Komplexitätstheorie

Volker Diekert
Institut für Informatik
Universität Stuttgart

8. Februar 1996

Vorwort

Zunächst erscheint mir eine *Warnung* angebracht. Dieses Skript ist kein Lehrbuch, sondern gibt den Stoff wieder, der im Sommersemester 1995 behandelt wurde. Im Gegensatz zu einem Lehrbuch sind die Beweise häufig sehr kurz, vielfach steht auch ein Hinweis auf Übungen. Dies bedeutet nicht, daß die Übungen unbedingt leicht sind. Ist ein Verständnis durch die Lektüre des Skripts nicht möglich, so soll auf die Hinzunahme eines Lehrbuchs nicht verzichtet werden.

Die Komplexitätstheorie hat sich in den letzten Jahren grundlegend verwandelt. Sie ist algorithmischer geworden, und durch Verwendung algebraischer Methoden konnten in der letzten Dekade wesentliche Fortschritte erzielt werden. Unsere Stoffauswahl basiert auf dem Abschnitt Komplexitätstheorie in [Die95] und dem Lehrbuch von C. Papadimitriou [Pap94]. Einige Themen aus diesem Lehrbuch finden sich auch bei U. Schöning [Sch95].

Mein Dank gilt den Hörern Holger Austinat, Martin Bauer, Michael Kaiser, Uwe Kubach und Tim Schmelmer sowie Herrn Markus Sing und Herrn Jochen Meßner, die dieses Skript erstellt haben.

Stuttgart, Februar 1996

Volker Diekert

Inhaltsverzeichnis

Kapitel 1

Einführung

Wir untersuchen in dieser Vorlesung Komplexitätsklassen, die durch Mehrband-Turingmaschinen definiert werden. Wichtige Klassen können auch durch andere Maschinenmodelle, wie etwa Registermaschinen, definiert werden. Der Stoff findet sich in Lehrbüchern wie [Pau78, HU79, BDG88, Rei90, Weg93, Pap94]. Dabei haben wir uns insbesondere an [Pap94] orientiert.

Wir betrachten ausschließlich *starke Platz- und Zeitschranken*, wie im folgenden erläutert. Genügt eine (nichtdeterministische) Maschine einer Platzschranke $f : \mathbb{N} \to \mathbb{N}$, so bedeutet dies, daß keine Rechnung existiert, die auf einer Eingabe der Länge n, $n \geq 0$, mehr als $f(n)$ Felder des Bandes benutzt. Für sublinearen Platz (d.h. $f(n) < n$ für ein n) gehen wir von einem separaten Eingabeband aus, das nicht überschrieben werden darf. Das linke und rechte Ende der Eingabe ist markiert, und der Lesekopf darf nicht über diese Markierungen hinauswandern. Damit wir die Lesekopfposition binär codiert auf das Band schreiben können, machen wir die Einschränkung, daß für alle Platzschranken $f \in \Omega(\log(n))$ gilt. Hierbei bezeichne $\log(n)$ die Funktion $\log(n) = \max\{1, \lceil \log_2(n) \rceil\}$. Die Wahl der Basis 2 ist unerheblich, da die Platzschranken nicht von multiplikativen Konstanten abhängen. Analog gilt für eine Zeitschranke $f : \mathbb{N} \to \mathbb{N}$, daß auf keiner Eingabe der Länge n, $n \geq 0$, die Maschine mehr als $f(n)$ Rechenschritte machen kann, bevor sie hält. Wir betrachten keine sublinearen Zeitschranken und folgen der Konvention, daß auch die Zeitschranke n die Durchführung von $n + 1$ Rechenschritten erlaubt. Damit kann stets das Ende eines Eingabewortes erkannt werden.

Für $f \in \Omega(\log(n))$ behandeln wir die Platzklassen DSPACE(f) und NSPACE(f); für $f(n) \geq n$ für alle n behandeln wir die Zeitklassen DTIME(f) und NTIME(f).

Definition 1.0.1 Sei $t : \mathbb{N} \to \mathbb{N}$, $s : \mathbb{N} \to \mathbb{N}$ Funktionen mit $t(n) \geq n$ für alle $n \in \mathbb{N}$ und $s \in \Omega(\log)$. Setze:

DTIME(t)	=	$\{L \subseteq \Sigma^* \mid$ es existiert eine deterministische Turingmaschine M mit $L = L(M)$, die auf allen Eingaben der Länge n nach höchstens $\max\{t(n), n{+}1\}$ Schritten hält.$\}$
NTIME(t)	=	$\{L \subseteq \Sigma^* \mid$ es existiert eine nichtdeterministische Turingmaschine M mit $L = L(M)$, die auf allen Eingaben der Länge n und bei allen Rechnungen nach höchstens $\max\{t(n), n+1\}$ Schritten hält.$\}$
DSPACE(s)	=	$\{L \subseteq \Sigma^* \mid$ es existiert eine deterministische Turingmaschine M mit $L = L(M)$, die auf allen Eingaben der Länge n höchstens $s(n)$ Arbeitsbandfelder benötigt.$\}$
NSPACE(s)	=	$\{L \subseteq \Sigma^* \mid$ es existiert eine nichtdeterministische Turingmaschine M mit $L = L(M)$, die auf allen Eingaben der Länge n bei allen Rechnungen höchstens $s(n)$ Arbeitsbandfelder benötigt.$\}$

Für eine (Komplexitäts-)Klasse $\mathcal{C}$ bezeichnen wir die Menge der Komplemente der in $\mathcal{C}$ enthaltenen Sprachen mit **co-**$\mathcal{C} = \{L \mid \Sigma^* \setminus L \in \mathcal{C}\}$.

1.1 Grundlegende Definitionen

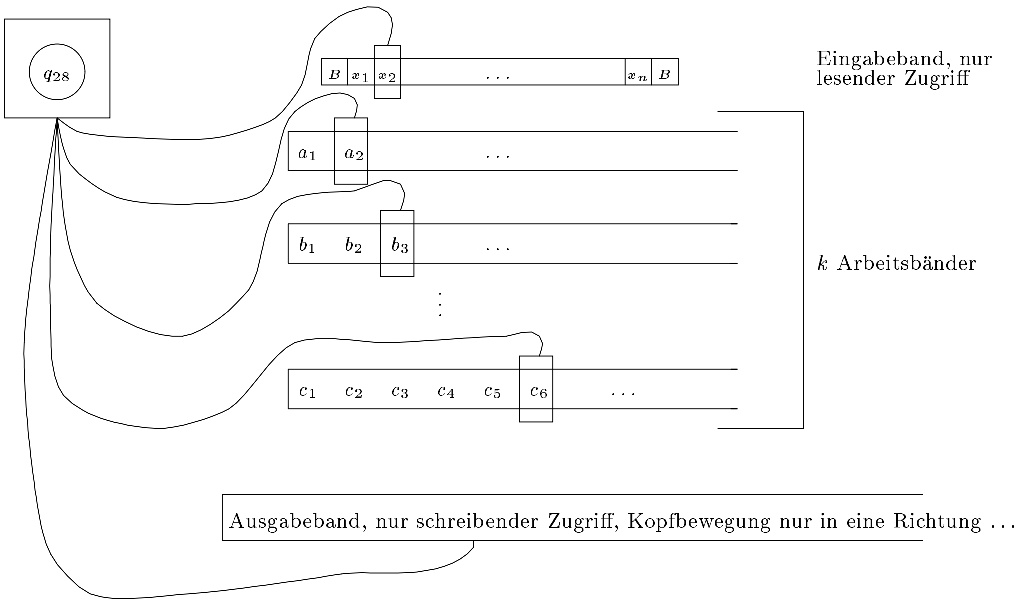
Definition 1.1.1 Eine Turingmaschine M ist ein Tupel $(Q, \Sigma, \Sigma', \Gamma, \delta, q_0, F, B)$

- Q : endliche Menge der Zustände
- $q_0 \in Q$: Startzustand
- $F \subseteq Q$: Menge der Endzustände
- Σ : Eingabealphabet
- Σ' : Ausgabealphabet
- Γ : Bandalphabet
- $B \in \Gamma - \Sigma$: Blanksymbol
- $\delta \subseteq Q \times (\Sigma \cup \{B\}) \times \Gamma^k \times Q \times \Gamma^k \times \{L, R, N\}^{k+1}$: endliche Übergangsrelation, falls M k separate Arbeitsbänder hat. L, R und N stehen für eine Bewegung des Kopfes nach links, rechts bzw. die neutrale Bewegung (verharren auf derselben Bandposition).

  Berechnet M eine Funktion, so wird δ um eine Ausgabekomponente $\Sigma' \cup \{\lambda\}$ erweitert.

  Ist M eine deterministische Turingmaschine, so ist δ eine partiell definierte Funktion $\delta : Q \times (\Sigma \cup \{B\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^{k+1}$.

Beispiel 1.1.2 Schematische Darstellung einer k-Band Turingmaschine



Definition 1.1.3 Eine Konfiguration α einer Turingmaschine ist ein Tupel $\alpha = (i, q, w_1, \dots, w_k)$, mit:

- i : Position des Lesekopfes, $0 \leq i \leq n+1$, wobei n die Eingabelänge bezeichnet. Die Positionen $i = 0$ und $i = n+1$ werden zum Erkennen von Anfang und Ende der Eingabe benötigt.
- $q \in Q$: aktueller Zustand der Turingmaschine
- $w_1, \dots, w_k \in (\Gamma \cup \hat{\Gamma})^*$: die k Bandinschriften, so daß gilt:
  1. Links und rechts von den w_i stehen nur Blanksymbole.
  2. Falls $w_i \neq \lambda$, trägt w_i die Position des Schreiblesekopfs auf dem i-ten Arbeitsband, etwa indem das entsprechende Zeichen mit einem Dach (ˆ) gekennzeichnet wird.

Beispiel 1.1.4 Die Konfiguration α in oberen Beispiel ergibt sich zu

$$\alpha = (2, q_{28}, a_1\hat{a}_2a_3 \dots a_{n_1}, b_1b_2\hat{b}_3b_4 \dots b_{n_2}, \dots, c_1c_2c_3c_4c_5\hat{c}_6c_7 \dots c_{n_k}).$$

Konventionen:

1. Die Länge $|\alpha|$ einer Konfiguration α ist definiert durch

$$|\alpha| = max\{|w_j| \mid \alpha = (i, q, w_1, \ldots, w_k), 1 \leq j \leq k\}.$$

2. Wir schreiben $\alpha \vdash \beta$, falls die Konfiguration α in einem Schritt nach β übergehen kann. Vorteilhaft an unserer Längendefiniton ist, daß gilt: $\alpha \vdash \beta \Rightarrow ||\alpha| - |\beta|| \leq 1$.

   Durch eine geeignete „Alphabetvergrößerung“ (alle Arbeitsbänder auf eines kodieren) ist es möglich, α durch $|\alpha|+1+\lceil\log_2(n+1)\rceil$ Zeichen zu beschreiben. Daraus folgt, daß, wenn f eine Platzschranke mit $f \in \Omega(\log(n))$ ist, der Platz $f(n)$ ausreicht, um α aufzuschreiben.

3. Für die Eingabe $x = x_1 x_2 \ldots x_n \in \Sigma^*$ bezeichnen wir mit

$$\text{Start}(x) = (1, q_0, \lambda, \ldots, \lambda)$$

   die Startkonfiguration, wobei gilt: $|\,\text{Start}(x)| = 0$.

   Beachte: Bei sublinearem Platz tragen Konfigurationen keine Informationen über das Eingabewort. Bei Zeit- oder Platzklassen mit Schranken $f \in \Omega(n)$ können wir das separate Eingabeband vernachlässigen und die Eingabe auf einem Arbeitsband kodieren, ohne dadurch in eine höhere Platzkomplexitätsklasse zu geraten.

4. Mit Accept bezeichnen wir die Menge der akzeptierenden Konfigurationen.

5. Mögliche Konvention ist, daß es genau eine bzw. eine fest vorgegebene Anzahl von akzeptierende Konfigurationen gibt. (Wähle z.B. $(q_f, 0, \lambda, \ldots, \lambda)$ als akzeptierende Konfiguration; das Löschen der Bandinhalte benötigt keinen zusätzlichen Platz und maximal soviel Zeit, wie die Turingmaschine schon bisher für die Rechnung benötigt hat.)

Definition 1.1.5

- Eine *Rechnung von M* ist eine Folge $\text{Start}(x) = \alpha_1, \alpha_2, \ldots, \alpha_m$ mit $\alpha_{i-1} \vdash \alpha_i$.
- Sie heißt *erfolgreich*, falls $\alpha_m \in \text{Accept}$.
- Ein *Protokoll* der Rechnung ist eine Folge der zugehörigen Übergänge, d.h. eine Folge $d_1 d_2 \ldots d_{m-1} \in \delta^*$, wobei d_i den Übergang von α_i nach α_{i+1} beschreibt.

Beispiel 1.1.6

- $\{a^n b^n c^n \mid n \geq 1\} \in \text{DSPACE}(\log(n))$
- $\{w\$w \mid w \in \Sigma^*\} \in \text{DSPACE}(\log(n))$
- Prim $= \{p \in 1\{0,1\}^* |$ p ist Binärcodierung einer Primzahl$\} \in \text{DSPACE}(n)$

Definition 1.1.7

- Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f \in \Omega(n)$ heißt *zeitkonstruierbar*, falls es eine deterministische Turingmaschine gibt, die auf eine Eingabe I^n (d.h. n ist unär kodiert) nach genau $f(n)$ Schritten hält.
- Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f \in \Omega(\log(n))$ heißt *platzkonstruierbar*, falls es eine deterministische Turingmaschine gibt, die auf eine Eingabe I^n genau $f(n)$ Felder auf den Arbeitsbändern markiert, dann hält und bei der Berechnung diesen Platz nicht verläßt.

Beispiel 1.1.8

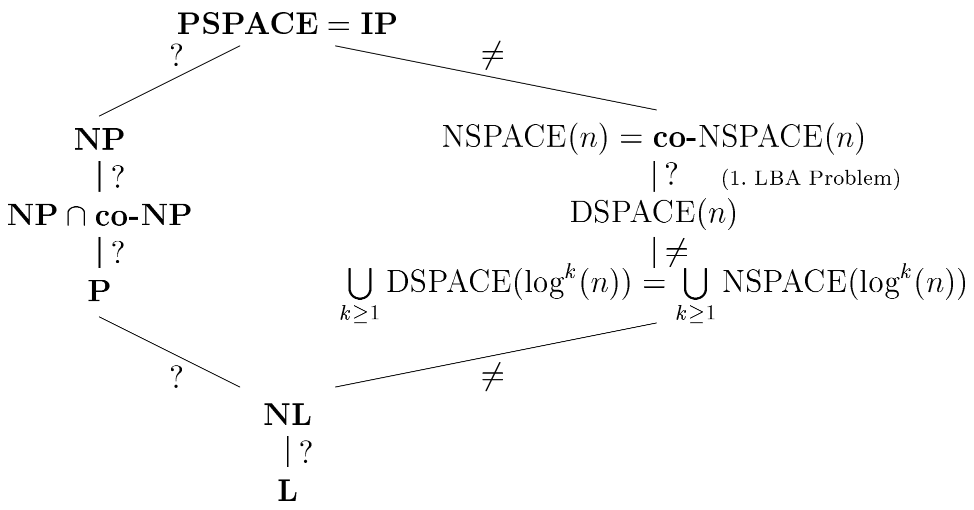
- Ist die Funktion ein Polynom, z.B. $f(n) = 17n^3 - 6n^2 + 3n + 23$, so ist sie zeitkonstruierbar.
- Funktionen wie $f(n) = 2^n$, $f(n) = 2^{2^n}$, $f(n) = 2^{p(n)}$, wobei $p(n)$ ein Polynom sei, sind zeitkonstruierbar.
- f ist zeitkonstruierbar $\Rightarrow$ f ist platzkonstruierbar.
- $\lceil \log_a^k(n) \rceil, a \geq 2, k \geq 1$ ist platzkonstruierbar.
- $\sqrt[5]{n}$ ist platzkonstruierbar.

1.2 Wichtige Komplexitätsklassen

Wir verwenden hier die gebräuchlichen Abkürzungen: $\mathbf{L} = \text{DSPACE}(\log(n))$, $\mathbf{NL} = \text{NSPACE}(\log(n))$, $\mathbf{P} = \bigcup_{k\geq 1} \text{DTIME}(n^k)$, $\mathbf{NP} = \bigcup_{k\geq 1} \text{NTIME}(n^k)$ und $\mathbf{PSPACE} = \bigcup_{k\geq 1} \text{DSPACE}(n^k) = \bigcup_{k\geq 1} \text{NSPACE}(n^k)$.

Im Zuge der Vorlesung (Satz von Savitch, vgl. Satz 2.2.1) werden wir sehen, daß **PSPACE** wohldefiniert ist.

Ein Ziel der Vorlesung ist, ein Verständnis für die Inklusionsbeziehungen in folgendem Diagramm zu gewinnen.



Bei den mit einem Fragezeichen versehenen Inklusionen ist die Gleichheit offen.

Mit Ausnahme des Paares ($\mathbf{NP}\cap\mathbf{co\text{-}NP}$, $\bigcup_{k\geq 1}$ DSPACE$(\log^k(n))$) ist es bekannt, daß die Sprachklassen im linken Zweig von den Sprachklassen des rechten Zweiges im obigen Diagramm paarweise verschieden sind. Nach heutigem Kenntnisstand können wir weder $\mathbf{L} = \mathbf{NP}$ noch $\mathbf{P} = \mathbf{PSPACE}$ ausschließen; nur beides zugleich ist unmöglich.

1.3 Varianten algorithmischer Probleme

Viele algorithmische Probleme treten in Varianten auf. Als typisches Beispiel sei zunächst das Problem des Handlungsreisenden erwähnt:

Beispiel 1.3.1 Ein Reisender will eine gegebene Anzahl von Städten besuchen, ohne dabei an einem Ort zweimal vorbeizukommen, und er will dabei den kürzesten Weg nehmen. Das Wegenetz kann als gerichteter Graph und die Wegstrecken als Gewichte auf den Kanten des Graphen aufgefaßt werden. Die Knoten stellen die Städte dar.

Sei $G = (V, E, \gamma : E \to \mathbb{N})$ ein gerichteter Graph mit Knotenmenge $V = \{1, ..., n\}$, Kantenmenge $E \subseteq V \times V$ und den Kantengewichten $\gamma(e) > 0$ für alle $e \in E$.

Ein Rundweg W ist gegeben durch eine Folge $W = (x_0, \ldots, x_n)$, $x_0 = x_n$, $x_i \neq x_j$ für $1 \leq i < j \leq n$ und $(x_{i-1}, x_i) \in E$ für $1 \leq i \leq n$.

Die Kosten $\gamma(W)$ des Rundweges W sind durch die Summe der Kantengewichte gegeben: $\gamma(W) = \sum_{i=1}^{n} \gamma(x_{i-1}, x_i)$.

(A) Entscheidungsvariante: Eingabe: $G = (V, E, \gamma : E \to \mathbb{N})$ und ein $k \geq 0$.

Frage: Existiert ein Rundweg mit Kosten $\leq k$? D.h., existiert ein Weg der alle Knoten genau einmal besucht und dessen Kosten höchstens k sind?

(B) Berechnungsvariante: Eingabe: $G = (V, E, \gamma : E \to \mathbb{N})$ und ein $k \geq 0$.

Falls ein Rundweg W mit $\gamma(W) \leq k$ existiert, berechne ein solches W.

(C) Optimierungsproblem: Eingabe: $G = (V, E, \gamma : E \to \mathbb{N})$.

Berechne einen kostenoptimalen Rundweg, falls ein Rundweg existiert.

In der Praxis wird man sich häufig nur für eine Lösung der Optimierungsvariante (C) interessieren. Es läßt sich jedoch leicht zeigen, daß eine effiziente Lösung des Entscheidungsproblems zu einer Lösung der Optimierungsvariante (C) führt. Ist (A) in Polynomialzeit lösbar, so auch (C):

Zunächst setzen wir $k_{\max} = \sum_{e \in E} \gamma(e)$. Dieser Wert ist in polynomialer Zeit berechenbar und existiert überhaupt ein Rundweg, so auch ein Rundweg W mit $\gamma(W) \leq k_{\max}$.

Eine erste Anfrage an die Entscheidungsvariante liefert damit die Existenz eines Rundwegs, die wir hiermit annehmen können. Sei $k_{opt} = \gamma(W_{opt})$ die Kosten eines kostenoptimalen Rundweges. Mit Hilfe binärer Suche und Anfragen am Entscheidungsproblem läßt sich k_{opt} in polynomialer Zeit ermitteln. Das binäre Suchverfahren ist notwendig, da der Betrag von k_{opt} exponentiell in der Eingabelänge sein kann. Die genaue Durchführung sei dem Leser als Übung überlassen.

Im nächsten Schritt berechnen wir nun eine optimale Lösung. Das folgende Programm ermittelt einen solchen optimalen Rundweg:

FUNCTION optimaler Rundweg
  $\mathcal{E}$:=LIST$(e_1, \ldots, e_m), e_i \in E$
  for $i := 1$ **to** m **do**
    if es gibt einen Rundweg W mit $\gamma(W) \leq k_{opt}$ und W benutzt nur Kanten aus $\mathcal{E} \setminus \{e_i\}$ **then**
      lösche e_i aus $\mathcal{E}$
    endif
  endfor
  return $\mathcal{E}$
ENDFUNC

Am Ende der for-Schleife besteht die Liste $\mathcal{E}$ genau aus den Kanten eines kostenoptimalen Weges. Hierfür beweist man mit Induktion die folgende Schleifeninvariante: Sei $\mathcal{E}_i$ die Liste, die entstanden ist, nachdem e_i betrachtet wurde. Dann enthält $\mathcal{E}_i$ einen kostenoptimalen Weg und ferner benützt jeder kostenoptimale Weg mit Kanten aus $\mathcal{E}_i$ alle Kanten aus $\{e_1, \ldots, e_i\} \cap \mathcal{E}_i$. Aus der Liste $\mathcal{E} = \mathcal{E}_m$

am Ende der for-Schleife können wir den Rundweg direkt ablesen. Die Lösbarkeit von (A) in polynomialer Zeit impliziert damit die Polyomialzeitberechenbarkeit von (C).

Beispiel 1.3.2 Wir betrachten das Problem einer Knotenüberdeckung (Vertex Cover) in einem ungerichteten Graphen.

Definition 1.3.3 (Vertex Cover, VC) Eine Knotenüberdeckung U eines Graphen $G = (V, E)$ ist eine Teilmenge $U \subseteq V$ so, daß jede Kante $e \in E$ mindestens einen Endpunkt in U hat.

In der Entscheidungsvariante (A) ist die Eingabe der Graph G und ein $k \in \mathbb{N}$. Man fragt nach der Existenz einer Knotenüberdeckung U mit $|U| \leq k$. In der Berechnungsvariante (B) soll auf die gleiche Eingabe ein solches U berechnet werden, falls es existiert. In den Optimierungsvarianten (C) fragt man nach einer optimalen Knotenüberdeckung, also nach einer Knotenüberdeckung $U \subseteq V$, wobei $|U| = k_{opt}$ so sei, daß für jede Knotenüberdeckung $U' \subseteq V$ $|U'| \geq k_{opt}$ gilt. Wie eben nimmt der Schwierigkeitsgrad von (A) nach (C) zu. Ein effizienter Algorithmus für (A) reicht erneut aus, um die Variante (C) effizient zu berechnen.

Angenommen, das Entscheidungsproblem (A) zur Knotenüberdeckung läßt sich in polynomialer Zeit berechnen. Dann läßt sich eine Knotenüberdeckung U minimaler Größe in polynomialer Zeit wie folgt konstruieren: Sei $G = (V, E)$ der Eingabegraph und $n = |V|$. In einem Durchlauf „**for** $i := 1$ **to** n **do**“ wird das minimale k_{opt} bestimmt. Es verbleibt, (B) mit $k = k_{opt}$ zu lösen. Hierfür durchläuft man die Liste aller Knoten und entfernt jeweils den betrachteten Knoten mit allen ausgehenden Kanten, falls in dem resultierenden Graph eine Knotenüberdeckung der Größe $k - 1$ existiert. Dabei wird k auf $k - 1$ gesetzt. Am Ende dieses Durchlaufs sind alle Kanten aus dem Graph entfernt. Insgesamt wurden k_{opt} Knoten aus dem Graph entfernt, die somit eine minimale Knotenüberdeckung bilden.

In Abschnitt 4.3 werden wir uns mit dem Problem VC erneut befassen. Dort werden wir zeigen, daß Vertex Cover **NP**-vollständig ist (Satz 4.3.12).

1.4 Grapherreichbarkeit

Ein wichtiges algorithmisches Problem ist die Frage nach der Existenz von Wegen in gerichteten Graphen. Sei $G = (V, E)$, $E \subseteq V \times V$, ein gerichteter Graph und seien $s, t \in V$ zwei Knoten. Die Frage, ob es einen gerichteten Pfad von s nach t gibt, ist leicht in polynomialer Zeit lösbar. Wir könnten sie auch als Frage nach der Existenz eines nichttrivialen Flusses in einem geeigneten Netzwerk formulieren.

Diese Polynomialzeitalgorithmen erfordern jedoch linearen Platz. Nichtdeterministisch läßt sich dieses Problem jedoch mit logarithmischen Platz lösen, d.h., das Problem gehört zu NSPACE($\mathcal{O}(\log(n))$), wie wir im folgenden beschreiben.

Sei $|V| = n$ und ohne Einschränkung seien die Knoten durch die Zahlen $1, \ldots, n$ repräsentiert, d.h., $V = \{1, ..., n\}$. Wir können ferner annehmen, daß $s = 1$ und $t = n$ gilt. (Der Fall $s = t$ ist trivial.)

Die Lösung wird durch das folgende nichtdeterministische Programm geliefert, wobei die Knotenvariablen u, v als Binärzahlen auf dem Arbeitsband gehalten werden.

FUNCTION Grapherreichbarkeit
  var $u := 1$
  while $u \neq n$ **do**
    wähle einen Knoten $v \in V$ mit $(u, v) \in E$
    $u := v$
  endwhile
  return „Es gibt einen Weg von 1 nach n.“
ENDFUNC

Das Programm kann nur dann terminieren, falls es einen Pfad von s nach t gibt. Terminiert es, so ist die Ausgabe korrekt und gibt es einen Pfad, so existiert ein terminierender Programmablauf. Mit Hilfe des Ansatzes von Savitch (vgl. Satz 2.2.1) läßt sich das obige nichtdeterministische Programm jedoch in DSPACE($\log^2(n)$) simulieren. Die Grundidee ist, daß, falls ein Weg von s nach t existiert, so auch ein schleifenfreier Weg, also ein Weg der Länge $\leq n$. Definiere daher ein Prädikat Reach(u, v, i) für $u, v \in V$ und $0 \leq i \leq \lceil \log_2(n) \rceil$ durch Reach$(u, v, i) \iff \exists$ Pfad von u nach v der Länge $\leq 2^i$. Für $i = 0$ läßt sich Reach(u, v, i) direkt in $\mathcal{O}(\log(n))$ Platz berechnen. Für $i \geq 1$ verwenden wir die Rekursion Reach$(u, v, i) \iff \exists w \in V$: Reach$(u, w, i-1)$ und Reach$(w, v, i-1)$. Denn für $i \geq 1$ gibt es genau dann einen Pfad von u nach v der Länge $\leq 2^i$, falls es einen Zwischenknoten $w \in V$ gibt und Pfade von u nach w und von w nach v der Länge $\leq 2^{i-1}$.

Etwas anders aufgeschrieben können wir Reach(u, v, i) durch das deterministische Programm in Abbildung 1.1 definieren.

Man beweist mit Induktion, daß ein Aufruf von Reach(u, v, i) den Platzbedarf $\mathcal{O}((i+1)\log(n))$ hat. Damit kann Reach$(u, v, \lceil \log_2(n) \rceil)$ in DSPACE($\mathcal{O}(\log^2(n))$) berechnet werden.

