}^n$ *den Wert* $C(x) \in \{0,1\}^r$ *ausgibt. Der Platzbedarf der Maschine ist linear.*

Beweis. Sei $x \in \{0,1\}^n$ der Eingabestring und C der Schaltkreis. Sei k die Anzahl aller Knoten in C. Die Maschine M benutzt ein Arbeitsband, auf dem ein Arbeitsvektor der Länge k bestehend aus Nullen, Einsen und Sternchen steht. Jeder Eintrag des Arbeitsvektors entspricht einem Knoten und dem derzeitigen Wissen der Maschine darüber, was an den Ausgängen des Knotens herauskommt. Ein Sternchen bedeutet dabei, dass die Maschine noch nicht weiß, was aus dem zugehörigen Knoten herauskommt.

Am Anfang schreibt die Maschinen an alle Positionen des Arbeitsvektors ein Sternchen, außer an die Positionen, die zu Eingabeknoten gehören. Dort trägt sie die zugehörigen Werte aus x ein.

Die Maschine geht k mal alle Knoten des Schaltkreises durch. Jedesmal, wenn sie einen Knoten findet, für den der Wert am Ausgang noch nicht bekannt ist, für den aber die Werte an den Eingängen bereits bekannt sind, wertet sie das Gatter aus und schreibt den resultierenden Wert in ihren Arbeitsvektor.

Nach k Schritten muss an jedem Knoten ein Wert vorliegen, da in jedem Schritt ja mindestens ein neuer Knoten ausgewertet wird. Die Werte in ihrem Arbeitsvektor an den Stellen, die zu den Ausgangsknoten gehören, kann die Maschine nun ausgeben.

Der Platzbedarf der Maschine ist linear, da der Arbeitsvektor höchstens so lang wie der Eingabeschaltkreis ist. $\square$

Es ist nicht bekannt, ob sich der Platzaufwand beim Auswerten von Schaltkreisen beispielsweise auf nur logarithmischen Platz reduzieren lässt.

5.3 Schaltkreise simulieren Turingmaschinen

Wir wenden uns nun der Frage zu, wie man Berechnungen von polynomiell zeitbeschränkten Turingmaschinen durch Berechnungen von Schaltkreisen ersetzen kann. Wir betrachten dazu eine Einband-DTM M mit Eingabealphabet $\Sigma = \{0, 1\}$. Sei $Q = \{q_1, \ldots, q_l\}$ die Zustandsmenge von M. M sei polynomiell zeitbeschränkt, und zwar sei $p(n)$ ein Polynom, so dass für alle Worte w die Rechenzeit von M bei Eingabe w höchstens $p(|w|)$ beträgt.

Die Berechnung von M bei Eingabe w, $|w| = n$, können wir als Folge von $p(|w|)+1$ vielen Konfigurationen $c_0, \ldots, c_{p(n)}$ darstellen. Stoppt die Berechnung nach weniger als $p(n)$ Schritten, so verlängern wir die Konfigurationenfolge durch wiederholtes Anhängen der letzten Konfiguration auf die passende Länge. Wir nummerieren die Bandfelder, indem wir dem Feld, auf dem in der Startkonfiguration der Schreib-/Lesekopf steht, die Nummer 0 geben und nach rechts aufsteigend, nach links absteigend durchzählen. Eine Konfiguration c_i geben wir an, indem wir für jedes $j = -p(n), \ldots, p(n)$ den Inhalt des j-ten Feldes binär kodiert angeben. Steht nach Schritt i der Kopf auf Feld j und ist die Maschine im Zustand q_k, so schreiben wir an die Stelle (i, j) zusätzlich den Binärcode von k; ist der Kopf nicht auf Feld j, so wird an der entsprechenden Stelle 0 eingetragen. Berechnungen von M bei Eingabe w lassen sich also als $\big(2p(n) + 1\big) \times \big(p(n) + 1\big)$-Matrix $C = (c_{i,j})$ darstellen. Dabei ist jedes $c_{i,j}$ ein Bitstring der Länge $s := \lceil \log(l + 1) \rceil + \lceil \log |\Gamma| \rceil$. Die ersten Bits in $c_{i,j}$ geben den aktuellen Zustand an und die Information, dass der Kopf sich an dieser Stelle befindet; oder die Information, dass der Kopf nicht auf dem Feld j steht. Die $\lceil \log |\Gamma| \rceil$ vielen weiteren Bits in $c_{i,j}$ kodieren das Zeichen aus Γ, das nach i Schritten im Feld j steht.

Der Wert von $c_{i,j}$ für $i \geqslant 1$ hängt nur von den drei Werten $c_{i-1,j-1}$, $c_{i-1,j}$ und $c_{i-1,j+1}$ ab; für die Randfelder mit $j = -p(n)$ oder $j = p(n)$ sogar nur von zwei Werten. Es gibt einen Schaltkreis B_M mit $3s$ Eingängen und s Ausgängen, der für alle i, j mit $i \geqslant 1$ und $-p(n) < j < p(n)$ den Wert von $c_{i,j}$ aus den drei Vorgängerwerten berechnet. Für den linken und rechten Rand seinen L_M und R_M die entsprechenden Schaltkreise mit $2s$ Eingängen.

Der Schaltkreis, der die Berechnung von M simuliert, lässt sich ganz gleichförmig aus Kopien der Basisbausteine B_M, L_M, R_M zusammensetzen. Nach der Schaltkreisschicht, die den letzten Schritt von M simuliert, wird für jedes j ein Schaltkreis A_M mit einem Ausgang angehängt, der 1 ausgibt genau dann, wenn auf den ersten $\lceil \log l \rceil$ Bits von $c_{p(n),j}$ die Nummer eines akzeptierenden Zustands steht. Die Ausgaben dieser A_M-Schaltkreise werden in einem baumartigen Oder-Schaltkreis zusammengeführt, so dass insgesamt 1 ausgegeben wird, wenn in der letzten Konfiguration $c_p(n)$ ein akzeptierender Zustand vorlag.

Diese Schaltkreiskonstruktion rechtfertigt folgenden Satz:

Satz 5.9. *Für jede DTM M mit polynomieller Laufzeitschranke p und Eingabealphabet $\Sigma = \{0, 1\}$ existiert für jede Eingabelänge n ein Schaltkreis C_n mit folgender Eigenschaft: Für alle Wörter $w \in \Sigma^*$ der Länge n akzeptiert M das Wort w genau dann, wenn $C_n(w) = 1$.*

Die Schaltkreise C_n haben dabei die Größe $O\big(p(n)^2\big)$ und lassen sich bei Eingabe 1^n in Zeit $O\big(p(n)^3\big)$ und mit Platzschranke $O(\log n)$ konstruieren.

5.4 POLYSIZE, P/poly

Definition 5.10 (POLYSIZE)**.** Eine Schaltkreisfamilie $(C_n)_{n \in \mathbb{N}}$ heißt *polynomiell größenbeschränkt*, falls sie ein Polynom als Größenschranke hat.

Die Sprachklasse POLYSIZE besteht aus den Sprachen $A \subseteq \{0,1\}^*$, für die eine polynomiell größenbeschränkte Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ existiert, die A entscheidet.

Die Schaltkreise aus der oben beschriebenen Konstruktion sind durch $O(p^2)$ größenbeschränkt und damit polynomiell größenbeschränkt. Verlangt man zusätzlich Logspace-Uniformität für die Schaltkreise, so gilt in obigem Korollar sogar Gleichheit. Ob auch NP in POLYSIZE enthalten ist, ist unbekannt.

Folgerung 5.11.

1. $\mathrm{P} \subsetneq \mathrm{POLYSIZE}$.
2. $\mathrm{P} =$ *logspace-uniform* POLYSIZE.

Damit eine Turingmaschine entscheiden kann, ob eine Eingabe x in einer Sprache $A \in \mathrm{POLYSIZE}$ ist, braucht sie den Schaltkreis C_n für die Wortlänge $|x| = n$ als *Zusatzinformation.* Liegt C_n vor, kann die Maschine C_n mit Eingabe x in polynomieller Zeit auswerten. Die Idee, Turingmaschinen für jede Wortlänge mit einer Zusatzinformation (engl. *advice*) auszustatten, führt zu folgender Definition:

Definition 5.12 (P/poly)**.**

1. Sei $h\colon \mathbb{N} \to \Sigma^*$ und $l\colon \mathbb{N} \to \mathbb{N}$. Wir sagen, h hat Länge l, wenn $|h(n)| = l(n)$ für alle n.
2. Eine Sprache A ist in der Klasse $\mathrm{P}/l(n)$, falls
   (a) eine Funktion $h\colon \mathbb{N} \to \Sigma^*$ der Länge l und
   (b) eine Sprache $B \in \mathrm{P}$

   existieren, so dass

   $$x \in A \text{ gdw } \langle x, h(|x|)\rangle \in B\,.$$

   Eine solche Funktion h nennt man *Advice-Funktion.*
3. Eine Sprache A ist in der Klasse P/poly, falls ein Polynom l existiert, so dass $A \in \mathrm{P}/l(n)$ ist.

Advice-Klassen im allgemeinen und insbesondere P/poly wurden 1980 von Karp und Lipton [KL80] eingeführt.

Einerseits ist die Kodierung eines Schaltkreises polynomieller Größe ein spezieller Advice polynomieller Länge. Es ist also $\mathrm{POLYSIZE} \subseteq \mathrm{P/poly}$. Aber auch die andere Inklusionsrichtung gilt: Seien A,B, h und p wie in Definition 5.12. Da B in P ist, gibt es eine polynomiell größenbeschränkte Schaltkreisfamilie $(C_n)_{n\in\mathbb{N}}$ für B. Man kann nun in dem Schaltkreis $C_{n+|h(n)|}$ die letzten $|h(n)|$ in konstante Gatter umwandeln, so dass sie den Advice $h(n)$ kodieren. Der so entstandene Schaltkreis C'_n hat n verbleibende Eingabegatter; er entscheidet A auf der Eingabelänge n. Außerdem ist C'_n nur polynomiell groß bezüglich n. Also ist A in POLYSIZE. Zusammengefasst gilt:

Satz 5.13.

$$\mathrm{POLYSIZE} = \mathrm{P/poly}$$

6 Vollständige Probleme

Vollständige Probleme in einer Sprachklasse sind „die schwierigsten Probleme“ in der Klasse. Ein vollständiges Problem hat nämlich die Eigenschaft, dass sich alle anderen Probleme aus der Sprachklasse auf dieses eine Problem reduzieren lassen.

Definition 6.1. Sei B eine Sprache und $\mathcal{C}$ eine Sprachklasse, es sei $\leqslant_{\mathrm{r}}$ eine Reduktion.

1. B heißt *schwierig* für $\mathcal{C}$ bezüglich $\leqslant_{\mathrm{r}}$-Reduktion, falls $A \leqslant_{\mathrm{r}} B$ für alle $A \in \mathcal{C}$.
2. B heißt *vollständig* für $\mathcal{C}$ bezüglich $\leqslant_{\mathrm{r}}$-Reduktion, falls $A \leqslant_{\mathrm{r}} B$ für alle $A \in \mathcal{C}$ *und* $B \in \mathcal{C}$.

Da *schwierig* auf englisch *hard* heißt und dies leicht falsch übersetzt wird, sagt man oft auch *hart* für $\mathcal{C}$ anstelle von *schwierig* für $\mathcal{C}$. Die wichtigste Reduktion für uns ist (insbesondere wenn es um vollständige Probleme geht) die Logspace-Many-One-Reduktion. Statt *vollständig für* $\mathcal{C}$ *bezüglich Logspace-Many-One-Reduktion* sagt man oft einfach *vollständig für* $\mathcal{C}$, oder noch kürzer $\mathcal{C}$*-vollständig.* Ein Problem ist also genau dann $\mathcal{C}$-vollständig, wenn es in $\mathcal{C}$ liegt und gleichzeitig $\mathcal{C}$-schwierig ist. Häufig ist es leicht zu zeigen, dass $B \in \mathcal{C}$ gilt, aber kompliziert zu zeigen, dass das Problem auch $\mathcal{C}$-schwierig ist.

Es gibt zwei Verfahren, wie man zeigen kann, dass ein Problem $B \in \mathcal{C}$ auch $\mathcal{C}$-vollständig ist.

1. *Bootstrapping-Methode.* Man kann direkt versuchen, ein Verfahren anzugeben, wie sich jedes Problem $A \in \mathcal{C}$ auf B reduzieren läßt. Dies ist im Allgemeinen recht schwierig: Man kann nur Eigenschaften von A benutzen, die alle Sprachen in $\mathcal{C}$ gemeinsam haben, B muss also gleichzeitig zu all diesen Sprachen A passen.
2. *Reduktionsmethode.* Man zeigt für ein bereits bekanntes $\mathcal{C}$-vollständiges Problem B', dass sich B' auf B reduzieren lässt.

Da sich bei der zweiten Methode jedes Problem in $\mathcal{C}$ auf B' reduzieren lässt, lässt sich wegen der Transitivität der Reduktion auch jedes Problem in $\mathcal{C}$ auf B reduzieren – und B ist folglich $\mathcal{C}$-schwierig.

6.1 NL-Vollständigkeit

Die Sprachklasse NL besitzt vollständige Probleme bezüglich Logspace-Many-One-Reduktion. Zunächst muss mittels Bootstrapping die Vollständigkeit *eines* Problems gezeigt werden. Dieses Problem ist das Graphproblem PATH (alias s-t-CON alias GAP):

Definition 6.2. PATH ist wie folgt definiert:

Eingabe: Ein gerichteter Graphen $G = (V, E)$ und zwei Knoten $s, t \in V$.
Frage: Existiert ein Weg von s nach t in G?

Satz 6.3. *Das gerichtete Erreichbarkeitsproblem* PATH *ist* NL*-vollständig.*

Beweis. PATH ist in NL, da eine NTM M bei Eingabe (G, s, t) einen Pfad von s aus raten kann. Dabei muss M sich immer nur maximal zwei Knotennummern merken. Dies erfordert nur logarithmischen Platz. M akzeptiert, wenn der Pfad irgendwann t erreicht.

Sei nun $A \in$ NL, sei M_A die NTM, die dies belegt. M_A sei so, dass es nur eine akzeptierende Endkonfiguration gibt. Die Reduktionsfunktion f bildet x auf ein Tripel (G_x, s_x, t_x) ab. Dabei ist G_x der Konfigurationsgraph von M_A bei Eingabe x, s_x die Startkonfiguration von M_A bei Eingabe x und t_x die akzeptierende Endkonfiguration von M_A bei Eingabe x. Die Funktion f ist in FL und M_A hat eine akzeptierende Berechnung bei Eingabe x gdw $(G_x, s_x, t_x) \in$ PATH. $\square$

Wir geben drei weitere NL-vollständige Probleme an; ein graphentheoretisches, ein aussagenlogisches und ein algebraisches Problem.

Definition 6.4. STRONGLYCONNECTED ist wie folgt definiert:

Eingabe: Ein gerichteter Graph $G = (V, E)$
Frage: Gibt es für alle $u, v \in V$ einen gerichteten Pfad von u nach v?

Satz 6.5. STRONGLYCONNECTED *ist* NL-*vollständig.*

Beweis. Eine NTM für STRONGLYCONNECTED rät für jedes Paar u, v einen Pfad und akzeptiert nur, wenn für jedes Paar ein Pfad gefunden wurde. Die Verwaltung konstant vieler Knotennummern braucht nur logarithmisch viel Platz.

Es bleibt zu zeigen, dass PATH sich auf STRONGLYCONNECTED reduzieren lässt. Die Reduktionsfunktion f bildet ein Tripel (G, s, t) mit $G = (V, E)$ auf einen Graphen $G' = (V', E')$ ab. Dabei ist $V' = V$, die neue Kantenmenge E' umfasst E und enthält zusätzliche Kanten: Für jedes $u \in V$, $u \neq s$ ist $(u, s) \in E'$, und für jedes $u \in V$, $u \neq t$ ist $(t, u) \in E'$. Es ist $f \in \mathrm{FL}$, und es ist leicht zu sehen, dass $(G, s, t) \in$ PATH gdw $G' \in$ STRONGLYCONNECTED. □

Definition 6.6. 2-UNSAT ist wie folgt definiert:

Eingabe: Eine aussagenlogische Formel φ in konjunktiver Normalform mit genau zwei verschiedenen Literalen pro Klausel

Frage: Ist φ nicht erfüllbar?

Satz 6.7. 2-UNSAT *ist* NL-*vollständig.*

Der Beweis lässt sich im Papadimitriou [Pap94] nachlesen. Da NL komplementabgeschlossen ist, ist auch das Problem 2-SAT NL-vollständig.

Als nächstes betrachten wir eine Variante des GENERABILITY-Problems, bei dem die zweistellige Verknüpfung *assoziativ* ist. Generability-Probleme werden in [BM91] behandelt.

Definition 6.8. ASSOGEN ist wie folgt definiert:

Eingabe: Eine endliche Menge W, ein zweistellige assoziative Operation $\circ$ auf W (durch eine Verknüpfungstafel), eine Startmenge $S \subseteq W$ und ein Zielelement $z \in W$.

Frage: Lässt sich z durch Verknüpfung aus S erzeugen? D.h., ist z enthalten in der kleinsten Teilmenge von W, die S umfasst und abgeschlossen ist unter der Verknüpfung $\circ$?

Satz 6.9. ASSOGEN *ist* NL-*vollständig.*

Beweis Ob eine durch Verknüpfungstafel gegebene Verknüpfung $\circ$ assoziativ ist, ist sogar deterministisch auf logarithmischen Platz überprüfbar. Da die Klammerung bei der Verknüpfung keine Rolle spielt, kann eine NTM M sukzessive Elemente $s_1, s_2, \ldots$ raten und Zwischenergebnisse $w_1, w_2, \ldots$ berechnen; dabei ist $w_1 = s_1$, $w_{i+1} = w_i \circ s_i$. M akzeptiert, sobald $w_i = z$ für ein i.

Es bleibt zu zeigen, dass PATH sich auf ASSOGEN reduzieren lässt. Die Reduktionsfunktion f bildet Tripel (G, s, t), $G = (V, E)$, wie folgt auf $W, \circ, S, z$ ab: W enthält alle Paare (u, v) mit $u, v \in V$, $u \neq v$. Die Menge S und die Verknüpfung $\circ$ werden so festgelegt, dass das Element (u, v) erzeugbar ist gdw in G ein Pfad von u nach v führt. Deshalb setzt man $S = E$ und

$$(u_1, v_1) \circ (u_2, v_2) = \begin{cases} (u_1, v_2) & : \text{ falls } v_1 = u_2 \\ (u_1, v_1) & : \text{ sonst} \end{cases}$$

Als Zielelement z wählt man (s, t). Dann ist z aus S erzeugbar gdw ein Pfad von s nach t existiert. Die Konstruktion der Verknüpfungstafel ist auf logarithmischem Platz durchführbar. Dass die Verknüpfung $\circ$ assoziativ ist, kann man leicht mit einer kleinen Fallunterscheidung überprüfen. □

6.2 P-Vollständigkeit

Wir wollen zeigen, dass das Schaltkreisauswertungsproblem, oder Circuit Value Problem, CVP vollständig für P ist. Da wir noch keine anderen P-vollständigen Probleme kennen, bleibt uns hier nur die Bootstrapping-Methode.

Definition 6.10.
Das Circuit Value Problem CVP ist wie folgt definiert:

Eingabe: Ein Schaltkreis C mit n Eingabeknoten und einem Ausgabeknoten, ein Bitstring $b_1 \cdots b_n$.

Frage: Gibt C bei Eingabe $b_1 \cdots b_n$ eine 1 aus, gilt also $C(b_1, \ldots, b_n) = 1$?

Satz 6.11. CVP *ist* P*-vollständig.*

Beweis Nach Satz 5.8 gilt CVP $\in$ P.

Sei nun $L \in$ P via M. Wir müssen eine Logspace-Many-One-Reduktion von L auf CVP angeben. Für diese Reduktion sei x eine Eingabe der Länge n. Die Reduktion gibt nun das Paar (C_n, x) aus, wobei C_n der Schaltkreis aus Satz 5.9 ist. Nach Satz 5.9 lässt sich dieser Schaltkreis auf logarithmischem Platz berechnen. Weiter gilt $C_n(x) = 1$ genau dann, wenn M das Wort x akzeptiert, was wiederum genau dann der Fall ist, wenn $x \in L$. Also gilt $x \in L$ genau dann, wenn $(C_n, x) \in$ CVP. Also gilt $L \leqslant_{\mathrm{m}}^{\log}$ CVP. $\square$

Die Vollständigkeit von CVP wurde zuerst von Ladner [Lad75] bewiesen. Es gibt sehr viele weitere Probleme, die P-vollständig sind. Eine gute Zusammenstellung findet sich in dem Buch *Limits to Parallel Computation* [GHR95]. Wir geben exemplarisch ein Formel-Erfüllbarkeitsproblem, ein weiteres Schaltkreis-Auswertungsproblem und drei weitere Probleme an.

Definition 6.12. Eine *Hornformel* ist eine boolesche Formel in konjunktiver Normalform, in der jede Klausel eine Hornklausel ist. Eine Disjunktion von Literalen ist eine *Hornklausel*, wenn höchstens eins der Literale positiv ist. Ein Literal ist *positiv*, wenn es eine Variable ist. Literale der Form $\neg x$ heißen *negativ.*

Hornklauseln können auch als Implikationen der Form $x_1 \wedge \cdots \wedge x_k \to x_{k+1}$ oder $x_1 \wedge \cdots \wedge x_k \to \neg x_{k+1}$ geschrieben werden.

Satz 6.13. *Erfüllbarkeit von Hornformeln* HORNSAT *ist* P*-vollständig. Sogar* HORN-3-SAT *ist* P*-vollständig.*

Beweis Wir reduzieren nun CVP auf HORNSAT. Die Reduktionsfunktion f erhält als Eingabe einen Schaltkreis C mit Eingabegattern $e_1, \ldots, e_n$ und weiteren Gattern $g_1, \ldots, g_{\max}$, dabei sei $g_{\max}$ gleichzeitig das Ausgabegatter, und ein Tupel von Eingabe-Bits $x = (x_1, \ldots, x_n)$. Produziert wird von f eine Formel $\varphi_{C,x}$ mit den Variablen $(e_i, 0)$ und $(e_i, 1)$ für $i \in \{1, \ldots, n\}$ und $(g_j, 0)$ und $(g_j, 1)$ für $j \in \{1, \ldots, \max\}$. Die Formel besteht aus drei Teilformeln:

$$\varphi_{C,x} = \varphi_{\text{Eingabe}} \wedge \varphi_{\text{Gatter}} \wedge \varphi_{\text{Ausgabe}}\,.$$

Dabei ist φ_{Eingabe} eine Konjunktion von $2n$ Literalen:

$$\begin{aligned} (e_i, b) \text{ ist Klausel in } \varphi_{\text{Eingabe}} \quad &\text{gdw} \quad x_i = b\,.\\ \neg(e_i, b) \text{ ist Klausel in } \varphi_{\text{Eingabe}} \quad &\text{gdw} \quad x_i \neq b\,.\end{aligned}$$

Dies bewirkt, dass für erfüllende Belegungen B von $\varphi_{C,x}$ gilt: $B(E_i, b) = \text{true} \Leftrightarrow x_i = b$.

φ_{Gatter} enthält zwei bis vier Klauseln pro Gatter g_j. Es wird nach konstanten, Oder-, Und- und Nicht-Gattern unterschieden. Konstante Gatter werden wie Eingabeknoten behandelt. Ist g_j ein Oder-Gatter mit Eingängen von den (Eingabe- oder weiteren) Gattern h und h', so werden aufgenommen:

$$\begin{array}{lcl} (h,0) \wedge (h',0) \to (g_j,0) & \textit{und} & (h,0) \wedge (h',0) \to \neg(g_j,1)\\ (h,0) \wedge (h',1) \to (g_j,1) & \textit{und} & (h,0) \wedge (h',1) \to \neg(g_j,0)\\ (h,1) \wedge (h',0) \to (g_j,1) & \textit{und} & (h,1) \wedge (h',0) \to \neg(g_j,0)\\ (h,1) \wedge (h',1) \to (g_j,1) & \textit{und} & (h,1) \wedge (h',1) \to \neg(g_j,0) \end{array}$$

Ist g_j ein Und-Gatter mit Eingängen von den Knoten h und h', so werden aufgenommen:

$$\begin{array}{lcl} (h,0) \wedge (h',0) \to (g_j,0) & \textit{und} & (h,0) \wedge (h',0) \to \neg(g_j,1)\\ (h,0) \wedge (h',1) \to (g_j,0) & \textit{und} & (h,0) \wedge (h',1) \to \neg(g_j,1)\\ (h,1) \wedge (h',0) \to (g_j,0) & \textit{und} & (h,1) \wedge (h',0) \to \neg(g_j,1)\\ (h,1) \wedge (h',1) \to (g_j,1) & \textit{und} & (h,1) \wedge (h',1) \to \neg(g_j,0) \end{array}$$

Ist g_j ein Nicht-Gatter mit Eingang vom Knoten h, so werden aufgenommen:

$$\begin{array}{lcl} (h,0) \to (g_j,1) & \textit{und} & (h,0) \to \neg(g_j,1)\\ (h,1) \to (g_j,0) & \textit{und} & (h,0) \to \neg(g_j,1) \end{array}$$

Die einzelnen Teilformeln sind Horn-Klauseln. Eine erfüllende Belegung muss die Variablen genau entsprechend dem jeweiligen Gatterwert belegen.