Als Übungsaufgabe berechne man den Zeitbedarf der rekursiven Berechnung von Reach$(u, v, \lceil \log_2(n) \rceil)$, wenn die Eingabe ein Graph mit n Knoten ist, der keinerlei Kanten hat. (Für $i > 0$ ist Reach(u, v, i) also stets falsch). Hinweis: Zeigen Sie den Zeitbedarf $\theta(n^i)$ für die Berechnung von Reach(u, v, i).

```
FUNCTION Reach(u, v, i)
  var b := FALSE
  if i = 0 then
    b := (u = v) or (u, v) ∈ E
  else
    for w := 1 to n do
      if not b then
        if Reach(u, w, i − 1) then
          b := Reach(w, v, i − 1)
        endif
      endif
    endfor
  endif
  return b
ENDFUNC
```

Abbildung 1.1: Die Reach-Funktion

In Abschnitt 3.3 untersuchen wir dieses Problem noch einmal. Dort werden wir in Satz 3.3.2 zeigen, daß das Grapherreichbarkeitsproblem **NL**-vollständig ist.

Kapitel 2

Beziehungen zwischen den Komplexitätsklassen

2.1 Grundlagen

Die Beweise für die Beziehungen in diesem Abschnitt finden sich in allen Standardlehrbüchern. Sie werden daher in diesem Skript nicht wiedergegeben oder nur angedeutet.

Satz 2.1.1

1. *Sei* $X \in \{D, N\}$, *dann gilt* $\mathrm{XSPACE}(\mathcal{O}(f)) = \mathrm{XSPACE}_{\text{1-Band}}(f)$.
2. $\mathrm{NTIME}(\mathcal{O}(f)) = \mathrm{NTIME}(f)$.
3. *Sei* $\epsilon > 0$ *und* $\forall n : f(n) \geq (1+\epsilon)n$, *dann gilt* $\mathrm{DTIME}(\mathcal{O}(f)) = \mathrm{DTIME}(f)$.
4. $\mathrm{DTIME}(n) \neq \mathrm{DTIME}(\mathcal{O}(n))$.

Der Punkt 1 kombiniert Bandreduktion mit Bandkompression. Die Punkte 2 und 3 bezeichnet man als Zeitkompression.

Der Satz von Hennie und Stearns ist ein Bandreduktionssatz für Zeitkomplexitätsklassen.

Satz 2.1.2 (Hennie-Stearns 1966) *Sei* $\varepsilon > 0$, $k \geq 1$ *und* $\forall n : f(n) \geq (1+\varepsilon)n$, *dann gilt:* $\mathrm{DTIME}_{k\text{-Band}}(f) \subseteq \mathrm{DTIME}_{\text{2-Band}}(f \cdot \log(f))$.

Satz 2.1.3 *Sei* $\forall n : f(n) \geq n$, *dann gilt*

$$\mathrm{DTIME}(f) \subseteq \mathrm{NTIME}(f) \subseteq \mathrm{DSPACE}(f).$$

Beweis: Zu zeigen ist NTIME(f) $\subseteq$ DSPACE(f).

Sei M eine nichtdeterministische $f(n)$-zeitbeschränkte Turingmaschine. Auf eine Eingabe $x \in \Sigma^*$ der Länge n können wir uns die Rechnung von M als einen Berechnungsbaum wie folgt vorstellen. Der Baum hat als Wurzel die Startkonfiguration Start(x), und die Kinder der inneren Knoten sind die jeweiligen Folgekonfigurationen.

Dieser Baum wird in einem Breitendurchlauf auf eine akzeptierende Konfiguration hin durchsucht. Dabei merkt man sich jeweils nur eine Konfiguration und das Protokoll mit dem diese Konfiguration erreicht wird. Die Tiefe $f(n)$ dieses Baumes entspricht der maximalen Pfadlänge und damit dem Platzbedarf der Breitensuche. ■

Satz 2.1.4 *Sei* $\forall n : f(n) \geq \log(n)$, *dann gilt*

$$\text{DSPACE}(f) \subseteq \text{NSPACE}(f) \subseteq \text{DTIME}(2^{\mathcal{O}(f)}).$$

Beweis: Sei M eine $f(n)$ platzbeschränkte nichtdeterministische Turingmaschine und $x \in \Sigma^*$ eine Eingabe der Länge n. Es gibt eine Konstante $c \geq 0$, die nur von M abhängt, so, daß die Anzahl der von Start(x) erreichbaren Konfigurationen durch $c^{f(n)}$ beschränkt ist. Beachte, daß $c^f \in 2^{\mathcal{O}(f)}$ gilt.

Wir erstellen jetzt eine Liste $\mathcal{L}$ aller von Start(x) erreichbaren Konfigurationen nach dem folgenden Schema: Zu Anfang enthält $\mathcal{L}$ nur die Startkonfiguration Start(x). Solange $\mathcal{L}$ eine Konfiguration α enthält, von der noch nicht alle direkten Nachfolgekonfigurationen in der Liste enthalten sind, wird $\mathcal{L}$ um diese direkten Nachfolger erweitert. Die Anzahl der Konfigurationen in der Liste $\mathcal{L}$ ist durch $c^{f(n)}$ beschränkt. In jedem Durchlauf werden also maximal $c^{f(n)}$ Konfigurationen α mit maximaler Länge $f(n)$ betrachtet. Das Durchsuchen, ob alle direkten Nachfolger bereits vorhanden sind, kostet pro Konfiguration höchstens die Zeit $\mathcal{O}(f(n)c^{f(n)})$. In jedem Durchlauf wird die Liste entweder vergrößert oder das Verfahren terminiert. Die Gesamtzeit ist also durch $\mathcal{O}(f(n)c^{2f(n)}) \subseteq \mathcal{O}(2^{\mathcal{O}(f)})$ beschränkt. (Man beachte auch $2^{\mathcal{O}(f)} = \cup_{c\geq 1}\mathcal{O}(2^{cf}) = \cup_{c\geq 1}\mathcal{O}(c^f)$ und daß es nicht darauf ankommt, $\mathcal{L}$ sehr effizient zu berechnen.) ■

Bemerkung 2.1.5 Es gilt

- $\mathbf{L} \subseteq \mathbf{NL} \subseteq$ DSPACE($\log^2(n)$) $\subseteq$ DTIME($n^{\mathcal{O}(\log(n))}$)
- $\mathbf{NL} \subseteq \mathbf{P}$
- $CS = LBA =$ NSPACE(n) $\subseteq$ DTIME($2^{\mathcal{O}(n)}$)

  Hierbei bezeichnet CS die Klasse der kontextsensitiven und LBA die Klasse der durch linear beschränkte Automaten akzeptierten Sprachen.
- DSPACE(n^2) $\subseteq$ DTIME($2^{\mathcal{O}(n^2)}$)

2.2 Der Satz von Savitch

Der Satz von Savitch [Sav70] besagt, daß eine nichtdeterministische platzbeschränkte Turingmaschine unter quadratischem Mehraufwand deterministisch simuliert werden kann. Diese platzeffiziente Simulation wird durch einen extremen Mehraufwand an Rechenzeit realisiert. Der Satz gilt für die hier verwendeten starken Platzschranken, ohne daß s als konstruierbar vorausgesetzt werden muß.

Satz 2.2.1 (Savitch, 1970) *Sei $s \in \Omega(\log(n))$. Dann gilt*

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

Beweis: Wir skizzieren den Beweis unter der Annahme, daß s platzkonstruierbar ist. Wie man sich von dieser Annahme befreit, überlassen wir den Übungen. Der Beweis verläuft analog zur Lösung des Graphenerreichbarkeitsproblems. In dem hier betrachteten Graphen sind die Knoten die Konfigurationen bis zu einer bestimmten Länge m. Die Kanten sind durch die direkte Nachfolgerelation gegeben. Die Frage ist, ob eine akzeptierende Konfiguration von der Startkonfiguration aus erreichbar ist.

Sei M eine s-platzbeschränkte nichtdeterministische Turingmaschine.
Mit $\alpha, \beta, \gamma, \ldots$ bezeichnen wir Konfigurationen und $\mathrm{Reach}(\alpha, \beta, i)$ bedeute, daß sich — auf Eingabe $w \in \Sigma^*$ — die Konfiguration α in höchstens 2^i Schritten in β überführen läßt. Wir können annehmen, daß M nur eine einzige akzeptierende Konfiguration Accept besitzt. Für $w \in \Sigma^*$ sei $\mathrm{Start}(w)$ die Startkonfiguration. Aus der Beschreibung von M können wir explizit eine Konstante c so gewinnen, daß für alle $w \in \Sigma^*$ gilt:

$$w \in L \iff \mathrm{Reach}(\mathrm{Start}(w), \mathrm{Accept}, c \cdot s(|w|)).$$

Das Prädikat $\mathrm{Reach}(\alpha, \beta, i)$ läßt sich für $i = 0$ direkt berechnen und für $i > 0$ verwenden wir das Rekursionsschema:

$$\begin{aligned}\mathrm{Reach}(\alpha, \beta, i) \quad \iff \quad & \exists \text{ Konfiguration } \gamma \text{ mit } |\gamma| \leq s(|w|): \\ & \mathrm{Reach}(\alpha, \gamma, i-1) \text{ und } \mathrm{Reach}(\gamma, \beta, i-1).\end{aligned}$$

Dieses Schema kann offensichtlich in ein deterministisches Programm übersetzt werden, sofern $s(|w|)$ bekannt ist. Dies ist erfüllt, wenn die Funktion s platzkonstruierbar ist. Für eine gewisse Konstante c' zeigt eine Induktion nach i, daß der Platzbedarf zur Berechnung von $\mathrm{Reach}(\alpha, \beta, i)$ durch $c' \cdot s(|w|) \cdot (i+1)$ abschätzt werden kann. Der maximale Platzbedarf $\mathcal{O}(s^2)$ ergibt sich unmittelbar. ∎

Um sich von der Forderung der Platzkonstruierbarkeit von s zu befreien, zeige man mit dem Ansatz von Savitch, daß sich der tatsächliche Platzbedarf einer s-platzbeschränkten Turingmaschine in $\mathrm{DSPACE}(s^2)$ berechnen läßt.

Wegen NSPACE(n^k) $\subseteq$ DSPACE(n^{2k}) $\subseteq \bigcup_{k\geq 1}$ DSPACE(n^k) folgt auch die Wohldefiniertheit von **PSPACE**:

Korollar 2.2.2 *Es gilt:* $\bigcup\limits_{k\geq 1} \mathrm{DSPACE}(n^k) = \bigcup\limits_{k\geq 1} \mathrm{NSPACE}(n^k)$.

2.3 Hierarchiesätze

Der Nachweis von unteren Schranken ist schwierig. Die generelle Idee ist, daß die Bereitstellung größerer Ressourcen an Platz und/oder Zeit zu größeren Klassen führen soll. Um jedoch zu beweisbaren Hierarchieresultaten zu gelangen, müssen mehrere Voraussetzungen erfüllt sein. Wir benötigen die Konstruierbarkeit der Platz- bzw. Zeitschranken sowie den Komplementabschluß. Außerdem müssen wir eine Mehrbandmaschine auf einer universellen Turingmaschine (mit einer festen Bandzahl) simulieren können.

Für deterministische Zeitklassen ist der Komplementabschluß trivial, eine deterministische Platzklasse DSPACE(f) ist nach [HU79] für $f \in \Omega(\log)$ gegen Komplement abgeschlossen. Die Simulation von beliebig vielen Bänder auf einem Band mit gleichem Platzverbrauch ist mit Hilfe des Bandreduktionssatzes möglich. Wir erhalten den deterministischen Platzhierarchiesatz:

Satz 2.3.1 (Platzhierarchiesatz)
Seien $s_1, s_2 : \mathbb{N} \to \mathbb{N}$ *Funktionen,* $s_1 \notin \Omega(s_2)$, $s_2 \in \Omega(\log(n))$ *und* s_2 *sei platzkonstruierbar. Dann gilt* DSPACE(s_2) $\setminus$ DSPACE(s_1) $\neq \emptyset$.

Bemerkung 2.3.2

- $s_1 \notin \Omega(s_2)$ bedeutet $\forall \epsilon > 0$ $\exists$ unendlich viele n mit $s_1(n) < \epsilon \cdot s_2(n)$.

  Seien etwa $s_1(n) = n$ und $s_2(n) = \begin{cases} n^2, & \text{falls } n \text{ gerade} \\ \log n, & \text{sonst} \end{cases}$,

  dann gilt: $s_2 \notin \Omega(s_1)$, $s_1 \notin \Omega(s_2)$.

- Aus dem Satz folgt etwa:

$$\begin{aligned} &\mathrm{DSPACE}(\log(n)) \subsetneq \mathrm{DSPACE}(\log^2(n)) \subsetneq \mathrm{DSPACE}(n) \\ \subseteq\ &\mathrm{NSPACE}(n) \subsetneq \mathrm{DSPACE}(n^{2.1}) \subsetneq \mathbf{PSPACE} \end{aligned}$$

Beweis: (des Platzhierarchiesatzes)
Wir wählen zunächst eine geeignete Kodierung von deterministischen Turingmaschinen, die eine effiziente Simulation erlaubt. Jedes Wort $x \in \Sigma^*$ soll als Kodierung einer Turingmaschine M_x interpretiert werden können (z.B. wählt man für nicht wohlgeformtes x eine Defaultmaschine). Wichtig ist folgende spezielle Konvention: wenn x die Turingmaschine M_x kodiert, so soll $y = x|^n$ für jedes $n \in \mathbb{N}$

eine Kodierung für dieselbe Turingmaschine sein, d.h. $M_x = M_y$. Als Folgerung hieraus kann gesagt werden, daß jede Turingmaschine Kodierungen für fast alle Längen besitzt.

Wir geben nun eine deterministische s_2-platzbeschränkte Turingmaschine M an mit $\overline{L(M)} \notin \text{DSPACE}(s_1)$. Damit folgt aus der Komplementabgeschlossenheit von $\text{DSPACE}(s_2)$, daß $\overline{L(M)} \in \text{DSPACE}(s_2) \setminus \text{DSPACE}(s_1)$.

Auf eine Eingabe der Länge n markiert M zunächst den Platz $s_2(n)$ auf seinem Arbeitsband. Sobald M ab jetzt den markierten Bereich verläßt, stoppt M. Damit ist M s_2-platzbeschränkt. M arbeitet nun wie eine universelle Turingmaschine. Auf eine Eingabe $y \in \Sigma^*$ simuliert M die von y kodierte Turingmaschine M_y auf der Eingabe y. M akzeptiert genau dann, wenn während der Simulation der markierte Bandbereich nicht verlassen und y von M_y akzeptiert wird. Um M_y zu simulieren, kopiert M den wesentlichen Anfangsteil x von y ($y = x\mathsf{I}^n$ für ein $n \in \mathbb{N}$) auf eine Spur des Arbeitsbandes. Auf einer anderen Spur des Arbeitsbandes merkt sich M die aktuelle Konfiguration von M_y mit Ausnahme der Kopfposition, die direkt auf dem Eingabeband simuliert werden kann. Um sich eine Konfiguration von M_y der Länge l zu merken, benötigt M damit $k \cdot (l+1)$ Platz für ein $k \in \mathbb{N}$, das nur von x bzw. $M_x = M_y$ abhängt.

Wir zeigen nun $\overline{L(M)} \notin \text{DSPACE}(s_1)$ durch Widerspruch: Sei $\overline{M}$ eine s_1-platzbeschränkte deterministische 1-Band Turingmaschine, die $\overline{L(M)}$ akzeptiert. Sei x eine Kodierung von $\overline{M}$ (z.B die kürzeste). Die Simulation von $\overline{M}$ auf eine Eingabe der Länge n benötigt damit höchstens $k \cdot (s_1(n) + 1)$-Platz für ein nur von x abhängiges k. Wähle nun $n \geq |x|$ so, daß $k \cdot (s_1(n) + 1) \leq s_2(n)$. Dies ist möglich, denn $s_1 \notin \Omega(s_2)$ und $s_2 \notin \mathcal{O}(1)$. Eine Simulation von $\overline{M}$ auf eine Eingabe der Länge n — z.B. $y = x\mathsf{I}^{n-|x|}$ — verläßt den markierten Bereich also nicht. Es gilt damit: M akzeptiert die Eingabe y genau dann, wenn $M_y = \overline{M}$ die Eingabe y akzeptiert. Aus $L(\overline{M}) = \overline{L(M)}$ ergibt sich der Widerspruch. ■

Bemerkung: in [Sip80] hat M. Sipser die Komplementabgeschlossenheit aller deterministischen Platzklassen gezeigt. In obigem Satz könnte also die Bedingung $s_2 \in \Omega(\log)$ durch $s_2 \notin \mathcal{O}(1)$ ersetzt werden.

Gemäß der Technik von Hennie und Stearns [HS66] ist die Simulation von beliebig vielen Bändern Zeiteffizient auf zwei Bändern möglich. Mit dem Abschluß von $\text{DTIME}(f)$ unter Komplementbildung ergibt sich analog der deterministische Zeithierarchiesatz.

Satz 2.3.3 (Deterministischer Zeithierarchiesatz, ohne Beweis)
Seien $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ *Funktionen,* $t_1 \cdot \log(t_1) \notin \Omega(t_2)$, $t_2 \in \Omega(n \log(n))$ *und* t_2 *sei zeitkonstruierbar. Dann gilt* $\text{DTIME}(t_2) \setminus \text{DTIME}(t_1) \neq \emptyset$.

Als weitere Folgerung hieraus ergibt sich:

$$\begin{aligned} &\mathrm{DTIME}(\mathcal{O}(n)) \subsetneq \mathrm{DTIME}(\mathcal{O}(n^2)) \subsetneq \mathbf{P} \\ \subsetneq\ &\mathrm{DTIME}(\mathcal{O}(2^n)) \subsetneq \mathrm{DTIME}(\mathcal{O}((2+\varepsilon)^n)) \end{aligned}$$

Bei den oben erwähnten Hierarchiesätzen haben wir stets eine Konstruierbarkeitsvoraussetzung mitgeführt. Dies läßt sich nach dem folgenden Lückensatz [Bor72] nicht umgehen.

Satz 2.3.4 (Borodin, 1972) *Sei r eine totale, berechenbare Funktion, $r(n) \geq n$ für alle n. Dann existiert effektiv eine totale, berechenbare Funktion $s : \mathbb{N} \to \mathbb{N}$ mit der Eigenschaft*

$$\mathrm{DTIME}(s) = \mathrm{DTIME}(r \circ s)\,.$$

Beweis: Sei $M_1, M_2, \ldots$ eine Aufzählung aller Turingmaschinen, und sei $t_i(n)$ der tatsächliche maximale Zeitbedarf einer Rechnung von M_i auf einer Eingabe der Länge n. Es gilt: $t_i(n) \in \mathbb{N} \cup \{\infty\}$.

Betrachte nun die Menge $N_n = \{t_k(n) | 1 \leq k \leq n\} \subseteq \mathbb{N} \cup \{\infty\}$. Diese Menge ist endlich, also existiert folglich eine Zahl s_n und ein Intervall $[s_n, r(s_n)]$ (eine *Lücke*) mit $N_n \cap [s_n, r(s_n)] = \emptyset$. Ein s_n, welches diese Bedingung erfüllt, ist etwa $s_n = \max\{t_k(n) \,|\, 1 \leq k \leq n,\, t_k(n) < \infty\}$. Dieser Wert wird jedoch i. allg. zu groß (und auch unberechenbar) sein. Einen passenden und berechenbaren Wert s_n finden wir auf die Eingabe n wie folgt:

```
FUNCTION s(n)
  s := 0
  repeat
    s := r(s) + 1
  until ∀k ≤ n : [t_k(n) < s or t_k(n) > r(s)]
  return s
ENDFUNC
```

Bemerkung 2.3.5 Die Funktion s wächst zunächst monoton. Im allgemeinen kann s jedoch nicht platzkonstruierbar sein.

Man zeigt jetzt, daß $\mathrm{DTIME}(s) = \mathrm{DTIME}(r \circ s)$ gilt:

Da $r(n) \geq n$ für alle n, ist $\mathrm{DTIME}(s) \subseteq \mathrm{DTIME}(r \circ s)$ klar.

Sei $L \in \mathrm{DTIME}(r \circ s)$, und sei M_k eine $(r \circ s)$-zeitbeschränkte deterministische Turingmaschine mit $L = L(M_k)$. Dann gilt: $\forall n : t_k(n) \leq r(s(n))$. Für alle $n \geq k$ gilt daher nach Berechnung von s: $t_k(n) < s(n)$. Damit gilt $t_k(n) \leq s(n)$ für fast alle n, und schließlich $L \in \mathrm{DTIME}(s)$. ∎

Korollar 2.3.6 *Es gibt berechenbare Funktionen s mit*

$$\mathrm{DTIME}(s) = \mathrm{NSPACE}(s) = \mathrm{DTIME}(2^{\mathcal{O}(s)}) = \mathrm{NSPACE}(2^{2^{2^{2^{2^{s}}}}}) .$$

Der Lückensatz 2.3.4 von Borodin ist auf den ersten Blick sehr überraschend. Naiv könnte man dem Irrtum verfallen, daß beispielsweise $\mathrm{DTIME}(2^s)$ für jede Funktion $s : \mathbb{N} \to \mathbb{N}$ mächtiger als $\mathrm{DTIME}(s)$ ist.

2.4 Der Komplementabschluß nichtdeterministischer Platzklassen

Bereits im Jahre 1964 stellte Kuroda die Frage, ob die Familie der kontextsensitiven Sprachen unter Komplementbildung abgeschlossen sei (2. LBA-Problem). In der Terminologie der Komplexitätstheorie ist dies die Frage, ob $\mathrm{NSPACE}(n) = \mathbf{co}\text{-}\mathrm{NSPACE}(n)$ gilt. Sie wurde nach mehr als zwanzig Jahren durch zwei unabhängige Arbeiten [Sze88, Imm88] positiv gelöst.

Satz 2.4.1 (Szelepcsényi, Immerman, 1987) *Sei $f \in \Omega(\log(n))$, dann gilt*

$$\mathrm{NSPACE}(f) = \mathbf{co}\text{-}\mathrm{NSPACE}(f) .$$

Beweis: Die verwendete Beweistechnik wird durch induktives nichtdeterministisches Zählen beschrieben. Wir führen dies näher aus und benötigen einige Bezeichnungen. Sei M eine nichtdeterministische f-platzbeschränkte Turingmaschine und $w \in \Sigma^*$ ein Eingabewort. Mit α, α', β, $\ldots$ bezeichnen wir Konfigurationen. Wir nehmen eine längenlexikographische Ordnung $\leq$ auf der Menge der Konfigurationen an. Sei α_0 die kleinste Konfiguration bzgl. $\leq$. Wir können zugleich annehmen, daß α_0 die einzige akzeptierende Konfiguration der Maschine M ist. Es kommt also darauf an, festzustellen, ob α_0 von der Startkonfiguration $\mathrm{Start}(w)$ aus erreichbar ist. Mit $R(*)$ bezeichnen wir die Menge aller erreichbaren Konfigurationen und mit $r(*)$ deren Anzahl:

$$\begin{aligned} R(*) &= \{\alpha \mid \alpha \text{ ist von } \mathrm{Start}(w) \text{ aus erreichbar}\}, \\ r(*) &= |R(*)|. \end{aligned}$$

Für $k \geq 0$ definieren wir analog:

$$\begin{aligned} R(k) &= \{\alpha \mid \alpha \text{ ist von } \mathrm{Start}(w) \text{ in höchstens } k \text{ Schritten erreichbar}\}, \\ r(k) &= |R(k)|. \end{aligned}$$

Da wir keine Konstruierbarkeitsvoraussetzung an f stellen wollen, benötigen wir für $k \geq 0$ noch den Wert

$$m(k) \quad = \quad \max\{|\alpha| \mid \alpha \in R(k)\}.$$

Die Berechnung von $r(*)$ ergibt sich induktiv wie folgt (unter der Annahme, daß jedes $r(k)$ berechenbar ist):

```
FUNCTION r(*)
   k := 0
   while r(k) < r(k + 1) do
      k := k + 1
   endwhile
   return r(k)
ENDFUNC
```

Die Berechnung von $r(k+1)$ aus $r(k)$ erfolgt in drei Schritten, die alle nach demselben Schema verlaufen. Der Erfolg der Rechnungen ist teilweise nur möglich, falls w nicht von M akzeptiert wird.

Zunächst berechnen wir den Wert $m(k)$ mit Hilfe des Wertes $r(k)$:

```
FUNCTION m(k)
   α := α₀
   m := |α₀|
   repeat r(k) times
      berechne ein beliebiges α' ∈ R(k)
      if α' ≤ α then
         „FEHLER“ ⇒ Programmabbruch
      else
         α := α'
         m := |α'|    (* = max{m, |α'|} aufgr. der Ordnung *)
      endif
   endrepeat
   return m
ENDFUNC
```

Man beachte: Angenommen α_0 ist in k Schritten von Start(w) aus erreichbar (w gehört also zu $L(M)$), dann muß dieses Programmstück mit „Fehler“ abbrechen. Es stehen nämlich in $R(k)$ nicht genügend Konfigurationen zur Verfügung, die alle echt größer als α_0 sind. Damit läßt sich $m(k)$ nicht berechnen und dies führt zum Abbruch der Berechnung von $r(*)$ (falls $w \in L(M)$).

Sei jetzt β eine beliebige Konfiguration. Es läßt sich mit Hilfe des Wertes $r(k)$ nichtdeterministisch testen, ob $\beta \in R(k+1)$ gilt oder nicht:

```
FUNCTION Erreichbarkeit(k + 1, β)
  bool := FALSE
  α := α₀
  repeat r(k) times
    berechne ein beliebiges α' ∈ R(k)
    if β ist von α' aus in höchstens einem Schritt erreichbar then
      bool := TRUE
    endif
    if α' ≤ α then
      „FEHLER“ ⇒ Programmabbruch
    else
      α := α'
    endif
  endrepeat
  if bool = TRUE then
    return 'β ∈ R(k + 1)'
  else
    return 'β ∉ R(k + 1)'
  endif
ENDFUNC
```

Es verbleibt, den Wert $r(k+1)$ zu berechnen. Wir wissen $r(0) = 1$ und können induktiv annehmen, daß $r(k)$ und $m(k)$ bereits bekannt sind. Wir betrachten nur Konfigurationen β mit $|\beta| \leq m(k)+1$, da $m(k+1) \leq m(k)+1$ immer erfüllt ist.

```
FUNCTION r(k + 1)
  r := 0
  forall β mit |β| ≤ m(k) + 1 do
    if β ∈ R(k + 1) then
      r := r + 1
    endif
  endforall
  return r
ENDFUNC
```

Eine erfolgreiche Berechnung von $r(*)$ ist genau dann möglich, wenn das Eingabewort w nicht von M akzeptiert wird. Man beachte dazu: Wenn w von M akzeptiert wird, so hält *jede* Rechnung mit einer Fehlermeldung, da die Funktion

$m(k)$, sobald sie die akzeptierende Konfiguration α_0 in $R(k)$ vorfindet, nicht mehr erfolgreich durchlaufen werden kann. Sowie also der Wert $r(*)$ berechnet wurde, kann w als zum Komplement gehörig akzeptiert werden.

Da für ein $c > 0$, das von M abhängt, $R(c^{f(n)+\log(n)}) = R(*)$ für Eingaben der Länge n, wird $r(k)$ maximal für $k = 0, \dots, c^{f(n)+\log(n)} + 1$ berechnet. Um $r(k)$ zu berechnen, benötigt man zusätzlich zur Speicherung von k noch $\mathcal{O}(f(n))$ Platz. Da $k \leq c^{f(n)+\log(n)} + 1$ in $\mathcal{O}(f + \log) = \mathcal{O}(f)$ Platz gespeichert werden kann, ist $\overline{L} \in \mathrm{NSPACE}(f)$. ■

2.5 Die Translationstechnik

Mit Hilfe des Translationssatzes läßt sich eine Enthaltenseinsbeziehung zwischen kleinen Komplexitätsklassen in eine Beziehung zwischen großen Klassen überführen. Die Idee ist das Ausstopfen (Padding) einer Sprache.

Definition 2.5.1 Sei $L \subseteq \Sigma^*$ eine Sprache, $f : \mathbb{N} \to \mathbb{N}$ eine Funktion mit $f(n) \geq n$ für alle $n \geq 0$ und $\$ \notin \Sigma$ ein neues Symbol. Definiere eine Sprache $\mathrm{Pad}_f(L)$ über dem erweiterten Alphabet $\Sigma \cup \{\$\}$ durch

$$\mathrm{Pad}_f(L) = \{w\$^{f(|w|)-|w|} \mid w \in L\}.$$

Man beachte, daß auf diese Weise jedem Wort aus L mit Länge n ein Wort aus $w\$^*$ der Länge $f(n)$ zugeordnet wird.

Satz 2.5.2 *Seien g und f, $f(n) \geq n$, $g(n) \geq n$ für alle $n \geq 0$, zwei zeitkonstrierbare Funktionen. Für $L \subseteq \Sigma^*$ gilt dann:*

1. $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(\mathcal{O}(g)) \iff L \in \mathrm{DTIME}(\mathcal{O}(g \circ f))$,
2. $\mathrm{Pad}_f(L) \in \mathrm{NTIME}(\mathcal{O}(g)) \iff L \in \mathrm{NTIME}(\mathcal{O}(g \circ f))$.

Beweis: Der Beweis wird hier für DTIME geführt; der Beweis für NTIME verläuft analog.

„⇒“: Sei $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(\mathcal{O}(g))$. Auf Eingabe $w \in \Sigma^*$ berechne das Wort $w\$^{f(|w|)-|w|}$ in der Zeit $f(|w|)$. Teste jetzt in $\mathrm{DTIME}(g(f(|w|)))$, ob $w\$^{f(|w|)-|w|}$ zu $\mathrm{Pad}_f(L)$ gehört. Nach Definition gilt

$$w\$^{f(|w|)-|w|} \in \mathrm{Pad}_f(L) \iff w \in L.$$

Also folgt $L \in \mathrm{DTIME}(\mathcal{O}(g \circ f))$.

„$\Leftarrow$“: Sei $L \in \mathrm{DTIME}(\mathcal{O}(g \circ f))$. Auf eine Eingabe $x \in (\Sigma \cup \{\$\})^*$ teste zunächst, ob $x \in w\$^*$ für ein $w \in \Sigma^*$ gilt. Sei $n = |w|$ und $m = |x|$. Berechne $g(m)$ in der Zeit $g(m)$ und lasse für die weiteren Berechnungen eine „Uhr“ mitlaufen, die die Berechnung verwerfend anhält, falls die Zeit $c \cdot g(m)$ überschritten wird, wobei $c > 1$ eine geeignete Konstante sei. Für korrekte Eingaben $x \in Pad_f(L)$ wird diese Zeit reichen, um zu akzeptieren. Teste jetzt, ob $m = f(n)$ gilt. Falls $m = f(n)$ ist diese Berechnung in m Schritten möglich, da f zeitkonstruierbar ist. Jetzt teste in der Zeit $c \cdot g(f(n)) = c \cdot g(m)$, ob $w \in L$ gilt. Dies zeigt: $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(\mathcal{O}(g))$. ∎

Bemerkung 2.5.3 Ein analoges Resultat (mit entsprechend veränderten Voraussetzungen für f, g) gilt auch für DSPACE und NSPACE:

Satz 2.5.4 *(ohne Beweis) Sei $g \in \Omega(\log(n))$, $f(n) \geq n$ für alle $n \geq 0$. Auf eine unäre Eingabe 1^n sei der binär codierte Wert von $f(n)$ in $\mathrm{DSPACE}(g \circ f)$ berechenbar. Für $L \subseteq \Sigma^*$ gilt dann:*

1. $L \in \mathrm{DSPACE}(g \circ f) \iff \mathrm{Pad}_f(L) \in \mathrm{DSPACE}(g)$,
2. $L \in \mathrm{NSPACE}(g \circ f) \iff \mathrm{Pad}_f(L) \in \mathrm{NSPACE}(g)$.

Der Zusammenfall einer Hierarchie von Komplexitätsklassen ist daher am ehesten weit oben zu erwarten. Wollen wir Separationsresultate zeigen, so bestehen hierfür die besten Aussichten am unteren Ende einer Hierarchie.

Korollar 2.5.5 $\mathrm{DSPACE}(n) \neq \mathrm{NSPACE}(n) \implies \mathbf{L} \neq \mathbf{NL}$.

Beweis: Angenommen $\mathbf{L} = \mathbf{NL}$. Sei $L \in \mathrm{NSPACE}(n)$. Dann gilt $\mathrm{Pad}_{\exp}(L) \in \mathrm{NSPACE}(\log) = \mathbf{NL} = \mathbf{L} = \mathrm{DSPACE}(\log)$. Aus Satz 2.5.4 ergibt sich $L \in \mathrm{DSPACE}(\exp \circ \log) = \mathrm{DSPACE}(n)$. ∎

Mit Hilfe der Translationstechnik läßt sich in einigen Fällen die Verschiedenheit von Komplexitätsklassen nachweisen.

Korollar 2.5.6 *Sei $\mathbf{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k)$ die Klasse der polynomial berechenbaren Probleme. Dann gilt $\mathbf{P} \neq \mathrm{DSPACE}(n)$.*

Beweis: Wir wählen eine Sprache $L \in \mathrm{DSPACE}(n^2) \setminus \mathrm{DSPACE}(n)$ und die Padding–Funktion $f(n) = n^2$. Dann gilt $\mathrm{Pad}_f(L) \in \mathrm{DSPACE}(n)$. Wäre nun $\mathrm{DSPACE}(n) = \mathbf{P}$, so wäre $\mathrm{Pad}_f(L) \in \mathrm{DTIME}(n^k)$ für ein $k \geq 1$ und $L \in \mathrm{DTIME}(n^{2k}) \subseteq \mathbf{P} = \mathrm{DSPACE}(n)$. Dies ist ein Widerspruch. ∎

Bemerkung 2.5.7

- Insbesondere gilt, daß $\mathbf{P}$ nicht gleich der Sprachklasse der kontextsensitiven Sprachen ist.
- Sowohl $\mathrm{DSPACE}(\log(n)) = \mathbf{P}$, $\mathrm{DSPACE}(n) \subset \mathbf{P}$ oder $\mathbf{P} \subset \mathrm{DSPACE}(n)$ sind nach heutigem Wissen möglich.

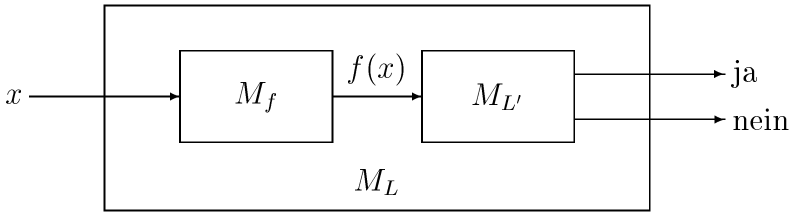
Kapitel 3

Reduktionen und vollständige Probleme

3.1 Polynomialzeitreduktionen

Seien $L \subseteq \Sigma^*$ und $L' \subseteq \Sigma'^*$ zwei (Entscheidungs-)Probleme. Unter einer Reduktion des Problems L auf das Problem L' verstehen wir eine totale berechenbare Abbildung $f : \Sigma^* \to \Sigma'^*$ mit der Eigenschaft $x \in L \iff f(x) \in L'$.

Angenommen, wir kennen bereits einen Algorithmus zur Lösung von L'. Dann können wir das Problem L lösen, indem wir zunächst für eine Eingabe $x \in \Sigma^*$ den Wert $f(x)$ berechnen und danach mit dem Algorithmus für L' entscheiden, ob $f(x) \in L'$ gilt. Das Verfahren ist unten bildlich dargestellt. Ist zudem die Reduktion f leicht zu berechnen und das Problem L' effizient zu lösen, so ist auch L effizient zu lösen.



Wir sprechen von einer Polynomialzeitreduktion, wenn sich f durch eine deterministische polynomialzeitbeschränkte Turingmaschine berechnen läßt.

Sei beispielsweise $L' \in \mathrm{DTIME}(n^2)$ und L auf L' mittels einer Abbildung f reduzierbar, die in kubischer Zeit berechenbar ist, dann liegt L in $\mathrm{DTIME}(n^6)$. Dies ergibt sich wie folgt: In jedem Schritt zur Berechnung von $f(x)$ kann höchstens ein Ausgabezeichen produziert werden. Damit gilt $|f(x)| \leq |x|^3$. Der Algorithmus für L' löst nach der Annahme das Problem „$f(x) \in L'$?“ in quadratischer Zeit, benötigt also höchstens die Zeit $|f(x)|^2 \leq (|x|^3)^2 = |x|^6$. Die Gesamtlaufzeit kann durch $\mathcal{O}(|x|^3 + |x|^6)$ abgeschätzt werden. Da $\mathrm{DTIME}(n^6) = \mathrm{DTIME}(\mathcal{O}(n^6))$ gilt, folgt die Behauptung.

Reduktionen können also für den Entwurf effizienter Algorithmen benutzt werden, insbesondere wenn eine Bibliothek zur Lösung von Standardproblemen zur Verfügung steht.

Beispiel 3.1.1 Maximum Matching:
Sei $G = (A, B, E)$ ein bipartiter Graph, d.h., $V = A \dot{\bigcup} B$ und $E \subseteq A \times B$. Ein *Matching* M ist eine Teilmenge $M \subseteq E$ so, daß keine zwei Kanten aus M einen gemeinsamen Endknoten haben. Wir zeigen, wie das Problem, ein Matching maximaler Größe zu berechnen, sehr effizient auf die Berechnung eines Maximalflusses in einem Netzwerk reduziert werden kann.

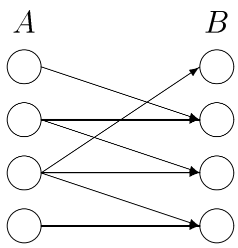
Ein Netzwerk sei ein gerichteter Graph $G = (V, E)$ mit einer Quelle $s \in V$ und einer Senke $t \in V$, wobei $s \neq t$. Jede Kante $e \in E$ hat eine Kapazität $c(e) > 0$. Ein Fluß F ist eine Abbildung $F : E \to \mathbb{N}$ so, daß für alle $v \in V \setminus \{s, t\}$ und $e \in E$ gilt:

- $\sum_{(x,v)\in E} F(x, v) = \sum_{(v,y)\in E} F(v, y)$ (Flußerhaltung), und
- $F(e) \leq c(e)$.

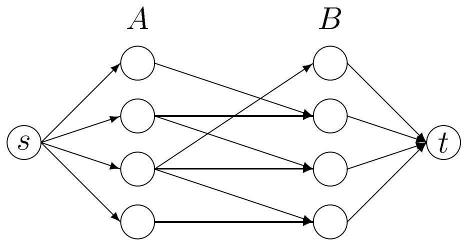
Es ist bekannt, daß ein Fluß maximaler Größe in polynomialer Zeit berechnet werden kann. Dies hängt mit dem bekannten Max-Flow=Min-Cut-Theorem von Ford-Fulkerson zusammen und wird hier nicht behandelt. Die Reduktion von Maximum Matching auf Max-Flow startet mit einem bipartiten Graphen $G = (A, B, E)$. Wir lesen E als gerichtete Kanten von A nach B. Wir nehmen zwei neue Knoten s und t hinzu, sowie Kanten (s, a) und (b, t) für alle $a \in A$, $b \in B$. Alle Kanten erhalten die Kapazität 1. Damit haben wir ein Netzwerk definiert und können einen Fluß F: $E \cup (\{s\} \times A) \cup (B \times \{t\}) \to \mathbb{N}$ maximaler Größe berechnen. Schließlich erhalten wir das Maximum Matching $M \subseteq A \times B$ durch $M = \{e \in E \mid F(e) = 1\}$. Dieses Vorgehen wird in Abb. 3.1 an einem Beispielgraphen durchgeführt.

Reduktionen stellen auch ein wichtiges Hilfsmittel dar, um zu Aussagen über untere Schranken zu gelangen. Dies führt auf den Begriff der **NP**-Vollständigkeit und die zentrale Frage der Komplexitätstheorie, ob **P** = **NP** gilt. S. Cook hat in einer bahnbrechenden Arbeit [Coo71] gezeigt, daß sich jedes Problem aus **NP** in polynomialer Zeit auf das Erfüllbarkeitsproblem aussagenlogischer Ausdrücke SAT reduzieren läßt. Wir gehen hierauf in Abschnitt 4.2 näher ein.

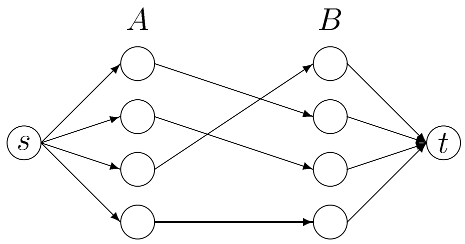
Sei $G = (A, B, E)$ ein bipartiter Graph:



Füge eine Quelle s und eine Senke t an:



Berechne einen maximalen Fluß:



Hieraus ergibt sich die Lösung des Matchingproblems:

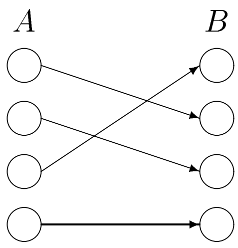


Abbildung 3.1: Maximum-Matching

3.2 Reduktion in logarithmischem Platz

Viele praktisch wichtige Reduktionen lassen sich in logarithmischem Platz berechnen. Dies führt auf den Begriff der logspace-Reduktion, den wir im weiteren verwenden. Er läßt sich auch auf Klassen unterhalb von **P** anwenden und erlaubt eine feinere Einteilung als unter Verwendung von polynomialen Zeitreduktionen.

Definition 3.2.1

1. Ein logarithmisch platzbeschränkter Transduktor (logspace-Transducer) ist eine deterministische Turingmaschine M mit einem Eingabeband (Zweiweg-nur-lesend), einem logarithmisch in der Eingabelänge platzbeschränkten Arbeitsband und einem separaten Ausgabeband (Einweg-nur-schreibend). In jedem Rechenschritt von M darf höchstens ein Zeichen auf das Ausgabeband geschrieben werden.

2. Eine Abbildung $f : \Sigma^* \to \Sigma'^*$ heißt logspace berechenbar, falls es einen logspace-Transducer M gibt, der auf jede Eingabe $x \in \Sigma^*$ hält und nach dem Halten den Wert $f(x) \in \Sigma'^*$ auf dem Ausgabeband stehen hat.

3. Ein Problem $L \subseteq \Sigma^*$ heißt logspace reduzierbar auf $L' \subseteq \Sigma'^*$, falls es eine logspace berechenbare Abbildung $f : \Sigma^* \to \Sigma'^*$ gibt mit $x \in L \iff f(x) \in L'$. In Zeichen schreiben wir hierfür $L \leq_m^{\log} L'$, der untere Index m steht hier für many-one[1].

Bemerkung 3.2.2

1. Jede logspace berechenbare Abbildung $f : \Sigma^* \to \Sigma'^*$ ist in polynomialer Zeit berechenbar. Insbesondere gibt es ein $k \geq 0$ so, daß $|f(x)| \leq |x|^k$ für alle $x \in \Sigma^*$.

2. Die Relation $\leq_m^{\log}$ ist transitiv, d.h., mit $L \leq_m^{\log} L' \leq_m^{\log} L''$ gilt auch $L \leq_m^{\log} L''$. Die analoge Aussage für polynomiale Zeitreduktion ist trivial. Bei der Hintereinanderausführung von logspace berechenbaren Reduktionen $f : \Sigma^* \to \Sigma'^*$ und $g : \Sigma'^* \to \Sigma''^*$ gibt es Probleme. Auf eine Eingabe $w \in \Sigma^*$ mit $|w| = n$ hat die Ausgabe $f(w)$ höchstens die Länge n^k, k fest. Die Berechnung von g auf $f(w)$ erfordert damit den Arbeitsplatz $\mathcal{O}(\log(n^k)) = \mathcal{O}(\log(n))$. Das Problem besteht darin, daß es keinen Platz gibt, um $f(w)$ auf das Arbeitsband zu schreiben. Eine Lösung dafür findet sich, wie folgt: Wenn man in der Berechnung von g das $i-$te Bit von $f(w)$ benötigt, wird die Berechnung von f auf w neu gestartet und mit Hilfe eines Binärzählers nur das $i-$te Bit ausgegeben. Man beachte,

[1] many-one bedeutet, daß mehrere Worte aus Σ^* auf das selbe Wort in Σ'^* abgebildet werden können.

daß dieser Binärzähler nur logarithmischen Platz benötigt. Nun kann ein Rechenschritt für g simuliert werden. Damit erfordert die Berechnung von $g(f(w))$ nur den Platz $\mathcal{O}(\log(n))$, d.h. die Funktion $g \circ f$ ist logspace berechenbar. Trivialerweise gilt: $w \in L \iff g(f(w)) \in L''$, womit insgesamt die Transitivität von $\leq_m^{\log}$ gezeigt ist.

3. Logspace- und polynomiale Zeit-Reduktionen sind genau dann gleichmächtig, wenn $\mathbf{L} = \mathbf{P}$ gilt.

Beispiel 3.2.3 Sei $f(n) = n^k$. Dann ist f logspace berechenbar. Also gilt mit Hilfe der Reduktionsabbildung $w \mapsto w\$^{|w|^k - |w|}$, daß $L \leq_m^{\log} \mathrm{Pad}_f(L)$. Umgekehrt gilt $\mathrm{Pad}_f(L) \leq_m^{\log} L$ für $L \neq \Sigma^*$.

3.3 Vollständige Probleme

Definition 3.3.1

1. Sei $\mathcal{C}$ eine Komplexitätsklasse. Ein Problem $L \subseteq \Sigma^*$ heißt *hart* für $\mathcal{C}$ oder kurz $\mathcal{C}$-hart (bzgl. logspace-Reduktionen), falls sich jedes Problem aus $\mathcal{C}$ mittels einer logspace berechenbaren Abbildung auf L reduzieren läßt, d.h., $\forall K \in \mathcal{C} : K \leq_m^{\log} L$.

2. Sei $\mathcal{C}$ eine Komplexitätsklasse. Ein Problem $L \subseteq \Sigma^*$ heißt *$\mathcal{C}$-vollständig* (bzgl. logspace-Reduktionen), falls L hart für $\mathcal{C}$ ist und zusätzlich $L \in \mathcal{C}$ gilt.

Wir geben ein erstes Beispiel:

Satz 3.3.2 *Das Grapherreichbarkeitsproblem ist* **NL**-*vollständig.*

Beweis: Das Enthaltensein des Grapherreichbarkeitsproblems in **NL** ist klar. Sei $L \in \mathbf{NL}$, und sei M eine nichtdeterministische logspace-platzbeschränkte Turingmaschine mit $L = L(M)$. Die Reduktion f berechnet jetzt auf eine Eingabe w den Graph der möglichen Konfigurationen. Knoten sind alle Konfigurationen mit Länge $\leq \log(n)$, falls $n = |w|$. Jede solche Konfiguration wird in $c \cdot \log(n)$ Platz repräsentiert, für ein $c > 0$, welches nur von M abhängt.

Die Kanten sind Paare (α, β), wenn M in einem Schritt von der Konfiguration α in die Konfiguration β übergehen kann. Setzte $s = \mathrm{Start}(w)$ und $t = \mathrm{Accept}$. Auf eine Eingabe hin werden also ein Graph $G(w)$ sowie zwei Knoten s und t berechnet, und es gilt:

$$w \in L(M) \iff \text{in } G(w) \text{ gibt es einen gerichteten Pfad von } s \text{ nach } t\,.$$

Damit haben wir eine logspace berechenbare Reduktion, was schließlich die **NL**-Vollständigkeit beweist. ∎

Kapitel 4

NP-Vollständigkeit

In diesem Abschnitt wollen wir uns mit **NP**-vollständigen Problemen beschäftigen. Bevor wir auf den in der Komplexitätstheorie zentralen Satz von Cook eingehen, betrachten wir zunächst zwei einfachere Sätze.

4.1 Einführende Betrachtungen

Satz 4.1.1 *Falls es überhaupt eine* **NP***-vollständige Sprache gibt, so auch eine in* NTIME(n)*:*

$$\exists L : L \textit{ ist } \mathbf{NP}\textit{-vollständig} \Rightarrow \exists \tilde{L} : \tilde{L} \in \mathrm{NTIME}(n) \wedge \tilde{L} \textit{ ist } \mathbf{NP}\textit{-vollständig.}$$

Beweis: Sei L ein **NP**-vollständiges Problem, dann existiert ein Polynom $p(n) = n^k, k > 0$ mit $L \in \mathrm{NTIME}(n^k)$, daher muß nach Satz 2.5.2 $\mathrm{Pad}_{p(n)}(L)$ in NTIME(n) liegen. Nun ist $L' \leq_m^{\log} L \leq_m^{\log} \mathrm{Pad}_{p(n)}(L)$ für alle $L' \in \mathbf{NP}$. Folglich muß mit der Transitivität von $\leq_m^{\log}$ auch $L' \leq_m^{\log} \mathrm{Pad}_{p(n)}(L)$ gelten. Also ist $\mathrm{Pad}_{p(n)}(L)$ **NP**-vollständig. ∎

Der Satz erscheint überraschend, besagt er doch, daß falls es **NP**-vollständige Probleme gibt, sich solche auch bereits ganz unten in der echten Zeithierarchie von **NP** finden lassen.

Daß es tatsächlich **NP**-vollständige Probleme gibt, zeigt der folgende Satz, der zwar leicht zu beweisen ist, aber leider ein für die Praxis uninteressantes Problem, welches einen modifizierten Interpreter darstellt, zum Gegenstand hat.

Sei U eine universelle Turingmaschine, $\langle w, M\rangle$ eine Codierung eines Wortes $w \in \Sigma^*$ und einer nichtdeterministischen Turingmaschine M. Dann simuliert U bei Eingabe $\langle w, M\rangle$ Rechnungen von M auf w. Betrachte jetzt

$$L_{Gen} = \{\langle w, M, \mathsf{I}^m \rangle \mid w \in \Sigma^*, M \text{ nichtdeterministische Turingmaschine}, m \in \mathbb{N} \text{ und } U \text{ erkennt in einer Simulation, die höchstens } m \text{ Schritte dauert, daß } w \in L(M) \text{ gilt.}\}$$

Satz 4.1.2 *L_{Gen} ist* **NP***-vollständig.*

Beweis: Offensichtlich gilt $L_{Gen} \in \mathrm{NTIME}(n)$, also auch $L_{Gen} \in \mathbf{NP}$. Somit müssen wir nur noch zeigen, daß L_{Gen} **NP**-hart ist. Sei hierzu $L \in \mathbf{NP}$ beliebig und M eine polynomial zeitbeschränkte Turingmaschine mit $L = L(M)$. Man wähle $p(n) = n^k + c$ so, daß U auf die Eingabe $\langle w, M \rangle$ höchstens $p(|w|)$ Schritte rechnet, wobei zu beachten ist, daß U zur Simulation etwas mehr Zeit als M selbst benötigen kann, diese aber immer noch polynomial in der Eingabelänge ist. Die Reduktion von L auf L_{Gen} berechnet nun in logarithmischem Platz auf eine Eingabe $w \in \Sigma^*$ die Ausgabe $f(w) = \left\langle w, M, \mathsf{I}^{|w|^k} \right\rangle$. Es gilt: $w \in L(M) \iff f(w) \in L_{Gen}$. ∎

4.2 Satz von Cook

Sei Σ_0 das Alphabet $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, 0, 1, (,), x\}$. Mit einer kontextfreien Grammatik kann man die formale Sprache $\mathbb{A}$ der aussagenlogischen Ausdrücke über der Variablenmenge $V = x1\{0,1\}^*$ definieren. Die Sprache $\mathbb{A} \subseteq \Sigma_0^*$ ist deterministisch kontextfrei. Damit ist in linearer Zeit entscheidbar, ob ein Wort aus Σ_0^* einen aussagenlogischen Ausdruck darstellt. Die Sprache SAT (Satisfiability) ist definiert durch

$$\mathrm{SAT} = \left\{ F \in \mathbb{A} \;\middle|\; F \text{ ist erfüllbar} \right\}.$$

Satz 4.2.1 (Cook) *Das Problem* SAT *ist* **NP***-vollständig.*

Beweis:

Es ist klar, daß SAT $\in$ **NP** gilt: Auf eine Eingabe $F \in \Sigma_0^*$ testen wir zunächst (in linearer Zeit), ob $F \in \mathbb{A}$ gilt. Danach schreiben wir die Variablen aus F heraus und belegen diese nichtdeterministisch mit 0 oder 1. Dies entspricht dem Raten einer Belegung. Schließlich wird F bzgl. dieser geratenen Belegung ausgewertet und akzeptiert, falls die Auswertung den Wert 1 ergibt.

Wir wenden uns jetzt dem schwierigen Teil zu und wollen zeigen, daß SAT hart für **NP** ist. Sei $L \in \mathbf{NP}$ die auf SAT zu reduzierende Sprache. Zu einem $w \in \Sigma^*$ konstruieren wir eine Formel $F(w)$, die genau dann erfüllbar ist, wenn $w \in L$ gilt. Die Konstruktion geschieht durch eine logspace berechenbare Abbildung $f : \Sigma^* \to \mathbb{A} \subseteq \Sigma_0^*$ mit der Eigenschaft, $w \in L \iff F(w) \in \mathrm{SAT}$. Es wird sich zeigen, daß die Länge von $F(w)$ und die Anzahl der in $F(w)$ vorkommenden Variablen im wesentlichen quadratisch in der Länge von w ist.

Da $L \in \mathbf{NP}$, existiert eine polynomial zeitbeschränkte nichtdeterministische Turingmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, \{\$\}, B)$ mit $L = L(M)$. Unsere Konstruktion hängt von M ab, man beachte aber, daß für die gewählte Sprache L, die Maschine M fest gewählt werden kann.

Wir wollen nur eine geeignet normierte polynomial zeitbeschränkte nichtdeterministische Turingmaschine M betrachten. Wir treffen die folgenden Vereinbarungen, die keine Einschränkung bedeuten. Bei M handelt es sich um eine Einband-Turingmaschine mit Startzustand q_0 und genau einem Endzustand \$. Auf eine Eingabe $w \in \Sigma^*$ macht M genau $p = p(|w|)$ Schritte, wobei $p(n)$ die polynomiale Zeitschranke ist.

Die Maschine akzeptiert w genau dann, wenn nach p Schritten eine Konfiguration aus $\$\Gamma^p$ erreicht wird. Durch Hinzunahme weiterer Zustände können wir erreichen, daß für einen Übergang $(q, a, q', a', T) \in \delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R, N\}$ die Zeichen a, a' und die Bewegungsrichtung T des Kopfes eindeutig durch $q \in Q$ festgelegt sind. Nur in dem möglichen Folgezustand q' haben wir eine nichtdeterministische Wahl. Die Einzelheiten dieser Normierung werden nicht weiter ausgeführt.

Unter dieser Voraussetzung können wir für jeden Übergang einer Konfiguration α zu α' sicherstellen, daß je drei aufeinanderfolgende Zeichen $a_{i-1}a_ia_{i+1}$ aus α das entsprechende i-te Zeichen a'_i aus α' eindeutig festlegen.

Wir erhalten damit eine endliche Tabelle Δ, die alle diese möglichen Viertupel $(a_{i-1}, a_i, a_{i+1}, a'_i) \in (Q \cup \Gamma)^4$ enthält.

Man betrachte hierfür auch die Abbildung 4.1: Die Maschine befindet sich im Zustand q und liest das Zeichen a. Links von a befindet sich das Zeichen b. Es wird die Transition (q, a, q', a', L) benutzt, wobei a, a' und L bereits durch q festliegen. Diese Transition liefert für alle $b, c, d \in \Gamma$ die folgenden Tabelleneinträge: (c, b, q, q'), (b, q, a, b), (q, a, d, a') und (b, c, d, c).

Zeichennummern:				$i-1$	i	$i+1$		
α	$=$	$\cdots$	$\cdots$	b	q	a	$\cdots$	$\cdots$
α'	$=$	$\cdots$	$\cdots$	q'	b	a'	$\cdots$	$\cdots$

Abbildung 4.1: Übergang von α zu α' mittels einer Transition $(q, a, q', a', L) \in \delta$.