Die Formel φ_{Ausgabe} besteht nur aus dem Literal $(g_{\max}, 1)$, wobei $g_{\max}$ das Ausgabegatter ist.

Eine erfüllende Belegung muss genau die Variablen mit *true* belegen, die den Wert des Gatters bei der Schaltkreisauswertung wiederspiegeln. Wird also g_j im Schaltkreis zu 1 ausgewertet, so muss eine erfüllende Belegung $(g_j, 1)$ mit *true* und $(g_j, 0)$ mit *false* belegen. Wegen φ_{Ausgabe} ist $\varphi_{C,x}$ also nur erfüllbar, wenn $(g_{\max}, 1)$ zu 1 ausgewertet wird.

Die Formel $\varphi_{C,x}$ kann mit einer logarithmisch platzbeschränkten DTM berechnet werden. □

Ein Schaltkreis ist *monoton*, wenn er keine Nicht-Gatter enthält. Ein monotoner Schaltkreis hat also nur Eingabe-, Ausgabe-, Und-, Oder- und konstante Gatter. Das Auswertungsproblem für monotone Schaltkreise heißt MCVP.

Satz 6.14. MCVP *ist* P*-vollständig.*

Beweisskizze Es muss ein Konstruktionsverfahren angegeben werden, das zu einem gegebenen Schaltkreis C mit Eingabe $b = (b_1, \ldots, b_n)$ einen monotonen Schaltkreis Schaltkreis C' und eine Eingabe b' konstruiert, so dass $C(b) = C'(b')$. Dies geschieht in zwei Phasen. In der ersten Phase wird das Vorkommen von Nicht-Gattern im Schaltkreis eingeschränkt, so dass Nicht-Gatter nur noch Ausgrad 1 haben und Nicht-Gatter nicht mehr direkt aufeinanderfolgen.

In der zweiten Phase wird dem Schaltkreis ein zweiter Schaltkreis H mit der gleichen Graphstruktur und der Eingabe b eine Eingabe b^H zur Seite gestellt. Jedem Gatter g_i in C ist ein Gatter h_i in H zugeordnet.

- Ist g_i ein Oder-Gatter, so ist h_i ein Und-Gatter.
- Ist g_i ein Und-Gatter, so ist h_i ein Oder-Gatter.
- Ist g_i ein Nicht-Gatter, so ist auch h_i ein Nicht-Gatter.
- Ist g_i ein Eingabeknoten für Eingabe b_i, so ist h_i Eingabeknoten für Eingabe b_i^H. Es wird festgelegt: $b_i^H = 1 \Leftrightarrow b_i = 0$.
- Das Ausgabegatter des gesamten Schaltkreises ist gleich dem Ausgabegatter des bisherigen Schaltkreises C.

Die Konstruktion ist so gemacht, dass bei der Auswertung des Schaltkreises für alle i gilt:

$$g_i \text{ hat Wert } 0 \Longleftrightarrow h_i \text{ hat Wert } 1 \ .$$

Man kann also auf den negierten Wert von Gatter g_i zugreifen, indem man direkt auf Gatter h_i zugreift. Für alle Gatterpaare g_i, g_j, für die ein Pfad von g_i über ein Nicht-Gatter zu g_j führt, entfernt man das Nichtgatter und die zwei beteiligten Kanten und fügt eine neue Kante von h_i zu g_j ein. Die analoge Operation führt man auch für Gatterpaare h_i, h_j durch. Der so entstandene Schaltkreis C' enthält keine Nicht-Gatter und liefert bei Eingabe b, b^H den gleichen Wert wie C bei Eingabe b. □

Satz 6.15. *Es sind* P*-vollständig:*

1. Generability*, Erzeugbarkeit mit einer durch Verknüpfungstafel gegebenen Operation. Das Problem bleibt* P*-vollständig, wenn man sich auf kommutative Verknüpfungen einschränkt.*
2. HDS*, High Degree Subgraph. Eingabe ist ein Graph G und eine Zahl k. Existiert ein induzierter Teilgraph von G, in dem jeder Knoten einen Grad $\geqslant k$ hat?*
3. CTQ*, Corporate Takeover Query, Firmenkontrolle.*

   Eingabe: *Eine Liste von Aktiengesellschaften $G_1, \ldots, G_n$; für jedes Paar i, j die Angabe, welchen Bruchteil der Aktien von Gesellschaft G_j die Firma G_i besitzt; zwei Firmen G_k und G_l.*

   Frage: *Kontrolliert G_k die Gesellschaft G_l ?*

   Dabei ist kontrolliert *die kleinste Relation mit folgenden Eigenschaften:*

   (a) *Jede Gesellschaft kontrolliert sich selbst.*

   (b) *Kontrolliert G die Gesellschaften $G_{i_1}, \ldots, G_{i_r}$ und besitzen $G_{i_1}, \ldots, G_{i_r}$ zusammen mehr als die Hälfte der Aktien von G', so kontrolliert G auch G'.*

6.3 NP-Vollständigkeit

Von besonderem Interesse ist die Frage, welche Probleme NP-vollständig sind. Für sehr viele natürliche, in der Praxis auftretende Probleme hat sich ihre NP-Vollständigkeit beweisen lassen. Für solche Probleme sind keine deterministischen Polynomialzeit-Algorithmen bekannt – und kaum jemand erwartet, dass sich jemals solche Algorithmen werden finden lassen.

Im Jahr 1971 zeigte Cook die NP-Vollständigkeit von SAT, dem Erfüllbarkeitsproblem für aussagenlogische Formeln, mittels Bootstrapping. Später konnte dann mittels der Reduktionsmethode die NP-Vollständigkeit von CIRCUITSAT (Schaltkreiserfüllbarkeit) nachgewiesen werden:

Definition 6.16. CIRCUITSAT ist wie folgt definiert:

Eingabe:	Ein Schaltkreis C mit n Eingabeknoten und einem Ausgabeknoten.
Frage:	Existiert ein Bitstring $b_1 \cdots b_n$, so dass $C(b_1, \ldots, b_n) = 1$?

Im Folgenden wollen wir allerdings genau umgekehrt vorgehen. Wir zeigen also zuerst mittels Bootstrapping, dass CIRCUITSAT ein NP-vollständiges Problem ist, und reduzieren es dann auf SAT.

Satz 6.17. *Das Schaltkreis-Erfüllbarkeitsproblem* CIRCUITSAT *ist* NP*-vollständig.*

Beweis. Es gilt CIRCUITSAT $\in$ NP: Bei Eingabe eines Schaltkreises rät man zunächst eine Belegung für die Eingänge des Schaltkreises. Nach Satz 5.8 kann dann in Polynomialzeit entschieden werden, ob der Schaltkreis zu 1 auswertet. In diesem Fall wird der Schaltkreis akzeptiert, andernfalls wird er verworfen.

Sei nun $A \in \mathrm{NP}$ via einer Maschine M mit Zeitschranke p. Wir müssen zeigen: $A \leqslant_{\mathrm{m}}^{\log}$ CIRCUITSAT. Wir dürfen annehmen, dass die Maschine M in jedem Schritt zwischen genau zwei möglichen Nachfolgezuständen nichtdeterministisch einen auswählt.

Wir konstruieren zunächst eine deterministische Maschine M', die als Eingabe ein Wort x der Länge n bekommt, sowie einen Berechnungspfad der Maschine M bei Eingabe x der Länge höchstens $p(n)$. Der Berechnungspfad wird hierbei kodiert als die Folge der nichtdeterministischen Entscheidungen, die die Maschine M bei Eingabe x während der Berechnung fällt. Dieser Code ist also eine Folge von Bits der Länge $p(n)$. Die Maschine M' akzeptiert nun das Wort x zusammen mit dem Code eines Berechnungspfads, falls die Maschine M das Wort x auf diesem Berechnungspfad akzeptiert. Es ist leicht einzusehen, dass M' dies in Polynomialzeit überprüfen kann.

Für die Reduktion von A auf CIRCUITSAT bemühen wir nun wieder Satz 5.9 über die Simulation von Turingmaschinen durch Schaltkreise. Bei Eingabe x konstruieren wir einen Schaltkreis C, so dass M' das Wort x zusammen mit einem Bistring b genau dann akzeptiert, wenn $C(x, b) = 1$. Nun ersetzen wir alle Eingangsknoten in C, die zu den Bits von x gehören, durch Null- oder Eins-Gatter wie folgt: Falls das i-te Bit von x eine 0 ist, so ersetze den i-ten Eingangsknoten durch ein Null-Gatter, sonst durch ein Eins-Gatter. Für den so entstandenen Schaltkreis C' gilt nun, dass $C'(b) = 1$ genau dann gilt, wenn M' das Wort x zusammen mit b akzeptiert. Damit ist die Reduktion vollständig beschrieben. Man überzeugt sich leicht, dass sie in logarithmischem Platz durchführbar ist.

Der Schaltkreis C' ist nun genau dann erfüllbar, wenn es einen Bitstring b gibt, so dass M' das Wort x zusammen mit b akzeptiert. Dies ist aber genau dann der Fall, wenn es einen Berechnungspfad gibt, auf dem M das Wort x akzeptiert. Also gilt $C' \in$ CIRCUITSAT genau dann, wenn $x \in A$. $\square$

Um nun die NP-Vollständigkeit weiterer Probleme zu zeigen, können wir im Folgenden die Reduktionsmethode einsetzen. Beginnen wir mit einer Reduktion von CIRCUITSAT auf die spezielle Variante 3-SAT des Erfüllbarkeitsproblems SAT.

Definition 6.18 (Varianten von SAT)**.**

- CNFSAT

Eingabe:	Eine Formel φ in konjunktiver Normalform (CNF).
Frage:	Ist φ erfüllbar?

- 3-SAT

Eingabe:	Eine Formel φ in CNF mit genau drei unterschiedlichen Literalen pro Klausel.
Frage:	Ist φ erfüllbar?

Satz 6.19. *Es gilt* CIRCUITSAT $\leqslant_m^{\log}$ 3-SAT.

Beweis Sei C ein Schaltkreis mit n Knoten. Wir müssen nun in logarithmischem Platz eine Formel φ in konjunktiver Normalform konstruieren, so dass φ genau dann erfüllbar ist, wenn eine Belegung b für C existiert mit $C(b) = 1$. Wir tun dies genau so wie im Beweis der P-Vollständigkeit von HORNSAT, Satz 6.13. Anders als dort bilden wir für jeden Eingabeknoten e_i, dessen Wert ja nicht festliegt, nur die Formeln

$$\begin{array}{rcl} (e_i, 0) & \vee & (e_i, 1) \\ \neg(e_i, 0) & \vee & \neg(e_i, 1) \end{array}$$

hinzu, die garantieren, dass eine erfüllende Belegung B für jedes i genau eine der zwei Variablen $(e_i, 0)$ und $(e_i, 0)$ mit *true* belegt. Man beachte übrigens, dass die erste der beiden Formeln keine Horn-Klausel ist. □

Satz 6.20 (Cook 1971)**.** *Das Erfüllbarkeitsproblem* SAT *ist* NP*-vollständig.*

Beweis. Es ist klar, dass SAT ∈ NP. Nun gilt nach dem vorherigen Satz CIRCUITSAT $\leqslant_\mathrm{m}^{\log}$ 3-SAT. Da aber trivialerweise 3-SAT $\leqslant_\mathrm{m}^{\log}$ SAT gilt, folgt CIRCUITSAT $\leqslant_\mathrm{m}^{\log}$ SAT. Also ist nach der Reduktionsmethode SAT auch NP-vollständig. □

Ursprünglich hatte Cook die Vollständigkeit von SAT mit der Bootstrapping-Methode gezeigt, dann in zwei Schritten SAT $\leqslant_\mathrm{m}^{\log}$ CNFSAT und CNFSAT $\leqslant_\mathrm{m}^{\log}$ 3-SAT gezeigt, und so die NP-Vollständigkeit von 3-SAT gefolgert. Wir geben im Folgenden den Beweis für CNFSAT $\leqslant_\mathrm{m}^{\log}$ 3-SAT wieder.

Satz 6.21. *Es gilt* CNFSAT $\leqslant_m^{\log}$ 3-SAT.

Beweis Sei ψ eine Formel in konjunktiver Normalform. Wandle jede Klausel in eine erfülllungsgleiche Konjunktion von Dreier-Klauseln um. Sei $\varphi = \varphi_1 \vee \cdots \vee \varphi_m$ eine Klausel. Wir unterscheiden vier Fälle.

1. Falls $m = 3$, so sind wir fertig.
2. Falls $m = 2$, so werwende neue Variable z. Ersetze φ durch die Formel
$$\varphi' := (v_1 \vee v_2 \vee z) \wedge (v_1 \vee v_2 \vee \neg z).$$
3. Falls $m = 1$, so verwende neue Variablen z_1 und z_2. Ersetze φ durch
$$\varphi' := (v_1 \vee z_1 \vee z_2) \wedge (v_1 \vee z_1 \vee \neg z_2) \wedge (v_1 \vee \neg z_1 \vee z_2) \wedge (v_1 \vee \neg z_1 \vee \neg z_2).$$
4. Falls $m \geq 4$, so verwende neue Variable $z_1, \dots, z_{m-3}$. Ersetze φ durch
$$\varphi' := (v_1 \vee v_2 \vee z_1) \wedge (\neg z_1 \vee v_3 \vee z_4) \wedge \cdots \wedge (\neg z_{m-3} \vee v_{m-1} \vee v_m).$$

□

Für die meisten Probleme, die in NP sind und für die kein deterministischer Polynomialzeit-Algorithmus bekannt ist, lässt sich zeigen, dass sie NP-vollständig sind. Wir geben noch drei dieser Probleme an:

Definition 6.22 (Einige Graph-Probleme)**.**

- VERTEXCOVER
  Eingabe: Ein ungerichteter Graphen $G = (V, E)$ und eine binär kodierte Zahl k.
  Frage: Gibt es eine Teilmenge $V' \subseteq V$ mit $|V'| \leq k$, so dass für jede Kante $(u, v) \in E$ mindestens einer der Knoten u und v in V' liegt?
- INDEPENDENTSET
  Eingabe: Ein ungerichteter Graph G und eine binär kodierte Zahl k.
  Frage: Enthält G eine unabhängige Menge V' mit $|V'| \geq k$?

- Clique

  Eingabe: Ein ungerichteter Graph G und eine binär kodierte natürliche Zahl k.

  Frage: Enthält G eine Clique (einen vollständigen Untergraphen) mit k Knoten?

Satz 6.23. *Es gilt*

$$3\text{-Sat} \leqslant_m^{\log} \text{VertexCover} \leqslant_m^{\log} \text{IndependentSet} \leqslant_m^{\log} \text{Clique}.$$

Also sind die Problem VertexCover*,* IndependentSet *und* Clique *alle* NP*-vollständig.*

Ohne Beweis. Die dritte Reduktion bildet $G = (V, E)$ auf den Komplementgraphen $G' = (V, V^2 \setminus E)$ ab.

6.4 PSPACE-Vollständigkeit

Auch die Sprachklasse PSPACE besitzt interessante vollständige Probleme. Viele Graphenprobleme, die NL-vollständig sind, werden PSPACE-vollständig, wenn man zu ihrer *Succinct*-Version übergeht. Das Konzept der *succinct representation* von Graphproblemen stammt aus [GW83]. Ein Graph $G = (V, E)$ als Eingabe wird hier nicht durch eine Adjazenzmatrix oder Adjazenzliste kodiert, sondern durch einen Schaltkreis C_E mit $2n$ Eingängen, der bei Eingabe zweier Knotennummern i und j (als n-stellige Binärzahlen) genau dann eine 1 ausgibt, wenn $(i, j) \in E$. Die Eingabegraphen haben bei dieser Art der Kodierung immer 2^n Knoten für ein gewisses n; dies ist aber keine wesentliche Einschränkung. Die *succinct representation* des Erreichbarkeitsproblems Path lässt sich dann wie folgt definieren:

Definition 6.24. succinctPath ist wie folgt definiert:

Eingabe: Ein $n \in \mathbb{N}$, zwei n-stellige Knotennummern s und t, ein Schaltkreis C_E mit $2n$ Eingabeknoten und einem Ausgabeknoten

Frage: Gibt es in dem durch C_E repräsentierten gerichteten Graphen $G = (\{0,1\}^n, E)$ einen Pfad von s nach t?

Satz 6.25. *Das Problem* succinctPath *ist* PSPACE*-vollständig.*

Beweis Eine NTM, die succinctPath akzeptiert, arbeitet wie folgt: Beginnend mit s rät sie eine Knotenfolge $s = v_0, v_1, v_2, \ldots$. Sie merkt sich immer nur zwei aufeinanderfolgende Knoten (linearer Platzaufwand). Sie prüft jeweils mit C_E, ob die Knoten durch eine gerichtete Kante verbunden sind (wenn nein, wird verwerfend abgebrochen). Wird so der Knoten t erreicht, wird akzeptiert. Also ist succinctPath $\in$ PSPACE.

Sei nun A ein beliebiges Problem in PSPACE. Sei M eine polynomiell platzbeschränkte Ein-Band-DTM mit $A = L(M)$. Sei $p(n)$ die Platzschranke für M. Die Reduktionsfunktion f bildet eine Eingabe x auf ein Tupel (n_x, s_x, t_x, C_x) ab. Dabei ist n_x so gewählt, dass sich Konfigurationen von M bei Eingaben der Länge $|x|$ sich mit n_x Bits darstellen lassen. s_x ist die Beschreibung der Startkonfiguration von M bei Eingabe x, t_x die akzeptierende Endkonfiguration von M (mit n_x Bits dargestellt). Der Schaltkreis C_x berechnet für zwei Knoten i, j im Konfigurationsgraphen zur Eingabelänge $|x|$, ob j Nachfolgekonfiguration von i ist. Die Eingabe x wird von M akzeptiert gdw es im Konfigurationsgraphen einen Pfad von s_x nach t_x gibt gdw $(n_x, s_x, t_x, C_x) \in$ succinctPath. □

Auch ein Formel-Erfüllbarkeitsproblem erweist sich als PSPACE-vollständig. Dafür braucht man eine Möglichkeit, Formeln verkürzt darzustellen. Wir erweitern dafür die Syntax aussagenlogischer Formeln um Quantoren.

Definition 6.26. Die Menge QBF der quantifizierten booleschen Formeln, und der Wert quantifizierter boolescher Formeln unter einer Variablenbelegung B, sind wie folgt definiert:

1. Variable sind in QBF. Die Konstanten 0 und 1 sind in QBF. Sind φ und φ' in QBF, so auch $\varphi \wedge \varphi'$, $\varphi \vee \varphi'$ und $\neg\varphi$. Der Wert unter Belegung B ist wie bei gewöhnlichen booleschen Formeln definiert.

2. Ist φ in QBF und x eine Variable, so ist $\forall x.\varphi$ in QBF. Die Formel $\varphi|_{x=0}$ entstehe, indem man in φ jedes Vorkommen von x durch 0 ersetzt, $\varphi|_{x=1}$ entstehe durch Ersetzen von x durch 1. Es wird festgelegt, dass

$$B(\forall x.\varphi) = 1 \quad \text{gdw} \quad B(\varphi|_{x=0}) = 1 \text{ und } B(\varphi|_{x=1}) = 1 \ .$$

3. Ebenso ist $\exists x.\varphi$ in QBF. Es wird festgelegt, dass

$$B(\exists x.\varphi) = 1 \quad \text{gdw} \quad B(\varphi|_{x=0}) = 1 \text{ oder } B(\varphi|_{x=1}) = 1 \ .$$

Definition 6.27. Das Problem QBF-SAT ist wie folgt definiert:

Eingabe: Eine quantifizierte boolesche Formel φ

Frage: Gibt es eine erfüllende Belegung B für φ?

Satz 6.28. QBF-SAT *ist* PSPACE-*vollständig.*

Dieser Satz wurde in [MS73] bewiesen. Ein Beweis lässt sich nachlesen in [BDG93].

7 Approximationsalgorithmen

Wir wollen in der Regel nicht nur die Existenz von (guten) Problemlösungen entscheiden, sondern solche Lösungen tatsächlich finden. Wir untersuchen deshalb jetzt *Optimierungsprobleme*. Ideal wäre es, deterministische polynomiell zeitbeschränkte Algorithmen finden, die optimale Lösungen liefern. Könnte man aber Optimierungsprobleme, die mit NP-vollständigen Entscheidungsproblemen verbunden sind, in polynomieller Zeit lösen, so wäre P = NP.

Deshalb sucht man nach *Approximationsverfahren*, die nur angenäherte Lösungen berechnen. Mit einer geeigneten *Fehlerdefinition* lässt sich die *Güte* solcher Verfahren bewerten. Optimierungsprobleme mit Approximationsverfahren bestimmter Güte lassen sich zu *Approximationsklassen* zusammenfassen. Für solche Klassen wird gezeigt, dass sie bezüglich Inklusion eine *echte Hierarchie* bilden (sofern P $\neq$ NP). In die Klassen lassen sich *konkrete Optimierungsprobleme einordnen*.

7.1 Definition von Optimierungsproblemen

Ein typisches Beispiel für ein Optimierungsproblem ist das Traveling Salesperson Problem MinTSP. Gegeben ist ein vollständiger Graph, in dem die Kanten mit Entfernungen gelabelt sind. Gesucht ist eine kürzeste Rundreise im Graphen. Vier Angaben spezifizieren das Problem: Erstens die Menge der zulässige Eingaben, hier: Graphen mit Kantengewichten. Zweitens die Menge von zulässigen Lösungen, hier: mögliche Rundreisen. Drittens sind zulässige Lösungen bewertet, hier: mit der Weglänge der Rundreise. Viertens muss gesagt werden, ob eine Lösung mit der *niedrigsten* oder *höchsten* Bewertung gesucht ist, hier: mit der niedrigsten. Dies motiviert die folgende Definition:

Definition 7.1 (Optimierungsproblem)**.**
Ein Optimierungsproblem F ist ein 4-Tupel $F = (I, S, m, \mathrm{typ})$ mit

1. $I \subseteq \Sigma^*$, die Menge der zulässigen Eingaben
2. $S \subseteq I \times \Sigma^*$, die Lösungsrelation. $S(x) = \{y \mid (x, y) \in S\}$ heiß t Menge der zulässigen Lösungen zu x. Es soll gelten: $\forall x \in I.\ S(x) \neq \emptyset$.
3. $m : S \to \mathbb{N}^+$, die *Kostenfunktion (oder Zielfunktion)*. Der Wert $m(x, y)$ muss nur definiert sein, wenn $y \in S(x)$ ist. Dass Kosten positive natürliche Zahlen sind, ist technisch hilfreich, aber keine entscheidende Einschränkung.
4. $\mathrm{typ} \in \{\min, \max\}$, je nachdem, ob ein Minimierungs- oder ein Maximierungsproblem vorliegt.

Definition 7.2 (optimaler Wert, lösender Algorithmus)**.**
Wir bezeichnen den optimalen Wert für eine Eingabe $x \in I$ mit $\mathrm{opt}(x)$. Also:

$$\mathrm{opt}(x) := \begin{cases} \max_{y \in S(x)}(m(x, y)) & , \text{ falls typ} = \max \\ \min_{y \in S(x)}(m(x, y)) & , \text{ falls typ} = \min \end{cases}$$

Ein Algorithmus A *löst* F gdw für alle $x \in I$: $A(x) \in S(x)$ und $m(x, A(x)) = opt(x)$.

Uns interessieren besonders *nichtdeterministisch* und *deterministisch polynomielle* Optimierungsprobleme (wegen ihrer Beziehung zu NP und P).

Definition 7.3 (**OptNP**, **OptP**)**.**
OptNP ist die Klasse aller Optimierungsprobleme F, für die gilt:

1. $I \in P$.
2. Es existiert ein Polynom q, so dass für alle $x \in I$ und alle $y \in S(x)$ gilt: $|y| \leqslant q(|x|)$.
3. $S \in P$.
4. $m \in \mathrm{FP}$.

OptP ist die Klasse aller Optimierungsprobleme F, für die gilt:

1. $F \in$ OptNP.
2. Es existiert ein Polynomialzeit-Algorithmus A, der F löst.

7.2 Entscheidungs- vs. Optimierungsprobleme

Meistens ist es leicht, zu einem NP-Entscheidungsproblem ein OptNP-Optimierungsproblem anzugeben und umgekehrt. Viele bekannte Probleme lassen sich sowohl als Entscheidungsproblem formulieren als auch als Optimierungsproblem (z.B. TSP, KNAPSACK, CLIQUE, ...).
Für diese Probleme gilt dann in der Regel: Gäbe es einen Polynomialzeitalgoritmus, der das Entscheidungsproblem löst, so ließe sich auch das Optimierungsproblem effizient lösen (und umgekehrt). Allgemein gilt folgender Satz:

Satz 7.4. P = NP *gdw* OptP = OptNP.