Um uns eine Sonderbehandlung für den Rand zu ersparen, nehmen wir wie üblich ein unendliches Band an, das außerhalb des betrachteten Bereichs mit dem Blanksymbol B beschrieben ist.

Ferner sei das Band nach links beschränkt, d.h.,M bewegt den Kopf nie nach links über die Eingabe hinaus, was wir durch eine geeignete Faltung des Bandes erreichen können. Eine Rechnung von M können wir nun als Matrix beschreiben:

$$\begin{array}{ccccccc}
\alpha_0 & = & \alpha_{0,0} & \alpha_{0,1} & \ldots & \alpha_{0,p(n)} \\
\alpha_1 & = & \alpha_{1,0} & \alpha_{1,1} & \ldots & \alpha_{1,p(n)} \\
 & & & \vdots & & \\
\alpha_{p(n)} & = & \alpha_{p(n),0} & \alpha_{p(n),1} & \ldots & \alpha_{p(n),p(n)}
\end{array}$$

Nach diesen Vorbereitungen kommen wir jetzt zur Beschreibung der Formel. Wir gehen von einem Eingabewort $w \in \Sigma^*$ der Länge n aus und bezeichnen wie eben $p = p(n)$. Für jedes Tripel (a,i,t), $a \in Q \cup \Gamma$, $0 \leq i,t \leq p$ führen wir eine Variable $x(a,i,t)$ ein. Die Interpretation ist, daß zum Zeitpunkt t das i-te Zeichen der Konfiguration ein a ist. Wir führen zusätzliche Variablen $x(B,-1,t)$ und $x(B,p+1,t)$ für alle $0 \leq t \leq p$ ein. Die Idee ist, daß an den entsprechenden Stellen während der ganzen Rechnung ein B steht. Dies wird von folgender Formel gewährleistet, die nur von der Länge $|w|$ der Eingabe abhängt:

$$G(|w|) \quad = \quad \bigwedge_{0 \leq t \leq p} \Big(x\,(B,-1,t) \wedge x\,(B,p+1,t) \Big)$$

Die Formel X stellt sicher, daß für jedes Paar (i,t) genau eine Variable $x(a,i,t)$ wahr ist (zu jedem Zeitpunkt kann auf einem Bandfeld nur ein Zeichen stehen). Die Formel hängt wieder nur von der Länge der Eingabe ab.

$$X\,(|w|) \quad = \quad \bigwedge_{0 \leq i,t \leq p} \Bigg(\bigvee_{a \in Q \cup \Gamma} \Big(x\,(a,i,t) \wedge \bigwedge_{b \neq a} \neg x\,(b,i,t) \Big) \Bigg)$$

Wir beschreiben jetzt die Formel für die Startkonfiguration auf dem Eingabewort $w = w_1 \ldots w_n \in \Sigma^*$, $w_i \in \Sigma$:

$$S(w) \quad = \quad \Bigg(x\,(q_0,0,0) \wedge \bigwedge_{i=1}^{n} x\,(w_i,i,0) \wedge \bigwedge_{i=n+1}^{p} x\,(B,i,0) \Bigg)$$

Die Beschreibung eines korrekten Übergangs basiert auf der obigen Normierung, daß je drei nebeneinanderliegende Zeichen einer Konfiguration das sich ergebende mittlere Zeichen der Folgekonfiguration bestimmen. Dies wurde durch die Tabelle

Δ ausgedrückt. Wir erhalten damit eine Formel, die (bei festem M) nur von der Länge von w abhängt.

$$D(|w|) = \bigwedge_{0 \le i,t \le p} \bigvee_{(a,b,c,d) \in \Delta} \Big(x(a, i-1, t) \wedge x(b, i, t) \wedge x(c, i+1, t) \wedge x(d, i, t+1) \Big)$$

Sei jetzt $R(w)$ die Formel $R(w) = G(|w|) \wedge X(|w|) \wedge S(w) \wedge D(|w|)$.

Es ergibt sich eine natürliche Bijektion zwischen der Menge der $R(w)$ erfüllenden Belegungen und der Menge derjenigen Rechnungen von M auf die Eingabe w, die aus p Rechenschritten bestehen. Diese Aussage erhält man mit einem Induktionsbeweis nach der Rechenzeit t.

Der Wert der Variablen $x(\$, 0, p)$ einer erfüllenden Belegung gibt an, ob die entsprechende Rechnung erfolgreich war. Unsere Endformel lautet daher $F(w) = R(w) \wedge x(\$, 0, p)$.

Die Formel $F(w)$ ist genau dann erfüllbar, wenn $w \in L$ gilt. Die Zahl der Variablen von $F(w)$ ist in $\mathcal{O}(p^2)$ und die Länge in $\mathcal{O}(p^2 \log p)$, da das Schreiben der Indizes logarithmischen Platzbedarf benötigt. ■

Bemerkung 4.2.2 Die Formel $F(w)$ hat eine sehr regelmäßige Gestalt und sie ist bis auf innere Formeln konstanter Länge bereits in konjunktiver Normalform.

4.3 Weitere NP-vollständige Probleme

4.3.1 SAT $\cap$ KNF

Bevor wir nun Formeln in konjunktiver Normalform näher betrachten, wollen wir zunächst einige Begriffe wiederholen.

Definition 4.3.1 Ein *Literal* $\tilde{x}$ ist eine Variable oder die Negation einer Variablen. Statt $\neg x$ schreiben wir auch $\overline{x}$. Außerdem sei $\overline{\overline{x}} = x$.

Sei KNF die Menge der aussagenlogischen Ausdrücke in konjunktiver Normalform:

$$\text{KNF} = \{F \in \mathbb{A} \mid F \text{ ist Konjunktion von Disjunktionen von Literalen}\}$$

Aus der Logik ist bekannt, daß sich jede aussagenlogische Formel in eine semantisch äquivalente Formel in disjunktiver Normalform überführen läßt. So sind beispielsweise die beiden folgenden Formeln semantisch äquivalent:

$$F = \bigwedge_{i=1,\dots,k} \left(\bigvee_{j=1,\dots,m} (\tilde{x}_{i,j}) \right) \equiv F' = \bigvee_{j \in \{1,\dots,m\}^{\{1,\dots,k\}}} \left(\bigwedge_{i=1,\dots,k} \tilde{x}_{i,j(i)} \right)$$

Man beachte hierbei, daß F die Größe $m \cdot k$ hat, während F' die Größe $m^k \cdot k$ besitzt. Da für Formeln von einer Gestalt wie F' (sog. disjunktive Normalform) die Erfüllbarkeit deterministisch in quadratischer Zeit zu entscheiden ist, für Formeln in KNF jedoch gemäß dem nachfolgenden Satz $\mathbf{NP}-$vollständig ist, ist es nicht verwunderlich, daß F' superpolynomiale Länge bzgl. F besitzt, denn sonst wäre KNF$\cap$SAT polynomial auf DNF$\cap$SAT reduzierbar, womit man $\mathbf{P} = \mathbf{NP}$ bewiesen hätte.

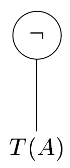
Satz 4.3.2 SAT $\cap$ KNF *ist* **NP**-*vollständig.*

Beweis: Die Zugehörigkeit zu **NP** ist trivial: Für eine Formel kann in linearer Zeit getestet werden, ob sie in KNF vorliegt und wir wissen bereits, daß SAT in **NP** liegt. Wir betrachten zwei mögliche Beweise der **NP**-Härte. Im Beweis der **NP**-Vollständigkeit von SAT haben wir eine Formel konstruiert, die bis auf innere Teilformeln konstanter Länge bereits in KNF war. Wir können also durch eine weitere logspace-berechenbare Abbildung die Formel auf die gewünschte Gestalt bringen und sind fertig.

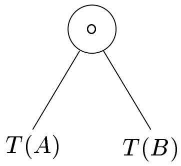
Interessanter ist aber ein direkter Beweis des Satzes. Hierzu reduzieren wir SAT auf SAT $\cap$ KNF, d.h. wir geben eine logspace-berechenbare Abbildung $f : \mathbb{A} \to$ KNF an mit: $F \in$ SAT $\iff f(F) \in$ SAT $\cap$ KNF. Wir verwandeln also eine beliebige Formel F in eine erfüllbarkeitsäquivalente Formel F' in konjunktiver Form. Wir können eine Formel $F \in \mathbb{A}$ als einen Baum $T(F)$ auffassen, der sich rekursiv folgendermaßen aufbauen läßt:

(1) Für eine Variable x sei $T(x) = x$.

(2) Ist F die Negation einer Formel A, also $F = \neg A$, so habe $T(F)$ folgende Gestalt:



(3) Besteht F aus der Verknüpfung zweier Teilformeln A, B, d.h. $F = A \circ B$ mit $\circ \in \{\Longleftrightarrow, \Rightarrow, \wedge, \vee\}$, so habe $T(F)$ folgende Gestalt:



Für die Formel $F = \Big(\big((\neg(\neg\neg x_1 \wedge x_2)) \Longleftrightarrow (x_2 \vee x_3)\big) \wedge \big(x_1 \Rightarrow (\neg x_2 \vee x_3)\big)\Big)$ erhalten wir beispielsweise einen Baum $T(F)$ wie in Abbildung 4.2 dargestellt.

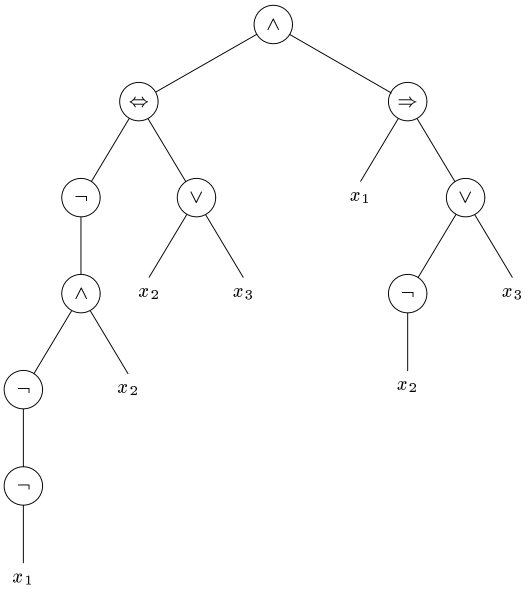


Abbildung 4.2: Baum zur Formel F

Nun ordnen wir jedem inneren Knoten von $T(F)$ eine neue Variable zu. Damit gehört zu *jedem* Knoten eine Variable. Eine mögliche Namensgebung erhalten wir, indem wir einen inneren Knoten mit der zugehörigen Teilformel A von F identifizieren und die neue Variable dann $v(A)$ nennen. Alternativ könnten wir die Knoten auch durchnumerieren, womit sich durch einfaches Zählen ein Durchlauf durch den Baum gewinnen läßt, ohne daß ein Speichern der Pfade erforderlich wäre.

Für jede Formel F bezeichnen wir die semantisch äquivalente Formel in konjunktiver Normalform mit KNF(F). Wir wenden KNF nur auf Formeln konstanter Länge an. Nun benötigen wir eine Hilfsfunktion f', die noch nicht die eigentliche Reduktion darstellt:

Definition 4.3.3 (Hilfsfunktion f') Die Hilfsfunktion $f' : \mathbb{A} \to \mathrm{SAT} \cap \mathrm{KNF}$ definieren wir rekursiv:

(1) Ist $F = x$, so sei $v(F) = x$ und $f'(F) := \mathrm{KNF}\big(v(F) \iff x\big)$. In diesem Fall ist $f'(F)$ eine triviale Tautologie.

(2) Ist $F = A \circ B$ mit $\circ \in \{\iff, \Leftarrow, \wedge, \vee\}$, so ist

$$f'(F) := \Big(\mathrm{KNF}\big(v(F) \iff (v(A) \circ v(B))\big) \wedge f'(A) \wedge f'(B)\Big)$$

(3) Ist $F = \neg A$, so ist

$$f'(F) := \Big(\mathrm{KNF}\big(v(F) \iff \neg v(A)\big) \wedge f'(A)\Big)$$

Die Überführung in konjunktive Normalform ist einfach, trotzdem wollen wir diese hier einmal explizit angeben. Nach entsprechender Umformung erhalten wir:

$$f'(F) = \Big(\big(\neg v(F) \vee v(A)\big) \wedge \big(v(F) \vee \neg v(A)\big) \wedge f'(A)\Big)$$

Mit dieser Definition ergeben sich folgende Eigenschaften von f':

Lemma 4.3.4

(1) $f'(F)$ ist stets erfüllbar.

(2) Sei σ eine erfüllende Belegung von $f'(F)$ und σ' die Restriktion von σ auf die Variablen von F. Dann gilt: $\sigma(F) = \sigma'(F) = \sigma(v(F))$.

(3) Sei $V(F)$ die Menge der in F vorkommenden Variablen und $\sigma' : V(F) \to \{0,1\}$ eine beliebige Belegung. Dann existiert eine erfüllende Belegung $\sigma : V(f'(F)) \to \{0,1\}$ mit $\sigma'(x) = \sigma(x)$ für alle Variablen $x \in V(F)$.

Nach (2) muß für die erfüllende Belegung in (3) $\sigma(v(F)) = \sigma'(F)$ gelten. Der Beweis von (2) und (3) mittels struktureller Induktion sei dem Leser zur Übung überlassen. (1) folgt unmittelbar aus (3).

Bemerkung 4.3.5 Anschaulich können wir uns den Baum $T(F)$ als Schaltnetz vorstellen, wobei die Blätter die Eingänge bilden. Die Wurzel $v(F)$ stellt den Ausgang dar und die inneren Knoten entsprechen Logikgattern. Geben wir nun eine bestimmte Belegung σ in dieses Netz ein, so wird diese ebenenweise von unten nach oben propagiert und wir erhalten am Ausgang $\sigma(F)$.

Die eigentliche Reduktion $f : \mathbb{A} \rightarrow$ KNF können wir jetzt definieren durch $f(F) := (f'(F) \wedge v(F))$, denn $f(F)$ ist erfüllbarkeitsäquivalent zu F, da gilt:

(i) Falls σ eine erfüllende Belegung von $f(F)$ ist, so muß $\sigma(v(F)) = \sigma(f'(F)) = 1$ gelten. Womit nach Lemma 4.3.4 $\sigma(F) = 1$ folgt, d.h. $F \in$ SAT.

(ii) Umgekehrt existiert eine Belegung σ' der Variablen in F mit $\sigma'(F) = 1$, falls $F \in$ SAT. Wir können σ' zu σ so fortsetzen, daß $\sigma(f'(F)) = 1$ gilt, was aber $\sigma(v(F)) = 1$ impliziert und schließlich $\sigma(f(F)) = 1$, also $f(F) \in$ SAT liefert. ∎

4.3.2 3-SAT

Selbst wenn wir noch speziellere Formeln betrachten und verlangen, daß jede Klausel genau drei Literale enthält, bleibt das Problem der Erfüllbarkeit **NP**-vollständig.

Definition 4.3.6 Sei 3-KNF die Menge der Formeln in konjunktiver Form mit genau drei Literalen je Klausel:

$$\text{3-KNF} := \{F \in \text{KNF} \mid \text{Jede Klausel in } F \text{ enthält genau drei Literale}\}$$

3-SAT sei die Teilmenge der davon erfüllbaren Formeln:

$$\text{3-SAT} := \text{3-KNF} \cap \text{SAT}$$

Korollar 4.3.7 *3-SAT ist* **NP***-vollständig.*

Beweis: Erneut ist 3-SAT $\in$ **NP** trivial: Für eine Formel kann in linearer Zeit getestet werden, ob sie in 3-KNF vorliegt und SAT $\in$ **NP**. Um zu zeigen, daß 3-SAT **NP**-hart ist, reduzieren wir SAT $\cap$ KNF in logarithmischem Platz auf 3-SAT. Dazu müssen wir drei Fälle unterscheiden:

(1) Die abzubildende Formel F enthält Klauseln $(\tilde{x})$ mit nur einem Literal. Dann führen wir jeweils eine neue Variable y ein und ersetzen $(\tilde{x})$ durch $(\tilde{x} \vee y) \wedge (\tilde{x} \vee \overline{y})$, was auf die Erfüllbarkeit von F natürlich keine Auswirkungen hat. Dies ist offensichtlich eine logspace-Abbildung und führt uns zu Fall 2.

(2) F enthält Klauseln $(\tilde{x} \vee \tilde{y})$ mit zwei Literalen. Diese Klauseln überführen wir durch eine weitere logspace-Abbildung in eine Konjunktion $(\tilde{x} \vee \tilde{y} \vee z) \wedge (\tilde{x} \vee \tilde{y} \vee \overline{z})$ mit je einer neuen Variable z, wodurch sich an der Erfüllbarkeit von F ebenfalls nichts ändert. Es verbleibt der 3. Fall.

(3) F enthält noch Klauseln mit mehr als drei Literalen. Sei also $c = (\tilde{x}_1 \vee \tilde{x}_2 \vee \cdots \vee \tilde{x}_k)$ eine solche Klausel mit $k \geq 4$ Literalen. In diesem Fall führen wir $k-3$ neue Variablen $v(\tilde{x}_3), v(\tilde{x}_4), \ldots, v(\tilde{x}_{k-2}), v(\tilde{x}_{k-1})$ ein und ersetzen c durch

$$\begin{aligned} c' &= \big(\tilde{x}_1 \vee \tilde{x}_2 \vee v(\tilde{x}_3)\big) \wedge \bigwedge_{i=3}^{k-2} \big(\overline{v(\tilde{x}_i)} \vee \tilde{x}_i \vee v(\tilde{x}_{i+1})\big) \\ &\quad \wedge \big(\overline{v(\tilde{x}_{k-1})} \vee \tilde{x}_{k-1} \vee \tilde{x}_k)\big) \end{aligned}$$

Die Erfüllbarkeitsäquivalenz von c und c' ist nun etwas schwieriger zu sehen. Wir zeigen daher: $c \in \text{SAT} \iff c' \in \text{SAT}$. Sei also σ eine erfüllende Belegung der Variablen der Klausel c. Dann muß $\sigma(\tilde{x}_i) = 1$ für mindestens ein Literal $\tilde{x}_i$ aus c gelten. Womit wir σ so zu σ' erweitern können, daß σ' c' erfüllt, indem wir $\sigma'(v(\tilde{x}_j)) = 1$ für $j \leq i$ und $\sigma'(v(\tilde{x}_j)) = 0$ für $j > i$ wählen.

Umgekehrt müssen wir zeigen, daß eine erfüllende Belegung σ' der Klausel c' auch c erfüllt, d.h. für mindestens ein Literal $\tilde{x}_i$ gilt $\sigma(\tilde{x}_i) = 1$. Aus der Annahme des Gegenteils folgt, daß $\sigma'(v(\tilde{x}_3)) = 1$ sein muß, um die erste Klausel zu erfüllen. Mit Induktion erhalten wir, daß für alle $v(\tilde{x}_i)$ $\sigma'(v(\tilde{x}_i)) = 1$ gelten muß. Also ist die letzte Klausel im Widerspruch zur Annahme nur erfüllbar, wenn $\sigma'(\tilde{x}_i) = 1$ für mindestens ein $\tilde{x}_i$ gilt. ■

Bemerkung 4.3.8 Bei der Reduktion $\text{SAT} \leq_m^{\log} \text{KNF} \cap \text{SAT}$ im Beweis zu Satz 4.3.2 wurden nur Klauseln mit höchstens drei Literalen erzeugt. Daher folgt auch ohne den letzten Schritt im obigen Beweis die Behauptung des Korollars 4.3.7.

4.3.3 Lineare Programmierung über ganzen Zahlen

Satz 4.3.9 *Die lineare Programmierung über $\mathbb{Z}$*

$$\text{LinProg}(\mathbb{Z}) := \{\langle A, b\rangle \mid A \in \mathbb{Z}^{m\times n},\ b \in \mathbb{Z}^{m\times 1},\ \exists x \in \mathbb{Z}^{n\times 1} : Ax \geq b\}$$

ist **NP**$-$*vollständig.*

Beweis: Im Gegensatz zu den meisten anderen **NP**-vollständigen Problemen ist hier die Zugehörigkeit zu **NP** der schwierigere Teil des Beweises.

Wir erwähnen, daß LinProg($\mathbb{Z}$) auch bei binärer Codierung der Eingabe in **NP** liegt. Sei hierzu α der betragsmäßig größte Eintrag in der Matrix A oder dem Vektor b. Dann hat die Eingabe $\langle A, b\rangle$ die Mindestgröße $(m \cdot n + \lceil \log_2(\alpha)\rceil)$, selbst wenn alle anderen Einträge den Wert Null hätten. Sei weiter B eine $k \times k$

Teilmatrix von A, dann gilt $|det(B)| \leq k! \cdot |\alpha|^k \leq |\alpha \cdot k|^k$. Die Binärdarstellung von $det(B)$ erfordert also höchstens $k \cdot (\lceil \log_2(\alpha) \rceil + \lceil \log_2(k) \rceil)$ Bits. Der Rest des Beweises folgt aus der linearen Algebra. Eine Ausführung ist u.a. in [HU79] zu finden.

Die Reduktion 3-SAT $\leq_m^{\log}$ LinProg($\mathbb{Z}$) ist einfach und zeigt die **NP**-Härte auch bei unärer Codierung.

Sei $F = c_1 \wedge c_2 \wedge \cdots \wedge c_q$ eine Formel in 3-KNF. Seien $x_1, \ldots, x_n$ die Variablen in F. Wir bilden jetzt das folgende System von Ungleichungen über den Variablen $x_i, \overline{x_i}, 1 \leq i \leq n$:

(1) $x_i \geq 0, \;\; 1 \leq i \leq n$

(2) $\overline{x_i} \geq 0, \;\; 1 \leq i \leq n$

(3) $x_i + \overline{x_i} \geq 1, \;\; 1 \leq i \leq n$

(4) $-x_i - \overline{x_i} \geq -1, \;\; 1 \leq i \leq n$

(5) $\tilde{x}_{j1} + \tilde{x}_{j2} + \tilde{x}_{j3} \geq 1$, für jede Klausel $c_j = (\tilde{x}_{j1} \vee \tilde{x}_{j2} \vee \tilde{x}_{j3})$.

(3) und (4) stellen sicher, daß $x_i + \overline{x_i} = 1$ gilt. Wegen (5) muß in jeder Klausel c_j mindestens ein Literal $\tilde{x}_{ij}$ den Wert 1 erhalten. Offensichtlich ist dieses System von Ungleichungen genau dann lösbar, wenn F erfüllbar ist. Insgesamt erhalten wir $4n + q$ Ungleichungen in $2n$ Variablen, d.h. A ist eine $(4n + q) \times 2n$-Matrix, $x \in \mathbb{Z}^{2n \times 1}$, $b \in \mathbb{Z}^{(4n+q) \times 1}$. Unsere Reduktion ist also logspace-berechenbar und LinProg($\mathbb{Z}$) ist selbst bei unärer Kodierung **NP**-hart. ■

Bemerkung 4.3.10 LinProg($\mathbb{Q}$)$\in$ **P**. Dieser Nachweis ist sehr schwierig und beruht auf der *Ellipsoidmethode* von Khachiyan.

4.3.4 Exkurs: Binärcodierung von Primzahlen $\in$ NP $\cap$ co-NP

Ein weiteres Problem, bei dem die **NP**-Zugehörigkeit schwierig zu zeigen ist, ist zu testen, ob eine gegebene Zeichenkette über $\{0, 1\}$ Binärcodierung einer Primzahl ist. Sei also Prim die Menge der Binärcodierungen von Primzahlen:

$$\text{Prim} := \{bin(p) \in \{0,1\}^+ \mid bin(p) \text{ ist Binärcodierung einer Primzahl}\}$$

Satz 4.3.11 Prim $\in$ **NP** $\cap$ **co-NP**.

Beweis: Die Zugehörigkeit Prim $\in$ **co-NP** ist trivial. Eine entsprechende Turingmaschine rät zwei Faktoren $r, s < p$ und verifiziert $r \cdot s = p$.

Für den Nachweis Prim $\in$ **NP** benutzen wir den kleinen Satz von Fermat, nach dem gilt:

$$p \text{ ist Primzahl} \iff \exists x : 0 \leq x \leq p \text{ mit } x^{p-1} \equiv 1 \bmod p$$
$$\text{und } x^i \not\equiv 1 \bmod p,\ \forall i \in \{1, \ldots, p-2\}$$

Eine Turingmaschine, die den Primzahltest durchführt, arbeitet nach folgendem rekursiven Schema:

(1) p ist Primzahl, falls $p = 2$ oder $p = 3$, sonst weiter mit (2).

(2) Rate eine Primfaktorzerlegung für $p - 1 = p_1 \cdot \ldots \cdot p_k$, $k \geq 2$.

(3) Überprüfe dabei, ob $p_j \geq 2$ und rekursiv, ob $bin(p_j) \in$ Prim, für jedes $j \in \{1, \ldots, k\}$.

(4) Bestimme wiederum nichtdeterministisch ein $x \in \{0, \ldots, p\}$ und überprüfe, ob $x^{p-1} \equiv 1 \bmod p$ gilt.

(5) Verifiziere, ob für alle $j \in \{1, \ldots, k\}$ die Ungleichung $x^{\frac{p-1}{p_j}} \not\equiv 1 \bmod p$ gilt.

Die Turingmaschine akzeptiert nach erfolgreichem Durchlaufen der Rekursion.

Wäre jetzt $x^i \equiv 1 \bmod p$ für ein $i \in \{1, \ldots, p-2\}$, dann gäbe es auch ein minimales $i > 0$ mit dieser Eigenschaft. Das hieße, daß $\{x^j \mid 0 \leq j < i\}$ eine Untergruppe der Größe i der multiplikativen Gruppe $\mathbb{Z}/p\mathbb{Z} \setminus \{0\}$ wäre, woraus sich ergäbe, daß i Teiler von $p-1$ ist. Somit gäbe es ein p_j mit $x^{\frac{p-1}{p_j}} \equiv 1 \bmod p$. Also genügt es in Schritt (5) zu verifizieren, ob für alle $j \in \{1, \ldots, k\}$ die Ungleichung $x^{\frac{p-1}{p_j}} \not\equiv 1 \bmod p$ gilt.

Wir betrachten für eine Primzahl p eine erfolgreiche Rechnung dieser nichtdeterministischen Turingmaschine. Diese Rechnung kann mit einem Baum verglichen werden, der die rekursive Struktur des Programms symbolisiert. An den inneren Knoten dieses Baumes werden die Schritte (1) – (5) ausgeführt, an den Blättern nur (1). Wir nennen diesen Baum auch Beweisbaum für die Primzahleigenschaft von p. Wir zeigen nun, daß dieser Beweisbaum maximal $2 \cdot \lfloor \log_2(p) \rfloor - 1$ Knoten besitzt. Für $p = 2$ oder $p = 3$ ist dies klar, denn der Baum besitzt nur einen Knoten und $2 \cdot \lfloor \log_2(2) \rfloor - 1 = 2 \cdot \lfloor \log_2(3) \rfloor - 1 = 1$. Sei nun $p > 3$ eine Primzahl und $p - 1 = p_1 \cdot \ldots \cdot p_k$, $k \geq 2$, eine richtig geratene Primfaktorzerlegung. Der Beweisbaum für p setzt sich zusammen aus den Beweisbäumen für $p_1, \ldots, p_k$ und dem Wurzelknoten. Mit Induktion ergibt sich die maximale Größe

$$
\begin{aligned}
1+\sum_{i=1}^{k}(2\cdot\lfloor\log_2(p_i)\rfloor-1) &\leq 2\cdot\lfloor\log_2(p_1\cdot\ldots\cdot p_k)\rfloor-(k-1)\\
&\leq 2\cdot\lfloor\log_2(p)\rfloor-1,\quad \text{da } k\geq 2.
\end{aligned}
$$

Um einen Abbruch des Programms in jedem Fall zu gewährleisten, können wir damit einen globalen Zähler einführen, der zu Beginn auf $2\cdot\lfloor\log_2(p)\rfloor-1$ gesetzt und jeweils vor Ausführen von Schritt (1) um 1 erniedrigt wird. Wird der Zählerwert 0, so wird die Berechnung abgebrochen.

Nun müssen wir noch zeigen, daß dieser Algorithmus höchstens polynomiale Zeit benötigt. Da nur mit Zahlen gerechnet wird, die kleiner als p sind, können wir eine Addition in $\mathcal{O}(\log p)$ Schritten, eine Multiplikation bzw. Division in $\mathcal{O}(\log^2 p)$ Schritten und eine Potenzierung in $\mathcal{O}(\log^3 p)$ Schritten erledigen. Hierbei potenzieren wir durch wiederholtes Quadrieren, d.h. wir bilden die Potenzen $x^2, x^4, x^8, \ldots, x^y$ (dabei wird jeweils modulo p gerechnet). Da $k \leq \log p$, benötigen die Multiplikationen in (2) $\mathcal{O}(\log^3 p)$ Schritte. Insgesamt benötigen damit (2), (4) und (5) $\mathcal{O}(\log^3 p)$ Zeit. Es ergibt sich eine Zeitkomplexität von $\mathcal{O}(\log^4 p)$. ■

4.3.5 Vertex Cover

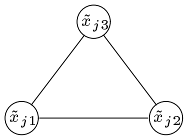
Nun wollen wir noch einige **NP**-vollständige Graphenprobleme betrachten. Besonders interessant ist hierbei das Problem, eine Knotenüberdeckung (Vertex Cover) in einem Graphen zu finden. Da es sich häufig recht einfach auf andere Probleme reduzieren läßt, ist es gut geeignet, um die **NP**-Vollständigkeit vieler anderer Probleme zu zeigen. Das Problem Vertex Cover (VC) wurde bereits in Def. 1.3.3 (Seite 8) definiert.

Satz 4.3.12 *Vertex Cover* (VC) *ist* **NP**-*vollständig.*

Beweis: Der Nachweis VC $\in$ **NP** ist leicht. Eine nichtdeterministische Turingmaschine, die VC löst, rät eine Knotenüberdeckung $U \subseteq V$ mit $|U| = k$ und überprüft für alle $e \in E$ in polynomialer Zeit, ob auch tatsächlich ein Endpunkt in U liegt.

Die **NP**-Härte zeigen wir durch die Reduktion 3-SAT $\leq_m^{\log}$ VC. Sei hierzu $F = c_1 \wedge \ldots \wedge c_q$ eine Formel in 3-KNF. Dann müssen wir einen Graphen $G(F) = (V, E)$ so konstruieren, daß für ein bestimmtes k (auf das wir später noch genauer eingehen werden) gilt: $F \in$ 3-SAT $\iff$ $(G(F), k) \in$ VC.

Zunächst bilden wir zu jeder Klausel $c_j = (\tilde{x}_{j1} \vee \tilde{x}_{j2} \vee \tilde{x}_{j3})$ den folgenden Graphen $G(c_j)$:



Danach bilden wir die disjunkte Vereinigung $\dot{\bigcup}_{j=1}^{q} G(c_j)$ aller dieser Teilgraphen $G(c_j)$ und erhalten daraus den gesuchten Graphen $G(F)$ durch Einfügen weiterer Kanten $(x, \overline{x})$ (x ist eine Variable aus F), falls $x, \overline{x}$ in verschiedenen Teilgraphen $G(c_j)$ liegen.

Beispiel 4.3.13 Für die Formel $F = (x \vee y \vee z) \wedge (\overline{x} \vee s \vee \overline{r}) \wedge (\overline{y} \vee \overline{s} \vee \overline{z}) \wedge (x \vee s \vee r)$ führt diese Konstruktion zu einem Graphen $G(F)$ wie in Abbildung 4.3 dargestellt.

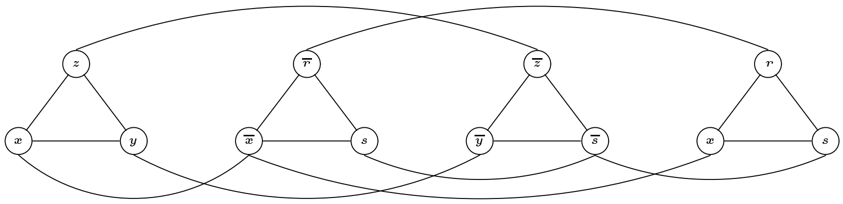


Abbildung 4.3: Graph $G(F)$ zur Formel F

In $G(F)$ kann es keine Knotenüberdeckung U mit weniger als $2q$ Knoten geben, da in jedem der q Dreiecke mindestens 2 Knoten zu U gehören müssen. Tatsächlich gilt sogar, daß $F \in$ 3-SAT $\iff$ $(G(F), 2q) \in$ VC, wie wir im folgenden zeigen werden.

Sei $\sigma(F) = 1$ für eine Belegung σ. Dann wird in jeder Klausel also mindestens ein Literal $\tilde{x}_{ji}$ wahr. Für U wählen wir aus dem entsprechenden Teilgraphen $G(c_j)$ genau zwei Knoten so aus, daß wir die Knoten, die zu nicht erfüllten Literalen gehören, auf jeden Fall aufnehmen. Dann gilt aber $|U| = 2q$ und U ist ein Vertex-Cover.

Umgekehrt gilt: Falls U ein Vertex-Cover mit $|U| = 2q$ ist, so gehören in jedem Teilgraphen $G(c_j)$ genau zwei Punkte zu U. Wir setzen $\sigma(\tilde{x}) = 1$ für den Knoten $\tilde{x}$, der nicht zu U gehört. Da U ein Vertex Cover ist, definiert dies eindeutig einen Wahrheitswert für die zugehörige Variable x. Für die Literale, denen so noch kein Wert zugewiesen wird, wählen wir einen beliebigen Wert aus und setzen σ dadurch offensichtlich zu einer Belegung mit $\sigma(F) = 1$ fort. ∎

4.3.6 Hamilton-Kreis und Hamilton-Pfad

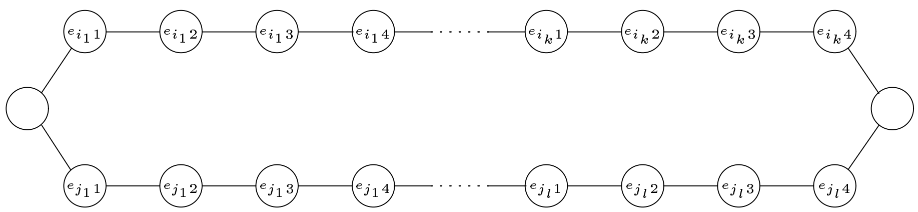
Zwei weitere häufig untersuchte Probleme sind die Probleme Hamilton-Kreis (HC) und Hamilton-Pfad (HP). Sie lauten: Gibt es in einem gegebenem ungerichten Graphen $G = (V, E)$ einen Rundweg bzw. einen Pfad, der alle Knoten genau einmal besucht?

Satz 4.3.14 *Die Probleme Hamilton-Kreis und Hamilton-Pfad sind* **NP**-*vollständig.*

Beweis: Wir beschränken uns hier auf den Beweis für das Problem Hamilton-Kreis. Trivial ist HC $\in$ **NP**. Den Nachweis, daß HC **NP**-hart ist, führen wir mittels 3-SAT $\leq_m^{\log}$ HC. Sei $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ eine Formel in 3-KNF mit den Klauseln $c_1, \ldots, c_m$ in den Variablen $x_1, \ldots, x_n$. Wir konstruieren einen Graphen $G(F)$, der genau dann einen Hamilton-Kreis hat, falls $F \in$ SAT gilt. Zunächst definieren wir zu jeder Klausel c einen Graphen $G(c)$ wie in Abbildung 4.4 dargestellt.

In $G(c)$ gibt es keinen Hamilton-Pfad von b_1 nach b_4, sowie man jedoch auf den Besuch von mindestens einem Weg $b_j - a_{j1} - a_{j2} - a_{j3} - a_{j4} - b_{j+1}$, $j \in \{1, 2, 3\}$ verzichtet, gibt es einen solchen Pfad von b_1 nach b_4.

Für eine Variable x sei $\{c_{i_1}, \ldots, c_{i_k}\}$ die Menge der Klauseln mit $x \in c_i$ und $\{c_{j_1}, \ldots, c_{j_l}\}$ die Menge der Klauseln mit $\overline{x} \in c_j$. Ohne Einschränkung können wir annehmen, daß $i_1 < \cdots < i_k, j_1 < \cdots < j_k$ sowie $\forall p, q : i_p \neq j_q$ gilt. Zu jeder Variablen x definieren wir nun einen Graphen $G(x)$ wie in folgendem Bild:



Den Graphen $G(F)$ bilden wir durch Zusammenfügen der bisher konstruierten Graphen nach dem in Abbildung 4.5 gezeigten Muster.

Zusätzlich benötigen wir für jede Klausel $c_k = (\tilde{x}_{k_1} \vee \tilde{x}_{k_2} \vee \tilde{x}_{k_3})$ noch Verbindungen zwischen dem Graphen $G(c_k)$ und den Graphen $G(x_{k_i})$, wie sie in Abbildung 4.6 dargestellt sind. Wir verbinden also für $i \in \{1, \ldots, 3\}$ und $q \in \{1, \ldots, 4\}$ den Knoten a_{iq} in $G(c_k)$ mit dem Knoten e_{kq} aus $G(x_{k_i})$ über einen neuen Zwischenknoten.

Um die Konstruktion nochmals zu verdeutlichen, ist in Abbildung 4.7 der gesamte Graph $G(F)$ für die Formel $F = \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{c_1} \wedge \underbrace{(\overline{x_1} \vee \overline{x_2} \vee x_3)}_{c_2}$ angegeben.

Nun ist nur noch zu zeigen, daß $F \in \mathrm{SAT}$ genau dann gilt, wenn es in $G(F)$ einen Hamilton-Kreis gibt. Auf einen formalen Beweis wollen wir hier verzichten. Wir geben nur anschaulich den zu einer erfüllenden Belegung gehörenden Hamilton-Kreis in $G(F)$ an. Der Weg führt über die erfüllten Literalen entsprechenden Knoten in den $G(x_i)$, wobei jedoch über die zuletzt eingefügten Verbindungen in jedem $G(c_k)$ genau eine Knotengruppe $a_{i1} - a_{i2} - a_{i3} - a_{i4}$ besucht wird. Dies ist möglich, da in jeder Klausel mindestens ein Literal erfüllt ist. Nachdem so alle $G(x_i)$ durchwandert wurden, werden die in den $G(c_k)$ verbleibenden Knoten und die zugehörigen Knoten in den $G(x_i)$ besucht. Anschließend endet der Weg im Startknoten, womit wir einen Hamilton-Kreis gefunden haben. Umgekehrt kann man sich verdeutlichen, daß jeder Hamilton-Kreis eine erfüllende Belegung definiert. Dies sei jedoch dem Leser überlassen. ■

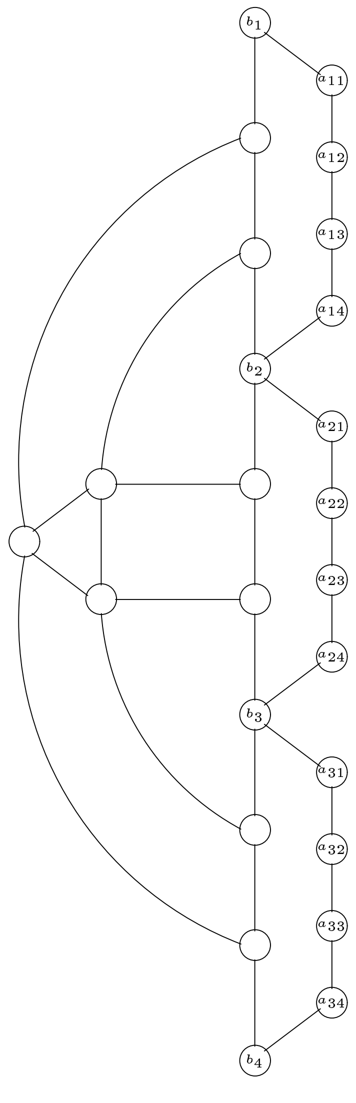


Abbildung 4.4: Teilgraph $G(c)$ für eine Klausel c

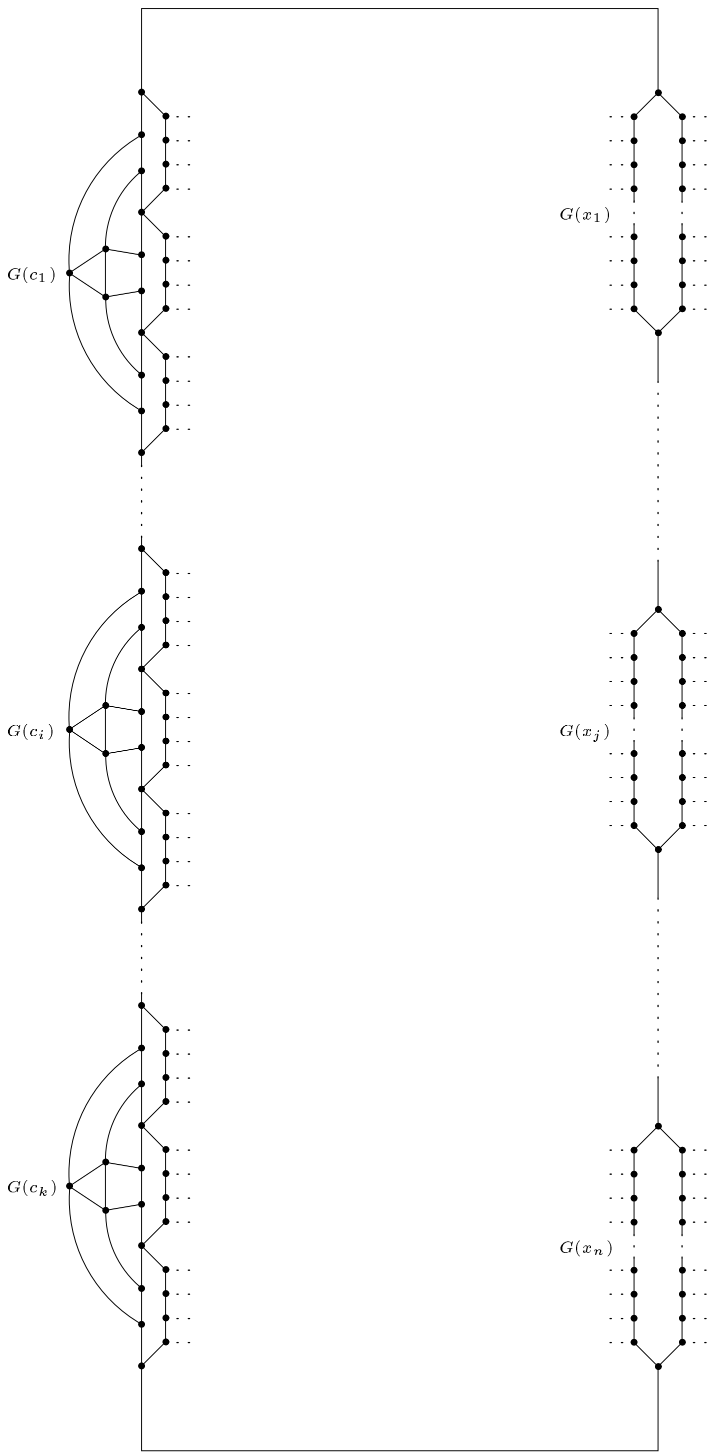


Abbildung 4.5: Anordnung der einzelnen Teilgraphen im Gesamtgraph $G(F)$

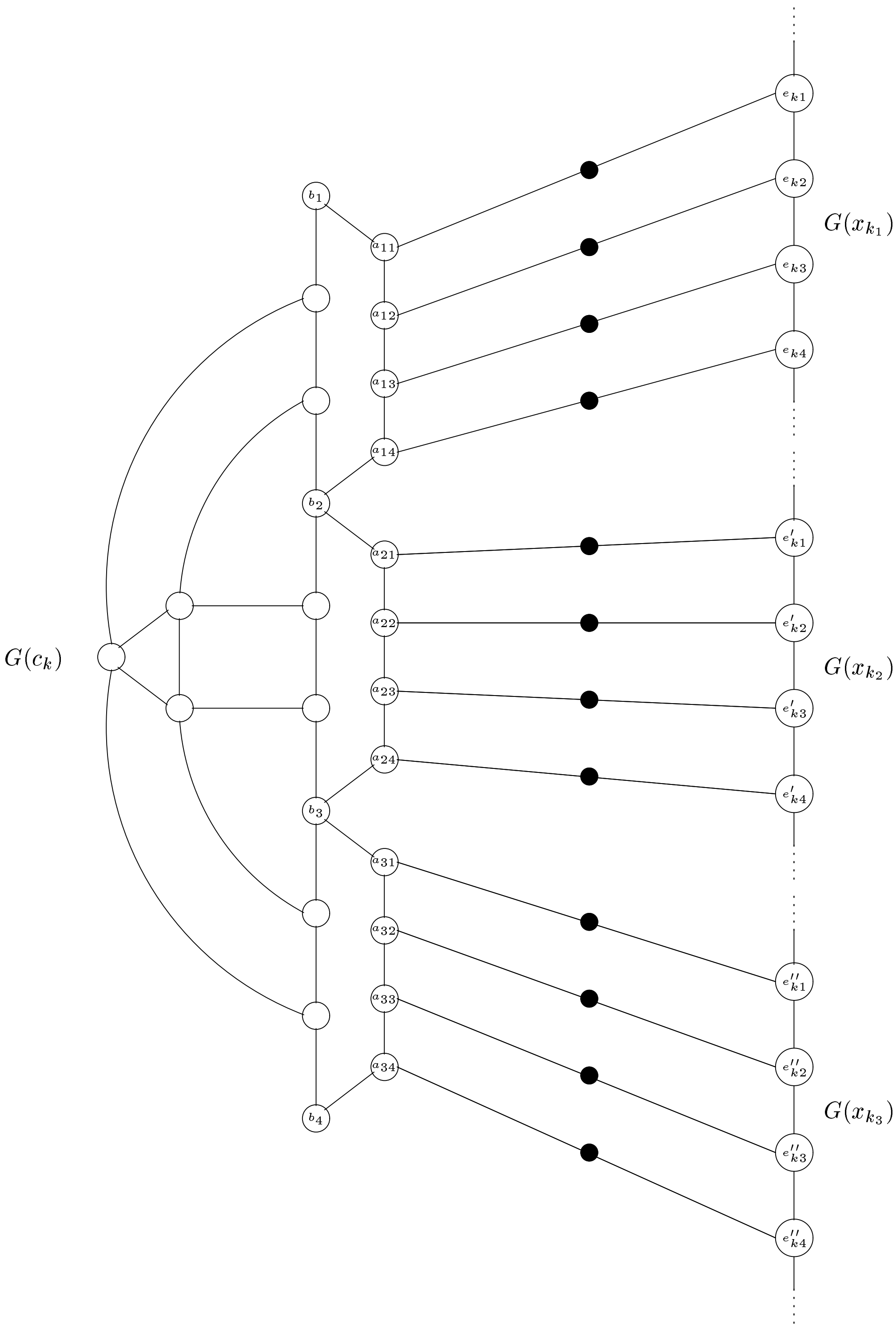


Abbildung 4.6: zusätzliche Verbindungen

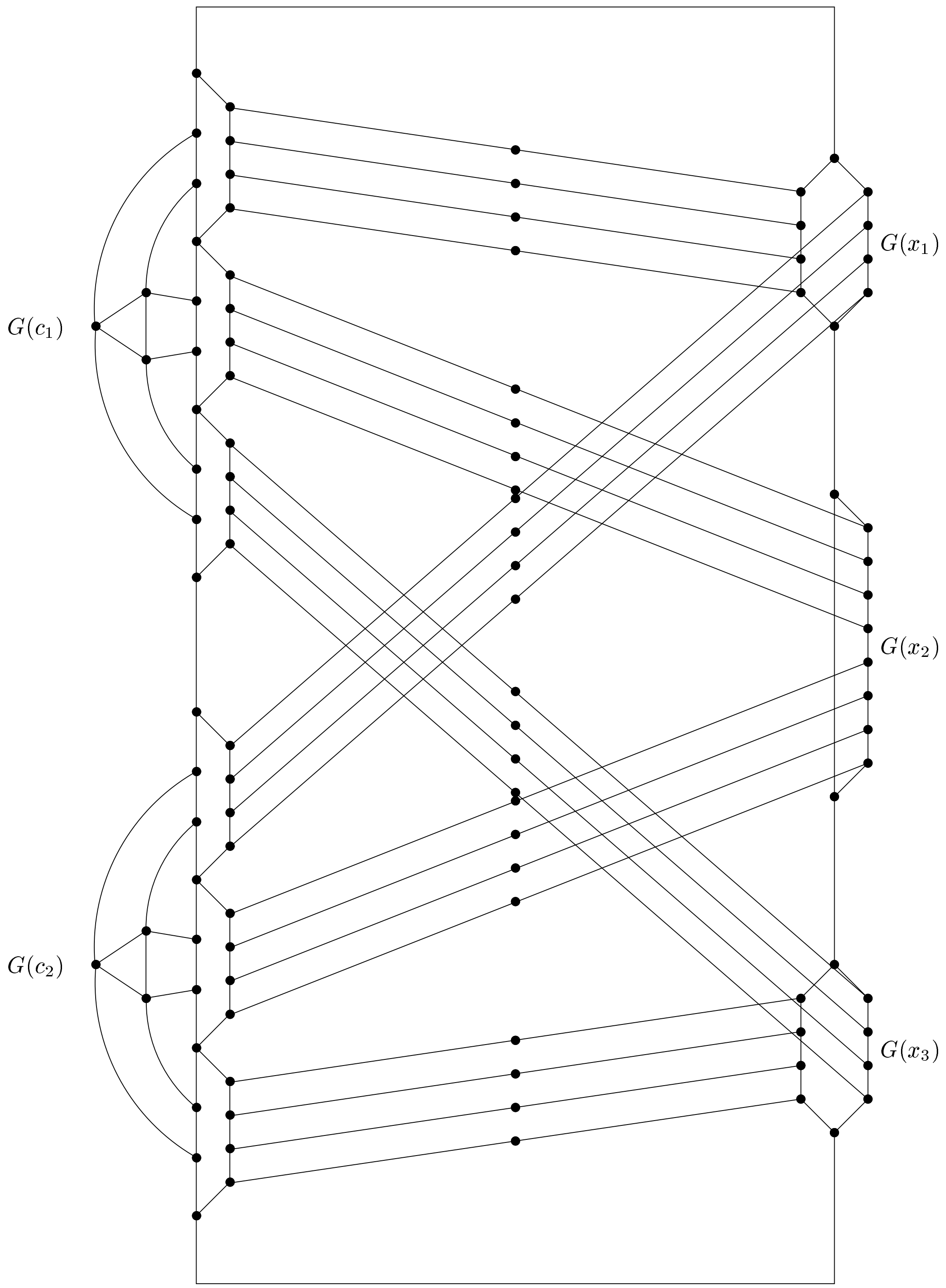


Abbildung 4.7: Graph zur Formel $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$

Bemerkung 4.3.15 HC ist bereits für planare Graphen **NP**-vollständig. Hat jeder Knoten in G mindestens den Grad $\frac{|V|}{2}$, so existiert stets ein Hamilton-Kreis. Gilt jedoch nur ein Mindestgrad von $\frac{|V|}{2} \cdot (1 - \varepsilon)$ für ein festes $\varepsilon > 0$, so wird das Problem **NP**-vollständig.

4.3.7 Problem des Handlungsreisenden

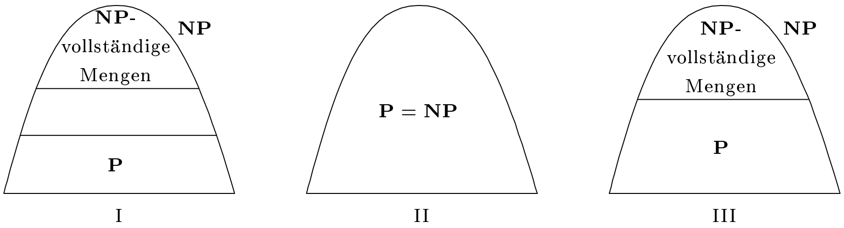
Wir erhalten direkt folgendes Ergebnis für das in Beispiel 1.3.1 untersuchte Problem des Handlungsreisenden (Traveling Salesman Problem, TSP)).

Korollar 4.3.16 *Die Entscheidungsvariante des Problem des Handlungsreisenden ist* **NP***-vollständig.*

Beweis: Klar ist TSP $\in$ **NP**. Die **NP**-Härte ist ebenfalls einfach, da HC eigentlich nur ein Spezialfall von TSP ist. Wir reduzieren also HC $\leq_m^{\log}$ TSP: Ist $G = (V, E)$ ein Graph, dann existiert genau dann ein Hamilton-Kreis, wenn ein Rundweg mit Kosten $\leq |V| = n$ existiert, wobei jede Kante die Kosten 1 erhält. ■

4.4 Nicht NP-vollständige Mengen innerhalb von NP $\setminus$ P

Wir wollen uns jetzt noch einige Gedanken zur Lage von **NP**-vollständigen Mengen bzgl. den Mengen **P** und **NP** machen. Dabei sind prinzipiell die folgenden drei Möglichkeiten gegeben:



Die dritte Möglichkeit können wir jedoch gemäß dem folgenden Satz ausschließen. Dies gilt auch schon bezüglich polynomieller Zeitreduktion.

Satz 4.4.1 *Ist* $\mathbf{P} \neq \mathbf{NP}$*, dann existiert effektiv eine Sprache* $L \in \mathbf{NP} \setminus \mathbf{P}$*, die nicht* **NP***-vollständig ist.*

Beweis: Sei $f : \mathbb{N} \to \mathbb{N}$ eine schwach monoton wachsende Funktion[1]. Definiere

$$L_f = \{x \in \Sigma^* | \, x \in \text{SAT} \wedge f(|x|) \text{ ist gerade}\}$$

Wir definieren jetzt eine spezielle Funktion $f : \mathbb{N} \to \mathbb{N}$ rekursiv durch ein deterministisches Programm, das $f(n)$ in $\mathcal{O}(n)$ Schritten berechnen wird. Insbesondere gilt damit $L_f \in \mathbf{NP}$.

Sei $M_1, M_2, \ldots$ eine Aufzählung aller deterministischen Turingmaschinen mit einer zusätzlichen polynomialen Zeitschranke. Das heißt, wir zählen eigentlich alle Paare (M_i, p_j) von deterministischen Turingmaschinen und Polynomen auf und erhalten daraus obige Aufzählung, indem M_i mit einem zusätzlichen Schrittzähler versehen wird, der sicherstellt, daß die polynomiale Zeitschranke p_j nicht überschritten wird. Man beachte dazu, daß Polynome zeitkonstruierbar sind. Sei analog $R_1, R_2, \ldots$ eine Aufzählung aller Polynomialzeitreduktionen. Wir verwenden ferner einen deterministischen Algorithmus zur Erkennung von SAT. Die Laufzeit dieses Algorithmus ist unerheblich und könnte daher auch exponentiell sein.

```
FUNCTION f(n)
  if n = 0 then
    return 0
  else (* n > 0 *)
    i := 0
    k := 0
    Die folgende Schleife wird genau n Befehls-Schritte lange abgearbeitet. Dann
    wird die Berechnung sofort mit dem auf die Schleife folgenden Befehl fortgesetzt.
    loop
      k := f(i)
      i := i + 1
    endloop
    (* Beachte: alle rekursiven Aufrufe f(i) geschehen nur mit i < n. *)
    (* Der Wert von k ist nach Durchlaufen dieser Schleife sehr klein. *)
    if k = 2j (* k gerade *) then
      suche für genau n Schritte in längenlexikographischer Reihenfolge ein x ∈ Σ*
      mit x ∈ L(M_j)ΔL_f.
    endif
    if k = 2j + 1 (* k ungerade *) then
      suche für genau n Schritte in längenlexikograph. Reihenfolge ein x ∈ Σ* mit
      (x ∈ SAT ∧ R_j(x) ∉ L_f) oder (x ∉ SAT ∧ R_j(x) ∈ L_f).
    endif
    if ein solcher Zeuge x ∈ Σ* wurde gefunden then
      return k + 1
    else
      return k
    endif
  endif
ENDFUNC
```

[1]Eine Funktion f heißt schwach monoton wachsend gdw. $(x_2 > x_1 \Rightarrow f(x_2) \geq f(x_1))$

Bemerkung 4.4.2 $f(n)$ ist wohldefiniert und wird in $\mathcal{O}(n)$ Schritten berechnet. Es gilt weiter $f(n) + 1 \geq f(n+1) \geq f(n)$ für alle $n \in \mathbb{N}$.