7.3 Fehler und Approximationsklassen

Ein Approximationsalgorithmus für ein Optimierungsproblem F berechnet für eine Eingabe $x \in I$ eine Lösung $y \in S(x)$. Wie gut die Approximation ist, hängt von dem Verhältnis von $m(x, y)$ und $opt(x)$ ab. Es gibt zahlreiche Möglichkeiten, die Approximierbarkeit zu messen (absoluter Fehler, relativer Fehler, Güte etc.). Ein oft verwendetes Maß ist die *Güte* (engl. *approximation ratio*), auch *performance ratio* oder *Approximationsgüte* genannt:

Definition 7.5 (Approximationsgüte)**.**
Ein Approximationsalgorithmus A für ein F hat *Approximationsgüte* r, falls für alle $x \in I$

$$\frac{\mathrm{opt}(x)}{m(x, A(x))} \leqslant r \text{ (bei Maximierungsproblemen)} \quad \text{bzw.} \quad \frac{m(x, A(x))}{\mathrm{opt}(x)} \leqslant r \text{ (bei Minimierungsproblemen).}$$

Approximationsgüten sind also Werte $\geqslant 1$. Je näher an 1, desto besser. Für manche Probleme gibt es Algorithmen, denen man neben der Probleminstanz die gewünschte Güte als Parameter übergeben kann. Ein Algorithmus A für ein Optimierungsproblem F, der bei Eingabe x und einer Zahl k eine Lösung für x mit einer Güte $\leqslant 1 + 1/k$ ausgibt, heißt *Approximationsschema* für F. Approximationsprobleme lassen sich nun in folgende Klassen einteilen:

Definition 7.6 (APX, PTAS, FPTAS)**.**
Ein Optimierungsproblem $F \in$ OptNP ist

1. in APX, falls ein $\varepsilon \geqslant 0$ und ein polynomiell zeitbeschränkter Approximationsalgorithmus A für F existieren, derart dass A Approximationsgüte $1 + \varepsilon$ hat.
2. in PTAS (*polynomial time approximation scheme*), falls für F ein Approximationsschema A existiert sowie für jedes k ein Polynom p_k, so dass die Laufzeit von A bei Eingabe (x, k) durch $p_k(|x|)$ beschränkt ist.
3. in FPTAS (*fully polynomial time approximation scheme*), falls für F ein Approximationsschema A existiert, dessen Laufzeit bei Eingabe (x, k) polynomiell in $|x|$ und k ist.

7.4 Beispiele für approximierbare Probleme

Wir geben für drei Optimierungsprobleme Approximationsalgorithmen an, und zwar für Max 3-SAT, für Min△TSP und für MaxKNAPSACK. Alle drei Algorithmen werden in dem Buch von Garey und Johnson [GJ79] besprochen, dass trotz (oder wegen?) seines Alters eine sehr gute Einführung in das Thema *Approximation* NP-*vollständiger Probleme* bietet.

7.4.1 Möglichst viele Klauseln erfüllen

Definition 7.7 (Max 3-Sat)**.**
Zu einer gegebenen aussagenlogischen Formel φ in 3-CNF soll eine Variablenbelegung gefunden werden, die so viele Klauseln wie möglich erfüllt (unabhängig davon, ob φ selber erfüllbar ist oder nicht). Der Wert einer Lösung ist die Anzahl der erfüllten Klauseln.

Im folgenden werden wir einen Algorithmus angeben, der Max 3-Sat mit Güte 2 approximiert. Sei φ eine Formel in 3-CNF mit n Klauseln, $x_1, \cdots, x_k$ die Variablen von φ. Sei B_T die Variablenbelegung, die alle Variablen mit True belegt; sei B_F die Variablenbelegung, die alle Variablen mit False belegt.

Approximationsalgorithmus für Max**3**-Sat

Eingabe: φ

Werte alle Klauseln von φ mit Belegung B_T aus.
Bestimme die Anzahl $\#(B_T)$ der Klauseln, die unter B_T zu True ausgewertet werden.
Falls $\#(B_T) \geqslant n/2$
dann gib B_T aus
sonst gib B_F aus.

Satz 7.8 (Max 3-Sat)**.** *Der obige Approximationsalgorithmus für* Max 3-Sat *hält Güte* 2 *ein. Also ist* Max 3-Sat $\in$ APX*.*

Beweis
Der Algorithmus ist polynomiell zeitbeschränkt. Mit diesem Algorithmus wird eine Belegung ermittelt, die mindestens die Hälfte aller Klauseln, also $\frac{n}{2}$ wahr macht, denn wird eine Klausel unter B_T False, so wird sie unter B_F zu True ausgewertet. Die ratio ist also $\leqslant \frac{\mathrm{opt}(\varphi)}{n/2} \leqslant \frac{n}{n/2} = 2$. Also liegt Max 3-Sat in APX, und zwar mit Güte 2. □

Diese Güte ist nicht die best-erreichbare. Ein einfacher Greedy-Algorithmus liefert für Max 3-Sat eine Güte von 4/3.

7.4.2 Möglichst kurze Rundreisen finden

Ein bekanntes ist das *Problem des Handlungsreisenden* TSP (engl. *T*raveling *S*ales *P*erson), bei dem es darum geht, möglichst kurze Rundreisen zu finden.

Definition 7.9. MinTSP ist wie folgt definiert:

Eingabe:	Ein vollständiger Graph $G = (V, E)$, $\lvert V \rvert = n$, bei dem jede Kante $(u, v) \in E$ mit einer Länge $c(u, v)$ gelabelt ist. .
Lösungen:	Eine Rundreise, d.h. eine Folge von Knoten $(u_1, \ldots, u_n, u_1)$ mit $\{u_1, \ldots, u_n\} = V$.
Ziel:	Minimiere $\sum_{i=1}^{n} c(u_i, u_{i+1})$; dabei sei $u_{n+1} := u_1$.

In vielen Fällen erfüllen die Instanzen die zusätzliche Bedingung, dass Wege mit Zwischenstationen nicht kürzer sind als direkte Verbindungen. Die Problemvariante Min$\triangle$TSP ist wie MinTSP definiert, nur dass für die Eingabe zusätzlich verlangt wird, dass die Dreiecksungleichung gilt. Das heißt, dass $c(u, v) + c(v, w) \geqslant c(u, w)$ für alle Knotentripel u, v, w gilt.

TSP mit Dreiecksungleichung wird auch als *metrisches* TSP bezeichnet. Für Min$\triangle$TSP gibt es bessere Approximationsalgorithmen als für das allgemeine MinTSP. Wenn nicht nur die Dreiecksungleichung gilt, sondern die Knoten sich in die euklidische Ebene (oder auch den k-dimensionalen euklidischen Raum) einbetten lassen, so dass die Kantenlängen den euklidischen Abständen der Knoten entsprechen, so spricht man vom *euklidischen* TSP. Dafür gibt es noch bessere Approximationsverfahren.

Wir geben den Christofides-Algorithmus aus [Chr76] für Min$\triangle$TSP an (benannt nach seinem Erfinder Nicos Christofides).

Christofides-Algorithmus für Min$\triangle$TSP

Eingabe: $G = (V, E)$, $\lvert V \rvert = n$, $c(u, v)$ für $u, v \in V$.

Konstruiere minimalen Spannbaum T für G.
Sei Odd die Menge der Knoten mit ungeradem Grad in T (|Odd| ist gerade).
Berechne minimales Matching M für durch Odd induzierten Teilgraphen von G.
Bilde Graphen $T + M$ mit Kanten aus T und M.
Berechne eine Eulertour tour_1 in $T + M$.
Verkürze tour_1 zu tour_2 durch Überspringen doppelt besuchter Knoten.

Ausgabe: tour_2

Satz 7.10. *Der Christofides-Algorithmus für* Min△TSP *ist polynomiell zeitbeschränkt und hält Güte* 3/2 *ein. Also ist* Min△TSP ∈ APX.

Beweis Ein minimaler Spannbaum lässt sich mit dem Algorithmus von Kruskal oder dem Algorithmus von Prim berechnen. Beide benötigen nur polynomielle Laufzeit. Ein minimales Matching lässt sich ebenfalls in polynomieller Zeit berechnen; dies wurde zuerst von Edmonds [Edm65] bewiesen, siehe zum Beispiel auch [Jun94]. Die Berechnung der Eulertour und die Verkürzung der Tour brauchen ebenfalls nur polynomielle Zeit.

Die Tour tour_2 ist kürzer als tour_1 (wegen der Dreiecksungleichung). Die Tour tour_1 ist so lang wie die Summe aus Kantensumme von T und Kantensumme von M. Nimmt man aus einer Rundreise eine Kante heraus, so erhält man einen Spannbaum. Die Kantensumme des minimalen Spannbaums T ist also kleiner als die Länge einer optimalen Rundreise. Betrachtet man in einer optimalen Rundreise nur die Knoten aus Odd und kürzt die verbindenden Pfade durch Kanten ab, so erhält man eine Rundreise durch Odd. Betrachtet man nur jede zweite Kante dieser verkürzten Rundreise, so erhält man zwei Matchings für Odd. Für mindestens eines davon ist die Kantensumme höchstens die Hälfte der optimalen Rundreise. Für das optimale Matching M kann die Kantensumme nur noch kleiner werden. □

7.4.3 Rucksäcke möglichst wertvoll füllen

Definition 7.11. Das Rucksack-Problem MaxKnapsack ist wie folgt definiert:

Eingabe:	Eine Menge von Objekten $\{1, \ldots, n\}$, eine Liste $\text{weight}(1), \ldots, \text{weight}(n)$ von Gewichten, eine Liste $\text{value}(1), \ldots, n)$ von Werten sowie einem Maximalgewicht $W_{\max}$.
Lösungen:	Eine Teilmenge $S \subseteq \{1, \ldots, n\}$, so dass $\sum_{i \in S} w_i \leqslant W_{\max}$.
Ziel:	Zielfunktion m sei definiert durch $m(x, S) = \sum_{i \in S} v_i$.

Eine Teilmenge S, die eine Lösung eines Knapsack-Problems ist, nennen wir auch eine *Packung*. Zunächst geben wir einen Algorithmus an, der eine *optimale* Lösung für MaxKnapsack liefert.

Im Algorithmus bezeichnet $\text{bestpacking}(i, v)$ eine Teilmenge der ersten i Objekte $\{0, \ldots, i\}$ mit *minimalem* Gewicht, die den Wert v hat, $\text{bestweight}(i, v)$ bezeichne das Gewicht einer solchen optimalen Packung mit Wert v. Sind für gegebenes i die Werte $\text{bestpacking}(i, v)$ und $\text{bestweight}(i, v)$ bekannt, so kann man für $i+1$ die Werte leicht bestimmen, denn $\text{bestweight}(i+1, v)$ ist gleich dem Minimum von $\text{bestweight}(i, v)$ und $\text{bestweight}(i, v - v_{i+1}) + w_{i+1}$. Ist $\text{bestweight}(i, v) \leqslant \text{bestweight}(i, v - v_{i+1}) + w_{i+1})$, wird das $(i+1)$-te Objekt nicht in der Packung aufgenommen, weil wir eine Packung mit dem gleichen Wert ohne das $(i+1)$-te Objekt aufbauen können, deren Gewicht kleiner als das Gewicht der Packung mit dem $(i+1)$-ten Objekt. Ist $\text{bestweight}(i, v - v_{i+1}) + w_{i+1}) < \text{bestweight}(i, v)$, wird das $(i + 1)$-te Objekt in der Packung aufgenommen, weil unter den ersten i Objekten eine Packung mit der Wert $v - v_{i+1}$ gefunden wurde, so dass diese Packung zusammen mit dem $i + 1$-ten Objekt kleineres Gewicht, aber gleichen Wert hat. Der Algorithmus berechnet also $\text{bestweight}(i + 1, v) = min\{\text{bestweight}(i, v), \text{bestweight}(i, v - v_{i+1}) + w_{i+1})\}$ für alle $0 \leqslant i \leqslant n - 1$ für alle $0 \leqslant v \leqslant n \cdot V_{\max}$.

Exakter Algorithmus für MaxKnapsack

Eingabe: $n, \text{weight}(1), \ldots, \text{weight}(n), \text{value}(1), \ldots, \text{value}(n), W_{\max}$.

$V_{\max} := max\{\text{value}(i) \mid 1 \leqslant i \leqslant n\}$. (Maximaler Einzelwert)
Für $i = 1$ bis $i = n$ tue: (Initialisierung für Wert 0)
    $\text{bestpacking}(i, 0) := \emptyset$ und $\text{bestweight}(i, 0) := 0$
Für $v = 1$ bis $v = n \cdot V_{\max}$ tue (Initialisierung für $i = 1$)

Falls $v = \text{value}(1)$
dann setze $\text{bestpacking}(1, v) := \{1\}$ und $\text{bestweight}(1, v) := \text{weight}(1)$
sonst $\text{bestpacking}(1, v)$ undefiniert und $\text{bestweight}(1, v) := \infty$
Für $i = 1$ bis $i = n - 1$ tue: (Sukzessive Konstruktion von Packungen für jeden Wert)
Für $v = 1$ bis $v = nV_{\max}$ tue
falls $\text{bestweight}(i, v) \leqslant \text{bestweight}(i, v - v_{i+1}) + w_{i+1}$).
dann $\text{bestweight}(i + 1, v) := \text{bestweight}(i, v)$
$\text{bestpacking}(i + 1, v) := \text{bestpacking}(i, v)$
sonst $\text{bestweight}(i + 1, v) := \text{bestweight}(i, v - v_{i+1}) + w_{i+1}$
$\text{bestpacking}(i + 1, v) := \text{bestpacking}(i, v - v_{i+1}) \cup \{i + 1\}$.
$v_{\text{best}} :=$ maximales v mit $\text{bestweight}(n, v) \leqslant W_{\max}$.
$\text{packing} := \text{bestpacking}(n, v_{\text{best}})$

Ausgabe: packing

Der obige Algorithmus liefert die optimale Lösung mit Zeitaufwand von etwa $O(n^2 V_{\max})$. Man sagt, der Algorithmus sei *pseudopolynomiell*. Pseudopolynomiell bedeutet hier, dass die Laufzeit des Algorithmus polynomiell im Wert, aber nicht in der Eingabelänge von V ist. Der Parameter n hingegen ist unär in der Liste der Gewichte kodiert. Wenn V sehr groß ist, z.B. $2^{|n|}$, dann ist der Zeitaufwand nicht mehr polynomiell in der Eingabelänge. Deshalb ist jetzt nach einem Algorithmus gefragt, der in polynomieller Zeit läuft und eine approximative Lösung mit akzeptablem Fehler ausgibt.

Die Idee dabei ist, die Werte der Objekte zu runden, das heißt, die letzten b Stellen der Werte nicht zu berücksichtigen. Lässt man den bereits vorliegenden exakten Algorithmus für diese modifizierte Instanz laufen, so ist er wesentlich schneller fertig. macht aber einen gewissen Rundungsfehler. Die Kunst ist nun, die Anzahl der vernachlässigten Stellen geschickt zu wählen.

Für jedes $\text{value}(i)$ bezeichne $\text{value}(b, i)$, die durch Ersetzung der letzten b Bits durch Nullen entstehende Zahl (z.B. für b=3 und $\text{value}(i) = 101101$ ist $\text{value}(b, i) = 101000$). Beachte: $\text{value}(i) \geqslant \text{value}(b, i)$ und $\text{value}(b, i) = 2^b(\text{value}(i) \text{ DIV } 2^b)$.

Eine gegebene Eingabe $n, \text{weight}(1), \ldots, \text{weight}(n), \text{value}(1), \ldots, \text{value}(n), W_{\max}$ ersetzen wir durch n, $\text{weight}(1), \ldots, \text{weight}(n)$,$\text{value}(b, 1), \ldots, \text{value}(b, n)$,$W_{\max}$ und lassen den obigen exakten Algorithmus mit der veränderten Eingabe laufen.

Approximationsalgorithmus für MaxKnapsack

Eingabe: $n, \text{weight}(1), \ldots, \text{weight}(n), \text{value}(1), \ldots, \text{value}(n), W_{\max}$, Approximationsparameter m.
Wähle passende Anzahl b zu vernachlässigender Stellen.
Setze für alle i $\text{value}(b, i) = 2^b(\text{value}(i) \div 2^b)$.
Bilde neue MaxKnapsack-Instanz
$n, \text{weight}(1), \ldots, \text{weight}(n), \text{value}(b, 1)/2^b, \ldots, \text{value}(b, n)/2^b, W_{\max}$.
Berechne packing mit dem exakten Algorithmus für die neue Instanz.

Ausgabe: packing

Der Zeitaufwand des Approximationsalgorithmus ist $O(n^2(V_{\max}/2^b)$.

Wie gut ist nun eine solche Näherung, das heißt: Wie groß ist der Fehler? Sei P_{opt} eine optimale Lösung, $P(b)$ die Näherungslösung bei Vernachlässigung der letzten b Stellen. Dann gilt:

$$\sum_{i \in P(b)} \text{value}(i) \geqslant \sum_{i \in P(b)} \text{value}(b, i) \geqslant \sum_{i \in P_{\text{opt}}} \text{value}(b, i) \geqslant \sum_{i \in P_{\text{opt}}} (\text{value}(i) - 2^b) \geqslant \left(\sum_{i \in P_{\text{opt}}} \text{value}(i) \right) - n2^b$$

Die erste Ungleichung gilt, da die Werte $\text{value}(b)$ durch Abrunden entstehen. Die zweite Ungleichung gilt, da $P(b)$ für die modifizierte Eingabe eine optimale Lösung ist. Die dritte Ungleichung gilt, da durch das abrunden die einzelnen Werte um höchstens $2^b - 1$ kleiner werden. Die letzte Ungleichung gilt, da P_{opt} höchstens n Elemente enthält.

Bezeichnet $\text{value}(P_{\text{opt}})$ den Wert der optimalen Packung und $\text{value}(P(b))$ den Wert der gefundenen Packung bei Rundung von b Stellen, so gilt also

$$\text{value}(P(b)) \geqslant \text{value}(P_{\text{opt}}) - n2^b \quad \text{und damit} \quad 1 + \frac{n2^b}{\text{value}(P(b))} \geqslant \frac{\text{value}(P_{\text{opt}})}{\text{value}(P(b))} \; .$$

Da alle Einzelgewichte nicht größer als $W_{\max}$ sind, gilt sicher $\text{value}(P(b)) \geqslant V_{\max} - 2^b$ (wobei $Vmax$ der größte Einzelwert ist). Erst recht gilt dann $\text{value}(P(b)) \geqslant V_{\max} - n2^b$. Dies liefert

$$\frac{\text{value}(P_{\text{opt}})}{\text{value}(P(b))} \leqslant 1 + \frac{n2^b}{V_{\max} - n2^b}$$

Damit die Güte kleiner als $1 + \frac{1}{m}$ bleibt, muss für b gelten

$$\frac{n2^b}{V_{\max} - n2^b} \leqslant \frac{1}{m}; \quad \text{also} \quad k \leqslant \frac{V_{\max} - n2^b}{n2^b} \quad ; \text{also} \quad n(k+1) \leqslant \frac{V_{\max}}{2^b}$$

Im Algorithmus wird zu gegebenem k, n und $V_{\max}$ das maximale b gewählt, das gerade noch $n(k+1) \leqslant V_{\max}/2^b$ erfüllt. Dann wird Güte $1+1/m$ eingehalten. Da andererseits wegen der Maximalität von b auch $2n(m+1) \geqslant V_{\max}/2^b$ gilt, liegt die Laufzeit des Approximationsalgorithmus in $O(n^3 m)$. Damit gilt:

Satz 7.12. *Der obige Approximationsalgorithmus ist ein fully polynomial approximation scheme für* MaxKnapsack. *Also ist* MaxKnapsack $\in$ FPTAS. *Die Laufzeit des Algorithmus für Güte* $1 + 1/m$ *ist in* $O(n^3 m)$.

7.5 Trennende Probleme

Es stellt sich die Frage, ob es sich bei den Klassen von Optimierungsproblemen eigentlich um verschiedene Klassen handelt. Falls P = NP ist, so ist OptP = OptNP und alle betrachteten Klassen sind gleich. Und wenn OptP $\neq$ OptNP wäre? Unter der Prämisse, dass P $\neq$ NP ist, wird lässt sich zum Beispiel zeigen, dass APX nicht gleich OptNP ist. Dies zeigt man, indem man für ein Problem aus OptNP zeigt, dass es nicht in APX liegt (es sei denn OptP = OptNP):

Satz 7.13. *Wenn* P $\neq$ NP, *dann ist Min*TSP *nicht in* APX *und damit* APX $\neq$ OptNP.

Beweis Wir zeigen, dass aus der Existenz eines Approximationsalgorithmus für MinTSP die Existenz eines (polynomiell zeitbeschränkten) Algorithmus für das Entscheidungsproblem Hamilton folgt. Da Hamilton NP-vollständig ist, folgt dann auch P = NP, ein Widerspruch zur Prämisse des Satzes.

Sei M ein Approximationsalgorithmus für MinTSP mit Güte $r \in \mathbb{N}$. Der Algorithmus, der Hamilton entscheidet, arbeitet wie folgt: Bei Eingabe eines Graphen $G = (V, E)$, $|V| = n$, wird eine Instanz von MinTSP erzeugt mit V als Knotenmenge. Die Längenfunktion w wird wie folgt festgelegt:

1. Ist $(u, v) \in E$, so ist $w(u, v) = 1$.
2. Ist $(u, v) \notin E$, so ist $w(u, v) = rn$.

Hat G einen Hamilton-Kreis, so hat eine optimale Tour in der so gebildeten Instanz Länge n; hat G keinen Hamilton-Kreis, so hat jede Tour mindestens Länge $(n-1) + rn$. Im ersten Fall findet M für die TSP-Instanz aber eine Tour, die höchstens Länge rn hat. An der Ausgabe von M kann also erkennen, ob G einen Hamilton-Kreis besitzt oder nicht. □

So wie MinTSP die Klassen OptP und APX trennt, so trennt MaxKnapsack FPTAS und OptP:

Satz 7.14. *Wenn* P $\neq$ NP, *dann ist* MaxKnapsack *nicht in* OptP *und damit* FPTAS $\neq$ OptP.

Auch für die beiden weiteren Inklusionen lassen sich trennende Probleme finden. Ist OptP $\neq$ OptNP, so liegt also folgende Situation vor:

APX	$\subsetneq$	OptNP	Beispiele für trennendes Problem:	MinTSP, MaxCLIQUE.
PTAS	$\subsetneq$	APX	Beispiele für trennendes Problem:	Min$\triangle$TSP, Max 3-Sat.
FPTAS	$\subsetneq$	PTAS	Beispiel für trennendes Problem:	1ReleaseScheduling
OptP	$\subsetneq$	FPTAS	Beispiel für trennendes Problem:	MaxKnapsack.

Die strikte Inklusion wird jeweils gezeigt, indem ein konkretes Optimierungsproblem angegeben wird, das in der rechten, aber nicht in der linken Klasse liegt.

Bei dem Beispiel für ein Problem, das PTAS von FPTAS trennt, handelt es sich um das spezielles Scheduling-Problem SingleMachineSchedulingWithReleaseDates, kurz 1ReleaseScheduling. Mehr zum Thema *Approximation von Scheduling Problemen* findet sich in [Hal95]

Definition 7.15. Das Minimierungsproblem 1ReleaseScheduling ist wie folgt definiert:

Eingabe: Eine Menge von Jobs $\{1, \ldots, n\}$,
eine Liste $\text{time}(1), \ldots, \text{time}(n)$ von *processing times* aus $\mathbb{N}$,
eine Liste $\text{release}(1), \ldots, \text{release}(n)$ von *release dates* aus $\mathbb{N}$ sowie
eine Liste $\text{due}(1), \ldots, \text{due}(n)$ von *due dates* aus $\mathbb{N}$.

Lösungen: Eine Reihenfolge der Objekte $i_1, \ldots, i_n$ und
eine Liste $\text{start}(i_1), \ldots, \text{start}(i_n)$ von *start dates* aus $\mathbb{N}$,
so dass $\text{start}(i_j) + \text{time}(i_j) = \text{start}(i_{j+1})$ für $1 \leqslant j < n$ und $\text{start}(i_j) \geqslant \text{release}(i_j)$ für $1 \leqslant j \leqslant n$.

Ziel: Minimiere $\max\{start(i_j) + \text{time}(i_j) - \text{due}(i_j) \mid 1 \leqslant j \leqslant n\}$.
Minimiere also die maximale Verspätung bei der Erledigung der Jobs.

8 Probabilistische Algorithmen

Das Verhalten eines probabilistischen Algorithmus hängt nicht nur von der Eingabe, sondern auch von zufälligen Entscheidungen ab. Der Algorithmus kann gewissermaßen sein weiteres Verhalten „vom Ergebnis von Münzwürfen“abhängig machen. Oft sind probabilistische Verfahren für ein Problem einfacher und/oder schneller als entsprechende deterministische Algorithmen. Dafür muss man in der Regel in Kauf nehmen, dass das probabilistische Verfahren nicht mit hundertprozentiger Wahrscheinlichkeit das richtige Ergebnis liefert. In anderen Fällen liefern die Algorithmen zwar das korrekte Ergebnis, aber die Verkürzung der Rechenzeit wird nur im Durchschnitt, nicht aber für alle Eingaben erreicht. Probabilistische Algorithmen nennt man auch randomisiert.

Ein gutes Buch über probabilistische Algorithmen ist das Buch „Randomized Algorithms“ von Motwani und Rhagavan [MR95]. Eine weitere gute Quelle ist das entsprechende Kapitel in [Pap94].