Seien L, K Mengen, dann bezeichnet

$$L \Delta K = L \setminus K \cup K \setminus L$$

die symmetrische Differenz. Man beachte: ist $L \in \mathcal{C}$ für eine der hier betrachteten Komplexitätsklassen $\mathcal{C}$ und ist $|L \Delta K| < \infty$, so gilt auch $K \in \mathcal{C}$.

Wir zeigen jetzt mit Widerspruch: wenn $\mathbf{P} \neq \mathbf{NP}$ gilt, so liegt L_f weder in $\mathbf{P}$ noch ist L_f $\mathbf{NP}$-vollständig.

Dazu überlegen wir zunächst:

1. $L_f \in \mathbf{P}$. Dann existiert ein j mit $L_f = L(M_j)$. Es gibt nun kein $x \in \Sigma^*$ mit $x \in L(M_j) \Delta L_f$. Aus der Berechnung von f folgt nun $f(n) \leq 2j$ für alle $n \in \mathbb{N}$.

2. L_f ist $\mathbf{NP}$-vollständig. Dann ist SAT $\leq_m^p L_f$. Das heißt, es gibt ein j so, daß $x \in \text{SAT} \Leftrightarrow R_j(x) \in L_f$ für alle $x \in \Sigma^*$. Damit gibt es kein $x \in \Sigma^*$ mit $(x \in \text{SAT} \wedge R_j(x) \notin L_f)$ oder $(x \notin \text{SAT} \wedge R_j(x) \in L_f)$. Aus der Berechnung von f folgt nun $f(n) \leq 2j + 1$ für alle $n \in \mathbb{N}$.

Ist also $L_f \in \mathbf{P}$ oder L_f $\mathbf{NP}$-vollständig, so ist f beschränkt und wird schließlich stationär.

Man beachte nun:

1. Ist f beschränkt und $f(n)$ gerade für fast alle n, so gilt $|L_f \Delta \text{SAT}| < \infty$ und L_f ist $\mathbf{NP}$-vollständig.

2. Ist f beschränkt und $f(n)$ ungerade für fast alle n, so ist L_f endlich und damit $L_f \in \mathbf{P}$.

Mit dieser Aussage und der Vorraussetzung $P \neq \mathbf{NP}$ erhalten wir den folgenden Widerspruch:

1. $f(n) = 2j$ für fast alle n. Dann gilt $L(M_j) = L_f$. Nach dem oben erwähnten 1. Fall ist L_f dann jedoch $\mathbf{NP}$-vollständig, das ist ein Widerspruch zu $L(M_j) \in \mathbf{P}$.

2. $f(n) = 2j + 1$ für fast alle n. Dann ist R_j eine Reduktion von SAT auf L_f. Dies steht im Widerspruch zum 2. Fall, da dann L_f endlich ist und wegen $\mathbf{P} \neq \mathbf{NP}$ die Sprache SAT nicht auf eine Sprache in $\mathbf{P}$ reduziert werden kann. ■

Kapitel 5

Vollständige Probleme für P und PSPACE

5.1 Leerheit kontextfreier Sprachen

Sei $L_{cfe} = \{\langle G\rangle \,|\, G$ ist eine kontextfreie Grammatik mit $L(G) \neq \emptyset\}$. Dabei stellen die spitzen Klammern $\langle\rangle$ eine geeignete Kodierung von Grammatiken dar, cfe steht für context–free–empty.

Satz 5.1.1 *L_{cfe} ist* **P***-vollständig.*

Beweis:

(i) $\boldsymbol{L_{cfe}} \in$ P: Teste für eine gegebene Grammatik G, ob das Startsymbol S produktiv ist.

(ii) $\boldsymbol{L_{cfe}}$ ist P-hart: Sei $L \in \mathbf{P}$ und $L(M) = L$ für eine $p(n)$ zeitbeschränkte Turingmaschine mit denselben Konventionen wie im Beweis des Satzes von Cook (Satz 4.2.1).

Insbesondere können wir Konfigurationen von M durch Wörter $\alpha \in \#\Gamma^*Q\Gamma^*\#$ beschreiben, wobei $\#$ ein spezielles Randsymbol bezeichne. Auf eine Eingabe $v = v_1 \cdots v_n \in \Sigma^*$ mit $|v| = n$ sei also $\#q_0v_1 \cdots v_nB \cdots B\#$ mit $p(n) - n$ Blanksymbolen die Startkonfiguration.

Sei $\#\alpha_0 \cdots \alpha_p\# \vdash_M \#\beta_0 \cdots \beta_p\#$ ein Einschrittübergang, dann können wir annehmen, daß $\alpha_{j-1}, \alpha_j, \alpha_{j+1}$ eindeutig β_j bestimmt, d.h. für eine endliche Tabelle Δ können wir dies durch $(\alpha_{j-1}, \alpha_j, \alpha_{j+1}, \beta_j) \in \Delta$ beschreiben.

Beachte: Insbesondere ist eine Kopfbewegungen der Turingmaschine nur vom Zustand abhängig und nicht vom gelesenen Zeichen.

Wir nehmen an, daß $v \in L(M)$ genau dann gilt, wenn M sich nach $p = p(|v|)$ Schritten im Zustand \$ befindet und der Kopf wieder ganz am linken Rand steht.

Das heißt, ist

$$s(v) = \#q_0v_1\cdots v_nB\cdots B\# \vdash_M \#\alpha_{10}\cdots\alpha_{1p}\# \vdash_M \cdots \vdash_M \#\alpha_{p0}\cdots\alpha_{pp}\#$$

eine Berechnung der Länge p, dann gilt $v \in L(M) \Leftrightarrow \alpha_{p0} = \$$. Zu $v \in \Sigma^*$ definieren wir jetzt die Grammatik $G(v) = G = (V, \emptyset, P, S)$ mit Variablenmenge V, leerem Terminalalphabet, Startsymbol $S \in V$ und der Regelmenge P.

Die Variablen heißen

$$V(a,t,j) \quad \text{für } a \in \Gamma \cup Q \cup \{\#\},\ 0 \leq t \leq p \text{ und } -1 \leq j \leq p+1\,.$$

Das Startsymbol S ist $V(\$, p, 0)$.

P enthält die folgenden Regeln:

$$\begin{array}{lrl}
(1) & V(d,t,j) \to & V(a,t-1,j-1)V(b,t-1,j)V(c,t-1,j+1) \\
 & & \text{für alle } (a,b,c,d) \in \Delta,\ 0 < t \leq p,\ 0 \leq j < p \\
(2) & V(\#,t,-1) \to \lambda & \text{für } 0 \leq t \leq p \\
(3) & V(\#,t,p+1) \to \lambda & \text{für } 0 \leq t \leq p \\
(4) & V(q_0,0,0) \to \lambda & \\
(5) & V(v_j,0,j) \to \lambda & \text{für } 1 \leq j \leq n \\
(6) & V(B,0,j) \to \lambda & \text{für } n < j \leq p
\end{array}$$

Nun gilt $L(G(v)) \neq \emptyset \Leftrightarrow v \in L$. Der formale Beweis für diese Aussage ist eine längliche Induktion, die wir hier nicht aufführen. ∎

5.2 Das Auswertungsproblem für Schaltkreise

Definition 5.2.1 Ein boolescher Schaltkreis C ist ein gerichteter Graph $C = (V, E)$. Die Knoten heißen Gatter und werden mit den Zahlen $1, \ldots, o$ bezeichnet. Sie haben jeweils den Eingangsgrad 0, 1 oder 2. Die Kanten $(i,j) \in E$ erfüllen die Bedingung $i < j$, d.h. der Graph ist insbesondere azyklisch. Jedes Gatter i sei von einer Sorte

$$s(i) \in \{\neg, \wedge, \vee, true, false\} \cup \{x_1, x_2, x_3, \ldots\}\,.$$

Die Sorte bestimmt den Eingangsgrad. Die Gatter der Sorte x_i heißen Eingangsgatter, das Gatter o (output) heißt Ausgangsgatter. Sind die Sorten der Eingangsgatter eine Teilmenge von $\{x_1, \ldots, x_n\}$, so definiert jeder Bitstring $x \in \{0,1\}^n$ einen Wert des Gatters $C(x) \in \{0,1\}$.

Schaltkreise berechnen boolesche Funktionen $\{0,1\}^n \to \{0,1\}$, und umgekehrt läßt sich jede boolesche Funktion durch einen Schaltkreis realisieren.

Ein Schaltkreis heißt *monoton*, falls er keine NOT-Gatter enthält. Solche Schaltkreise berechnen monotone Funktionen (d.h. eine Umwandlung eines Bits von 0 auf 1 kann kein Zurücksetzen des Ausgangswertes auf 0 erzeugen). Umgekehrt läßt sich jede monotone Funktion durch einen monotonen Schaltkreis berechnen. Die genaue Durchführung sei dem Leser zur Übung überlassen.

Ein Schaltkreis heißt *variablenfrei*, falls er keine Eingangsgatter hat. Insbesondere ist der Wert eines solchen Schaltkreises konstant 0 oder 1.

Bemerkung 5.2.2

1. Die Frage, ob es eine Belegung der Eingangsgatter gibt so, daß sich der Wert eines Schaltkreises zu 1 berechnet, ist **NP**-vollständig.

2. Die Auswertung eines Schaltkreises unter einer Belegung ist in polynomialer Zeit möglich.

Satz 5.2.3 *Das Auswertungsproblem (*CV*, Circuit Value) für Schaltkreise ist* **P***-vollständig. Dies gilt auch für monotone, variablenfreie Schaltkreise.*

Beweis:

(i) CV $\in$ **P:** Siehe obige Bemerkung.

(ii) CV **ist P-hart:**
Betrachte den Beweis zur **P**-Vollständigkeit von L_{cfe}. Zu einer Sprache $L \in \mathbf{P}$ und einem $v \in \Sigma^*$ wurde eine kontextfreie Grammatik $G(v)$ konstruiert. Die Grammatik hatte eine spezielle Form. Die Regeln waren vom Typ

$$A \to B_1B_2B_3|\cdots|Z_1Z_2Z_3$$

oder

$$A \to \lambda$$

Jede Variable wird als ein Gatter interpretiert. Startsymbol ist das Ausgangsgatter. Die Regeln

$$A \to B_1B_2B_3|\cdots|Z_1Z_2Z_3$$

werden in das in Abbildung 5.1 dargestellte Gatter übersetzt.

Man beachte das in dem so erzeugten Schaltkreis keine Schleifen auftreten, denn in $G(v)$ sind keine Ableitungen der Form $A \Rightarrow^+ uAv$ mit $u, v \in V^*$, $A \in V$

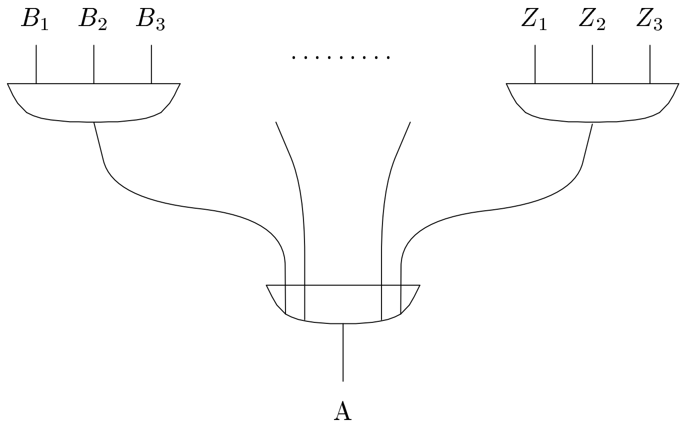


Abbildung 5.1: Gatter für die Regel $A \to B_1B_2B_3|\cdots|Z_1Z_2Z_3$

möglich. Auch ist die Anzahl der Produktion mit einer Variablen A auf der linken Seite nur von Δ abhängig. Der Eingangsgrad des zugehörigen Gatters A ist damit konstant (unabhängig von v). Der Schaltkreis läßt sich deshalb in einen Schaltkreis mit maximalem Eingangsgrad 2 übersetzten, wobei die Anzahl der Gatter nur um einen konstanten Faktor zunimmt. Die Variablen A mit einer Produktion $A \to \lambda$ werden zu Gattern mit dem Wert „true". Alle anderen Variablen A, die auf keiner linken Seite erscheinen, werden zu „false"–Gattern. Dieser so zu $G(v)$ konstruierte Schaltkreis liefert den Wert 1 genau dann, wenn $v \in L$. Hieraus folgt die Behauptung. ■

5.3 Quantifizierte Boolesche Formeln

Die Menge M der quantifizierten booleschen Formeln ist die kleinste Menge, die alle Variablen $x_1, x_2, \ldots$, die Konstanten 0 und 1 und mit $E, F \in M$ auch die Formeln (E), $\neg E$, $(E \wedge F)$, $(E \vee F)$, $\forall x E$ und $\exists x E$ für alle Variablen x enthält. Sie läßt sich durch eine eindeutige kontextfreie Grammatik über $\Sigma = \{x, 0, 1, (,), \neg, \wedge, \vee, \forall, \exists\}$ erzeugen. Dabei werden die Variablen durch $x1\{0,1\}^*$ dargestellt. Klammern können oft auch weggelassen werden, wenn man die Vorrangregel $\wedge$ vor $\vee$ und Assoziativregeln für $\wedge$ und $\vee$ berücksichtigt.

Die Erfüllbarkeit von quantifizierten booleschen Formeln wird durch die Existenz einer erfüllenden Belegung definiert. Eine Belegung ist eine Funktion $b : \{x_1, x_2, \ldots\} \to \{0, 1\}$. Diese kann für eine gegebene Formel F natürlich auf die in F vorkommenden Variablen eingeschränkt werden. Für $z \in \{0, 1\}$, führen wir noch die Bezeichnung $b[x_j \mapsto z]$ ein für eine Belegung mit $b[x_j \mapsto z](x_i) = b(x_i)$ für $i \neq j$ und $b[x_j \mapsto z](x_j) = z$. Damit können wir die Erfüllbarkeit einer

Formel F bezüglich einer Belegung b rekursiv definieren: b *erfüllt* die Formel F genau dann, wenn für eine Variable x_j und Formeln E, F_1, F_2 eine der folgenden Bedingungen zutrifft:

$$
\begin{array}{lll}
F = 1, & & \\
F = x_j & \text{und} & b(x_j) = 1, \\
F = (E) & \text{und} & b \text{ erfüllt } E, \\
F = \neg E & \text{und} & b \text{ erfüllt } E \text{ nicht}, \\
F = (F_1 \wedge F_2) & \text{und} & b \text{ erfüllt } F_1 \text{ und } F_2, \\
F = (F_1 \vee F_2) & \text{und} & b \text{ erfüllt } F_1 \text{ oder } F_2, \\
F = \exists x_j E & \text{und} & b[x_j \mapsto 0] \text{ oder } b[x_j \mapsto 1] \text{ erfüllt } E, \\
F = \forall x_j E & \text{und} & b[x_j \mapsto 0] \text{ und } b[x_j \mapsto 1] \text{ erfüllen } E.
\end{array}
$$

Wird eine Formel von jeder Belegung erfüllt, so heißt sie *gültig*.

Eine Variable x_i heißt *frei* in einer Formel F, falls $F = x_i$, falls eine Formel E existiert, in der x_i frei ist mit $F = (E)$, $F = \neg E$, $F = \exists x_j E$ oder $F = \forall x_j E$ für ein $j \neq i$ oder falls Formeln E und G mit $F = (E \wedge G)$ bzw. $F = (E \vee G)$ so existieren, daß x_i in G oder in F frei ist. Eine Variable die in F vorkommt und nicht frei ist, bezeichnet man als *gebunden*. Eine Formel ohne freie Variablen nennt man *geschlossen*. Damit ist klar, daß die Erfüllbarkeit einer geschlossenen Formel nicht von der Belegung abhängt, d. h. existiert eine erfüllende Belegung, so ist die Formel gültig.

Bezeichnung: QBF sei die Menge der geschlossenen quantifizierten booleschen Formeln, die gültig sind.

Satz 5.3.1 QBF *ist* **PSPACE**-*vollständig.*

Beweis: Der Beweis für QBF $\in$ **PSPACE** sei dem Leser zur Übung überlassen.

QBF ist **PSPACE**-hart: Sei $L \in$ **PSPACE**. Es gilt $L = L(M)$ für eine $p(n)$-platzbeschränkte 1-Band-Turingmaschine, wobei $p(n)$ ein geeignetes Polynom sei. Im weiteren stellen wir uns die Konfigurationen binär kodiert vor. Ohne Einschränkung ist die Länge aller Binärkodierungen der von einer Startkonfiguration $\text{Start}(w)$, $|w| = n$, aus erreichbaren Konfigurationen durch $p(n)$ und deren Anzahl durch $2^{p(n)}$ begrenzt.

Betrachte den Savitch-Ansatz:

$$\text{Reach}(\text{Start}(w), \text{Accept}, p(n)) \Leftrightarrow w \in L$$

$$\text{Reach}(\alpha, \beta, i) = \exists \gamma \left(\text{Reach}(\alpha, \gamma, i-1) \wedge \text{Reach}(\gamma, \beta, i-1) \right) \quad \text{für } i > 0$$

$$\text{Reach}(\alpha, \beta, 0) = \alpha \vdash_M^{\leq 1} \beta$$

Die direkte Einsetzung würde auf eine Formel exponentieller Länge führen. Wir behelfen uns daher mit einem Trick. Wir führen für die Konfigurationen Konfigurationsvariable $X, Y, U, V, \ldots$ ein und definieren für $i > 0$:

$$\text{Reach}(X,Y,i) := \exists U\, \forall V\, \forall W \big((V{=}X \wedge W{=}U) \vee (V{=}U \wedge W{=}Y)\big) \Rightarrow \text{Reach}(V,W,i-1)$$

In einer ersten Reduktion berechnen wir auf eine Eingabe $w \in \Sigma^*$ mit $|w| = n$ durch iteriertes Einsetzen eine Formel in den Konfigurationsvariablen $X, Y, \ldots$ und den Konstanten Start(w) und Accept mit atomaren Formeln der Gestalt Reach$(X, Y, 0)$. Sei $p = p(n)$. Jetzt ersetzen wir jede Konfigurationsvariable X (bzw. $Y, \ldots$) durch einen Block von p booleschen Variablen $(x_1, \ldots, x_p)$ (bzw. $(y_1, \ldots, y_p), \ldots$). Das heißt aus $\forall X$ wird der Block $\forall x_1\, \forall x_2 \cdots \forall\, x_p$, entsprechend wird $\exists X$ zu $\exists x_1\, \exists x_2 \cdots \exists x_p$.

Aus $V = X$ erzeugen wir die Formel

$$\bigwedge_{i=1,\ldots,p} (v_i \Leftrightarrow x_i)\,,$$

d.h. $V = X$ wird als Makro gelesen.

Aus den atomaren Formeln Reach$(X, Y, 0)$ wird wie im Beweis zum Satz von Cook (Satz 4.2.1) eine Formel aus der Übergangstabelle von M erzeugt. Wir erhalten so mit Hilfe von Standardverfahren eine geschlossene quantifizierte Formel, die genau dann erfüllbar ist, wenn $w \in L$. ∎

Kapitel 6

Orakel-Turingmaschinen / „relative“ Schwierigkeit von $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

Eine Orakel-Turingmaschine $M^?$ ist eine Turingmaschine mit drei zusätzlichen Zuständen „ja“, „nein“ und „?“ und einem weiteren Arbeitsband („Orakelband“). Die Rechnung (für Akzeptanz) einer solchen Maschine $M^?$ ist relativ zu einem Orakel $A \subseteq \Sigma^*$ definiert. Hierbei kann A eine beliebige, auch unentscheidbare Sprache sein.

Auf eine Eingabe $v \in \Sigma^*$ rechnet $M^?$ zunächst wie gewöhnlich. Gerät $M^?$ während der Rechnung in den Zustand „?“, so ist der Folgezustand „ja“, falls die augenblickliche Orakelbandinschrift zu A gehört, andernfalls „nein“. Die übliche Konvention sieht vor, daß die Orakelbandinschrift im selben Schritt gelöscht wird. Die Eingabe wird akzeptiert, falls $M^?$ in einen Endzustand gerät.

Liegt das Orakel $A \subseteq \Sigma^*$ fest, so bezeichnen wir die Maschine $M^?$ auch mit M^A.

Wir können jetzt Komplexitätslassen wie

$$\begin{aligned}
\mathbf{P}^A &= \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l}\text{Es gibt eine deterministische polynomial zeitbe-}\\ \text{schränkte Turingmaschine } M^A \text{ mit } L = L(M^A)\end{array} \right\} \\
\mathbf{NP}^A &= \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l}\text{Es gibt eine nichtdeterministische polynomial zeit-}\\ \text{beschränkte Turingmaschine } M^A \text{ mit } L = L(M^A)\end{array} \right\}
\end{aligned}$$

definieren.

Bemerkung 6.0.2

i) $\mathbf{NP} \cup \textbf{co-NP} \subseteq \mathbf{P}^{\mathrm{SAT}} \subseteq \mathbf{PSPACE}$ Es ist offen, ob „=“ gilt.

ii) $A \in \mathbf{P}^A$ für alle $A \subseteq \Sigma^*$, damit kann insbesondere $\mathbf{P}^A$ unentscheidbare Sprachen enthalten.

iii) Ist A entscheidbar, so ist $\mathbf{NP}^A$ eine Familie entscheidbarer Sprachen.

iv) Trivialerweise gilt $\mathbf{P}^A \subseteq \mathbf{NP}^A$.

Satz 6.0.3 *Es gibt ein Orakel $A \subseteq \Sigma^*$ in* **PSPACE** *mit* $\mathbf{P}^A = \mathbf{NP}^A$.

Beweis: Betrachte eine **PSPACE**-vollständige Sprache, etwa $A = \text{QBF}$. Dann gilt

$$\mathbf{PSPACE} \subseteq \mathbf{P}^{\text{QBF}} \subseteq \mathbf{NP}^{\text{QBF}} \subseteq \bigcup_{k \geq 1} \text{NSPACE}(n^k) = \mathbf{PSPACE}$$

Also gilt

$$\mathbf{PSPACE} = \mathbf{P}^{\text{QBF}} = \mathbf{NP}^{\text{QBF}}$$

■

Satz 6.0.4 *Es gibt ein entscheidbares Orakel $B \subseteq \{0,1\}^*$ mit* $\mathbf{P}^B \neq \mathbf{NP}^B$.

Beweis: Das Orakel B wird über $\{0,1\}^*$ so definiert, daß pro Länge höchstens ein Wort $w \in B$ existiert. Zu jeder Sprache $B \subseteq \{0,1\}^*$ sei L_B durch

$$L_B = \left\{ 1^n \;\middle|\; \exists w \in B \text{ mit } |w| = n \right\}$$

definiert.

Dann gilt offensichtlich: $L_B \in \mathbf{NP}^B$.

Zu zeigen ist daher nur: Es gibt eine entscheidbare Sprache B so, daß $L_B \notin \mathbf{P}^B$.

Sei $M_1^?, M_2^?, \ldots$ eine Aufzählung deterministischen Orakel-Turingmaschinen mit zusätzlicher polynomialer Zeitschranke. Man beachte, das dies effektiv möglich ist (vgl. mit dem Beweis von Satz 4.4.1). Wir nehmen an, daß jede solche Turingmaschine in der Aufzählung unendlich oft wiederholt wird. Dies kann z.B. dadurch erreicht werden, daß $M^?(i,j) := M_i^?$ aus einer ursprünglichen Aufzählung definiert wird und dann die Orakel-Turingmaschinen aus $\{M^?(i,j) | (i,j) \in \mathbb{N} \times \mathbb{N}\}$ aufgezählt werden. Wir definieren rekursiv das Orakel durch Mengen B_i mit $B = \bigcup_{i \geq 0} B_i$ und eine Ausnahmemenge X mit $B \cap X = \emptyset$:

- Für $i = 0$ setze $B_0 = X = \emptyset$.
- Für $i > 0$ betrachte 1^i und simuliere eine Rechnung von $M_i^?$ auf 1^i für $i^{\lceil \log i \rceil}$ Schritte. Wir nehmen an, daß B_{i-1} und eine Menge X mit $\bigcup_{j \leq i-1} B_j \cap X = \emptyset$ schon definiert sind.
  - Befragt $M_i^?$ im Laufe der Rechnung ein Wort w mit $|w| < i$, so wird die Rechnung entsprechend dem Orakel B_{i-1} fortgesetzt.

- Befragt $M_i^?$ im Laufe der Rechnung ein Wort w mit $|w| \geq i$, so setze die Rechnung in dem „nein"-Zustand fort und füge w zu X hinzu.
- Akzeptiert $M_i^?$ innerhalb der Zeitschranke $i^{\lceil \log i \rceil}$ die Eingabe 1^i, so setze $B_i := B_{i-1}$.
- Verwirft $M_i^?$ innerhalb der Zeitschranke $i^{\lceil \log i \rceil}$ die Eingabe 1^i, so betrachte das lexikographisch erste Wort b_i in $\{0,1\}^i \setminus X$. Setze dann

$$B_i = \begin{cases} B_{i-1} \cup \{b_i\} & \text{falls } b_i \text{ existiert} \\ B_{i-1} & \text{sonst} \end{cases}$$

Es gilt mit $B = \bigcup\limits_{i \geq 0} B_i$:

- $B_i \subseteq \{w \in \{0,1\}^* \mid |w| \leq i\}$
- B ist entscheidbar, denn für $|w| = i$ gilt: $w \in B \Leftrightarrow w \in B_i$.
- $B \cap X = \emptyset$.

Wir zeigen jetzt $L_B \notin \mathbf{P}^B$. Angenommen, es wäre $L_B = L(M^B)$ für eine deterministische, polynomial zeitbeschränkte Orakel-Turingmaschine M^B. Dann gibt es ein Polynom $p(n)$ so, daß M^B $p(n)$-zeitbeschränkt ist. Wähle jetzt i genügend groß so, daß $\forall n \geq i : p(n) \leq n^{\lceil \log i \rceil}$ und $M^B = M_i^B$ gilt. Dies ist möglich, da $M^?$ in der Aufzählung beliebig oft vorkommt.

1. Fall: $1^i \in L_B = L(M_i^B)$, dann gilt $B_{i-1} = B_i$ nach Konstruktion von B und damit $1^i \notin L_B$ nach Definition von L_B. Dies ist ein Widerspruch.

2. Fall: $1^i \notin L_B = L(M_i^B)$, d.h. M_i^B verwirft 1^i innerhalb der Zeitschranke $i^{\lceil \log i \rceil}$. Ist $\{0,1\}^i \setminus X \neq \emptyset$, so existiert ein $b_i \in B$ mit $|b_i| = i$ und damit $1^i \in L_B$. Dies ist ein Widerspruch.

Es ist daher noch zu zeigen, daß für alle zuvor genügend groß gewählten i $\{0,1\}^i \setminus X \neq \emptyset$ gilt.

Bei der Definition von X bis zum i-ten Schritt wurden maximal

$$\sum_{j=0}^{i} j^{\lceil \log j \rceil} \leq i \cdot i^{\lceil \log i \rceil} \leq 2^{\log i + \lceil \log i \rceil^2}$$

Orakelanfragen gestellt. X enthält also maximal $2^{\log i + \lceil \log i \rceil^2}$ Wörter der Länge $\leq i$. Für alle zuvor genügend groß gewählten i gilt nun

$$2^{\log i + \lceil \log i \rceil^2} < 2^i,$$

weshalb $\{0,1\}^i \setminus X$ nicht leer ist. ∎

Kapitel 7

Dünne Mengen und der Satz von Mahaney

Eine Menge $L \subseteq \Sigma^*$ heißt *dünn*, wenn ein Polynom $p(n)$ so existiert, daß

$$\Big|\{w \in L \big| |w| = n\}\Big| \leq p(n).$$

Beispiel 7.0.5 Das Orakel B aus Satz 6.0.4 mit $\mathbf{P}^B \neq \mathbf{NP}^B$ ist dünn.