8.1 Vom Nichtdeterminismus zum probabilistischen Algorithmus

Ausgangspunkt für die formale Definition probabilistischer Algorithmen ist die Idee, dass bereits nichtdeterministische Algorithmen eine Art von Zufall zu enthalten scheinen: Ein Wort x wird von einer nichtdeterministischen Maschine M genau dann akzeptiert, wenn es eine akzeptierende Berechnung *gibt*, die Wahrscheinlichkeit, einem akzeptierenden Pfad zu finden, nicht null ist..

Um zu überprüfen, ob ein Wort von der Maschine M akzeptiert wird, kann man r zufällig einen Berechnungspfad auswählen und kontrollieren, ob M das Eingabewort auf diesem Berechnungspfad akzeptiert. Falls $x \notin L(M)$, so wird bei der Kontrolle auf jeden Fall herauskommen, dass M das Wort auf dem geratenen Pfad nicht akzeptiert. Falls hingegen $x \in L(M)$ gilt, so besteht eine gewisse Chance, dass wir einen der akzeptierenden Berechnungspfade „erwischt“ haben.

Bei probabilistischen Algorithmen werden nun die Chancen dadurch verbessert, dass man die Anzahl der akzeptierenden Pfade erhöht. Man betrachtet Maschinen M, für die es zu Wörtern $x \in L(M)$ nicht nur wenigstens *einen* akzeptierenden Pfad gibt, sondern *sehr viele*.

Im folgenden Abschnitt wird die Idee einer „nichtdeterministischen Maschine mit vielen akzeptierenden Pfaden“ zunächst etwas formalisiert. Es wird dann auch gleich noch ein äquivalentes Modell für probabilistische Berechnungen vorgestellt.

8.2 Zwei Maschinenmodelle

Zwei Maschinenmodelle werden benutzt, um probabilistische Algorithmen zu formalisieren:

1. Nichtdeterministische Turingmaschine, deren Berechnungsbäume ausgeglichene Binärbäume sind.

   Jeder Knoten im Berechnungsbaum hat genau 2 oder 0 Nachfolger; alle Berechnungspfade haben die gleiche Länge. Ist $t_M(x)$ die Rechenzeit der Maschine M bei Eingabe x, so gibt es $2^{t_M(x)}$ Berechnungspfade. Die Anzahl der akzeptierenden Pfade bestimmt die Wahrscheinlichkeit, mit der x akzeptiert wird:

   $$\text{Prob}[M \text{ akz. } x] := \frac{\text{Anzahl akz. Pfade von } M \text{ bei Eingabe } x}{2^{t_M(x)}}.$$

   Es ist wichtig zu verstehen, worüber hier die Wahrscheinlichkeit genommen wird, nämlich über die Berechnungspfade. *Das Wort x ist nicht zufällig.* Vielmehr wird zu einer gegebenen Maschine M und einem gegebenen Wort x zufällig ein Berechnungspfad ausgewählt.

2. Probabilistische Turingmaschine: Zugriff auf ein gesondertes „Zufallsband“ durch eine ansonsten deterministische TM.

   Auf dem zweiten Eingabeband, dem „Zufallsband“, steht eine Folge von Nullen und Einsen. Der Kopf auf dem Zufallsband darf nur von links nach rechts bewegt werden, es darf nichts auf das Band geschrieben werden und das Band zählt wie das Eingabeband bei platzbeschränkten Berechnungen nicht mit. Die Anzahl der benötigten Zufallsbits ist allerdings eine zusätzliche Ressource neben der Rechenzeit und dem Platz. Es sei $r(n)$ minimal, so dass für alle Eingaben x der Länge n

für alle Bitstrings b der Länge $r(n)$ auf dem Zufallsband der Kopf nicht über das letzte Zeichen von b hinausgeht. Die Anzahl der Bitstrings der Länge $r(n)$, bei denen M das Wort x akzeptiert, bestimmt nun die Wahrscheinlichkeit, mit der x akzeptiert wird:

$$\text{Prob}[M \text{ akz. } x] := \frac{\left|\{\, b \in \{0,1\}^{r(|x|)} \mid M \text{ akz. } x \text{ mit } b \text{ auf dem Zufallsband} \,\}\right|}{2^{r(|x|)}}.$$

Die zwei Maschinenmodelle sind äquivalent im folgenden Sinne:

Satz 8.1. *Für jede Maschine M vom Typ 1 existiert eine Maschine M' vom Typ 2 mit denselben Laufzeit- und Platzschranken und*

$$\text{Prob}[M \textit{ akz. } x] = \text{Prob}[M' \textit{ akz. } x].$$

Umgekehrt existiert auch zu jeder Maschine M vom Typ 2 eine Maschine M' vom Typ 1 mit denselben Laufzeit- und Platzschranken und ebenfalls

$$\text{Prob}[M \textit{ akz. } x] = \text{Prob}[M' \textit{ akz. } x].$$

Beweis. Sei M eine nichtdeterministische Maschine vom Typ 1. Wir können diese durch eine deterministische Maschine M' vom Typ 2 ersetzen, die für jeden nichtdeterministischen Schritt von M einen deterministischen macht in Abhängigkeit von ihrem Zufallsband. Genauer entscheidet sie sich in jedem Schritt zwischen den jeweils genau zwei möglichen Folgezuständen der nichtdeterministischen Maschine, indem sie in den ersten dieser beiden Zustände geht, wenn auf dem Zufallsband eine 0 steht, und in den zweiten, wenn sie dort eine 1 findet.

Für die andere Richtung sei nun eine deterministische Turingmaschine mit Zufallsband gegeben. Diese können wir durch eine nichtdeterministische Maschine simulieren, die einfach immer eine nichtdeterministische Entscheidung fällt, wenn das Zufallsband gelesen werden soll. Wird hingegen gerade nichts vom Zufallsband gelesen, so machen wir einen normalen deterministischen Schritt, den man aber auch als eine entartete binäre nichtdeterministische Entscheidung auffassen kann. □

8.3 Einfache probabilistische Komplexitätsklassen

Wir führen nun über polynomiell zeitbeschränkte probablistische Turingmaschinen Komplexitätsklassen ein. In der folgenden Liste wird die Klassen NP und P nochmals mit aufgeführt, um deutlich zu machen, dass NP und P auch als probabilistische Klasse angesehen werden können.

Definition 8.2 (Probabilistische Klassen)**.** Eine Sprache A ist in der Klasse

1. RP, falls es eine polynomiell zeitbeschränkte probabilistische Turingmaschine M gibt mit

$$\begin{aligned} x \in A &\implies \text{Prob}[M \text{ akz. } x] \geqslant 1/2, \\ x \notin A &\implies \text{Prob}[M \text{ akz. } x] = 0. \end{aligned}$$

2. BPP, falls es eine polynomiell zeitbeschränkte probabilistische Turingmaschine M gibt mit

$$\begin{aligned} x \in A &\implies \text{Prob}[M \text{ akz. } x] \geqslant 2/3, \\ x \notin A &\implies \text{Prob}[M \text{ akz. } x] \leqslant 1/3. \end{aligned}$$

3. PP, falls es eine polynomiell zeitbeschränkte probabilistische Turingmaschine M gibt mit

$$\begin{aligned} x \in A &\implies \text{Prob}[M \text{ akz. } x] \geqslant 1/2, \\ x \notin A &\implies \text{Prob}[M \text{ akz. } x] < 1/2. \end{aligned}$$

Zum Vergleich die zwei bekannten Klassen:

4. NP, falls es eine polynomiell zeitbeschränkte probabilistische Turingmaschine M gibt mit

$$\begin{aligned} x \in A &\implies \text{Prob}[M \text{ akz. } x] > 0, \\ x \notin A &\implies \text{Prob}[M \text{ akz. } x] = 0. \end{aligned}$$

5. P, falls es eine polynomiell zeitbeschränkte probabilistische Turingmaschine M gibt mit

$$\begin{aligned} x \in A &\implies \text{Prob}[M \text{ akz. } x] = 1, \\ x \notin A &\implies \text{Prob}[M \text{ akz. } x] = 0. \end{aligned}$$

Die Klasse BPP ist offensichtlich unter Komplementbildung abgeschlossen; auch für PP kann man diese Abschlusseigenschaft leicht nachweisen (Übungsaufgabe). Es ist unbekannt, ob die Klasse RP unter Komplementbildung abgeschlossen ist. Deshalb wird noch eine Klasse eingeführt, die die RP-Probleme enthält, deren Komplement auch in RP ist:

Definition 8.3 (ZPP)**.**

$$\text{ZPP} := \text{RP} \cap \text{co-RP}\,.$$

Falls eine Sprache A in ZPP liegt, gibt es also sowohl einen RP-Algorithmus für A als auch einen RP-Algorithmus für das Komplement von A. Dies ist dazu äquivalent, dass es eine probabilistische Maschine gibt, die mit Wahrscheinlichkeit $\geqslant 1/2$ die richtige Antwort auf die Frage „$x \in A$?“ liefert, und in den verbleibenden Fällen mit „weiß nicht“antwortet. Eine solche Maschine liefert einem also manchmal keine Information, aber nie eine Fehlinformation. Dies rechtfertigt den Namen ZPP für *zero error probabilistic polynomial time.*

In der Definition von RP wird festgelegt, dass die Wahrscheinlichkeit, einen akzeptierenden Pfad zu erwischen, mindestens 1/2 betragen muss, falls es einen solchen Pfad gibt. Im Allgemeinen ist man aber an zuverlässigeren Aussagen interessiert. Der folgende Satz zeigt, dass wir unsere Chancen bei Sprachen in R-Klassen auf einfache Weise beliebig erhöhen können.

Satz 8.4 (Fehler reduzieren bei RP)**.**
Sei $A \in \text{RP}$ und sei q ein Polynom. Dann gibt es eine polynomiell zeitbeschränkte probabilistische Maschine M', die für alle x ($|x| = n$) folgendes erfüllt:

$$\begin{aligned} x \in A &\implies \text{Prob}[M' \textit{ akz. } x] \geqslant 1 - \frac{1}{2^{q(n)}}, \\ x \notin A &\implies \text{Prob}[M' \textit{ akz. } x] = 0. \end{aligned}$$

Beweis. Sei $A \in \text{RP}$ via M. Eine Maschine M' bei Eingabe x, $|x| = n$, führt die Maschine M $q(n)$ mal hintereinander aus. Sollte M bei einem der Durchläufe akzeptieren, so akzeptiert auch M'. Andernfalls verwirft M'.

Betrachten wir zunächst Prob$[M'$ akz. $x]$, falls $x \notin A$ gilt. Dann wird M bei keiner der Simulationen auf irgendeinem Pfad akzeptieren, und folglich wird ebenfalls M' das Wort x auf keinem Pfad akzeptieren. Also gilt Prob$[M'$ akz. $x] = 0$.

Betrachten wir nun den Fall $x \in A$. Jedesmal, wenn M' die Maschine M simuliert, besteht mindestens eine fifty-fifty Chance, dass M das Wort x akzeptiert. Die Wahrscheinlichkeit, dass wir jedes Mal Pech haben, ist deshalb höchstens $(1/2)^{q(n)}$. Also akzeptiert M' das Wort x mit der Wahrscheinlichkeit $1 - \frac{1}{2^{q(n)}}$. □

Als nächstes betrachten wir, wie die probablistischen Klassen relativ zu den bekannten Komplexitätsklassen liegen und dass sie unter Reduktion abgeschlossen sind.

Satz 8.5 (Inklusionen, Reduktionsabschluss)**.**

1. $\text{P} \subseteq \text{RP} \subseteq \text{NP} \subseteq \text{PP} \subseteq \text{PSPACE}$.

2. $\text{RP} \subseteq \text{BPP} \subseteq \text{PP}$.

3. RP, BPP *und* PP *sind unter Logspace-Many-One-Reduktion abgeschlossen.*

Beweis. Für die Inklusion RP $\subseteq$ BPP benutzt man, dass man bei RP die Fehlerwahrscheinlichkeit unter 1/3 drücken kann. Bei der Inklusion PP $\subseteq$ PSPACE benutzt man, dass man auf polynomiellen Platz die Berechnung einer polynomiell zeitbeschränkten Maschine für jeden der möglichen Zufallsstrings simulieren kann.

Die Aussagen über die Abgeschlossenheit unter Reduktion zeigt man „wie üblich". □

Es ist unbekannt, ob BPP $\subseteq$ NP gilt. Ebenso ist unbekannt, ob NP $\subseteq$ BPP gilt.

Ähnlich wie bei RP kann man auch bei BPP die Fehlerwahrscheinlichkeit durch Wiederholen der Berechnung reduzieren:

Satz 8.6 (Fehler reduzieren bei BPP)**.**
Sei $A \in$ BPP und sei q ein Polynom. Dann gibt es eine polynomiell zeitbeschränkte probabilistische Maschine M' mit

$$x \in A \implies \mathrm{Prob}[M' \text{ akz. } x] \geqslant 1 - \frac{1}{2^{q(n)}},$$
$$x \notin A \implies \mathrm{Prob}[M' \text{ akz. } x] \leqslant \frac{1}{2^{q(n)}}.$$

Beweisidee Sei $A \in$ BPP via M. Wie im RP-Fall simuliert nun die Maschine M' die Maschine M hinreichend oft. Allerdings ist es nun schwieriger zu entscheiden, was die Maschine M' am Ende ausgeben soll – schließlich können wir uns nicht mehr darauf verlassen, dass die Maschine M im Fall $x \notin A$ immer verwirft. Vielmehr besteht eine nicht unerhebliche Wahrscheinlichkeit, dass M ab und zu auch im Fall $x \notin A$ das Wort x akzeptiert. Würde also die Maschine M' einfach immer akzeptieren, sobald M dies tut, würden wir viel zu häufig Wörter akzeptieren, die nicht in A liegen.

Aus diesem Grund verfahren wir wie folgt:

Eingabe x. $|x| = n$.
count := 0.
tue $r(n)$ ***mal***
  Lasse M auf Eingabe x laufen.
  falls M akzeptiert
    count := *count* + 1
falls *count* $> r(n)/2$
  akzeptiere
sonst
  verwerfe

Die Idee ist also, dass wir „der Mehrheit" glauben. Wir wählen $r(n)$, die Anzahl der Simulationen, so, dass $r(n)$ ungerade ist und dass $r(n)$ ein Polynom ist. Die Funktion $r(n)$ muss natürlich von $q(n)$ abhängen. wir setzen:

$$r(n) := 2kq(n) + 1$$

Dabei ein k eine Konstante, deren Wert wir noch festlegen werden. Wir schätzen nun die Wahrscheinlichkeit $\mathrm{Prob}[M'(x) \neq \chi_A(x)]$ ab. Wir gehen dabei vom schlechtesten Fall aus, dass $\mathrm{Prob}[M(x) \neq \chi_A(x)]$ gleich 1/3 ist. Wir beschreiben den Ausgang der Simulationen durch Folgen aus $\{R, F\}^{r(n)}$, wobei R für $M(x) = \chi_A(x)$ und F für $M(x) \neq \chi_A(x)$ steht. Für eine solche Folge mit j vielen Rs ist die Wahrscheinlichkeit, dass sie eintritt,

$$\left(\frac{2}{3}\right)^j \left(\frac{1}{3}\right)^{r(n)-j} .$$

Es gibt $\binom{r(n)}{j}$ solche Folgen. Also ist

$$\begin{aligned}
\text{Prob}[M'(x) \neq \chi_A(x)] &\leqslant \sum_{j=0}^{j<r(n)/2} \left(\frac{2}{3}\right)^j \left(\frac{1}{3}\right)^{r(n)-j} \binom{r(n)}{j} \\
&\leqslant \sum_{j=0}^{k\,q(n)} \left(\frac{2}{3}\right)^{1/2\,r(n)} \left(\frac{1}{3}\right)^{1/2\,r(n)} \binom{r(n)}{j} \\
&\leqslant \sum_{j=0}^{k\,q(n)} \left(\frac{2}{9}\right)^{1/2\,r(n)} \binom{r(n)}{j} \leqslant \left(\frac{2}{9}\right)^{1/2r(n)} \frac{1}{2} 2^{r(n)} \\
&\leqslant \frac{1}{2} \left(\frac{8}{9}\right)^{1/2\,r(n)} \leqslant \left(\frac{8}{9}\right)^{k\,q(n)}
\end{aligned}$$

Ab $k = 6$ gilt $(8/9)^k \leqslant 1/2$. Dann ist

$$\text{Prob}[M'(x) \neq \chi_A(x)] \leqslant \left(\frac{1}{2}\right)^{q(n)} .$$

Die Laufzeit von M' bleibt polynomiell. □

8.4 Beispiele probabilistischer Algorithmen

Primzahlen

Als erstes Problem wollen wir das Primzahlproblem betrachten. Genauer wollen wir einen probabilistischen Algorithmus für das Problem Composites angeben. Inzwischen ist zwar bekannt, dass sogar Composites $\in$ P gilt [AKS02], aber der hier vorgestellte Algorithmus ist dennoch nicht wertlos, denn erstens ist er verhältnismäßig leicht zu verstehen; und vor allem ist seine Laufzeit wesentlich besser als die bisher bekannte Laufzeitschranke für den Algorithmus von Agrawal, Kayal und Saxena.

Der probabilistische Algorithmus benutzt den folgenden Satz 8.8 aus der Zahlentheorie, der eine Verfeinerung des Satzes von Fermat darstellt:

Satz 8.7 (Satz von Fermat)**.** *Ist p Primzahl, so gilt für alle $a \in \{1, \ldots, p-1\}$*

$$a^{p-1} \equiv 1 \mod p.$$

Satz 8.8. *Ist p Primzahl und sind u und k die Zahlen mit der Eigenschaft $p - 1 = u2^k$, u ungerade, so gilt für alle $a \in \{1, \ldots, p-1\}$*

$$a^u \equiv \pm 1 \textit{ oder es existiert ein } i \in \{1, \ldots, k-1\} \textit{ mit } a^{u2^i} \equiv -1. \qquad (*)$$

Beispiel 8.9. Betrachte wir den Fall $p = 5$. Dann ist $u = 1$ und $k = 2$. Für $i \in \{1, \ldots, k-1\}$ kommt also nur $i = 1$ in Frage.

a	1	2	3	4
$a^u \mod 5$	1	2	3	$4 \equiv -1$
$a^2 = a^{u2^i}$		$4 \equiv -1$	$9 \equiv -1$	

Beispiel 8.10. Betrachten wir andererseits den Fall $p = 9$. Dann ist $u = 1$ und $k = 3$. Für i kommen nun 1 und 2 in Frage.

a	1	2	3	4	5	6	7	8
$a^u \mod 9$	1	2	3	4	5	6	7	$8 \equiv -1$
$a^2 = a^{u2^1}$		4	0	7	7	0	4	
$a^4 = a^{u2^2}$		7	0	4	4	0	7	

Wir definieren nun Belege für Zusammengesetztheit:

Definition 8.11. Ist n zusammengesetzt, so heißt ein a, das nicht Bedingung $(*)$ erfüllt, „Beleg für die Zusammengesetztheit“ von n.

Ob ein gegebenes a ein Beleg für die Zusammengesetztheit von n ist, lässt sich in Polynomialzeit bezüglich $|\mathrm{bin}(n)|$ prüfen.

Aus der Zahlentheorie ist ebenfalls bekannt, wieviele Belege für die Zusammengesetztheit von n es gibt:

Satz 8.12. *Ist n eine ungerade zusammengesetzte Zahl, so sind mindestens $\frac{3}{4}$ der Zahlen aus $\{2, \ldots, n-1\}$ Belege für die Zusammengesetztheit von n.*

Folgerung 8.13.

$$\textsc{Composites} \in \mathrm{RP}$$

Symbolische Determinante

In vielen Anwendungen muss man als Teilaufgabe die Determinante einer gegebenen Matrix ausrechnen. Zur Erinnerung: Die Determinante $\det A$ einer $n \times n$-Matrix A ist definiert als

$$\det A := \sum_{\pi \in S_n} \sigma(\pi) \prod_{i=1}^{n} A_{i,\pi(i)} .$$

Dabei ist S_n die Menge aller Permutationen der Menge $\{1, \ldots, n\}$. Das Vorzeichen $\sigma(\pi)$ der Permutation π ist 1, falls sich π in geradzahlig viele Transpositionen zerlegen lässt, sonst ist es -1. Und $A_{i,j}$ ist der Matrixeintrag, der in der i-ten Zeile und j-ten Spalte steht.

Manchmal reicht es auch, nur zu bestimmen, ob die Determinante einer Matrix gleich 0 ist oder nicht. Zum Beispiel sind n Vektoren in einem n-dimensionalen Vektorraum genau dann linear unabhängig, wenn die Determinante der aus den n Vektoren als Spalten gebildeten Matrix ungleich 0 ist.

Für beide Aufgaben (die Funktionsberechnung und den „Null-Test“) bietet sich der Gauß-Algorithmus als Verfahren an. Mit Zeilenvertauschungen (wechselt nur das Vorzeichen der Determinante) und Additionen des Vielfachen einer Zeile zu anderen Zeile (ändert die Determinante nicht) bringt man die Matrix in obere Dreiecksform. Das Produkt der Elemente in der Diagonalen ist dann gleich der Determinante.

Sind die Matrixeinträge zum Beispiel rationale Zahlen (dargestellt als Brüche ganzer Zahlen in Binärdarstellung), so kann man deterministisch in polynomieller Zeit die Determinante ausrechnen. In anderen Fällen ist der Gauß-Algorithmus jedoch nicht effizient, weil während der Umformung die Einträge in der Matrix exponentiell groß werden können. Dies ist zum Beispiel der Fall, wenn die Matrix Polynome in mehreren Variablen enthält (über dem Grundkörper $\mathbb{Q}$ der rationalen Zahlen). Ein deterministisches Polynomialzeitverfahren ist für diese Problemstellung nicht bekannt. Für den Test, ob die Determinante einer solchen Matrix gleich dem Nullpolynom ist, ist aber immerhin ein probabilistischer Polynomialzeitalgorithmus bekannt. Dieser Algorithmus wird im Folgenden vorgestellt.

Die Grundidee des Algorithmus besteht darin, für die Variablen in der Matrix eine Belegung mit natürlichen Zahlen *zufällig auszuwählen.* Da ein Polynom, das nicht das Nullpolynom ist, nur wenig Nullstellen hat, ist es sehr unwahrscheinlich, dabei gerade eine Nullstelle des Polynoms zu treffen. Ist die Determinante der Matrix unter der geratenen Belegung null, ist die Determinante der Matrix selbst also höchstwahrscheinlich das Nullpolynom.

Um die Sache formaler angehen zu können, brauchen wir eine Aussage über die Anzahl der ganzzahligen Nullstellen eines Polynoms in einem vorgegebenen Bereich:

Lemma 8.14. *Sei $P(x_1, \ldots, x_m)$ ein Polynom in m Variablen, das nicht gerade das Nullpolynom ist. Der Grad jeder Variablen sei höchstens d.*

Sei $M \in \mathbb{N}$. Dann ist die Anzahl der Nullstellen $(x_1, \ldots, x_m)$ von P mit $x_i \in \{0, \ldots, M-1\}$ für jedes i höchstens

$$m \cdot d \cdot M^{m-1} .$$

Beweis. Induktion über m. Den Beweis findet man in [Pap94], Lemma 11.1. $\square$

Für die Wahrscheinlichkeit, dass ein Tupel $(i_1, \ldots, i_m)$ Nullstelle ist, gilt also

$$\operatorname*{Prob}_{x_i \in \{0,\ldots,M-1\}} [P(x_1, \ldots, x_m) = 0] \leqslant \frac{mdM^{m-1}}{M^m} = \frac{md}{M} .$$

Wir geben nun die Definition des Problems, für das wir einen probabilistischen Algorithmus angeben werden. Wir fragen dabei, ob die Determinante einer gegebenen Matrix *ungleich* dem Nullpolynom ist:

- Symbolische nichtverschwindende Determinante SYMBDET

  Eingabe: Eine $n \times n$-Matrix A, deren Einträge Polynome über $\mathbb{Q}$ sind. Alle vorkommenden Exponenten sind höchstens n.

  Frage: Ist $\det A \not\equiv 0$?

Die Frage ist also, ob man die in der Matrix vorkommenden Variablen so belegen kann, dass die Determinante der Matrix nicht null ist.

Beispiel 8.15. Folgende Matrix ist ein Element von SYMBDET

$$\begin{pmatrix} a^2b + c & b - c & 0 \\ 1 & 2 & -c^2 \\ d & e & f^2 \end{pmatrix} .$$

Beispiel 8.16. Folgende Matrix ist hingegen kein Element von SYMBDET, da die Determinante immer null ist:

$$\begin{pmatrix} a^2b & b + c & 5 \\ b & c + d & 3d \\ a^2b - b & b - d & 5 - 3d \end{pmatrix} .$$

Satz 8.17.

$$\text{SYMBDET} \in \text{RP} .$$

Beweis. Der probabilistische Algorithmus arbeitet wie folgt:

Eingabe: $n \times n$-Matrix A; Einträge sind Polynome über $\mathbb{Q}$.
Seien $x_1, \ldots, x_m$ die in A vorkommenden Variablen.
falls ein Exponent $> n$ vorkommt
  verwerfe
$M :=$ kleinste Zweierpotenz $\geqslant 2mn^2$.
rate $i_1, \ldots, i_m \in \{0, \ldots, M - 1\}$.
$A' := A(x_1/i_1, \ldots, x_m/i_m)$.
Werte die arithmetischen Ausdrücke in A' aus.
Berechne $\det A'$ mit dem Gauß-Algorithmus.
falls $\det A' \neq 0$
  akzeptiere .
sonst
  verwerfe .