Mahaney hat gezeigt, daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ keine dünne Sprache **NP**-hart sein kann. Folglich gibt es unter dieser Voraussetzung keine dünnen **NP**-vollständigen Sprachen. Eine weitere Folgerung ist, daß es, falls $\mathbf{P} \neq \mathbf{NP}$ ist, sehr schwierige (bzw. sogar unentscheidbare) Sprachen gibt, die trotzdem nicht **NP**-hart sind.

Eine Sprache über einem einelementigen Alphabet heißt *unär*.

Beispiel 7.0.6 Die Sprache L_B aus dem Beweis zu Satz 6.0.4 ist unär.

Eine Abschwächung des Satzes von Mahaney ist die Aussage, daß entweder $\mathbf{P} = \mathbf{NP}$ gilt oder es keine unäre Sprache gibt, die **NP**-hart ist. Aufgrund der verwendeten interessanten Beweistechnik zeigen wir zunächst diesen schwächeren Satz.

Satz 7.0.7 *Gilt* $\mathbf{P} \neq \mathbf{NP}$*, so gibt es keine unäre Sprache, die* **NP***-hart ist.*

Beweis: Grundidee ist das Anlegen einer Tabelle, in die die Werte bereits berechneter Funktionsaufrufe zusammen mit den Aufrufparametern abgelegt werden. Dadurch kann (möglicherweise) eine Beschleunigung der Berechnung rekursiver Funktionen erreicht werden: Vor einem Rekursionsaufruf wird in der Tabelle nachgeschaut, ob der Wert mit den aktuellen Parametern nicht schon vorliegt.

Die zweite Idee ist die Selbstreduktion von SAT. Sei $S : \Sigma^* \to \{0,1\}$ die charakteristische Funktion von SAT, d.h. $S(F) = 1 \Leftrightarrow F \in \text{SAT}$. Sei F_i ($i \in \{0,1\}$) die Formel, die entsteht, wenn die erste Variable durch i belegt wird. Dann berechnet sich $S(F)$ durch das folgende Programm mit exponentieller Laufzeit:

```
FUNCTION S(F)
  if F hat keine freien Variablen then
    v := Auswertung von F
    return v
  else
    if S(F_1) = 1 then
      return 1
    else
      return S(F_0)
    endif
  endif
ENDFUNC
```

Wir nehmen an, daß $L \subseteq \{1\}^*$ **NP**-hart sei und zeigen, daß dann SAT $\in$ **P** gelten muß.

Sei $H : \Sigma \to \{1\}^*$ eine Polynomialzeitreduktion mit $H(F) \in L \Leftrightarrow F \in \text{SAT}$. (Die Bezeichnung H soll an eine Hash-Funktion erinnern.)

Betrachte jetzt das Programm S' in Abbildung 7.1 zur Berechnung derselben Funktion. Dieses Programm ergibt sich aus S mit Hilfe der oben erwähnten Technik, eine Tabelle bereits berechneter Funktionswerte einschließlich ihrer Aufrufparameter zu verwenden.

Da L eine unäre Sprache ist, gibt es ein Polynom $p(n)$ so, daß für $|F| = n$ die Tabelle $\mathcal{L}$ höchstens $p(n)$ Einträge enthält. Wir können uns die Reduktion $H(F)$ als eine Zahl $\leq p(n)$ vorstellen und $\mathcal{L}$ als ein Feld implementieren:

$$\textbf{array } [\,0..p(n)] \textbf{ of } (\textbf{bool} \cup \{\text{undefiniert}\}).$$

Behauptung: Die Laufzeit von S' ist jetzt durch ein Polynom beschränkt.

Die Berechnung von S' können wir uns als einen Binärbaum vorstellen. An den Blättern wird entweder F ausgewertet oder ein Tabelleneintrag gefunden.

Sei m die Zahl der Knoten in diesem Baum. Es reicht zu zeigen, daß die Anzahl der inneren Knoten durch ein Polynom beschränkt ist. Denn die Anzahl der Blätter ist kleiner oder gleich der Anzahl der inneren Knoten $+1$.

Beweis der Behauptung: Seien x, y zwei innere Knoten in dem Berechnungsbaum. Seien $t_1 \cdots t_i \in \{0,1\}^i$, $t'_1 \cdots t'_j \in \{0,1\}^j$ die entsprechenden Pfade, die zu diesen Knoten führen. Diese Pfade entsprechen Belegungen $x_1 = t_1$, $x_2 = t_2$, $\ldots$, $x_i = t_i$ bzw. $x_1 = t'_1$, $x_2 = t'_2$, $\ldots$, $x_j = t'_j$.

```
FUNCTION S'(F)
  if F hat keine freien Variablen then
    v := Auswertung von F
    (* man erhält einen booleschen Wert v = S'(F) ∈ {0,1} *)
    Lege (H(F), v) in einer Tabelle ℒ ab
    return v
  else
    if (H(F), v) ∈ ℒ für ein v ∈ {0,1} then   (* Beschleunigung *)
      return v
    else
      (* F_1 und F_0 seien wie oben definiert *)
      if S'(F_1) = 1 then
        Lege (H(F), 1) in ℒ ab
        return 1
      else
        if S'(F_0) = 1 then
          Lege (H(F), 1) in ℒ ab
          return 1
        else
          Lege (H(F), 0) in ℒ ab
          return 0
        endif
      endif
    endif
  endif
ENDFUNC
```

Abbildung 7.1: Berechnung von S mittels Hash-Tabelle

Wir tragen diese Werte in F ein und erhalten zwei Formeln:

$$F' = F|_{x_k = t_k,\, 1 \le k \le i}$$

$$F'' = F|_{x_k = t'_k,\, 1 \le k \le j}$$

Sind die Knoten x und y unvergleichbar, d.h. ist $t_1 \cdots t_i$ kein Präfix von $t'_1 \cdots t'_j$ und umgekehrt, so gilt

$$H(F') \neq H(F'')\,.$$

Die Behauptung ergibt sich aus dem Programm: wenn x nach y besucht wird, so ist der Teilbaum von y ganz abgearbeitet und der Wert $(H(F''), v)$ in $\mathcal{L}$ eingetragen. Da x kein Blatt ist, muß dann $H(F') \neq H(F'')$ gelten.

Es gibt höchstens n freie Variable in F, damit ist die Tiefe des Baumes durch n begrenzt. Die Zahl der unvergleichbaren Knoten ist $\le |\mathcal{L}| \le p(n)$. Damit ist die

Zahl der inneren Knoten $\leq n \cdot |\mathcal{L}| \leq n \cdot p(n)$. Somit ist m, wie behauptet, durch ein Polynom begrenzt. ■

Das Problem bei allgemeinen dünnen Sprachen ist, daß es für eine Länge $|H(F)|$ exponentiel viele Einträge der Form $(H(F), 0)$ in $\mathcal{L}$ geben könnte. Daß dieses Problem umgangen werden kann, zeigt der Satz von Mahaney. Der ursprüngliche Beweis von Mahaney selbst ist ziemlich kompliziert und umfangreich, jedoch haben Ogiwara und Watanabe einen relativ leicht verständlichen Beweis gefunden, den wir hier angeben.

Satz 7.0.8 (Mahaney 1982) *Gilt* **P** $\neq$ **NP***, so gibt es keine dünnen* **NP***-harten Sprachen.*

Beweis: (nach Ogiwara und Watanabe (1990)) Es wird zunächst eine Menge SAT′ definiert:

$$\text{SAT}' = \Big\{\langle F, x\rangle \Big| \begin{array}{l} F \text{ ist boolesche Formel in den Variablen } x_1, \ldots, x_n, \\ x \in \{0,1\}^n \;\bigwedge\; \exists y \in \{0,1\}^n : (y \geq x \text{ (in lexikogra-} \\ \text{phischer Ordnung)} \wedge F(y) = 1) \end{array}\Big\}$$

Hierbei bedeutet $F(y) = 1$, daß der Bitstring y als Belegung der Variablen $x_1, \ldots, x_n$ interpretiert wird und F sich unter dieser Belegung zu wahr auswertet.

Behauptung: SAT′ ist **NP**-vollständig.

(i) **SAT′ ∈ NP:**
Rate eine Belegung $y \geq x$ und überprüfe, ob $F(y) = 1$ gilt.

(ii) **SAT′ ist NP-hart:**
O.B.d.A. enthalte F die Variablen $x_1, \ldots, x_n$. Wir reduzieren SAT auf SAT′ mit der Abbildung $F \mapsto \langle F, 0^n\rangle$. Es gilt offensichtlich:

$$F \in \text{SAT} \;\Leftrightarrow\; \langle F, 0^n\rangle \in \text{SAT}'$$

Für eine Formel F (in den Variablen $x_1, \ldots, x_n$) definieren wir eine Funktion $f : [0, \ldots, 2^n - 1] \to \{0, 1\}$ mit

$$f(x) \;=\; \begin{cases} 1, \text{ falls } \langle F, x\rangle \in \text{SAT}' \\ 0, \text{ sonst} \end{cases}$$

Der Definitionsbereich ist die Menge $\{0,1\}^n$, die wir als Binärzahlen in dem Intervall (exponentieller Länge) $[0, \ldots, 2^n - 1]$ interpretieren.

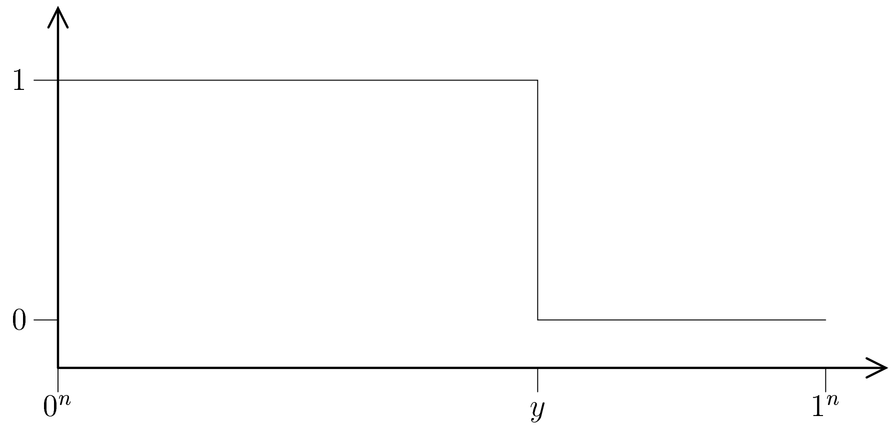


Abbildung 7.2: Treppenfunktion f

Der Graph dieser Funktion ist eine Treppenfunktion. Er ist in Abbildung 7.2 dargestellt. Der Wert y ist dabei die (lexikographisch) größte erfüllende Belegung für F.

Wir werden im folgenden zeigen: Wenn es eine dünne **NP**-harte Sprache $L \subseteq \{0,1\}^*$ gibt, auf die wir SAT′ in Polynomialzeit reduzieren können, dann liegt SAT in **P** und es folgt **P** = **NP**.

Die Länge einer Eingabe $\langle F, x\rangle$ definieren wir im folgenden zu $n = |F|$ und wir können ohne Einschränkung annehmen, daß die Formel F nur Variablen aus der Menge $x_1, \ldots, x_n$ enthält.

Sei $H : \Sigma^* \to \{0,1\}$ die bereits erwähnte Polynomialzeitreduktion von SAT′ auf L, also

$$\langle F, x\rangle \in \mathrm{SAT}' \;\Leftrightarrow\; H(\langle F, x\rangle) \in L$$

Da L nach Annahme eine dünne Sprache ist, gibt es ein Polynom $p(n)$ so, daß

$$\Big|\big\{H(\langle F, x\rangle) \mid \langle F, x\rangle \in \mathrm{SAT}' \;\wedge\; |\langle F, x\rangle| \le n\big\}\Big| \quad\le\quad p(n).$$

Sei I das Intervall $[0, \ldots, 2^n - 1]$. Wir unterteilen I in $2 \cdot p(n)$ gleichgroße Teilintervalle. Jedes Intervall wird durch die linke Grenze und die Länge repräsentiert. Für die linken Grenzen x_i wird $H(\langle F, x_i\rangle)$ berechnet und in einer Tabelle abgelegt. Wenn dabei zwei Tupel $\langle F, x_i\rangle$ und $\langle F, x_j\rangle$ auf denselben Wert abgebildet werden, findet eine Kollisionsauflösung statt:

> Sei ohne Einschränkung $x_i < x_j$. Dann werden alle Teilintervalle mit linken Grenzen $x_i, x_{i+1}, \ldots, x_{j-1}$ gelöscht, denn alle haben denselben Funktionswert $f(x_j)$.

> Man beachte, daß dabei das Intervall, das die größte erfüllende Belegung y enthält (sofern überhaupt eine erfüllende Belegung existiert), nicht gelöscht wird: denn aus $x_i < y < x_j$ würde sich $H(\langle F, x_i \rangle) \neq H(\langle F, x_j \rangle)$ ergeben.

Am Ende dieser Prozedur sind die Werte $H(\langle F, x_i \rangle)$ für alle linken Grenzen der verbleibenden Intervalle verschieden. Da es nur maximal $p(n)$ positive Antworten ($f(x_k) = 1$) geben kann, sind $\langle F, x_{p(n)+1} \rangle, \ldots, \langle F, x_l \rangle \notin \mathrm{SAT}'$ und die zugehörigen Intervalle mit den linken Grenzen $x_{p(n)+1}, \ldots, x_l$ können gelöscht werden, ohne das Intervall mit der größten erfüllenden Belegung y zu löschen. Am Ende bleiben also höchsten $p(n)$ Intervalle übrig.

Jetzt wird eine Intervallhalbierung bei den verbleibenden Intervallen durchgeführt. Dies erzeugt maximal $2 \cdot p(n)$ Intervalle, die nach dem gleichen Verfahren wiederum auf maximal $p(n)$ Intervalle verringert werden. Dann erfolgt eine erneute Intervallhalbierung usw.

Nach höchstens n Halbierungen gibt es nur noch 1-Punkt-Intervalle (da das Intervall I 2^n Elemente enthält) und das Verfahren der Intervallhalbierungen wird abgebrochen. Die bis zu diesem Zeitpunkt verbrauchte Zeit ist polynomial beschränkt.

Seien $x_1, \ldots, x_l$ ($l \leq p(n)$) die linken Grenzen der verbleibenden 1-Punkt-Intervalle. Jetzt kann in polynomialer Zeit für jedes i, $1 \leq i \leq l$, der Wert $F(x_i)$ berechnet werden. Ist $F(x_i) = 0$ für alle i, so ist F unerfüllbar. Sonst ist F erfüllbar (das größte x_i mit $F(x_i) = 1$, ergibt die größte erfüllende Belegung). Damit können wir in Polynomialzeit entscheiden, ob $F \in \mathrm{SAT}$. ■

Kapitel 8

Interaktive Beweissysteme

In diesem Kapitel betrachten wir das folgende Szenario:

> Es gibt zwei Personen in unserem System: Alice und Bob. **Alice** ist „allmächtig“ (mindestens jedoch eine **PSPACE**-Maschine). **Bob** ist „bieder und ehrlich“ (d.h. eine deterministische, polynomial-zeitbeschränkte Turingmaschine) und besitzt ein Würfel. Bob stellt Fragen an Alice. Dieses Frage-Antwort Wechselspiel wird als *Protokoll* bezeichnet.

Ein *interaktives Beweissystem* (IBS) ist ein solches Protokoll. Anschaulich versucht Alice für ein $x \in \Sigma^*$ stets, Bob von der Zugehörigkeit dieses Wortes zu einer Sprache L zu überzeugen. Eine Sprache L läßt sich durch ein IBS entscheiden, wenn Bob für ein $x \in L$ Alice mit hoher Wahrscheinlichkeit glaubt und sich für $x \notin L$ mit sehr geringer Wahrscheinlichkeit vom Gegenteil überzeugen läßt (Eine exaktere Definition dieser Begriffe wird im Abschnitt 8.2 gegeben). Wir werden zunächst zwei Spezialfälle interaktiver Beweissysteme genauer betrachten.

8.1 Zero-Knowledge-Proofs

Ein *Zero-Knowledge-Proof* ist dadurch gekennzeichnet, daß Bob zwar davon überzeugt wird, daß Alice einen Beweis für das betrachtete Problem kennt, aber trotzdem keinerlei Information über den Beweis oder eine mögliche Lösung des Problems erhält.

8.1.1 3-Färbbarkeit von Graphen

Definition 8.1.1 (3-Färbbarkeit eines Graphen) Ein Graph $G = (V, E)$ mit $E \subseteq \binom{V}{2}$ (also ohne Schlingen[1]) heißt 3-färbbar, falls es eine Funktion $c : V \to \{0, 1, 2\}$ gibt, so daß gilt: $(i, j) \in E \Rightarrow c(i) \neq c(j)$.

Unter der Annahme, daß das kryptologische RSA-Verfahren sicher ist, können wir den folgenden Satz beweisen.

Satz 8.1.2 *Für das 3-Färbbarkeitsproblem eines Graphen gibt es einen (interaktiven) Zero-Knowledge-Proof.*

Beweis: Wir nehmen nun an, daß Alice die 3-Färbbarkeit von Graphen entscheiden kann[2]. Ein Beweis für die 3-Färbbarkeit eines Graphen wäre die Angabe einer Lösung. Alice gibt im folgenden Beweissystem diese Lösung jedoch nicht preis, sondern überzeugt Bob nur davon, daß sie sie kennt. Dies wird erreicht durch die RSA-Verschlüsselung, die Bob — falls die Verschlüsselung sicher ist — nicht knacken kann.

Alice und Bob einigen sich auf das folgende Protokoll: Bei einem Graphen $G = (V, E)$ als Eingabe werden n^3 Runden durchgeführt (mit $n = |V|$). In jeder Runde wählt Alice eine Zufallspermutation der Farben, die wir mit $\pi : \{0, 1, 2\} \tilde{\to} \{0, 1, 2\}$ bezeichnen[3] und erzeugt ein RSA-Kryptosystem für jeden Knoten i. Alice wählt also jeweils große Primzahlen $p_i, q_i < 2^n$ und eine Zahl $s_i < 2^n$ mit $\mathrm{ggT}(s_i, \varphi(n_i)) = 1$, wobei wir $n_i = p_i q_i$ setzen und mit $\varphi(n_i) = (p_i - 1)(q_i - 1)$ die Eulersche φ-Funktion bezeichnen. Dies ist die Anzahl der zu n_i teilerfremden Zahlen in $[0, \ldots, n_i - 1]$. Algebraisch ausgedrückt, ist $\varphi(n_i)$ die Ordnung der Gruppe, die von den multiplikativ invertierbaren Elementen in $\mathbb{Z}/n_i\mathbb{Z}$ gebildet wird.

Alice berechnet ein $0 < e_i < n_i$ mit $s_i e_i \equiv 1 \mod (p_i - 1)(q_i - 1)$. Es gilt jetzt für alle $a \in \mathbb{N}$: $a^{s_i e_i} \equiv a \mod n_i$. Alice wählt desweiteren Zufallszahlen x_i mit $0 < 3x_i + 2 < n_i$.

Das Protokoll beginnt mit Alice. Sie sendet Bob eine Liste $(n_i, s_i, z_i)_{i=1,\ldots,n}$ mit $z_i \equiv (3x_i + \pi(c(i)))^{s_i} \mod n_i$.

Bob wählt nun zufällig eine Kante $(i, j) \in E$. Er fragt Alice nach p_k, q_k, e_k (für $k \in \{i, j\}$) sowie nach den Beweisen, daß die p_k Primzahlen sind. Er überprüft $n_k = p_k q_k$ und $s_k e_k \equiv 1 \mod (p_k - 1)(q_k - 1)$ und berechnet $y_k = z_k^{e_k} \mod n_k$. Man beachte, daß $z_k^{e_k} = 3x_k + \pi(c(k)) \mod n_k$. Er berechnet außerdem Werte

[1] $\binom{V}{2}$ ist die Menge aller 2-elementigen Teilmengen von V.

[2] Es ist bekannt, daß das Problem der 3-Färbbarkeit eines Graphen (also $\{G = (V, E) | G$ ist 3-färbbar$\}$) **NP**-vollständig ist.

[3] $\tilde{\to}$ steht für eine bijektive Abbildung

$c_k \in \{0,1,2\}$ durch $c_k = y_k \bmod 3$. Hat sich Alice korrekt verhalten, dann gilt: $c_k = \pi(c(k))$. Bob ist mit dieser Runde einverstanden, falls $c_i \neq c_j$ gilt.

Beachte, daß Alice pro Runde nur die Frage nach einer Kante beantwortet (um den Beweis nicht preiszugeben) und Alice nicht im voraus wissen darf, nach welcher Kante Bob fragt.

Falls Bob n^3 Runden einverstanden ist, akzeptiert er G als 3-färbbar. Er akzeptiert also auf jeden Fall, wenn G 3-färbbar ist. Über die Färbung von G hat er nichts gelernt. Wie groß ist dabei die Wahrscheinlichkeit, daß Alice unerkannt betrogen hat?

Angenommen, G ist nicht 3-färbbar. Dann gibt es mindestens eine Kante $(i,j) \in E$, an der Bob merken kann, daß Alice betrügt (zum Beispiel daran, daß eine der p_k keine Primzahl ist). Ist das RSA-System jedoch in Ordnung, so merkt Bob, daß $c_i = c_j$ und damit $\pi(c(i)) = \pi(c(j))$, also $c(i) = c(j)$ ist. Alice wird also pro Runde mindestens mit der Wahrscheinlichkeit $\frac{1}{|E|}$ entdeckt. Insgesamt bleibt sie folglich mit einer Wahrscheinlichkeit kleiner $\left(1 - \frac{1}{|E|}\right)^{n^3}$ unentdeckt. Wir zeigen $\left(1 - \frac{1}{|E|}\right)^{n^3} < 2^{-n}$, die Wahrscheinlichkeit, daß Bob einen nicht 3-färbbaren Graph als 3-färbbar akzeptiert ist also sehr klein. Es gilt:

$$\begin{array}{rrcll}
 & \left(1 - \frac{1}{|E|}\right)^{n^3} & \leq & e^{-n} & (< 2^{-n} \text{ für alle } n > 0) \\
\Leftrightarrow & n^2\left(\ln(|E|) - \ln(|E|-1)\right) & \geq & 1 & \\
\Leftarrow & \ln(|E|) - \ln(|E|-1) & \geq & \frac{1}{|E|} & (\text{wegen } |E| < n^2)
\end{array}$$

Es reicht daher zu zeigen: $\ln(|E|) - \ln(|E|-1) \geq \frac{1}{|E|}$. Die Korrektheit dieser Ungleichung folgt aus der Ableitung $\ln'(x) = \frac{1}{x}$. Wie man dem Graphen der ln-Funktion unmittelbar ansieht, gilt für alle $x > 1$ die Abschätzung $\ln'(x) < \ln(x) - \ln(x-1)$. Man beachte, daß $\ln'(x)$ streng monoton fallend ist und daher der Differenzenquotient $(\ln(x) - \ln(x-h))/h$ für $0 < h < x$ größer als $\ln'(x)$ ist. Die Behauptung folgt mit $x = |E|$. ■

Bemerkung 8.1.3 Da das 3-Färbbarkeitsproblem eines Graphen **NP**-vollständig ist, hat jedes **NP**-Problem einen Zero-Knowledge-Proof, der auf dem RSA-Verfahren basiert.

8.1.2 Die (Nicht-)Isomorphie von Graphen

Das Problem, ob zwei Graphen isomorph sind, liegt in **NP**. Es ist weder bekannt, ob es in **P** liegt, noch, ob es **NP**-vollständig ist. Es gibt allerdings Hinweise, die darauf hindeuten, daß dieses Problem nicht **NP**-hart ist (vgl. [Sch87]).

Es ist nicht bekannt, ob das Problem der Nichtisomorphie von Graphen in **NP** liegt. Intuitiv erscheint dieses Problem jedoch schwieriger als das der Isomorphie zweier Graphen. Bei der Isomorphie können wir einen Isomorphismus raten und überprüfen — eine Vorgehensweise, die uns bei der Nichtisomorphie nicht möglich ist.

Satz 8.1.4 *Die Nichtisomorphie von Graphen hat einen perfekten Zero-Knowledge-Proof.*

Beweis: Die Eingabe besteht aus 2 Graphen $G_0 = (V_0, E_0)$ und $G_1 = (V_1, E_1)$ mit $V_0 = V_1 = \{1, \dots, n\}$ (wäre $|V_0| \neq |V_1|$, dann kann der Vergleich der beiden Graphen sofort abgebrochen werden).

Wir nehmen an, daß Alice die Nichtisomorphie zweier Graphen entscheiden kann und behauptet: $G_0 \ncong G_1$ (d.h. G_0 ist nicht isomorph zu G_1). Falls die beiden Graphen doch isomorph sind, so kann sie mit dem folgenden Protokoll Bob nur mit der Wahrscheinlichkeit von 2^{-m} betrügen, wobei m polynomial in n gewählt werden kann.

Bob wählt m Permutationen $\pi_i \in \mathrm{Perm}(\{1, \dots, n\})$ und m Bits $b_i \in \{0, 1\}$ zufällig ($1 \leq i \leq m$). Dann übermittelt Bob an Alice in einer einzigen Runde die Liste der Paare $\langle G_0, \pi_i(G_{b_i}) \rangle_{1 \leq i \leq m}$. Bob verlangt von Alice jetzt einen Bitstring $b'_1 b'_2 \dots b'_m$ und akzeptiert, falls $\forall i \in \{1, \dots, m\}\ b_i = b'_i$. Sind G_0 und G_1 nicht isomorph, so kann Alice diesen Bitstring berechnen. Sind sie jedoch isomorph, so kann Alice den Bitstring $b_1 b_2 \dots b_m$ nicht rekonstruieren. Denn Alice sieht nur Paare von G_0 und einer Zufallspermutation von G_0. Der Bitstring $b'_1 b'_2 \dots b'_m$ muß also zufällig gewählt werden — Alice hat keine andere Wahl. Wenn Alice betrügen muß, dann ist die Wahrscheinlichkeit, nicht entdeckt zu werden, damit gleich 2^{-m}. ∎

8.2 IP = PSPACE

Definition 8.2.1 (Interaktives Beweissystem, IP)

(i) Ein **interaktives Beweissystem** (IBS) ist ein Protokoll (A, B) zwischen Alice und Bob. Alice ist eine **PSPACE**-Maschine und Bob eine randomisierte polynomial zeitbeschränkte Turingmaschine[4]. Alice und Bob erhalten die gleiche Eingabe x (mit $|x| = n$), die von beiden nur gelesen werden darf. In dem Protokoll werden polynomial viele Runden durchgeführt ($p(n)$) und

[4]Eine randomisierte Turingmaschine (RTM) ist eine deterministische Turingmaschine, die zusätzlich Zugriff auf Zufallszahlen hat. Man kann sich eine RTM als eine deterministische Turingmaschine vorstellen, die ein weiteres (Nur-Lese-)Band besitzt, auf dem unendlich viele Zufallsbits (0 oder 1) stehen und das nur in einer Richtung gelesen werden darf.

in jeder Runde werden polynomial lange Botschaften ausgetauscht (diese Botschaften seien $a_1, b_1, a_2, b_2, \ldots, a_{p(n)}, b_{p(n)}$). Jede Runde beginnt mit einer Botschaft von Alice, die von der Eingabe und dem Ergebnis vorheriger Runden abhängt ($a_i = A(x, a_1, b_1, \ldots, b_{i-1})$). Bobs Antwort hängt ebenfalls von der Eingabe und den bisher ausgetauschten Botschaften sowie von Zufallszahlen r_i ab, die in jeder Runde neu berechnet werden und Alice unbekannt sind ($b_i = B(x, a_1, b_1, a_2, \ldots, a_{i-1}, b_{i-1}, a_i, r_1, \ldots, r_i)$). In der letzten Runde sagt Bob „ja" oder „nein" (also $b_{p(n)} \in \{\text{ja, nein}\}$).