In der dritten Zeile wird geprüft, ob die Exponenten in der Matrix nicht zu groß sind, was einen Syntaxfehler darstellen würde. Dann wird M einerseits groß genug gewählt, dass die Wahrscheinlichkeit, die Eingabe fälschlich zu akzeptieren, klein genug wird; andererseits wählt man eine Zweierpotenz, damit es leicht fällt, *gleichverteilt* auf die Tupel $(i_1, \ldots, i_m)$ zu verzweigen. Jedem $i_j \in \{0, \ldots, M - 1\}$ kann man dann nämlich eindeutig einen Bitstring der Länge $\log M$ zuordnen, den man Bit für Bit raten kann. Da m und n in der Eingabelänge beschränkt sind, ist auch $(i_1, \ldots, i_m)$ polynomiell längenbeschränkt bezüglich n. Damit ist das Raten in polynomieller Zeit durchführbar.

In der siebten Zeile wird für jedes j jedes Vorkommen der Variablen x_j durch die geratene natürliche Zahl i_j ersetzt. Dann werden an jeder Matrixposition die entstandenen arithmetischen Ausdrücke ausgewertet. Dafür müssen natürliche Zahlen potenziert, multipliziert und addiert werden. Die Anzahl der auszuführenden Operationen ist durch die Größe der Eingabe beschränkt. Da die Exponenten durch n nach oben beschränkt sind, können auch durch das Potenzieren die berechneten Zahlen nicht zu groß werden. Die Auswertung ist deshalb in polynomieller Zeit durchführbar. Nun wird $\det A'$ mit dem Gauß-Algorithmus bestimmt. Dies geht in polynomieller Zeit, und da die Einträge in A' nur polynomiell groß sind bezüglich der Eingabelänge, auch in polynomieller Zeit bezüglich der Eingabelänge. Falls $\det A' \neq 0$ ist, so ist also das Polynom $\det A$ an der Stelle $(i_1, \ldots, i_m)$ ungleich 0; deshalb ist $\det A$ nicht das Nullpolynom und der Algorithmus akzeptiert zu Recht. Ist $\det A$ das Nullpolynom, so ist $\det A' = 0$ und der Algorithmus verwirft auf jeden Fall.

Wie groß ist nun die Wahrscheinlichkeit, dass $\det A \neq 0$ und der Algorithmus trotzdem verwirft? Dies geschieht nur, wenn $(i_1, \ldots, i_m)$ zufälligerweise eine Nullstelle von $\det A$ ist. Wir wenden Lemma 8.14 an. Da die Exponenten in A durch n beschränkt sind und in den Summanden der Determinante immer n Matrixeinträge aufmultipliziert werden, gilt $d \leqslant n^2$. Wir erhalten deshalb (unter Benutzung von $M \geqslant 2mn^2$):

$$\operatorname*{Prob}_{i_j \in \{0,\ldots,M-1\}} \big[\det A(i_1, \ldots, i_m) = 0\big] \leqslant \frac{md}{M} \leqslant \frac{mn^2}{M} \leqslant \frac{mn^2}{2mn^2} = \frac{1}{2}\,.$$

Der Algorithmus erfüllt also die RP-Bedingung aus Definition 8.2. □

Wie man aus dem Beweis sieht, ist die Schranke für die Exponentengröße in der Definition von SymbDet etwas zufällig gewählt. Es reicht, dass die Exponentengröße durch ein festes Polynom in n beschränkt ist.

8.5 Die Klasse #P und ihre Beziehung zu PP

Viele Sprachklassen zwischen P und PSPACE (wie NP, RP,BPP und PP) sind definiert über Bedingungen für die Anzahl der akzeptierenden Pfade nichtdeterministischer Polynomialzeitmaschinen. Wie schwer ist es eigentlich, diese Anzahl für eine Maschine M und Eingabe x zu bestimmen? Wir definieren zunächst eine passende Komplexitätsklasse, d.h. eine Funktionenklasse, die genau solche Anzahlfunktionen enthält. Wir werden sie dann zu anderen Sprach- und Funktionenklassen in Beziehung setzen.

Definition 8.18 (Anzahl der Pfade)**.**
Für eine nichtdeterministische Turingmaschine M mit Eingabealphabet Σ ist die Funktion $\#_M : \Sigma^* \to \mathbb{N}$ definiert durch

$$\#_M(x) := \text{Anzahl der akzeptierenden Pfade von } M \text{ bei Eingabe } x.$$

Definition 8.19 (#P)**.** Eine Funktion $f \colon \Sigma^* \to \{0,1\}^*$ ist in #P, falls eine polynomiell zeitbeschränkte NTM M existiert, für die für alle x gilt:

$$f(x) = \mathrm{bin}(\#_M(x)).$$

Satz 8.20.

$$\mathrm{P}(\#\mathrm{P}) = \mathrm{P}(\mathrm{PP})\,.$$

Beweisidee Um festzustellen, ob die Mehrheit der Berechnungspfade akzeptiert, ist eine #P-Anfrage nötig. Umgekehrt kann man die exakte Anzahl akzeptierender Pfade durch binäre Suche mit einem geeigneten PP-Orakel berechnen. □

8.6 BPP und Schaltkreisgröße

In diesem Abschnitt wollen wir das wichtige Resultat beweisen, dass alle Sprachen in BPP und damit auch alle Sprachen in RP und ZPP von einer lediglich polynomiell großen Schaltkreisfamilie berechnet werden können. Wegen Satz 5.13 heißt das, dass wir die Inklusion BPP $\subseteq$ P/poly zeigen.

Die überraschende Aussage des folgenden Satzes ist, dass es für jede Sprache in BPP für jede Wortlänge einen einzelnen Zufallsstring b gibt, so dass wir alle Wörter dieser Länge ganz leicht entscheiden könnten, wenn wir ihn zur Verfügung hätten.

Satz 8.21. *Für jedes $A \in \mathrm{BPP}$ existiert eine probabilistische, polynomiell zeitbeschränkte Turingmaschine M derart, dass für alle Längen n ein einziger Zufallsstring $b_n \in \{0,1\}^{r(n)}$ existiert mit folgender Eigenschaft: Jedes Wort x der Länge n wird von M mit diesem Zufallsstring b_n genau dann akzeptiert, wenn $x \in A$. Hierbei ist $r(n)$ die Anzahl der von M verbrauchten Zufallsbits.*

Beweis. Sei also $A \in \mathrm{BPP}$. Zunächst liefert uns nun Satz 8.6 eine Maschine M mit der Eigenschaft, dass $\mathrm{Prob}[\ M$ irrt bei $x] \leqslant 2^{-(n+1)}$. Sei x ein Wort der Länge n. Betrachten wir nun die Anzahl der Bitstrings $c \in \{0,1\}^{r(n)}$, für die M bei Eingabe x und Zufallsbits c das falsche Ergebnis liefert. M kann sich für höchstens $\frac{2^{r(n)}}{2^{n+1}}$ viele c irren.
Es gibt insgesamt 2^n viele Wörter der Länge n. Also gibt es insgesamt höchstens

$$\frac{2^{r(n)}}{2^{n+1}} 2^n = \frac{2^{r(n)}}{2}$$

viele Bitstrings c, *für die sich M für irgendein Wort x der Länge n irrt.* Da es aber $2^{r(n)}$ Kandidaten-Bitstrings der Länge $r(n)$ gibt, folgt, *dass sich für mindestens einen dieser Bitstrings die Maschine M bei keinem Wort x der Länge n irrt.* Tatsächlich ist dies sogar für mindestens die Hälfte aller c der Fall. □

Man beachte, dass dieser Beweis *nicht konstruktiv ist.* Wir wissen zwar, dass solche „tollen“ Bitstrings existieren (es gibt sie sogar in Massen). Wir wissen aber nicht, wie wir uns welche beschaffen können.

Folgerung 8.22.

$$\mathrm{BPP} \subseteq \mathrm{POLYSIZE}\,.$$

Beweis. Eine polynomiell zeitbeschränkte DTM M_B kann bei Eingabe $\langle x, b\rangle$ das Programm der Maschine M für Eingabe x mit Zufallsstring b simulieren. Wird als Advice $h(n)$ der String b_n nach dem obigen Satz gewählt, so gilt $x \in A \Leftrightarrow \langle x, b_n\rangle \in B$, wobei $B = L(M_B)$. □

9 Teilinformationsalgorithmen

9.1 Einführung

Polynomielle Teilinformationsklassen erweitern die Klasse P. Ein *Teilinformationsalgorithmus* für eine Sprache $A \subseteq \Sigma^*$ (und eine feste Eingabetupellänge m) bestimmt bei Eingabe von m Wörtern $x_1, \dots, x_m$ eine Information über $\chi_A(x_1, \dots, x_m)$, schließt also einige Werte für $\chi_A(x_1, \dots, x_m)$ aus. Der Algorithmus berechnet eine Menge D von Bitstrings, so daß $\chi_A(x_1, \dots, x_n) \in D$. Ein Informationstyp wird bestimmt durch eine Menge $\mathcal{D}$ von Mengen von Bitstrings. Solche $\mathcal{D}$ werden Familien genannt. Eine Sprache A ist in der Teilinformationsklasse P $[\mathcal{D}]$, falls es einen polynomiell zeitbeschränkten Teilinformationsalgorithmus gibt, der für alle Eingaben $(x_1, \dots, x_m)$ ein $D \in \mathcal{D}$ ausgibt mit $\chi_A(x_1, \dots, x_m) \in D$.

Definition 9.1 (Pools, Familien, Teilinformationsklassen)**.**

1. Ein *m-Pool* ist eine Teilmenge von $\{0, 1\}^m$.
2. Eine *m-Familie* ist eine Menge $\mathcal{D}$ von m-Pools, so dass $\mathcal{D}$ die Menge $\{0, 1\}^m$ überdeckt ; es muss also gelten: $\{b \mid \text{es existiert } D \in \mathcal{D} \text{ mit } b \in D\} = \{0, 1\}^m$
3. $A \in \mathrm{P}[\mathcal{D}]$ gdw ein $f \in \mathrm{FP}$ existiert, so dass
   (a) $f(x_1, \dots, x_m) \in \mathcal{D}$ und
   (b) $\chi_A(x_1, \dots, x_m) \in f(x_1, \dots, x_m)$.

Spezielle Arten von Teilinformationsalgorithmen wurden zunächst in der Rekursionstheorie eingeführt, z.B. in [Joc68], bevor das Konzept Ende der Siebziger [Sel79] in den Polynomialzeit-Bereich übertragen wurde . Eine gute Einführung in die Theorie der Teilinformationsklassen findet sich in [NT03].

9.2 Drei Beispiele

Wir geben drei relativ prominente Beispiele für Teilinformationklassen an:

Definition 9.2 (p-selektiv, approximierbar, leicht zählbar)**.**

- Wähle $m = 2$ und $\mathcal{D}$ bestehe nur aus den zwei Mengen $\{00, 01, 11\}$ und $\{00, 10, 11\}$. Sprachen in P $[\mathcal{D}]$ heißen dann *p-selektiv.*
- Sei $m \geqslant 2$ und $\mathcal{D}$ bestehe aus den Pools, in denen ein Bitstring der Länge m fehlt; also $\mathcal{D} = \{D \subseteq \{0, 1\}^m \mid |D| = 2^m - 1\}$. Sprachen in P $[\mathcal{D}]$ heißen dann *m-approximierbar.*
- Wähle $m = 2$ und $\mathcal{D}$ bestehe aus den zwei Mengen $\{00, 11\}$, $\{00, 10, 10\}$ und $\{01, 10, 11\}$. Sprachen in P $[\mathcal{D}]$ heißen dann *leicht 2-zählbar.*

Wir betrachten hauptsächlich den besonders gut untersuchten Fall der p-selektiven Sprachen. wie kommt der Name zustande? Das „p“steht für polynomiell zeitbeschränkt. „Selektiv”heißen die Sprachen, weil man aus zwei Worten x_1, x_2 ein Wort x_i auswählen kann, das in der Sprache ist, wenn überhaupt eins von beiden in der Sprache ist.

Definition 9.3 (Äquivalente Definition von p-selektiv)**.**
Eine Sprache A ist *p-selektiv*, falls es eine zweistellige Funktion $f \in \mathrm{FP}$ gibt, die für alle $x_1, x_2 \in \Sigma^*$ die folgenden Eigenschaften erfüllt:

(a) $f(x_1, x_2) \in \{x_1, x_2\}$.

(b) $A \cap \{x_1, x_2\} \neq \emptyset \;\Rightarrow\; f(x_1, x_2) \in A$.

Die Klasse der p-selektiven Sprachen liegt in gewissem Sinne quer zu den klassischen Komplexitätsklassen. Es gibt sogar nicht-entscheidbare p-selektive Sprachen; andererseits ist nicht ausgeschlossen, dass es schon in NP Sprachen gibt, die nicht p-selektiv sind. Allerdings wären solche Sprachen dann nicht NP-volständig.

9.3 P/poly und Teilinformation

Satz 9.4. *Jede p-selektive Sprache ist in* P/poly.

Beweis Sei A p-selektiv via $f \in \mathrm{FP}$. Ziel ist es nun, für jede Wortlänge n einen Advice, also eine Zusatzinformation, $h(n)$ zu bestimmen, so dass für jedes x die Frage „$x \in A$?“mithilfe von $h(|x|)$ entschieden werden kann.

Definiere für Wortlänge n für jede Wortmenge $V \subseteq \Sigma^n$ den gerichteten Graphen $G_V = (V, E_V)$, derart dass für $x, y \in V$, $x \neq y$, gilt:

$$(x, y) \in E_v \Leftrightarrow f(x, y) \subseteq \{00, 01, 10\} \ .$$

Es führt also eine Kante von x nach y, falls f die Information „$x \in A \Rightarrow y \in A$“liefert. Da G_V mindestens $\binom{|V|}{2}$ Kanten enthält, gibt es in G_V einen Knoten mit Ausgrad $\geqslant (|V| - 1)/2$. Ist $\Sigma^n \cap A \neq \emptyset$, so lässt sich nun iterativ eine Folge $v_1, \ldots, v_r$ bestimmen, so dass für $x \in \Sigma^n$ gilt:

$$x \in A \Leftrightarrow \exists i.\, (v_i, x) \in E_{\Sigma^n} \wedge v_i = x \ .$$

Da bei der Konstruktion der Folge $v_1, \ldots, v_r$ die Menge der noch nicht abgedeckten Worte sich mindestens halbiert, kann man zeigen, dass immer $r \leqslant n$ gilt. Der Advice $h(n)$ liefert nun die Information, ob $\Sigma^n \cap A$ leer ist, und wenn nicht, liefert $h(n)$ die Folge $v_1, \ldots, v_r$. Die Länge des Advice liegt in $O(n^2)$.

Der Algorithmus, der x, $|x| = n$, mit Advice $h(n)$ entscheidet, arbeitet wie folgt: Ist $\Sigma^n \cap A = \emptyset$, so wird x verworfen. Sonst wird geprüft, ob $\exists i.\, (v_i, x) \in E_{\Sigma^n} \wedge v_i = x$. Wenn ja, akzeptiere, sonst verwerfe. Da die Anzahl der v_i linear ist, und die Kantenrelation wegen $f \in \mathrm{FP}$ in Polynomialzeit entscheidbar ist, ist der Algorithmus polynomiell zeitbeschränkt. □

Nicht nur die p-selektiven Sprachen, sondern alle (nichttrivialen) Teilinformationsklassen sind in P/poly. Der Beweis des folgenden Satzes von Amir und Gasarch [AG88] ist aber um einiges komplizierter:

Satz 9.5. *Ist $\mathcal{D}$ eine m-Familie mit $\{0,1\}^m \notin \mathcal{D}$, so gilt:*

$$\mathrm{P}[\mathcal{D}] \subseteq \mathrm{P/poly} \ .$$

Gibt es umgekehrt auch für jede Sprache in P/poly auch einen polynomiellen Teilinformationsalgorithmus? Nein, das ist nicht der Fall. Aber wenn die Art der Teilinformation nicht zu eng gewählt wird, so lässt sich wenigstens jede Sprache in P/poly auf eine Sprache mit Teilinformationsalgorithmus reduzieren:

Satz 9.6. *Jede Sprache in* P/poly *lässt sich auf eine p-selektive Sprache $\leqslant_T^p$-reduzieren.*

Beweis Der Beweis vollzieht sich in zwei Schritten. Zunächst zeigt man, dass Sprachen in P/poly sich auf tally Sprachen reduzieren lassen. Eine Sprache ist tally, wenn sie nur Wörter aus $\{1\}^*$ enthält. Hier reichen sogar Truth-Table-Reduktionen aus.

Dann zeigt man, dass es zu jeder tally Sprache T eine p-selektive Sprache A gibt, so dass $T \leqslant_T^p A$. Sei T eine tally Sprache. Sei c_T dass unendliche Wort über $\{0, 1\}$, dass die charakteristische Funktion von T repräsentiert. D.h., es gilt

$$1^n \in T \Leftrightarrow c_T[n] = 1 \ .$$

Sei $\leqslant_{\mathrm{lex}}$ die Lexikon-Ordnung auf Wörtern (Achtung; nicht die Standard-Ordnung). Definiere A durch

$$w \in A \Leftrightarrow w \leqslant_{\mathrm{lex}} c_T \ .$$

Dann ist T auf A Turing-reduzierbar. □

Folgerung 9.7. P/poly *ist der Abschluss der p-selektiven Sprachen unter Polynomialzeit-Turing-Reduktion.*

Für die Klasse der leicht 2-zählbarenSprachen gilt ein solche Satz zum Beipiel nicht. Das kann man beweisen, indem man zeigt, dass leicht 2-zählbare Sprachen immer entscheidbar sind, P/poly dagegen auch nicht-entscheidbare Sprachen enthält.

9.4 Selbstreduzierbarkeit und Teilinformation

Definition 9.8 (Selbstreduzierbarkeit)**.** Eine Sprache A ist selbstreduzierbar, wenn eine polynomiell zeitbeschränkte Orakel-TM M existiert, die bei Eingabe w für jedes Orakel X nur Fragen q stellt mit $|q| \leqslant |w|$, so dass $A = L(M, A)$.

Sprachen in P sind natürlich selbstreduzierbar, denn in diesem Fall werden garkeine Orakelanfragen gebraucht, um die Sprache zu entscheiden. Sehr viel weiter oben in der Hierarchie der „gewöhnlichen" Komplexitätsklassen können selbstreduzierbare Sprachen auch nicht liegen:

Satz 9.9. *Selbstreduzierbare Sprachen sind in* PSPACE.

Praktisch alle bekannten Sprachen in PSPACE sind selbstreduzierbar. Insbesondere auch die üblichen NP-vollständigen Sprachen inklusive SAT.

Satz 9.10. SAT *ist selbstreduzierbar in Truth-Table-Art mit zwei Fragen, wobei die Antworten disjunktiv ausgewertet werden.*

Beweis Für eine Formel φ sei φ_0 die Formel, die entsteht, wenn man alle Vorkommmen der Variablen mit der kleinsten Nummer durch False ersetzt; und φ_1 entstehe durch ersetzen dieser Variablen durch True. Dann ist $\varphi \in \text{SAT}$ gdw $\varphi_0 \in \text{SAT}$ oder $\varphi_1 \in \text{SAT}$; und φ_0 und φ_1 sind kürzer als φ. □

Wir benutzen diese Selbstreduzierbarkeit, um zu zeigen, dass SAT nicht p-selektiv ist (es sei denn P = NP):

Satz 9.11. *Ist* SAT *p-selektiv, so ist* SAT $\in$ P.

Beweis Sei SAT p-selektiv via $f \in \text{FP}$. Benutze die Selbstreduzierbarkeit von SAT. Es wird Schritt für Schritt eine immer kleinere Formel ψ berechnet, so dass $\varphi \in \text{SAT} \Leftrightarrow \psi \in \text{SAT}$. Ist ψ die bisher berechnete Formel mit dieser Eigenschaft, so bildet man ψ_0 und ψ_1 und berechnet $f(\psi_0, \psi_1)$. Liefert dies die Information „$\psi_0 \in \text{SAT} \Rightarrow \psi_1 \in \text{SAT}$", so gilt $\psi \in \text{SAT} \Leftrightarrow \psi_1 \in \text{SAT}$. Ersetze ψ durch ψ_1. Im umgekehrten Fall ersetze ψ durch ψ_0. Man wiederholt dies bis man eine variablenfreie Formel ψ bestimmt hat. Man vereinfacht ψ zu True oder zu False. Im ersten Fall akzeptiert man φ, im zweiten Fall verwirfst man φ. □

Mit einer etwas trickreicheren Beweismethode lässt sich sogar der folgende Satz [BKS95] beweisen:

Satz 9.12. *Ist* SAT *m-approximierbar für ein $m \geqslant 2$, so ist* SAT $\in$ P.

10 Polynomielle Hierarchie

10.1 Definition und Eigenschaften

Man kann nicht nur deterministische Turingmaschinen mit der Möglichkeit ausstatten, Fragen an ein Orakel zu stellen. Auch bei nichtdeterministischen oder probabilistischen Maschinen kann man dies tun. Manche Komplexitätsklassen sind sehr robust unter solchen Operationen. Z. B. gilt

- $\mathrm{P}^{\mathrm{P}} = \mathrm{P}$.
- $\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP}$.
- $\mathrm{PSPACE}^{\mathrm{PSPACE}} = \mathrm{PSPACE}$.

(Das Ergebnis für PSPACE setzt voraus, dass auch das Orakelband der polynomielle Platzschranke unterliegt.) Für NP ist unbekannt, ob $\mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}$ gilt. Das hat auch eine positive Seite, denn dadurch kann man manche Probleme, die vermutlich nicht in NP sind, genauer klassifizieren, indem man sie in eine passende Stufe der *Polynomiellen Hierarchie* einordnet. Die Hierarchie wurde 1976 von Stockmeyer [Sto76] eingeführt.

Definition 10.1 (Polynomielle Hierarchie)**.**

1. $\Delta_0^p = \Sigma_0^p = \Pi_0^p := \mathrm{P}$.
2. Für $k \geqslant 0$: $\Delta_{k+1}^p := \mathrm{P}^{\Sigma_k^p}$; $\Sigma_{k+1}^p := \mathrm{NP}^{\Sigma_k^p}$; $\Pi_{k+1}^p := \text{co-}\Sigma_{k+1}^p$.
3. $\mathrm{PH} := \bigcup_{k\in\mathbb{N}} \Sigma_k^p$.

Wenn man neue Komplexitätsklassen einführt, untersucht man als erstes, ob gewisse Abschlusseigenschaften gelten:

Lemma 10.2 (Abschlusseigenschaften)**.**

1. *Für jedes $k \in \mathbb{N}$:*
   Δ_k^p, Σ_k^p und Π_k^p sind unter Join (disjunkter Vereinigung), Schnitt und Vereinigung abgeschlossen.
2. *Für jedes $k \in \mathbb{N}$:*
   Δ_k^p, Σ_k^p und Π_k^p sind unter $\leqslant_m^{\log}$-Reduktion abgeschlossen.
3. *Für jedes $k \in \mathbb{N}$:*
   Δ_k^p ist unter $\leqslant_T^p$-Reduktion abgeschlossen.
4. *Für jedes $k \in \mathbb{N}$:*
   Σ_k^p und Π_k^p sind unter disjunktiv ausgewerteter $\leqslant_{tt}^p$-Reduktion abgeschlossen; ebenso unter konjunktiv ausgewerteter $\leqslant_{tt}^p$-Reduktion .

Beweis Zu 1.: Wir zeigen den Abschluss unter Join durch Induktion über k. P ist unter Join abgeschlossen. Sei $A_0 \in \Sigma_{k+1}^p$ via einer OTM M_0 und einem Orakel $B_0 \in \Sigma_k^p$; sei $A_1 \in \Sigma_{k+1}^p$ via einer OTM M_1 und einem Orakel $B_1 \in \Sigma_k^p$. Eine Orakel-Maschine M für $A_0 \oplus A_1$ verwendet ein Orakel für $B_0 \oplus B_1$ und arbeitet wie folgt: Bei Eingabe x wird zunächst geprüft, ob $x = u0$ oder $x = u1$ für ein u. Im ersten Fall ist $x \in A_0 \oplus A_1$ gdw $u \in A_0$. Deshalb wird M_0 mit Orakel B_0 bei Eingabe u simuliert. Fragen an B_0 werden allerdings an die entsprechenden Fragen an $B_0 \oplus B_1$ ersetzt. Der Fall $x = u1$ wird analog behandelt. Das verwendete Orakel $B_0 \oplus B_1$ ist laut Induktionsvoraussetzung in Σ_k^p.

Der Abschluss und Vereinigung und Schnitt wird ähnlich durch Induktion über k gezeigt, und zwar unter Verwendung der Abgeschlossenheit unter Join.

Zu 2.: Wird gezeigt wie die Abgeschlossenheit von P, NP und co-NP unter $\leqslant_{\mathrm{m}}^{\log}$-Reduktion. Dass die Maschinen auch noch Zugriff auf ein Orakel haben, ändert nichts an der Beweismethode.

Zu 3.: Wird gezeigt wie die Abgeschlossenheit von P unter $\leqslant_{\mathrm{T}}^{\mathrm{p}}$-Reduktion. Dass die Maschine auch noch Zugriff auf ein Orakel hat, ändert nichts an der Beweismethode.

Zu 4.: Wird gezeigt wie die Abgeschlossenheit von NP und co-NP unter $\leqslant_{\mathrm{dtt}}^{\mathrm{p}}$-Reduktion (disjunktiver Truth-Table-Reduktion) und $\leqslant_{\mathrm{ctt}}^{\mathrm{p}}$-Reduktion (konjunktiver Truth-Table-Reduktion). Dass die Maschine auch noch Zugriff auf ein Orakel hat, ändert nichts an der Beweismethode.

Die Abgeschlossenheit von NP unter $\leqslant_{\mathrm{dtt}}^{\mathrm{p}}$-Reduktion zeigt man zum Beispiel wie folgt: Sei $A \leqslant_{\mathrm{dtt}}^{\mathrm{p}} B$ via M_{red} und $B \in \mathrm{NP}$ via M_B. Eine NTM M_A für A arbeitet bei Eingabe x wie folgt: Zunächst wird M_{red} bei Eingabe x simuliert. Seien $q_1, \dots, q_l$ die Fragen, die M_{red} erzeugt. Nun wird nacheinander M_B auf jedem q_i simuliert. Wird bei einer der Simulationen q_i von M_B akzeptiert, so akzeptiert M_A die Eingabe x. □

Als nächstes betrachtet man das Verhältnis zu bekannten Klassen. Auch über das Verhältnis von PH zu den Klassen BPP, PP und P/poly ist einiges bekannt, dazu später mehr, aber am naheliegendsten ist es, das Verhältnis zu PSPACE zu betrachten.

Satz 10.3. PH $\subseteq$ PSPACE.

Beweis Man zeigt durch Induktion über k, dass Σ_k^p in PSPACE ist.

Induktionsanfang. $\Sigma_0^p = \mathrm{P}$ ist in PSPACE enthalten.

Induktionsschritt. Nach Voraussetzung sei $\Sigma_k^p \subseteq \mathrm{PSPACE}$. Ist $A \in \Sigma_{k+1}^p$, so existiert eine polynomiell zeitbeschränkte nichtdeterministische OTM M_A und ein Orakel $B \in \Sigma_k^p$ mit $A = L(M_A, B)$. Nach Voraussetzung ist B in PSPACE. Sei M_B eine polynomiell platzbeschränkte DTM, die B entscheidet. Ersetzt man in Berechnungen von M_A die Orakelfragen durch Berechnungen von M_B, so ist die entstehende nichtdeterministische Maschine insgesamt polynomiell platzbeschränkt. □

Wäre PH = PSPACE, so würde die Polynomielle Hierarchie kollabieren:

Satz 10.4. *Ist* PH = PSPACE, *so existiert ein* $k \in \mathbb{N}$ *mit* $\Sigma_k^p = \mathrm{PSPACE}$.

Beweis Übungsaufgabe.

Ist es eigentlich möglich, dass einzelne Stufen der Polynomiellen Hierarchie zusammenfallen, es aber trotzdem weiter oberhalb noch echt verschiedene Stufen gibt? Die Antwort lautet nein:

Satz 10.5. *Ist für ein* k Σ_k^p

10.2 Quantorencharakterisierung

Man kann die Stufen der Polynomiellen Hierarchie mithilfe von Formeln mit alternierenden längenbeschränkten Quantoren charakterisieren. Diese Charakterisierung erleichtert es wesentlich, konkrete Entscheidungsprobleme in die passende Stufe der Hierarchie einzuordnen.

Definition 10.6. Formeln mit polynomieller Längenschranke und die Operatoren $\exists$ und $\forall$ auf Sprachklassen werden eingeführt.

1. Für jede Relation $R \subseteq \Sigma^* \times \Sigma^*$ und jedes Polynom p erfülle ein Wort x die Formel

   $$\exists^p y.\, \langle x, y \rangle \in R$$

   genau dann, wenn ein y existiert mit $|y| \leqslant p(|x|)$, so dass $\langle x, y \rangle \in R$ gilt.

2. Analog für $\forall^p y.\, \langle x, y \rangle \in R$.

3. Für jede Klasse $\mathcal{C}$ ist die Klasse $\exists\mathcal{C}$ definiert wie folgt: $A \in \exists\mathcal{C}$ gdw ein Polynom p und ein $R \in \mathcal{C}$ existieren mit

   $$x \in A \Leftrightarrow \exists^p y.\, \langle x, y \rangle \in R\ .$$

4. Analog für die Klasse $\forall\mathcal{C}$.

Satz 10.7 (Quantorencharakterisierung von **NP**). *Es gilt* $\exists \cdot \mathrm{P} = \mathrm{NP}$.

Beweis „$\subseteq$" Sei $A \in \exists \cdot \mathrm{P}$. Sei p ein Polynom und $B \in \mathrm{P}$, so dass $x \in A \Leftrightarrow \exists^p y.\langle x, y\rangle \in B$. Sie $B = L(M_B)$ für eine deterministische polynomiell zeitbeschränkte Maschine M_B. Eine nichtdeterministische polynomiell zeitbeschränkte Maschine M_A für A arbeitet auf Eingabe x wie folgt: Sie rät ein y mit $|y| \leqslant p(|x|)$, simuliert M_B auf Eingabe $\langle x, y\rangle$ und akzeptiert genau dann, wenn M_B akzeptiert. Offensichtlich ist $A = L(M_A)$ und damit $A \in \mathrm{NP}$.

„$\supseteq$" Sei $A \in \mathrm{NP}$. Sei $A = L(M_A)$ für eine NTM M_A, deren Laufzeit durch das Polynom p beschränkt ist, und

$$B = \{\langle x, y\rangle \mid y \text{ kodiert einen akzeptierenden Berechnungspfad von } M_A \text{ auf Eingabe } x\} .$$

Offenbar ist $B \in \mathrm{P}$, und es gilt $x \in A$ gdw. es ein y mit $|y| \leqslant p(|x|)$ gibt mit $\langle x, y\rangle \in B$. Also ist $A \in \exists \cdot \mathrm{P}$. □

Im folgenden Lemma wird eine Rechenregel aufgestellt, wie der co-Operator mit $\exists$- und dem $\forall$-Operator vertauscht. Und es wird festgestellt, dass die Anwendung des $\exists$- oder des $\forall$-Operators auf eine Klasse in allen nicht-pathologischen Fällen diese Klasse zumindest nicht verkleinert.

Lemma 10.8.

1. *$A \in \exists \cdot \mathcal{C}$ gdw $\overline{A} \in \forall \cdot$ co-$\mathcal{C}$. D. h., dass co-$\exists \cdot \mathcal{C} = \forall \cdot$ co-$\mathcal{C}$.*
2. *Für jede Klasse $\mathcal{C}$ mit der Eigenschaft $A \in \mathcal{C} \Rightarrow \{\langle x, y\rangle \mid x \in A\} \in \mathcal{C}$ gilt $\mathcal{C} \subseteq \exists \cdot \mathcal{C}$ und $\mathcal{C} \subseteq \forall \cdot \mathcal{C}$.*

Mit diesem Lemma und Satz 10.7 folgt insbesondere $\forall \cdot \mathrm{P} = \text{co-}\mathrm{NP}$.

Satz 10.9. *Für alle $k \geqslant 1$ gilt:*

a) $\exists \cdot \Sigma_k^p = \Sigma_k^p$;
b) $\forall \cdot \Pi_k^p = \Pi_k^p$.

Beweis Es genügt wegen Lemma 10.8 die erste Gleichung zu zeigen. Die zweite Gleichung folgt dann durch Übergang zu den Komplement-Klassen. Die Inklusion $\Sigma_k^p \subseteq \exists \cdot \Sigma_k^p$ gilt wegen Lemma 10.8. Es bleibt zu zeigen, dass $\exists \cdot \Sigma_k^p \subseteq \Sigma_k^p$.

Sei $A \in \exists \cdot \Sigma_k^p$. Seien p und $B \in \Sigma_k^p$ so, dass $x \in A \Leftrightarrow \exists^p\, y.\langle x, y\rangle \in B$. Sei $B = L(M_B, D)$, wobei $D \in \Sigma_{k-1}^p$. Eine OTM M_A für A mit Orakel D arbeitet wie folgt: Bei Eingabe x rate ein y mit $|y| \leqslant p(|x|)$. Simuliere M_B mit Orakel D bei Eingabe $\langle x, y\rangle$. Falls M_B akzeptiert, akzeptiere, sonst verwerfe. Offensichtlich ist $A = L(M_A, D)$ und damit ist $A \in \Sigma_k^p$. □

Satz 10.10. *Für alle $k \geqslant 0$ gilt:*

a) $\exists \cdot \Pi_k^p = \Sigma_{k+1}^p$;
b) $\forall \cdot \Sigma_k^p = \Pi_{k+1}^p$.

Beweis Es genügt wegen Lemma 10.8 die erste Gleichung zu zeigen. Die zweite Gleichung folgt dann durch Übergang zu den Komplement-Klassen.

Beweis durch Induktion über k. Es genügt wegen Lemma 10.8 jeweils die Gleichung a) zu zeigen. Die Gleichung b) folgt dann durch Übergang zu den Komplement-Klassen.

Induktionsanfang. Dies ist die Aussage von Satz 10.7 über die Quantorencharakterisierung von NP.
Induktionsschritt. Die Gleichungen a) und und b) seien bis zur Stufe $k-1$ bereits erfüllt ($k \geqslant 1$). Wir müssen zwei Inklusionsrichtungen zeigen.

„$\subseteq$" Sei $A \in \exists \cdot \Pi_k^p$. Also existiert ein Polynom p und ein $B \in \Pi_k^p$ mit

$$x \in A \Leftrightarrow \exists^p y.\langle x, y\rangle \in B .$$

Wir geben eine Maschine M_A an, die A mit Orakel B entscheidet. Bei Eingabe x rät M_A ein y mit $|y| \leqslant p(|x|)$. Dann stellt M_A die Frage „$\langle x, y\rangle \in B$". Wenn ja, akzeptiere. Wenn nein, verwerfe. Die Maschine M_A belegt, dass $A \in \mathrm{NP}(\Pi_k^p) = \mathrm{NP}(\Sigma_k^p) = \Sigma_{k+1}^p$.

„$\supseteq$“ Sei $A \in \Sigma_{k+1}^p$. Es sei $A = L(M_A, B)$ mit $B \in \Sigma_k^p$. Wir suchen nun nach einer existenziell quantifizierten Formel, die A charakterisiert. In der Formel werden wir zu Sprachen S die abgeleitete Sprache S^* verwenden, die Tupel beliebiger Länge von Worten aus S enthält:

$$S^* := \{\langle z_1, \ldots, z_l\rangle \mid \text{ für jedes } i \text{ ist } z_i \in S\}$$

Man beachte, dass wenn S aus Σ_k^p (bzw. aus Π_k^p) ist, wegen Lemma 10.2 auch S^* aus Σ_k^p (bzw. aus Π_k^p) ist.

Es gilt nun

$x \in A \Leftrightarrow$ es existiert eine akzeptierende Berechnung von M_A bei Eingabe x, wobei Orakelfragen bzgl. Orakel B beantwortet werden

$\Leftrightarrow$ $\exists^p s.$ sodass
1. s hat die Form $y\#z_1, \ldots, z_l\#w_1, \ldots, w_m$, und
2. y kodiert eine akzeptierende Berechnung von M_A bei Eingabe x, in der genau die Fragen in $\{z_1, \ldots, z_l, w_1, \ldots, w_m\}$ vorkommen und für jedes z_i wird mit Antwort „ja“ weitergerechnet und für jedes w_j wird mit Antwort „nein“ weitergerechnet, und
3. für jedes der z_i gilt $z_i \in B$, und
4. für jedes der w_j gilt $w_j \in \overline{B}$

Welche Komplexität haben die vier Bedingungen innerhalb des Existenzquantors? Das 1. und das 2. Prädikat sind deterministisch in polynomieller Zeit entscheidbar. Das 3. Prädikat ist äquivalent zu $\langle z_1, \ldots, z_l\rangle \in B^*$, also handelt es sich (leider) um ein Σ_k^p-Prädikat. Wir wenden deshalb hier die Induktionsvoraussetzung an: Es gibt ein Polynom q und ein $C \in \Pi_{k-1}^p$ derart, dass

$$z \in B^* \Leftrightarrow \exists^q t.\langle z, t\rangle \in C\ .$$

Da $\Pi_{k-1}^p \subseteq \Pi_k^p$, ist auch $C \in \Pi_k^p$. Das 4. Prädikat ist äquivalent zu $\langle w_1, \ldots, w_m\rangle \in \overline{B}^*$, also handelt es sich (wie erhofft) um ein Π_k^p-Prädikat.

Wir setzen nun die Teilformel für das 3. Prädikat in die Formel ein, ziehen den inneren Existenzquantor nach vorne und fassen die zwei Existenzquantoren zusammen. Das neue Schranken-Polynom $\tilde{p}$ entsteht durch geeignete Kombination von p und q. Es gilt

$x \in A \Leftrightarrow$ $\exists^p s.$ so dass
1. s hat die Form $y\#z_1, \ldots, z_l\#w_1, \ldots, w_m$ und
2. y kodiert eine akzeptierende Berechnung ... (wie oben) und
3. $\exists^q t.\langle z_1, \ldots, z_l, t\rangle \in C$ und
4. $\langle w_1, \ldots, w_m\rangle \in \overline{B}^*$

$\Leftrightarrow$ $\exists^{\tilde{p}} \tilde{s}.$ so dass
1. $\tilde{s}$ hat die Form $y\#z_1, \ldots, z_l\#t\#w_1, \ldots, w_m$ und
2. y kodiert eine akzeptierende Berechnung ... (wie oben) und
3. $\langle z_1, \ldots, z_l, t\rangle \in C$ und
4. $\langle w_1, \ldots, w_m\rangle \in \overline{B}^*$

Jede der vier Bedingungen ist nun eine Π_k^p-Prädikat, und da Π_k^p nach Lemma 10.2 unter Schnitt abgeschlossen ist, hat die Formel insgesamt die Form $\exists^{\tilde{p}} \tilde{s}.\ \langle x, \tilde{s}\rangle \in \tilde{B}$ mit $\tilde{B} \in \Pi_k^p$ und damit ist $A \in \exists \cdot \Pi_k^p$. □

Aus dem obigen Satz 10.10 folgt nun durch Induktion die Charakterisierung der Stufen der Polynomiellen Hierarchie:

Folgerung 10.11 (Quantorencharakterisierung der Polynomiellen Hierarchie)**.**
Für $n \geqslant 1$ ist eine Sprache A in Σ_k^p gdw ein $(k+1)$-stelliges Prädikat $R \in \mathrm{P}$ und Polynome $p_1, \ldots, p_n$ existieren, derart dass für alle x, $|x| = n$,

$$x \in A \Leftrightarrow \exists^{p_1} y_1\ \forall^{p_2} y_2\ \ldots Q^{p_k} y_k.\ (x, y_1, \ldots, y_k) \in R\ .$$

Dabei wechseln sich Existenz- und All-Quantoren ab. Der Quantor Q ist Existenzquantor gdw k gerade.

10.3 Polynomielle Hierarchie und probabilistische Klassen

Der folgende Satz ist nach der Einordnung in P/poly ein weiteres Indiz, dass Probleme in BPP nicht allzu schwierig sind. Manche Autoren vermuten sogar, dass sich in den nächsten Jahren wird beweisen lassen, dass BPP = P gilt. Der folgende Satz wurde 1983 von Lautemann [Lau83] bewiesen.

Satz 10.12. BPP $\subseteq \Sigma_2^p$.

Beweis Sei $A \in$ BPP. Sei M_A eine probabilistische Maschine, die A entscheidet. Das Polynom p sei eine Rechenzeitschranke für M_A. Nach Satz 8.6 dürfen wir annehmen, dass M_A exponentiell kleine Fehlerschranke einhält, etwa

$$\text{Prob}\left[M_A(x) \neq \chi_A(x)\right] \leqslant \frac{1}{2^n} \ .$$

Wir versuchen jetzt Zugehörigkeit zu A in der Form $x \in A \Leftrightarrow \exists y \forall z. \langle x, y, z \rangle \in B$ zu beschreiben, wobei die Quantoren noch polynomiell längenbeschränkt sein müssen und B in P liegen muss.

Sei nun x ein Eingabewort mit $|x| = n$. Wir betrachten das Verhalten von M_A für die $2^{p(n)}$ vielen Random Strings der Länge $p(n)$. Wir führen eine Bezeichnung ein für die Menge der zur Akzeptanz von x führenden Zufallsstrings:

$$W_x := \{y \mid M_A \text{ akzeptiert } x \text{ mit Zufallstring } y\}$$

Damit gilt

$$x \in A \Leftrightarrow |W_x| \geqslant \left(1 - \frac{1}{2^n}\right) 2^{p(n)}$$

$$x \notin A \Leftrightarrow |W_x| \leqslant \frac{1}{2^n} 2^{p(n)} \ .$$

Die gesuchte Formel muss also eine Eigenschaft beschreiben, die sehr große Teilmengen von $\Sigma^{p(n)}$ erfüllen, sehr kleine aber nicht.

Wir betrachten die bitweise Addition modulo 2 (mit anderen Worten: die X-Or-Verknüpfung) auf Bitstrings $(a_1 \ldots a_m) \oplus (b_1 \ldots b_m) = (a_1 \oplus b_1) \ldots (a_m \oplus b_m)$. Wir halten zwei Eigenschaften fest: Für festes $b \in \{0,1\}^m$ ist die durch $a \mapsto a \oplus b$ definierte Abbildung eine Bijektion auf $\{0,1\}^m$. Es ist außerdem $a \oplus b = c \Leftrightarrow a = b \oplus c$.

Seien $t_1, \ldots, t_s \in \{0,1\}^{p(n)}$. Wir sagen, $(t_1, \ldots, t_s)$ überdeckt mit W_x ganz $\{0,1\}^{p(n)}$, falls für alle $b \in \{0,1\}^{p(n)}$ ein j und ein $y \in W_x$ existieren mit $y \oplus t_j = b$. Anders formuliert: $(t_1, \ldots, t_s)$ überdeckt mit W_x ganz $\{0,1\}^{p(n)}$, falls für alle $b \in \{0,1\}^{p(n)}$ ein j existiert mit $b \oplus t_j \in W_x$. Wenn man s richtig wählt, kann man erreichen, dass man mit sehr kleinem W_x auf keinen Fall ganz $\{0,1\}^{p(n)}$ überdecken kann, bei geschickter Wahl der t_i aber mit jedem sehr großen W_x die Menge $\{0,1\}^{p(n)}$ überdeckt. Wir wählen $s = p(n)$ und stellen die folgende Behauptung auf. Für alle x ab einer gewissen Mindestlänge gilt:

$$x \in A \Leftrightarrow \exists t_1, \ldots, t_{p(n)} \in \{0,1\}^{p(n)}. \forall b \in \{0,1\}^{p(n)}. \bigvee_{j=1}^{p(n)} (b \oplus t_j \in W_x) \qquad \textit{(Behauptung)}$$

Beweis der Behauptung: Betrachten wir zunächst den Fall $x \in A$. Für ein gegebenes b ist die Wahrscheinlichkeit (unter Gleichverteilung)

$$\text{Prob}_{t \in \{0,1\}^{p(n)}}\left[b \oplus t \notin W_x\right] \leqslant \frac{1}{2^n} \ .$$

Nun schätzen wir die Wahrscheinlichkeit für eine ganze Folge von t_i ab, dass W_x nie getroffen wird:

$$\text{Prob}_{t_1, \ldots, t_{p(n)}}\left[\bigwedge_{i=1}^{p(n)} (b \oplus t_i \notin W_x)\right] < \left(\frac{1}{2^n}\right)^{p(n)} = 2^{-np(n)} \ .$$

Dann ist

$$\text{Prob}_{t_1, \ldots, t_{p(n)}}\left[\exists b \in \{0,1\}^{p(n)}. \bigwedge_{i=1}^{p(n)} (b \oplus t_i \notin W_x)\right] \leqslant 2^{p(n)} 2^{-np(n)} = 2^{(-n+1)p(n)} \ .$$

Insbesondere ist für $n \geqslant 2$ diese Wahrscheinlichkeit < 1, und damit die Wahrscheinlichkeit für das entgegengesetzte Ereignis > 0:

$$\mathrm{Prob}_{t_1,\ldots,t_{p(n)}}\left[\forall b \in \{0,1\}^{p(n)}.\ \bigvee_{i=1}^{p(n)} (b \oplus t_i \in W_x)\right] > 0\ .$$

Also existiert eine Folge $t_1, \ldots, t_{p(n)}$, die mit W_x ganz $\{0,1\}^{p(n)}$ überdeckt.

Betrachten wir nun den Fall $x \notin A$. Dann ist $|W_x| \leqslant 2^{-n}2^{p(n)}$. Überdeckt werden von $t_1, \ldots, t_{p(n)}$ mit W_x also höchstens $2^{-n}2^{p(n)}p(n)$. Für genügend großes n ist dies echt kleiner als $2^{p(n)}$; es werden also nicht alle Strings überdeckt. Damit ist die *Behauptung* gezeigt.

Die in der Behauptung angegebene Formel ist eine $\exists\forall$-Formel, die gebundenen Variablen sind polynomiell längenbeschränkt, und die Eigenschaft im Inneren der Formel ist in Polynomialzeit entscheidbar. Nach Satz 10.10 ist A also in Σ_2^p. $\square$

Folgerung 10.13. *Ist* NP = P*, so ist auch* BPP = P*.*

Die Klasse BPP liegt also gemessen an der Polynomiellen Hierarchie „weit unten", die probabilistische Klasse PP liegt dagegen gemessen an PH „ganz oben". Der direkte Vergleich wird erschwert dadurch, dass man nicht weiß, ob PP unter Turing-Reduktion abgeschlossen ist, deshalb vergleicht man PH mit dem Abschluss von PP unter dieser Reduktion.

Satz 10.14 (Satz von Toda)**.** PH $\subseteq \mathrm{R}_T^{\mathrm{p}}(\mathrm{PP})$.

Beweis: Lang, kompliziert und ausgesprochen clever. Nachzulesen in Originalpapier [Tod91].

10.4 Polynomielle Hierarchie und P/poly

In Kapitel 9 haben wir gesehen, dass aus der P-Selektivität von Sat NP = P folgen würde. Sprachen in P/poly sind ja auf p-selektive Sprachen Turing-reduzierbar. Reicht vielleicht schon die Annahme, dass Sat in P/poly liegt, um NP = P zu folgern? Dies zu beweisen, ist noch nicht gelungen. Karp und Lipton [KL80] konnten aber immerhin die folgende Implikation beweisen:

Satz 10.15. *Wenn* Sat $\in$ P/poly*, dann ist* PH $= \Sigma_2^p$*.*

Der Beweis lässt sich auch in [Pap94] nachlesen.

Der Kollaps der Polynomiellen Hierarchie auf die nullte Stufe konnte also aus Sat $\in$ P/poly bisher nicht gefolgert werden, aber immerhin der Kollaps auf die zweite Stufe.

11 Beweismethoden

Im Folgenden wollen wir einige der behandelten Ergebnisse noch einmal Revue passieren lassen. Der Blickwinkel ist dabei aber ein anderer, da wir diesmal nicht nacheinander die verschiedenen Berechnungsmodelle abhandeln, sondern entlang der Methoden vorgehen, die zur Verfügung stehen, um Fragen im Bereich der Komplexität von Berechnungsproblemen anzugehen.

Die Ausgangsfrage ist folgende: Wie groß ist – für ein gegebenes Problem A, einen gewissen Maschinentyp und eine gewisse Ressourcenart – der zur Lösung von A benötigte Ressourcenverbrauch höchstens und mindestens? Wir fragen also nach *oberen* und nach *unteren* Schranken für den Berechnungsaufwand. Wenden wir uns zunächst den oberen Schranken zu.

11.1 Obere Schranken

Wie zeigt man obere Schranken? Wie zeigen wir also für eine Sprache A und eine Komplexitätsklasse $\mathcal{C}$, dass $A \in \mathcal{C}$ gilt? Verschiedene Lösungsansätze bieten sich an:

1. **Algorithmus angeben.** Man gibt direkt einen Algorithmus an, der A entscheidet. Man schätzt den Ressourcenverbrauch des Algorithmus ab.
2. **Inklusion ausnutzen.** Man zeigt für eine andere, eventuell leichter zu handhabende Klasse $\mathcal{C}'$, dass $A \in \mathcal{C}'$. Dann benutzt man die Inklusion $\mathcal{C}' \subseteq \mathcal{C}$ und folgert $A \in \mathcal{C}$. Beispiele:
    - NL $\subseteq$ NC. Man fragt sich, ob es einen schnellen parallelen Algorithmus für A gibt. Es mag zunächst einfacher sein, eine nichtdeterministische Maschine anzugeben, die A entscheidet und mit wenig Speicherplatz auskommt. Dann verwendet man die Inklusion NL $\subseteq$ NC.
    - NPSPACE $\subseteq$ PSPACE. Man fragt sich, ob ein (deterministischer) Algorithmus mit polynomiellem Platzbedarf für ein Problem existiert. Oft ist es leichter, zunächst nach einem nichtdeterministischen Algorithmus zu suchen und dann den Satz von Savitch, Satz 3.4 anzuwenden.
3. **Reduktion angeben.** Man gibt eine Reduktion von A nach B an für ein B bekannter Komplexität. Dann schließt man aus der Komplexität von B und dem Aufwand der Reduktion auf die Komplexität von A. Ein Spezialfall ist besonders interessant: Ist $B \in \mathcal{C}$ und ist $\mathcal{C}$ unter der Reduktion abgeschlossen, so ist auch $A \in \mathcal{C}$. Nicht immer gelingt es, eine Many-One-Reduktion zu finden. Dann können auch Truth-Table- oder Turing-Reduktionen von Nutzen sein.
4. **Syntaktische Beschreibung angeben.** Man schließt aus der Komplexität einer syntaktischen Beschreibung des Problems auf seine Berechnungskomplexität. (Dieser Ansatz wird in diesem Skript wenig, nämlich nur im Zusammenhang mit der Charakterisierung der Polynomiellen Hierarchie, behandelt.) Varianten:
    (a) **Beschreibung durch eine Grammatik.** Ist etwa A durch eine kontextfreie Grammatik beschreibbar, so ist A in P (und sogar in NC_2). Die Klasse NSPACE(n) enthält genau die Sprachen, die durch eine kontextsensitive Grammatik beschreibbar sind.
    (b) **Beschreibung durch eine prädikatenlogische Formel.** Instanzen wie Graphen, Hypergraphen, Wörter, oder Zahlenfolgen kann man als Strukturen zu einer passenden prädikatenlogischen Syntax auffassen. Gerichtete Graphen $G = (V, E)$ kann man etwa als eine Struktur auf"fassen über einer Signatur, die ein zweistelliges Relationssymbol E enthält. Zu einer Sprache wie zum Beispiel einer Menge von Graphen mit einer bestimmten Eigenschaft kann man dann in vielen Fällen eine prädikatenlogische Formel ϕ angeben, deren Modellklasse gerade diese Sprache ist.

    Für Formeln ϕ in Prädikatenlogik erster Stufe sind die zugehörigen Sprachen (also die Modellklassen) immer in L. Der Satz von Fagin [Fag74] besagt, dass in NP genau diejenigen Probleme sind, die durch eine zweitstufige Formel beschreibbar sind, bei der zweitstufige Variablen nur durch Existenzquantoren gebunden sind.

    Ähnliche Charakterisierungen gibt es auch für andere Komplexitätsklassen wie z. B. P und NL. Für Informationen zu diesem Thema schaue man in das Buch von Immerman [Imm98].

(c) **Beschreibung durch polynomiell längenbeschränkte Formeln.** Bei der Charakterisierung der Polynomiellen Hierarchie, Folgerung 10.11, werden auch Formeln verwendet. Quantifiziert wird allerdings über längenbeschränkte Worte und im Inneren der Formel steht ein Prädikat, dessen Zugehörigkeit zu P nachgewiesen werden muss. Es handelt sich also bei dieser Charakterisierung um eine Mischform aus syntaktischen und klassisch maschinenbezogenen Bestandteilen.

11.2 Untere Schranken

Untere Schranken für die Komplexität von Sprachen (oder anderen Berechnungsproblemen) sind in der Regel schwerer zu finden als obere Schranken. Wenn man für eine Sprache A und eine Klasse $\mathcal{C}$ zeigen will, dass $A \notin \mathcal{C}$ gilt, bieten sich wieder mehrere Vorgehensweisen an:

1. **Reduktion und Vollständigkeit ausnutzen.** Man gehe wie folgt vor: Man sucht eine echte Oberklasse $\mathcal{C}'$ von $\mathcal{C}$. Man zeigt, dass $\mathcal{C}$ unter einer Reduktion $\leqslant_{\mathrm{r}}$ abgeschlossen ist. Man findet ein vollständiges Problem A' in $\mathcal{C}'$. Man zeigt, dass $A' \leqslant_{\mathrm{r}} A$. Dann ist $A \notin \mathcal{C}$, denn sonst wäre auch A in $\mathcal{C}$ und damit $\mathcal{C}' \subseteq \mathcal{C}$.

   Das Verfahren funktioniert natürlich auch, wenn man direkt für die Sprache A zeigt, dass sie vollständig ist für $\mathcal{C}'$. Man kann dass Verfahren auch so modifizieren, dass man auf andere Weise (nicht über Vollständigkeit für eine Klasse $\mathcal{C}'$) zeigt, dass A' nicht in $\mathcal{C}$ liegt.

2. **Trennende Eigenschaft finden.** Gib eine Eigenschaft aller Sprachen in $\mathcal{C}$ an. Zeige, dass A diese Eigenschaft nicht hat. So geht man beispielsweise vor, wenn man mithilfe von Pumping-Lemmata zeigt, dass Sprachen nicht in REG oder nicht in CFL sind.

3. **Informationstheoretisch argumentieren.** Zeige direkt am Maschinenmodell, dass es für jede $\mathcal{C}$-Maschine M mindestens zwei Eingaben $x \in A$ und $y \notin A$ geben muss, für die M das gleiche Akzeptanzverhalten zeigt. Die Argumentation ist bei dieser Methode oft informationstheoretischer Art. Man zeigt zum Beispiel, dass M wegen mangelnder Ressourcen nicht genug Informationen über einen Engpass (*Bottleneck*) schaffen kann, um x und y zu unterscheiden. Mit dieser Technik kann man zum Beispiel zeigen, dass die Sprache $A = \{ww \mid w \in \Sigma^*\}$ nicht in $\mathrm{DSPACE}(\log\log n)$ liegt, oder dass die Palindrome nicht von einer Einband-Turingmaschine in Zeit $o(n^2)$ erkannt werden können.

Wenn man keine absoluten unteren Schranken für die Komplexität von A zeigen kann, muss man sich mit schwächeren Resultaten zufrieden geben. Oft kann man untere Schranken zeigen unter der Voraussetzung, dass zwei Komplexitätsklassen nicht zusammenfallen. Die Aussage „falls $\mathrm{P} \neq \mathrm{NP}$, so ist $A \notin \mathrm{P}$“ trifft zum Beispiel auf alle NP-vollständigen Sprachen A zu.

Auch eine Aussage der Art „A ist mindestens so schwer wie B”, also zum Beispiel $B \leqslant_{\mathrm{m}}^{\log} A$, kann man als eine Art unterer Schranke auf”fassen.

11.3 Inklusionen von Klassen

Sowohl um obere als auch um untere Schranken zu zeigen, ist es wichtig, die Inklusionsbeziehungen zwischen Klassen zu kennen. Für zwei gegebene Klassen $\mathcal{C}$ und $\mathcal{C}'$ möchte man also einerseits wissen, ob vielleicht $\mathcal{C} \subseteq \mathcal{C}'$, andererseits, ob vielleicht $\mathcal{C} \neq \mathcal{C}'$. Eine Inklusion $\mathcal{C} \subseteq \mathcal{C}'$ lässt sich auf verschiedene Arten zeigen; zum Beispiel so:

1. **Ressourcenschranken erhöhen.** Für ein festes Maschinenmodell vergrößert sich die Klasse durch eine Erhöhung der Ressourcenschranke. Zum Beispiel ist $\mathrm{DTIME}(t) \subseteq \mathrm{DTIME}(t')$, falls $t(n) \leq t'(n)$ für alle n.

2. **Simulation.** Man simuliert die Maschine M, die belegt, dass $A \in \mathcal{C}$ ist, durch eine Maschine M' von dem Typ, über den die Klasse $\mathcal{C}'$ definiert ist, und schätzt den Aufwand von M' ab. Viele Varianten dieses Prinzips können auftreten:

- M und M' sind die gleiche (oder fast die gleiche) Maschine. Beispiel: L $\subseteq$ NL
- Das Akzeptanzverhalten von M' ist gegenüber dem von M geändert. Beispiel: P $\subseteq$ co-P.
- Alle Pfade einer nichtdeterministischen Maschine M werden nacheinander von M' simuliert. Beispiel: NP $\subseteq$ PSPACE.
- Parallele Rechnungen werden sequenzialisiert. Ein Schritt vieler Prozessoren von M wird von M' für jeden Prozessoren einzeln nacheinander ausgeführt. Beispiel: NC $\subseteq$ P.
- Ein Schritt einer deterministischen Turingmaschine M wird durch eine Schicht von Schaltkreisgattern simuliert. Beispiel: P $\subseteq$ POLYSIZE oder BPP $\subseteq$ POLYSIZE.

3. **Konfigurationsgraph untersuchen.** Eine nichtdeterministische Maschine akzeptiert ein Wort genau dann, wenn es einen Pfad von der Anfangskonfiguration zu einer akzeptierenden Endkonfiguration gibt. Dies wird benutzt beim Beweis von

   - NSPACE$(s) \subseteq$ DSPACE(s^2) (Satz von Savitch).
     Anwendung: PSPACE = NPSPACE.
   - NSPACE$(s) \subseteq$ DTIME$(2^{O(s)})$.
     Anwendung: NL $\subseteq$ P.

4. **Vollständige Probleme einordnen.** Man zeigt, dass es bezüglich einer Reduktion, z.B. $\leqslant_{\mathrm{m}}^{\log}$, ein vollständiges Problem A in $\mathcal{C}$ gibt. Man zeigt, dass $\mathcal{C}'$ unter $\leqslant_{\mathrm{m}}^{\log}$ abgeschlossen ist, und dass A in $\mathcal{C}'$ liegt.

11.4 Klassen trennen

Das zwei Klassen $\mathcal{C}$ und $\mathcal{C}'$ ungleich sind, lässt sich ebenfalls auf verschiedene Arten zeigen:

1. **Diagonalisierung.** Das ist *die* grundlegende Methode zu Klassentrennung. Man konstruiert eine Sprache $A \in \mathcal{C}'$ mit $A \notin \mathcal{C}$, indem man alle potentiellen $\mathcal{C}$-Maschinen, die A akzeptieren könnten, austrickst. Das heißt, man baut eine Maschine M', $L(M') = A$, die jede $\mathcal{C}$-Maschine M auf jeweils einem passenden Wort w_M simuliert; und M' akzeptiert w_M genau dann, wenn M das Wort w_M verwirft.

   So wird zum Beispiel Satz 2.6, der Platzhierarchiesatz für deterministische Maschinen, bewiesen.

2. **Unterschiedliche Eigenschaften.** Man zeigt, dass sich $\mathcal{C}$ und $\mathcal{C}'$ bezüglich einer geeignet gewählten Eigenschaft unterscheiden. Dafür bietet sich zum Beispiel an:

   - $\mathcal{C}$ ist komplementabgeschlossen, $\mathcal{C}'$ nicht.
   - $\mathcal{C}$ ist abgeschlossen unter einer geeigneten Reduktion, $\mathcal{C}'$ nicht. Beispiel: E $\neq$ EXP.
   - $\mathcal{C}$ hat vollständige Probleme bezglich einer geeigneten Reduktion, $\mathcal{C}'$ nicht. Beispiel: P $\neq$ POLYLOG.

3. **Anzahlargumente.** Es gibt zum Beispiel überabzählbare Sprachen, aber nur abzählbar viele entscheidbare Sprachen. Also gibt es nichtentscheidbare Sprachen.

   Mit Anzahlargumenten kann man auch zeigen, dass nicht alle Sprachen in POLYSIZE sind. Es gibt nämlich 2^{2^n} viele Funktionen $f\colon \{0,1\}^n \to \{0,1\}$. Sei $g(n)$ eine Funktion, die stärker wächst als jedes Polynom, aber wesentlich schwächer als 2^n. Dann gibt es ab einem gewissen n_0 für jedes $n \geqslant n_0$ eine Funktion $f_n\colon \{0,1\}^n \to \{0,1\}$, für die mehr als $g(n)$ Gatter zur Berechnung gebraucht werden.

11.5 Bedingte Ergebnisse

Oft kann man nicht beweisen, ob zwei Klassen gleich oder verschieden sind. Insbesondere bei P und NP ist dies ja seit langem ungeklärt. Dann kann man manchmal aber immerhin Beziehungen zwischen den verschiedenen offenen Fragen herstellen. Besonders gerne stellt man, wenn möglich, eine Äquivalenz zur Aussage „P = NP“ her.

Um solche Sätze mit Bedingung zu beweisen, kann man wieder bereits genannte Techniken benutzen, zum Beispiel:

- Simulation,
- Abgeschlossenheit von Klassen oder
- Einordnung von vollständigen Problemen.

Beispiele für solche bedingten Aussagen sind

- Wenn PH eine echte Hierarchie bildet, so ist PH $\neq$ PSPACE.
- P = NP gdw alle Sprachen in NP p-selektiv sind.

Auch die echten Inklusionen innerhalb von OptNP gelten nur unter der Bedingung P $\neq$ NP. Man beweist die Echtheit, indem man zu einem NP-vollständigen Problem A ein passendes Optimierungsproblem F angibt, derart dass das Finden einer approximativen Lösung für Instanzen von F einen Entscheidungsalgorithmus für Instanzen von A liefert.

11.6 Weitere Methoden

Einige wichtige Beweisverfahren der Komplexitätstheorie, die auch in diesem Skript Verwendung finden, kommen in der bisherigen Auflistung nicht vor oder ihre Bedeutung kommt nicht genügend zur Geltung:

1. **Padding.** Die „künstliche“ Verlängerung von Eingabeworten wird zum Beipiel verwendet in Diagonalisierungsbeweisen wie im Beweis des Platzhierarchiesatzes. Durch das Padding der Eingabe wird in solchen Fällen gesichert, dass eine Turingmaschine, die andere Maschinen simuliert, für passenden Eingaben genug Platz oder Zeit hat, diese Simulation bis zu Ende durchzuführen.

   Auch Reduktionsfunktionen können ein Padding ausführen. Dies ist die Kernidee des Beweises, dass sich Sprachen in EXP auf Sprachen in E reduzieren lassen.

   Die Technik kann man auch verwenden, um bedingte Inklusionen von Klassen zu beweisen. Zum Beispiel wird die im Skript nicht behandelte Implikation „NP = P $\Rightarrow$ NE = E“ mit Padding gezeigt.

2. **Fehler drücken durch Iteration.** Sowohl für RP als auch BPP kann durch Wiederholen des Zufallsexperiments (Berechnung der Maschine für einen Zufallsstring) die Fehlerwahrscheinlichkeit exponentiell klein gemacht werden.

3. **Diagonalisierung.** Diagonalisierung wird nicht nur genutzt, um Sprachklassen zu trennen. Auch um zu zeigen, dass verschiedene Reduktionsarten unterschiedlich mächtig sind, wird die Technik eingesetzt. Beim Trennen von Klassen ist auch nicht immer das Austricksen einer Maschine nach ressourcenbeschränkter Simulation auf *einem* Wort die passende Methode. Dies sieht man etwa an der Diagonalisierung im Beweis der Echtheit der Hierarchie von Advice-Klassen bei steigendem Advice.

4. **Probabilistische Existenzbeweise.** Im Beweis, dass BPP $\subseteq \Sigma_2^p$ gilt, (Satz 10.12) wird benutzt, dass ein Tupel polynomieller Größe von Bitstrings existiert, die mit W_x die Menge aller Bitstrings überdecken. Dieses Tupel wird nicht explizit konstruiert, sondern es wird wahrscheinlichkeitstheoretisch argumentiert. Es wird gezeigt, dass bei zufälliger Wahl eines solchen Tupels die Wahrscheinlichkeit, eines mit den gewünschten Eigenschaften zu finden, größer als 0 ist. Also gibt es ein solches Tupel.

Dies ist ein einfaches Beispiel der sogenannnten Probabilistischen Methode. Da hier als Wahrscheinlichkeitsverteilung die Gleichverteilung gewählt wurde, kann man den Beweis auch noch als reines Häufigkeitsargument auffassen. In anderen Beweisen argumentiert man aber oft mit speziellen, dem Problem angepassten Wahrscheinlichkeitsverteilungen. Auf diese Art hat man in vielen Bereichen der Mathematik nicht-konstruktive Existenzbeweise führen können.

Ausführlich behandelt wird das Thema der probabilistischen Existenzbeweise in der Graphentheorie und Kombinatorik im Buch von Alon und Spencer [AS92].

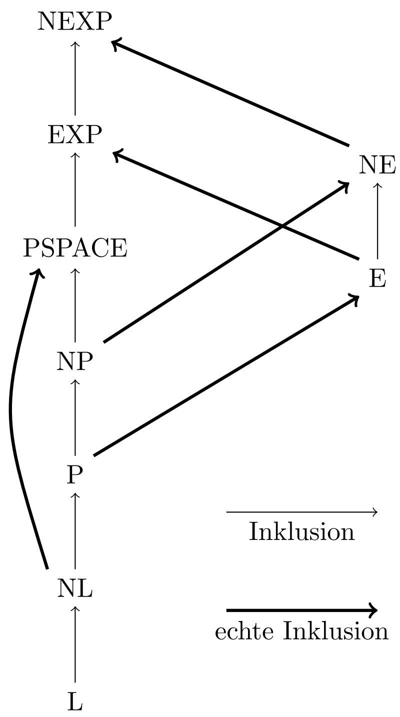
5. **Operatoren auf Sprachklassen.** Im Kapitel 10 wurden der $\exists$- und der $\forall$-Operator eingeführt. Ein weiterer Operator auf Klassen ist der „co“-Operator, der aus einer Klasse $\mathcal{C}$ die Klasse der Komplemente der Sprachen aus $\mathcal{C}$. Für diese Operatoren lassen sich Rechenregeln beweisen wie $\mathrm{co} \cdot \forall = \exists \cdot \mathrm{co}$ oder $\exists \cdot \exists = \exists$. Man kann auch andere Klassen wie z. B. BPP durch Anwendung eines Operators auf eine bereits vorliegende Klasse erzeugen. Sich so zeitweise von Maschinenmodellen zu lösen und mit Rechenregeln für Operatoren zu operieren, hat sich als sehr erfolgreich erwiesen. Der Beweis des Satzes von Toda, Satz 10.14, beruht wesentlich auf dieser Technik.

12 Inklusionsdiagramme

Wir stellen die Inklusionsbeziehungen verschiedener Komplexitätsklassen in Diagrammen dar.

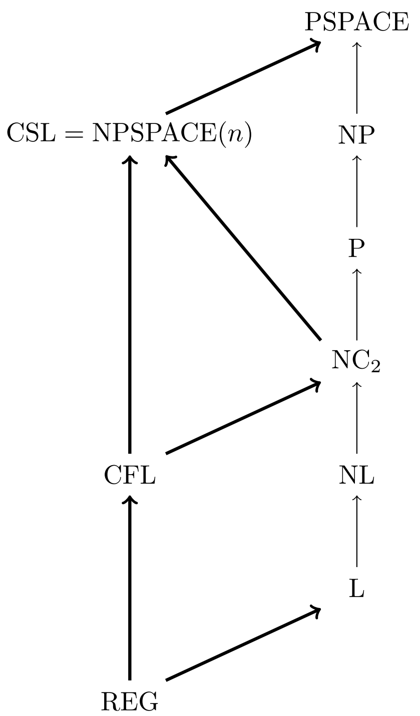
Das Diagramm rechts stellt die Beziehungen zwischen den gebräuchlichsten Zeit- und Platzklassen dar. Die Inklusionen folgen aus

- dem Übergang von deterministischen zu nichtdeterministischen Turingmaschinen,
- der Erhöhung der Ressourcenschranke,
- der Tatsache, dass jede beliebige Zeitschranke gleichzeitig eine Platzschranke darstellt, oder
- Satz 3.2, der deterministische Zeitschranken durch eine exponentielle Funktion bezüglich der nichtdeterministischen Platzschranken abschätzt.



Die Inklusion NL $\subseteq$ PSPACE ist echt wegen des Hierarchiesatzes für nichtdeterministische Platzklassen (und der Gleichheit PSPACE = NPSPACE).

Die Inklusionen E $\subseteq$ EXP und NE $\subseteq$ NEXP sind echt wegen der Zeithierarchiesätze. Man kann die Echtheit aber auch damit begründen, dass E und NE nicht unter polynomieller Many-One-Reduktion abgeschlossen sind, EXP und NEXP aber schon.

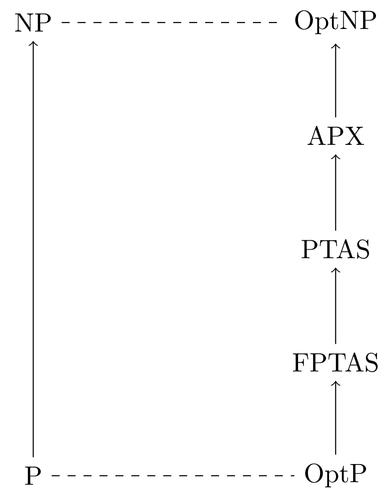


Das Diagramm zur Linken zeigt die Beziehungen zu den formalsprachlichen Klassen. REG bezeichnet die Klasse der regulären, CFL die der kontextfreien und CSL die Klasse der kontextsensitiven Sprachen. Die Echtheit der Inklusionen REG $\subseteq$ CFL und CFL $\subseteq$ CSL lässt sich mit sogenannten Pumping-Lemmata zeigen.

Die Inklusion CFL $\subseteq NC_2$ wurde von Ruzzo [Ruz81] bewiesen. Einen Beweis für die Inklusion CFL $\subseteq$ P kann man zum Beispiel in [Sch92] nachlesen. Die Inklusionen CFL $\subseteq NC_2$ und REG $\subseteq$ L sind echt, da zum Beispiel die Sprache $\{\, a^n b^n c^n \mid n \in \mathbb{N} \,\}$ in L ist, aber nicht einmal kontextfrei.

Da kontextsensitive Grammatiken keine verkürzenden Regeln enthalten, kann man Zugehörigkeit eines Wortes w zu einer kontextsensitiven Sprache feststellen, indem man eine Folge von Ableitungsregeln rät und testet, ob mit diesen Regeln w erzeugt wird. Überschreitet die Länge des abgeleiteten Wortes $|w|$, kann man die Berechnung abbrechen. Für den Beweis der umgekehrten Inklusion benutzt man, dass sich die Übergangsregeln einer Turingmaschine mithilfe von Grammatikregeln beschreiben lassen. Damit ist nach dem Satz von Immerman-Szelepcsényi, Satz 3.6, CSL unter Komplementbildung abgeschlossen. REG ist ebenfalls komplementabgeschlossen, CFL jedoch nicht.

Geht man von Entscheidungsproblemen zu Optimierungsproblemen über, so erhält man aus den Klassen P und NP die Klassen OptP und OptNP.
Durch Betrachtung von Approximationsalgorithmen bzw: -schemata verschiedener Qualität erhält man die nebenstehende Hierarchie.
Ist P $\neq$ NP, so ist auch jede Inklusion in der Kette OptP $\subseteq$ FPTAS $\subseteq$ PTAS $\subseteq$ APX $\subseteq$ OptNP echt. Es lassen sich jeweils die Klassen trennende Probleme angeben; siehe Abschnitt 7.5.

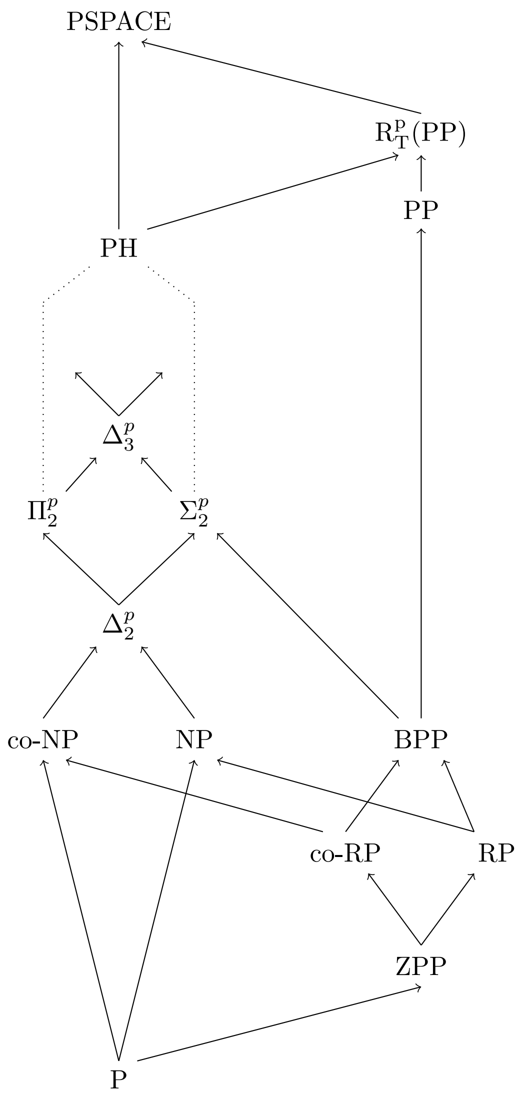


Rechts sind die Beziehungen für die probabilistischen Polynomialzeit-Klassen und die Polynomielle Hierarchie dargestellt. Es ist unbekannt, ob eine der Inklusionen echt ist, da nach heutigem Kenntnisstand sogar PSPACE = P möglich ist.

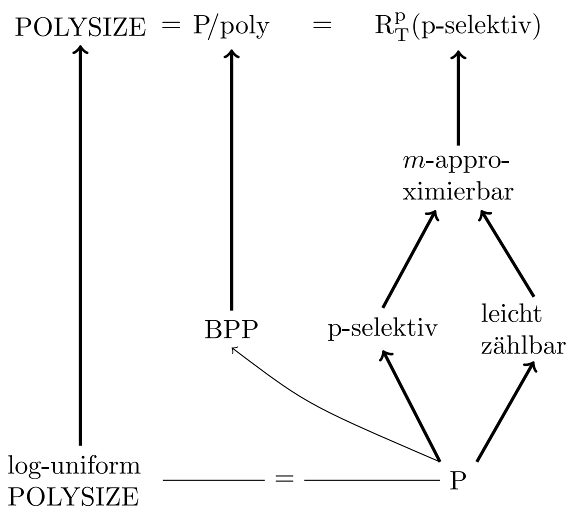
Für RP und NP (und die Σ_k^p-Klassen darüber) ist nicht bekannt, ob sie unter Komplementbildung abgeschlossen sind.

Alle diese Klassen sind unter polynomieller Many-One-Reduktion abgeschlossen. Die Klassen Δ_k^p, ZPP und BPP (und natürlich PSPACE) sind sogar unter $\leqslant_{\mathrm{T}}^{\mathrm{p}}$-Reduktion abgeschlossen.

Vollständige Probleme sind bekannt für PP (zum Beispiel MajSat), für PSPACE und für die Klassen in der Polynomiellen Hierarchie, aber nicht für BPP, RP und ZPP.



Dieses Diagramm setzt Advice-Klassen, Schaltkreisklassen und Teilinformationsklassen in Beziehung.
Sowohl P als auch P/poly, bei der die Polynomialzeitmaschine polynomiell viele Advice-Bits verwenden darf, lassen sich, siehe Satz 5.13 und Folgerung 5.11, durch Schaltkreise polynomieller Größe charakterisieren. Bei P muss man aber eine Uniformitätsbedingung an die Schaltkreisfamilie stellen.
P/poly ist außerdem der Turing-Abschluss der Klasse der p-selektiven Sprachen. Die verschiedenen Teilinformationsklassen bilden eine feinere Untergliederung von P/poly. Sprachen außerhalb von P können nicht gleichzeitig selbstreduzierbar sein und einen polynomiellen Teilinformationsalgorithmus haben.
Folgerung 8.22 besagt, dass BPP $\subseteq$ P/poly. Die Inklusion ist echt, da P/poly sogar unentscheidbare Sprachen enthält.



A Liste von Berechnungsproblemen

A.1 Worteigenschaften

[Word 1] PALINDROME

Eingabe: Ein Wort $w \subseteq \Sigma^*$
Frage: Ist w ein Palindrom, d. h. ist $w[i] = w[n-i]$ für alle $i \in \{1, \dots, n\}$, wobei $n = |w|$?

[Word 2] PARITY

Eingabe: Ein Wort $w \subseteq \{0,1\}^*$
Frage: Ist die Anzahl der 1en in w gerade, d. h. ist $\#_1(w) = 0 \mod 2$?

A.2 Maschinensimulation

[Mach 1] HALTINGPROBLEM Halteproblem (ohne Ressourcenschranke)

Eingabe: Ein Paar (m, w) von Worten, wobei m die Darstellung einer DTM M ist
Frage: Stoppt M bei Eingabe w ?

[Mach 2] NTM-ACCEPTANCE NTM-Simulation, unärer Zeitschranke

Eingabe: $\langle M, x, 1^t \rangle$, wobei M die Kodierung einer nichtdeterministischen online-Turingmaschine mit zwei Bändern ist mit Eingabealphabet $\Sigma = \{0,1\}$ und Arbeitsalphabet $\Gamma = \{0, 1, \square\}$, x eine über Σ kodierte Eingabe für M und $t \in \mathbb{N}$.
Frage: Akzeptiert M die Eingabe x auf mindestens einem Pfad in höchstens t Schritten?

Das Halteproblem Mach1 ist nicht entscheidbar, aber rekursiv aufzählbar. Das Problem Mach2 ist NP-vollständig. Es wird auch „das generische NP-vollständige Problem"genannt.

A.3 Aussagenlogische Formeln

Auswertung

[Form 1] Formelauswertung

Eingabe: Ein aussagenlogische Formel φ mit n Variablen und eine Bitfolge $b_1, \dots, b_n$
Frage: Ist φ wahr unter der Variablenbelegung $b_1, \dots, b_n$ (wobei 1 als TRUE und 0 als FALSE interpretiert wird)?

Formelauswertung ist offensichtlich in P und sogar in L..

Erfüllbarkeit

[Form 2] SAT

Eingabe: Eine aussagenlogische Formel ϕ
Frage: Existiert eine erfüllende Belegung für ϕ?

[Form 3] SAT als Konstruktionsproblem

Eingabe: Eine aussagenlogische Formel ϕ
Ausgabe: Eine erfüllende Belegung für ϕ

[Form 4] CNFSAT

Eingabe: Eine aussagenlogische Formel ϕ in konjunktiver Normalform
Frage: Existiert eine erfüllende Belegung für ϕ?

[Form 5] 3-SAT

Eingabe: Eine aussagenlogische Formel ϕ in konjunktiver Normalform mit genau drei verschiedenen Literalen pro Klausel

Frage: Existiert eine erfüllende Belegung für ϕ?

[Form 6] 2-SAT

Eingabe: Eine aussagenlogische Formel ϕ in konjunktiver Normalform mit genau zwei Literalen pro Klausel

Frage: Existiert eine erfüllende Belegung für ϕ?

[Form 7] MAJSAT

Eingabe: Eine aussagenlogische Formel ϕ

Frage: Wird ϕ von mehr als der Hälfte der möglichen Belegungen erfüllt?

Die Entscheidungsprobleme SAT, SATCNF und 3-SAT sind NP-vollständig. Das Problem 2-SAT ist co-NL-vollständig (und damit NL-vollständig). MAJSAT ist vollständig für PP.

A.4 Schaltkreise

Auswertung

[Circ 1] CVP Schaltkreisauswertung – Circuit Value Problem

Eingabe: Ein boolescher Schaltkreis C mit n Eingabegattern und einem Ausgabegatter und eine Bitfolge $b_1, \ldots, b_n$

Frage: Gibt C bei Eingabe $b_1, \ldots, b_n$ eine 1 aus?

CVP ist P-vollständig.

Erfüllbarkeit

[Circ 2] CIRCUITSAT

Eingabe: Eine boolescher Schaltkreis C mit n Eingabegattern und einem Ausgabegatter

Frage: Existiert ein Eingabe-Bitstring $b_1 \ldots b_n$, so dass C eine 1 ausgibt (also $C(b_1 \ldots b_n) = 1$)?

[Circ 3] $\exists\forall$CVP Schaltkreisauswertung – Circuit Value Problem

Eingabe: Ein boolescher Schaltkreis C mit $2m$ Eingabegattern $e_1, \ldots, e_{2m}$ und einem Ausgabegatter.

Frage: Gibt es eine Bitfolge $b_1, \ldots, b_m$, so dass für alle Bitfolgen $b_{m+1}, \ldots, b_{2m}$ gilt: $C(b_1, \ldots, b_{2m}) = 1$?

CIRCUITSAT ist NP-vollständig, $\exists\forall$CVP ist vollständig für Σ_2^p.

A.5 Graphen

Wege finden

[Graph 1] PATH Erreichbarkeit

Eingabe: Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
Frage: Existiert ein gerichteter Pfad von s nach t?

[Graph 2] PATH als Konstruktionsproblem

Eingabe: Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
Ausgabe: Ein gerichteter Pfad von s nach t

[Graph 3] UPATH Ungerichtete Erreichbarkeit

Eingabe: Ein ungerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
Frage: Existiert ein Pfad von s nach t?

[Graph 4] EULER

Eingabe: Ein ungerichteter Graph $G = (V, E)$ ($|V| = n$, $|E| = m$)
Frage: Gibt es einen Eulerkreis in G?
D.h. gibt es eine Permutation $e_1 = \langle u_1, v_1\rangle, \ldots, e_m = \langle u_m, v_m\rangle$ der Kanten in E, so dass $v_i = u_{i+1}$ für $1 \leqslant i < n$ und $v_m = u_1$?

[Graph 5] CONNECTED Zusammenhang

Eingabe: Ein ungerichteter Graph $G = (V, E)$
Frage: Gibt es für alle $u, v \in V$ einen Pfad von u nach v?

[Graph 6] STRONGLYCONNECTED Starker Zusammenhang

Eingabe: Ein gerichteter Graph $G = (V, E)$
Frage: Gibt es für alle $u, v \in V$ einen gerichteten Pfad von u nach v?

[Graph 7] HAMILTON

Eingabe: Ein Graph $G = (V, E)$ ($|V| = n$)
Frage: Gibt es einen Hamiltonkreis in G?
D.h. gibt es eine Permutation $v_1, \ldots, v_n$ der Knoten in V, so dass $\langle v_i, v_{i+1}\rangle \in E$ für $1 \leqslant i < n$ und $\langle v_n, v_1\rangle \in E$?

[Graph 8] TSP Travelling Salesperson Problem – Handlungsreisendenproblem

Eingabe: Ein Graph $G = (V, E)$, eine Gewichtsfunktion $w : E \to \mathbb{N}$ und eine natürliche Zahl k
Frage: Gibt es eine Rundreise mit Gewicht höchstens k? D.h. existiert eine Permutation $v_1, \ldots, v_n$ der Knoten in V, derart dass $\langle v_i, v_{i+1}\rangle \in E$ für $1 \leqslant i < n$ und $\langle v_n, v_1\rangle \in E$ und

$$w(\langle v_n, v_1\rangle) + \sum_{i=1}^{n-1} w(\langle v_i, v_{i+1}\rangle) \leqslant k?$$

Das Problem PATH (auch bekannt unter der Namen s-t-CON, GAP und REACH) ist NL-vollständig. Das Problem UPATH (auch bekannt unter den Namen s-t-UCON, UGAP und REACH_u) ist sogar in L. Ein Graph hat einen Eulerkreis, wenn er zusammenhängend ist und jeder Knoten geraden Grad hat. Das „Schwierige“ am Problem EULER ist also der Test, ob der Graph zusammenhängend ist. EULER liegt in NL und ist logspace-many-one-reduzierbar auf UPATH. Das Problem HDS ist P-vollständig. Die Entscheidungsprobleme HAMILTON und TSP sind NP-vollständig.

Knotenmengen maximieren

[Graph 6] IndependentSet

Eingabe: Ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$
Frage: Existiert eine unabhängige Menge $V' \subseteq V$ in G mit $|V'| \geqslant k$?

[Graph 7] IndependentSet als Konstruktionsproblem

Eingabe: Ein Graph $G = (V, E)$
Ausgabe: Eine unabhängige Menge $V' \subseteq V$ maximaler Größe (V' ist unabhängige Menge, wenn alle $v_1, v_2 \in V'$ nicht durch eine Kante aus E verbunden sind.)

[Graph 8] Clique

Eingabe: Ein Paar (G, k); G ein ungerichteter Graph, k eine natürliche Zahl (binär kodiert)
Frage: Enthält G eine Clique (einen vollständigen Teilgraphen) der Größe (Knotenzahl) k ?

[Graph 9] VertexCover

Eingabe: Ein Graph $G = (V, E)$ und eine natürliche Zahl $k \leqslant |V|$
Frage: Gibt es eine Teilmenge V' von V mit $|V'| \leqslant k$, so dass für jede Kante $(u, v) \in E$ mindestens einer der Knoten u und v in V' enthalten ist?

[Graph 10] DominatingSet

Eingabe: Ein Graph $G = (V, E)$ und eine natürliche Zahl $k \leqslant |V|$
Frage: Gibt es eine Teilmenge V' von V mit $|V'| \leqslant k$, so dass für jeden Knoten $v \in V$ gilt: $v \in V'$ oder es existiert ein $v' \in V'$ mit $(v, v') \in E$?

IndependentSet, Clique, VertexCover und DominatingSet sind NP-vollständig.

Weitere Graphprobleme

[Graph 1] HDS High Degree Subgraph

Eingabe: Ein ungerichteter Graph $G = (V, E)$, eine natürliche Zahl k
Frage: Enthält G einen induzierten Teilgraphen, in dem jeder Knoten einen Grad $\geqslant k$ hat ?

[Graph 2] CTQ Corporate Takeover Query, Firmenkontrolle

Eingabe: Eine Liste von Aktiengesellschaften $G_1, \ldots, G_n$; für jedes Paar i, j die Angabe, welchen Bruchteil der Aktien von Gesellschaft G_j die Firma G_i besitzt; zwei Indizes k und l.
Frage: Kontrolliert C_k die Gesellschaft C_l?

Dabei ist *kontrolliert* ist die kleinste Relation mit folgenden Eigenschaften:

(a) Jede Gesellschaft kontrolliert sich selbst.

(b) Kontrolliert G die Gesellschaften $G_{i_1}, \ldots, G_{i_r}$ und besitzen $G_{i_1}, \ldots, G_{i_r}$ zusammen mehr als die Hälfte der Aktien von G', so kontrolliert G auch G'.

Die Probleme HDS und CTQ sind P-vollständig.

A.6 Zahlentheorie und Arithmetik

Arithmetische Operationen

[Num 1] Addition

Eingabe: Zwei natürliche Zahlen m, n (binär kodiert)
Ausgabe: Die Summe $m + n$ (binär kodiert)

[Num 2] Addition einer Zahlenliste

Eingabe: Eine Liste natürlicher Zahlen $n_1, \ldots, n_l$ (n_i binär kodiert, l variabel)
Ausgabe: Die Summe $\sum_{i=1}^{l} n_i$ (binär kodiert)

[Num 3] Multiplikation

Eingabe: Zwei natürliche Zahlen m, n (binär kodiert)
Ausgabe: Das Produkt $m \cdot n$ (bin kodiert)

[Num 4] Multiplikation einer Zahlenliste

Eingabe: Eine Liste natürlicher Zahlen $n_1, \ldots, n_l$ (n_i binär kodiert, l variabel)
Ausgabe: Das Produkt $\prod_{i=1}^{l} n_i$ (binär kodiert)

[Num 5] Division

Eingabe: Zwei natürliche Zahlen m, n (binär kodiert), $n \neq 0$
Ausgabe: Der Quotient $\lceil \frac{m}{n} \rceil$ (binär kodiert)

Es ist leicht zu sehen, dass Addition, Addition einer Zahlenliste und Multiplikation in FL sind. Sie sind sogar mit Schaltkreisen logarithmischer Tiefe lösbar, also in FNC_1 Dass auch Multiplikation einer Liste von Zahlen und Division in FL und sogar in FNC_1 sind, wurde erst 1995 gezeigt [Chi95][CDL00][All01].

Matrizenrechnung

[Num 8] Matrizenmultiplikation

Eingabe: Zwei $n \times n$-Matrizen A, B (binär kodierte natürliche Zahlen als Einträge)
Ausgabe: Das Produkt $A \cdot B$

[Num 9] Berechnung von A^n für $n \times n$-Matrizen

Eingabe: Eine $n \times n$-Matrix A (binär kodierte natürliche Zahlen als Einträge)
Ausgabe: Die Matrix A^n (also A n-mal aufmultipliziert).

[Num 10] Det Determinante berechnen

Eingabe: Eine $n \times n$-Matrix A mit Einträgen aus $\mathbb{Q}$.
Ausgabe: Der Wert der Determinante von A.

[Num 11] SymbDet Symbolische nichtverschwindende Determinante

Eingabe: Eine $n \times n$-Matrix A, deren Einträge Polynome über $\mathbb{Q}$ in mehreren Variablen sind. Alle vorkommenden Exponenten sind höchstens n.
Frage: Ist $\det A \not\equiv 0$?

Matrixmultiplikation und Berechnung von A^n für eine $n \times n$-Matrix liegen in der parallelen Klasse FNC (sogar in FNC_2). NC_2 umfasst NL, FNC ist FP enthalten. Determinantenberechnung ist in FP; man benutzt hierbei den Gauß-Algorithmus, um die Matrix in obere Dreiecksform zu bringen. SymbDet liegt in RP.

Faktorisieren und Verwandtes

[Num 1] PRIMES

Eingabe: Eine natürliche Zahl n (binär kodiert)
Frage: Ist n eine Primzahl?

[Num 2] COMPOSITES

Eingabe: Eine natürliche Zahl n (binär kodiert)
Frage: Ist n zusammengesetzt, d.h. existieren $a, b < n$ mit $ab = n$?

[Num 3] Faktorisieren

Eingabe: Eine natürliche Zahl n (binär kodiert)
Ausgabe: Zwei Zahlen $a, b < n$ mit $ab = n$ oder „n ist prim“

Die Entscheidungsprobleme PRIMES und COMPOSITES liegen in P [AKS02]. Ein verhältnismäßig einfacher Algorithmus belegt, dass PRIMES $\in$ co-RP. Aus der Tatsache PRIMES $\in$ P lässt sich nicht direkt ein Polynomialzeitalgorithmus für das Konstruktionsproblem Faktorisieren herleiten. Es ist noch unbekannt, ob sich Zahlen in Polynomialzeit in ihre Faktoren zerlegen lassen.

A.7 Algebra

[Alg 1] GENERABILITY GEN

Eingabe: Eine endliche Menge W, ein zweistellige Operation $\circ$ auf W (durch eine Verknüpfungstafel), eine Startmenge $S \subseteq W$ und ein Zielelement $t \in W$
Frage: Lässt sich t durch Verknüpfung aus S erzeugen? D.h., ist t enthalten in der kleinsten Teilmenge von W, die S umfasst und abgeschlossen ist unter der Verknüpfung $\circ$?

[Alg 2] COMMGEN

Eingabe: Eine endliche Menge W, ein zweistellige kommutative Operation $\circ$ auf W (durch eine Verknüpfungstafel), eine Startmenge $S \subseteq W$ und ein Zielelement $t \in W$
Frage: Lässt sich t durch Verknüpfung aus S erzeugen? D.h., ist t enthalten in der kleinsten Teilmenge von W, die S umfasst und abgeschlossen ist unter der Verknüpfung $\circ$?

[Alg 3] ASSOGEN

Eingabe: Eine endliche Menge W, ein zweistellige assoziative Operation $\circ$ auf W (durch eine Verknüpfungstafel), eine Startmenge $S \subseteq W$ und ein Zielelement $t \in W$
Frage: Lässt sich t durch Verknüpfung aus S erzeugen? D.h., ist t enthalten in der kleinsten Teilmenge von W, die S umfasst und abgeschlossen ist unter der Verknüpfung $\circ$?

GENERABILITY und COMMGEN sind P-vollständig, ASSOGEN ist NL-vollständig.

A.8 Packungen und Scheduling

[Pack 1] KNAPSACK

Eingabe: Eine Liste $w_1, w_2, \ldots, w_n$ von Gewichten und eine Liste $v_1, v_2, \ldots, v_n$ von Werten sowie ein Maximalgewicht W; außerdem ein Mindestwert V.
Frage: Gibt es eine Teilmenge $S \subseteq \{1, \ldots, n\}$, so dass $\sum_{i \in S} w_i \leqslant W$; und gleichzeitig $\sum_{i \in S} v_i \geqslant V$?

[Pack 2] MaxKnapsack

Eingabe: Eine Liste $w_1, w_2, \ldots, w_n$ von Gewichten und eine Liste $v_1, v_2, \ldots, v_n$ von Werten sowie ein Maximalgewicht W.

Lösungen: Eine Teilmenge $S \subseteq \{1, \ldots, n\}$, so dass $\sum_{i \in S} w_i \leqslant W$.

Ziel: Maximiere $m(x, S) = \sum_{i \in S} v_i$.

[Pack 3] BinPacking

Eingabe: Eine Liste $w_1, w_2, \ldots, w_n$ von Gewichten, ein Maximalgewicht W und ein $k \in \mathbb{N}$.

Frage: Kann man die Gewichte in k Behälter (Bins), die jeweils Fassungsvermögen W haben, packen?

[Pack 4] MinBinPacking

Eingabe: Eine Liste $w_1, w_2, \ldots, w_n$ von Gewichten, ein Maximalgewicht W.

Lösungen: Eine Verteilung der Gewichte auf Behälter (Bins). D.h., eine Partion $B_1, \ldots, B_k$, so dass für jedes j gilt $\sum_{i \in B_j} w_i \leqslant W$.

Ziel: Minimiere k, die Anzahl der Bins.

[Pack 5] MinSchedulingWithSpeedFactors

Eingabe: Eine Liste von n Jobs mit Bearbeitungszeiten $t_1, \ldots, t_n \in \mathbb{N}$ und von k Prozessoren mit Speed-Factors $s_1, \ldots, s_k \in \mathbb{N}$. Wird Job i auf Prozessor j ausgeführt, so dauert das $t_i \cdot s_j$ Zeiteinheiten.

Lösungen: Eine Verteilung der Jobs auf die Prozessoren; also eine Partition $J_1 \cup \cdots \cup J_k = \{1, \ldots, n\}$.

Ziel: Minimiere die Zeit, bis alle Prozessoren fertig sind. Minimiere also $\max_j \sum_{i \in J_j} t_i \cdot s_j$.

Literatur

[AB07] Sanjeev Arora and Boaz Barak. *Complexity Theory: A Modern Approach.* to appear, 2007.

[AG88] Amihood Amir and William Gasarch. Polynomial terse sets. *Information and Computation*, 77, 1988.

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur-208016, India, August 2002.

[All01] Eric Allender. The division breakthroughs. In *Bulletin of the EATCS*, volume 74, 2001.

[AS92] Noga Alon and Joel H. Spencer. *The probabilistic method.* Wiley, New York, 1992.

[BC94] Daniel P. Bovet and Pierluigi Crescenzi. *Introduction to the Theory of Complexity.* Prentice Hall, New York, 1994. vergriffen, aber online verfügbar: http://www.algoritmica.org/piluc/.

[BDG90] José Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity II.* Springer, 1990.

[BDG93] José Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I.* Springer, 1993. Zweite Auflage.

[BKS95] Richard Beigel, Martin Kummer, and Frank Stephan. Approximable sets. *Information and Computation*, 120(2), 1995.

[BM91] David A. Mix Barrington and Pierre McKenzie. Oracle branching programs and logspace versus P. *Information and Computation*, 95(1):96–115, 1991.

[CDL00] Andrew Chiu, George I. Davida, and Bruce E. Litow. NC^1 division, 2000. verfügbar über `http://www.cs.jcu.edu.au/~bruce` .

[Chi95] Andrew Chiu. Complexity of parallel arithmetic using the chinese remainder theorem. Master's thesis, U. Wisconsin-Milwaukee, 1995. G. Davida, supervisor.

[Chr76] Nicos Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report 388, GSIA, Carnegie-Mellon University, Pittsburgh, 1976.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press Cambridge, 1990.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings 3rd Symposium Science (FOCS)*, pages 151–158, 1971.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity.* John Wiley and Sons, 2000.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with 0, 1 vertices. *Journal of Research National Bureau of Standards*, 69B:125–130, 1965.

[Fag74] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity and Computation*, volume 7, pages 43–73. SIAM-AMS, 1974.

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation – P-Completeness Theory.* Oxford University Press, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[GW83] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control (now Information and Computation)*, 56:183–198, 1983.

[Hal73] Paul R. Halmos. *Naive Mengenlehre.* Vandenhoeck & Ruprecht, 3. Auflage, 1973.

[Hal95] Leslie Hall. Appproximation algorithms for scheduling. In Dorit S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*. PWS, 1995.

[HU79] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[Imm98] Neil Immerman. *Descriptive Complexity*. Springer, 1998.

[Joc68] C. Jockusch, Jr. Semirecursive sets and positive reducibility. *Trans. Amer. Math. Soc.*, 131, 1968.

[Jun94] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI Wissenschaftsverlag, 3. Auflage, 1994.

[KL80] Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th Symposium on Theory of Computing (STOC)*, pages 302–309. ACM Press, 1980.

[Lad75] Richard E. Ladner. The circuit value problem is log-space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[Lau83] Clemens Lautemann. BPP and the polynomial hierarchy. *Information Processing Letters*, 17, 1983.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[MS73] Albert R. Meyer and Larry J. Stockmeyer. Word problems requiring exponential time. In *ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM Press, 1973.

[NT03] Arfst Nickelsen and Till Tantau. Partial information classes. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 34, 2003.

[NT04] Arfst Nickelsen and Till Tantau. Berechnungsmodelle und Komplexität. Vorlesungsskript, 2004. TU Berlin, `http://tal.cs.tu-berlin.de/lv/ss2004/basis_thi/skript2004.ps`.

[Odi89] Piergiorgio Odifreddi. *Classical Recursion Theory I*. North Holland, 1989.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Rei99] Rüdiger Reischuk. *Einführung in die Komplexitätstheorie*. Teubner Verlag Stuttgart, 1999.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[Sch85] Uwe Schöning. *Complexity and Structure*. Springer-Verlag, 1985.

[Sch92] Uwe Schöning. *Theoretische Informatik kurz gefaßt*. BI-Wissenschaftsverlag, 1992.

[Sel79] Alan Selman. P-selective sets, tally languages and the behaviour of polynomial time reducibilities on NP. *Mathematical Systems Theory*, 13:55–65, 1979.

[Sto76] Larry Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Tod91] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[Wag94] Klaus Wagner. *Einführung in die Theoretische Informatik*. Springer, 1994.

[WW86] Klaus Wagner and Gerd Wechsung. *Computational Complexity*. VEB Deutscher Verlag der Wissenschaften, 1986.