(ii) Ein IBS (A, B) entscheidet eine Sprache $L \subseteq \Sigma^*$, falls für alle Eingaben $x \in \Sigma^*$ folgendes gilt:

   (1) Ist $x \in L$, so akzeptiert Bob die Eingabe mit einer Wahrscheinlichkeit $\geq (1 - 2^{-n})$.

   (2) Ist $x \notin L$, so gibt es kein IBS (A', B) so, daß Bob die Eingabe mit einer höheren Wahrscheinlichkeit als 2^{-n} akzeptiert.

(iii) **IP** ist die Menge der Sprachen, die durch ein IBS entschieden werden können:

$$\mathbf{IP} \;:=\; \{L \subseteq \Sigma^* \mid \exists \text{ IBS } (A, B) : (A, B) \text{ entscheidet } L\}$$

Beispiele 8.2.2 Die bereits betrachteten Protokolle zur 3-Färbbarkeit von Graphen (Abschnitt 8.1.1) und zur Graphen-Nichtisomorphie (Abschnitt 8.1.2) sind interaktive Beweissysteme.

Bemerkung 8.2.3 Es gilt **NP** $\subseteq$ **IP**, da Alice die erfolgreiche Rechnung einer **NP**-Maschine Bob zur Überprüfung vorlegen kann. Für ein spezielles Problem aus **NP** kann Alice auch eine Lösung angeben: Für SAT übermittelt Alice beispielsweise eine erfüllende Belegung an Bob. Bob wertet die Formel unter dieser Belegung aus und akzeptiert, wenn sich als Wert der Formel „wahr" ergibt.

Lemma 8.2.4 **IP** *ist unter polynomialer Zeitreduktion abgeschlossen:*

$$L \leq_m^p L' \text{ und } L' \in \mathbf{IP} \;\Rightarrow\; L \in \mathbf{IP}$$

Beweis: Der Beweis zu diesem Lemma ist trivial, denn Bob kann die Reduktion $L \leq_m^p L'$ berechnen, da er eine polynomial zeitbeschränkte Turingmaschine ist. ■

Satz 8.2.5 **IP** $\subseteq$ **PSPACE**.[5]

[5] Auch eine Allmächtigkeit, die wir für Alice erlauben, verändert die Größe von **IP** nicht, d.h. wäre Alice mächtiger als eine **PSPACE**-Maschine, dann würde trotzdem **IP** $\subseteq$ **PSPACE** gelten. Die Beschränkung eines IBS liegt im Austausch polynomial langer polynomial vieler Botschaften.

Beweis: Wir skizzieren hier nur knapp die Beweisidee (für die genaue Ausführung siehe Literatur). Ein Protokoll zwischen Alice und Bob kann durch einen Berechnungsbaum polynomialer Tiefe dargestellt werden, der sämtliche möglichen Interaktionen zwischen Alice und Bob simuliert. ■

Satz 8.2.6 (Shamir, 1990) IP = PSPACE.

Beweis:

Die leichte Richtung (**IP** $\subseteq$ **PSPACE**) ist nach Satz 8.2.5 bereits bekannt. Wir müssen also nur noch **PSPACE** $\subseteq$ **IP** beweisen. Nach Lemma 8.2.4 reicht es zu zeigen, daß QBF $\in$ **IP** gilt, da QBF **PSPACE**-vollständig ist (vgl. Satz 5.3.1). Wir werden im folgenden sogar ein interaktives Beweissystem konstruieren, in dem Bob die Zufallszahlen bekannt gibt und am Ende (mit Wahrscheinlichkeit 1) „ja" sagt, falls $x \in$ QBF gilt. (Um diesen ziemlich umfangreichen und technisch komplizierten Beweis besser verstehen zu können, folgt im Anschluß ein Beispiel, an dem das Protokoll zwischen Alice und Bob im einzelnen nachvollzogen wird.)

Eingabe ist also eine geschlossene quantifizierte Formel F über $\wedge, \vee, \forall, \exists$ und den Literalen $\tilde{x_1}, \tilde{x_2}, \ldots$ mit $\tilde{x_i} \in \{x_i, \overline{x_i}\}$ (wir nehmen ohne Einschränkung an, daß Negation nur auf Variablen und nicht auf größere Teilausdrücke angewandt wird).

Eine solche Formel F wird wie folgt in einen arithmetischen Ausdruck $a(F)$ umgeformt. Wir ersetzen

$\overline{x}$ durch $(1 - x)$

$\vee$ durch $+$

$\wedge$ durch $\cdot$ („Malpunkt")

$\forall x$ durch $\prod\limits_{x=0,1}$

$\exists x$ durch $\sum\limits_{x=0,1}$

Mit dieser Übersetzung gilt:

$$\begin{aligned} F \in \text{QBF} &\Leftrightarrow a(F) > 0 \\ F \notin \text{QBF} &\Leftrightarrow a(F) = 0 \end{aligned}$$

Beispiel 8.2.7 Sei $F = \forall x \exists y \Big((x \vee \overline{y}) \wedge \exists z (\overline{x} \wedge z) \Big)$. Dann ist

$$\begin{aligned}
a(F) &= \prod_{x=0,1} \Big(\sum_{y=0,1} \Big((x + (1-y)) \cdot \sum_{z=0,1} ((1-x) \cdot z) \Big) \Big) \\
&= \prod_{x=0,1} \Big(\sum_{y=0,1} \Big((x + (1-y)) \cdot (1-x) \Big) \Big) \\
&= \prod_{x=0,1} \Big((1-x^2) + (x - x^2) \Big) \\
&= \prod_{x=0,1} \Big(-2x^2 + x + 1 \Big) \\
&= 1 \cdot 0 \\
&= 0
\end{aligned}$$

Das bedeutet, daß die Auswertung der Formel F „falsch“ ergibt.

Für die Negation von F gilt: $\neg F = \exists x \forall y \Big((\overline{x} \wedge y) \vee \forall z (x \vee \overline{z}) \Big)$. Damit folgt

$$\begin{aligned}
a(\neg F) &= \sum_{x=0,1} \Big(\prod_{y=0,1} \Big((1-x) \cdot y + \prod_{z=0,1} (x + 1 - z) \Big) \Big) \\
&= \sum_{x=0,1} \prod_{y=0,1} \Big((1-x) \cdot y + x^2 + x \Big) \\
&= \sum_{x=0,1} (x^2 + x) \cdot (1 + x^2) \\
&= 4
\end{aligned}$$

Die Auswertung der Formel $\neg F$ ergibt also „wahr“.

Es stellt sich die Frage, wie groß die Werte von $a(F)$ werden können. Im folgenden sei n die Länge der Eingabeformel, also $|F| = n$. Es gilt dann:

Lemma 8.2.8 $a(F) \leq 2^{2^n}$.

Beweis: Bei den Operationen werden die Werte für Teilformeln addiert oder multipliziert. Die Behauptung folgt mit Induktion wegen $2^{2^{n-1}} \cdot 2^{2^{n-1}} = 2^{2^n}$. ∎

Beispiel 8.2.9 Sei $F = \forall x_1 \dots \forall x_k \exists y \exists z (y \vee z)$. Dann ist

$$\begin{aligned}
a(F) &= \prod_{x_1=0,1} \cdots \prod_{x_k=0,1} \sum_{y=0,1} \sum_{z=0,1} (y+z) \\
&= \prod_{x_1=0,1} \cdots \prod_{x_k=0,1} \sum_{y=0,1} (2y+1) \\
&= \prod_{x_1=0,1} \cdots \prod_{x_k=0,1} (4) \\
&= 4^{2^k}
\end{aligned}$$

An diesem Beispiel sehen wir, daß $a(F)$ tatsächlich den Wert $2^{2^{\Omega(n)}}$ erreichen kann. Solche Zahlen haben aber auch in Binärcodierung noch exponentielle Länge und können folglich nicht in einer polynomial langen Botschaft übermittelt werden. Sie sind selbst für Alice zu groß.

Um dieses Problem zu umgehen, werden wir sämtliche Rechnungen modulo einer Primzahl p durchführen. Aus dem schwachen Primzahlsatz folgt, daß ein geeignetes p existieren muß.

Lemma 8.2.10 (schwacher Primzahlsatz, ohne Beweis)
Zwischen 2^n und 2^{3n} liegen mindestens 2^n verschiedene Primzahlen.

Wir wenden dieses Lemma für genügend große n an. Dann ist es eine formale Konsequenz aus dem sehr tiefliegenden Primzahlsatz, der in der analytischen Zahlentheorie bewiesen wird und eine wesentlich schärfere Abschätzung über die Zahl der Primzahlen bis zu einer gewissen Größe liefert[6].

Seien $p_1, \dots, p_k$ die Primzahlen zwischen 2^n und 2^{3n}. Dann gilt: $m = p_1 \cdot p_2 \cdot \dots \cdot p_k > (2^n)^{(2^n)}$ und damit natürlich auch $m > a(F) \in 2^{2^{\Omega(n)}}$. Damit gilt:

$$\begin{aligned}
F \notin \text{ QBF } &\Leftrightarrow a(F) \equiv 0 \bmod m \quad (\text{da } a(F) = 0) \\
F \in \text{ QBF } &\Leftrightarrow \exists \text{ Primzahl } p, 2^n < p < 2^{3n} : a(F) \not\equiv 0 \bmod p
\end{aligned}$$

Die zweite Zeile folgt aus der Tatsache, daß $m = p_1 \cdot p_2 \cdot \dots \cdot p_k > a(F)$ gilt und $a(F)$ damit nicht alle p_i als Teiler haben kann.

Falls nun $F \in$ QBF gilt, dann berechnet Alice das kleinste $p > 2^n$ mit der Eigenschaft $a(F) \not\equiv 0 \bmod p$. Sie sendet den Wert p zusammen mit dem Beweis dafür,

[6]Sei $\pi(x) = \#\{p \leq x | p \text{ ist Primzahl}\}$. Dann verhält sich $\pi(x)$ asymptotisch wie $\frac{x}{\ln(x)}$. Damit wäre eine noch exaktere Abschätzung möglich, die für den hier geführten Beweis aber keine Bedeutung hat.

daß p eine Primzahl ist, an Bob. Man beachte, daß die Länge der Binärcodierung von p in $\mathcal{O}(n)$ liegt. Alle weiteren Rechnungen werden von nun an modulo p durchgeführt.

Jetzt wird aus einem arithmetischen Ausdruck $a(F)$ ein Polynom wie folgt berechnet (wir nehmen o.B.d.A. an, daß die Formel F mit einem Quantor beginnt): wenn F von der Form $F = \forall x F'$ (bzw. $F = \exists x F'$) ist, so gilt: $a(F) = \prod_{x=0,1} a(F'(x))$ (bzw. $a(F) = \sum_{x=0,1} a(F'(x))$). Wenn in $a(F)$ das erste Zeichen $\prod$ (bzw. $\sum$) gestrichen wird, so erhalten wir ein Polynom in der freien Variablen x. Der Grad dieses Polynoms kann jedoch exponentiell sein. Dieses Problem kann beseitigt werden, indem zu einfachen Formeln übergegangen wird.

Definition 8.2.11 (einfache Formel)
Eine Formel heißt *einfach*, wenn zwischen der Quantifizierung $Qx,\ Q \in \{\forall, \exists\}$ und jedem Auftreten der Variablen x höchstens ein $\forall$-Quantor auftritt.

Lemma 8.2.12 *Jede geschlossene quantifizierte Formel läßt sich (durch eine Polynomialzeitreduktion) in eine äquivalente einfache Formel verwandeln.*

Beweis: Sei k die maximale Zahl der $\forall$-Quantoren, die zwischen der Quantifizierung und dem Auftreten einer Variablen erscheinen. F hat also die Form

$$F = \ldots Qx \ldots \forall z_1 \ldots \forall z_2 \ldots \forall z_k \ldots F'(x)$$

F wird durch die folgende Formel $\tilde{F}$, die eine weitere neue Variable y enthält:

$$\tilde{F} = \ldots Qx \ldots \forall z_1 \exists y \Big((x \leftrightarrow y) \wedge \ldots \forall z_2 \ldots \forall z_k \ldots F'(y) \Big)$$

Für eine Variable muß dieses Verfahren maximal k mal durchgeführt werden. Dabei werden nur $\exists$-Quantoren eingefügt, die maximale Anzahl der $\forall$-Quantoren ändert sich dadurch nicht. Bei l Variablen terminiert dieses Verfahren folglich nach maximal $k \cdot l$ Iterationen und die resultierende Formel ist einfach. ∎

Ab jetzt können wir annehmen, daß F mit einem Quantor beginnt, alle Quantoren bis zum rechten Rand binden und F einfach ist. (Beginnt die ursprüngliche Formel mit einem Quantor, so gilt dies auch für die umgewandelte Formel.)

Lemma 8.2.13 *Sei $F = \forall x F'(x)$ (bzw. $F = \exists x F'(x)$) eine einfache Formel. Dann ist $a(F'(x))$ ein Polynom vom Grad $\leq 2n$.*

Beweis: Der Beweis erfolgt durch Induktion über den Formelaufbau. $F'(x)$ ist von der Form $F_1(x) \vee F_2(x)$, $\exists y\ F_1(x)$, $F_1(x) \wedge F_2(x)$ oder $\forall y\ F_1(x)$. Damit ergeben sich die folgenden Fälle:

(i) $a(F'(x)) = a(F_1(x)) + a(F_2(x))$
$$\Rightarrow \quad \mathrm{Grad}(a(F'(x))) \leq \max\{\mathrm{Grad}(a(F_1(x))), \mathrm{Grad}(a(F_2(x)))\}$$

(ii) $a(F'(x)) = \sum_{y=0,1} a(F_1(x))$
$$\Rightarrow \quad \mathrm{Grad}(a(F'(x))) \leq \max\{\mathrm{Grad}(a(F_1(x))|_{y=0}), \mathrm{Grad}(a(F_1 x))|_{y=1})\}$$

(iii) $a(F'(x)) = a(F_1(x)) \cdot a(F_2(x))$
$$\Rightarrow \quad \mathrm{Grad}(a(F'(x))) = \mathrm{Grad}(a(F_1(x))) + \mathrm{Grad}(a(F_2(x)))$$

(iv) $a(F'(x)) = \prod_{y=0,1} a(F_1(x))$
$$\Rightarrow \quad \mathrm{Grad}(a(F'(x))) = \mathrm{Grad}(a(F_1(x))|_{y=0}) + \mathrm{Grad}(a(F_1(x))|_{y=1})$$

Bezogen auf die Eingabelänge der Formel verdoppelt sich der Grad nur bei (iv).

Da die Formel einfach ist, erfolgt ein solcher Verdoppelungsschritt nur ein einziges Mal. Man beachte, daß auch bei (iii) die Grade addiert werden, aber dies korrespondiert genau zur Addition der Längen der Teilformeln. Die Behauptung ergibt sich aus diesen Überlegungen. ■

Nach der obigen Vorrunde, in der sich Bob davon überzeugt hat, daß es sich bei p um eine Primzahl $\geq 2^n$ handelt, wird das Protokoll in mehreren Beweisrunden gespielt. Die Zählung der Runden beginnt bei Eins.

Sei $F = \forall x F'(x)$ (bzw. $F = \exists x F'(x)$), $a(F) = \prod_{x=0,1} F'(x)$ (bzw. $a(F) = \sum_{x=0,1} F'(x)$) und $p(x) = a(F'(x))$ das Polynom in x.

Die erste Runde beginnt damit, daß Alice den Wert $a(F) \bmod p$ an Bob übermittelt. Bob verlangt daraufhin einen Beweis für die Korrektheit dieses Wertes. Diesen gibt Alice dadurch, daß sie Bob das Polynom $p(x)$ mitteilt, indem sie die Koeffizienten angibt.

Hat F die Gestalt $F = \forall x F'(x)$, dann überprüft Bob $a(F) \equiv p(0) \cdot p(1) \bmod p$. Ist $F = \exists x F'(x)$, so überprüft er $a(F) \equiv p(0) + p(1) \bmod p$.

Bob wählt jetzt zufällig eine Zahl $r \in \mathbb{Z}/p\mathbb{Z}$, teilt diese Zahl Alice mit und berechnet $p(r) \bmod p$, d.h., er wertet das Polynom $p(x)$ bei r aus. Außerdem setzt er r in den arithmetischen Ausdruck $a(F'(x))$ für x ein. Dieser Ausdruck ist jetzt von der Form

$$a(F'(r)) \quad = \quad b + c \cdot a(G)\,,$$

wobei G der Teilausdruck von F' ist, der mit dem ersten Quantor von F' beginnt. Ist $c = 0$ (falls kein weiterer Quantor mehr auftritt oder sich der Term c zu 0 auswertet), so überprüft Bob, ob $p(r) = b$ gilt und akzeptiert im positiven Fall. Ist $c \neq 0$, dann berechnet Bob $a' = (p(r) - b) \cdot c^{-1}$. Damit ist diese Runde beendet.

Jetzt beginnt die zweite Runde. Wenn Alice das korrekte Polynom $p(x)$ angegeben hatte, dann muß $a(G(r)) \equiv a' \mod p$ gelten. Bob verlangt jetzt von Alice den Beweis für die Richtigkeit dieses Wertes, der wiederum durch die Angabe eines Polynoms $q(y)$ erfolgt. Bob überprüft erneut $a(G) \equiv q(0) \cdot q(1) \mod p$ bzw. $a(G) \equiv q(0) + q(1) \mod p$, wählt eine weitere Zufallszahl s und berechnet $a(G'(s))$, was Alice in der dritten Runde beweisen muß usw.

Bei diesem Protokoll handelt es sich um ein IBS. Ist $F \in$ QBF, dann akzeptiert Bob mit Wahrscheinlichkeit 1. Ist $F \notin$ QBF und Alice versucht zu betrügen, dann muß der erste Wert $a(F) \mod p$ falsch sein (denn sonst wäre $a(F) \equiv 0 \mod p$ und offensichtlich $F \notin$ QBF). Da Bob das übermittelte Polynom $p(x)$ an $x = 0$ und $x = 1$ auswerten kann, ist auch dieses Polynom falsch (das korrekte Polynom sei $q(x)$). Dann ist $p(x) - q(x)$ ein Polynom vom Grad $\leq 2n$ und hat damit höchstens $2n$ Nullstellen, da p eine Primzahl ist[7]. Es gilt also $p(x) = q(x)$ an maximal $2n$ Stellen $x \in \{0, \ldots, p-1\}$. Damit ist die Wahrscheinlichkeit, daß $p(r) = q(r)$ gilt $\leq \frac{2n}{2^n}$, da $p > 2^n$ ist.

Alice wird also mit sehr hoher Wahrscheinlichkeit in die Behauptung eines falschen Wertes $a(G(r))$ geschickt und muß dann die nächste Runde mit einem falschen Polynom beginnen. Diese Situation wiederholt sich ständig. Je mehr Runden jedoch gespielt werden, desto besser für Alice: Wenn Bob einmal eine Zufallszahl r trifft, für die $p(r) = q(r)$ gilt, dann ist der von ihm berechnete Wert a' der korrekte arithmetische Wert für den verbliebenen Teilausdruck und Alice kann beruhigt fertig spielen — ohne erneute Täuschungsmanöver. Es werden jedoch maximal n Runden durchgeführt. Die Wahrscheinlichkeit für Alice, entdeckt zu werden, ist am Ende mindestens $\left(1 - \frac{2n}{2^n}\right)^n \geq 1 - n \cdot \frac{2n}{2^n}$, denn es gilt: $(1-x)^n \geq 1 - nx$ für $0 \leq x \leq 1$. Die Wahrscheinlichkeit, daß Bob betrogen werden kann, ist also kleiner als $\frac{2n^2}{2^n}$. Wird das Protokoll zweimal durchlaufen, dann sinkt diese Wahrscheinlichkeit auf $\left(\frac{2n^2}{2^n}\right)^2 = \frac{4n^4}{2^{2n}} < 2^{-n}$ ab einem genügend großen n. ■

Beispiel 8.2.14 Wir wollen das Protokoll zwischen Alice und Bob an einem Beispiel nachvollziehen, das dem Buch [Pap94] entnommen ist.

Wir gehen von folgender (einfachen) Formel aus:

$$F \;=\; \forall x\Big(\exists y\Big((x \vee y) \wedge \forall z\Big((x \wedge z) \vee (y \wedge \overline{z}) \vee \exists w\Big((z \vee (y \wedge \overline{w}))\Big)\Big)\Big)\Big)$$

[7]Wäre p keine Primzahl, dann könnte ein Polynom vom Grad n mehr als n Nullstellen haben, wenn modulo p gerechnet wird. Als Beispiel sei das Polynom $p(x) = x \cdot (x+1) \mod 6$ angegeben. Hier gilt: $p(0) \equiv 0 \mod 6$, $p(2) = 6 \equiv 0 \mod 6$, $p(3) = 12 \equiv 0 \mod 6$, $p(5) = 30 \equiv 0 \mod 6$. $p(x)$ ist vom Grad 2, hat aber 4 Nullstellen (bei der Rechnung modulo 6).

Der zugehörige arithmetische Ausdruck $a(F)$ ist

$$a(F) = \prod_{x=0,1} \Big(\sum_{y=0,1} \Big((x+y) \cdot \prod_{z=0,1} \Big(x \cdot z + y \cdot (1-z) + \sum_{w=0,1} \Big(z + y \cdot (1-w) \Big)\Big)\Big)\Big)$$

Der Ausdruck $a(F)$ ergäbe ausgewertet 96. Der Wert ist ungleich 0, d.h. die Formel F ist wahr.

Nun folgt die Beschreibung des eigentlichen Protokolls zwischen Alice (A) und Bob (B). (Dabei sei $a(F)$ ein arithmetischer Ausdruck, der mit $\prod$ bzw. $\sum$ beginnt, $a(F')$ derselbe Ausdruck, bei dem das erste $\prod$ bzw. $\sum$ gestrichen wurde, und $a(G)$ der Teilausdruck von $a(F)$, der mit dem zweiten $\prod$ oder $\sum$ beginnt.)

(Vorrunde) Alice übermittelt Bob zunächst die Primzahl p und den zugehörigen Primzahlbeweis. Sämtliche Rechnungen werden im folgenden modulo dieser Primzahl durchgeführt. In unserem Beispiel[8] sei diese Primzahl 13.

(A) Nun beginnt die *erste Runde*, in der Alice zunächst den Wert $a(F)$ mod p, also 96 mod 13 = 5, an Bob sendet. Bob erhält außerdem die Koeffizienten des Polynoms $p(x) = 2x^2 + 8x + 6$.

(B) Der Ausdruck $a(F)$ beginnt mit einem $\prod$, also überprüft Bob, ob $a(F) \equiv p(0) \cdot p(1) \bmod p$ gilt. In unserem Fall war $a(F) \equiv 5 \bmod 13$ und $p(0) \cdot p(1)$ ist $6 \cdot 16 \equiv 5 \bmod 13$. Der von Bob berechnete Wert stimmt also mit dem von Alice behaupteten überein.

Nun will Bob das führende $\prod$ streichen, um die Überprüfung mit der Teilformel $a(G)$ fortsetzen zu können. Diese enthält jetzt aber die freie Variable x. Bob wählt deswegen eine Zufallszahl r mod 13, z.B. $r = 9$, und setzt diese für x ein. Damit erhalten wir einen neuen Ausdruck (den wir im weiteren mit $a(G)$ bezeichnen):

$$a(G) = \sum_{y=0,1} \Big((9+y) \cdot \prod_{z=0,1} \Big(9 \cdot z + y \cdot (1-z) + \sum_{w=0,1} \Big(z + y \cdot (1-w) \Big)\Big)\Big)$$

Nun berechnet Bob $p(9) = 2 \cdot 9^2 + 8 \cdot 9 + 6 \equiv 6 \bmod 13$. Wenn Alice das korrekte Polynom übergeben hatte, dann muß dieser Wert mit dem Wert $a(G)$ übereinstimmen. Bob wird also für die nächste Runde den Nachweis verlangen, daß $a(G) \equiv p(9) \mod 13$ (also $a(G) \equiv 6 \mod 13$) gilt.

(A) Es beginnt die *zweite Runde*. Alice hat die Aufgabe, $a(G) \equiv 6 \bmod 13$ zu zeigen. Hierfür streicht sie das $\sum$ und erhält damit den Ausdruck $a(G'(y)) \equiv 2y^3 + y^2 + 3y \bmod 13$, dessen Koeffizienten wiederum an Bob übermittelt werden. Bob sieht also das Polynom $p'(y) = 2y^3 + y^2 + 3y$.

[8] Um das Protokoll noch einigermaßen nachvollziehbar zu gestalten, wird an dieser Stelle auf die Forderung $2^n < p < 2^{3n}$ verzichtet.

(B) Bob überprüft, daß $p'(0) + p'(1) \equiv 6 \bmod 13$ gilt. Er bildet jetzt den Ausdruck $a(G')$, indem er die freie Variable y durch eine weitere Zufallszahl, hier durch $r = 3$, ersetzt. Dieser Ausdruck $a(G')$ ist von der Form $12 \cdot a(H)$, wobei H die folgende Gestalt hat:

$$a(H) \quad = \quad \prod_{z=0,1} \Big(9 \cdot z + 3 \cdot (1-z) + \sum_{w=0,1} \big(z + 3 \cdot (1-w)\big)\Big)$$

In der obigen Sprechweise ist $a(G') = b + c \cdot a(H)$ mit $b = 0$ und $c = (9+r) = 12$. Bob berechnet also $p'(3) = 2 \cdot 3^3 + 3^2 + 3 \cdot 3 \equiv 7 \bmod 13$ und dann $a(H) = 7 \cdot (12^{-1}) \equiv 7 \cdot 12 \equiv 6 \bmod 13$. Die Bestimmung von $12^{-1} \equiv 12 \bmod 13$ bereitet Bob keinerlei Schwierigkeiten und ist beispielsweise mit dem Algorithmus von Euklid möglich. In der nächsten Runde wird er den Nachweis $a(H) \equiv 6 \bmod 13$ verlangen.

(A) Es beginnt die *dritte Runde*. Alice hat die Aufgabe, $a(H) \equiv 6 \bmod 13$ zu zeigen. Sie berechnet das Polynom $p''(z) = 8z + 6$ und liefert Bob die Koeffizienten.

(B) Bob prüft zunächst, daß $p''(0) \cdot p''(1) = 6 \cdot 14 \equiv 6 \cdot 1 \equiv 6 \bmod 13$ ist.

Erneut wählt er eine Zufallszahl (etwa $r = 7$). Er streicht das $\prod$ in $a(H)$ und erhält durch Einsetzen von r für z einen Ausdruck $9 \cdot 7 + 3 \cdot (1-7) + a(I) \equiv 6 + a(I) \bmod 13$ mit

$$a(I) \quad = \quad \sum_{w=0,1} \big(7 + 3 \cdot (1-w)\big).$$

Nach der Auswertung von $p''(7) \equiv 10 \bmod 13$ berechnet Bob also $a(I) \equiv 10 - 6 \equiv 4 \bmod 13$ und wird in der nächsten Runde den Nachweis $a(I) \equiv 4 \bmod 13$ verlangen.

(A) Wir kommen nun zur *letzten Runde*. Alice berechnet das Polynom $p'''(w) = -3w + 10$ und übermittelt es an Bob.

(B) Bob testet $p'''(0) + p'''(1) \equiv 4 \bmod 13$ und ist damit von der Richtigkeit überzeugt.

Bob wählt wieder eine Zufallszahl r und wertet das Polynom $p'''(r)$ aus. Er setzt r in den arithmetischen Ausdruck $-3w + 10 = -3r + 10$ ein und erhält eine Zahl modulo 13, da kein weiterer Quantor auftritt. In diesem Beispiel stimmt diese Zahl mit $p'''(r)$ überein, denn Alice hatte ja genau das Polynom $p'''(w) = -3w + 10$ übermittelt.

Bob ist zufrieden und akzeptiert die Formel als wahr, nachdem ein zweiter Durchgang die Betrugswahrscheinlichkeit wie gefordert vermindert hat.

Literaturverzeichnis

[BDG88] J. Balcázar, J. Díaz, J. Gabarró. *Structural Complexity I.* Springer-Verlag, Berlin-Heidelberg-New York, 1988.

[Bor72] A. Borodin. Computational complexity and the existence of complexity gaps. *Journal of the Association of Computing Machinery*, 19:158–174, 1972.

[Coo71] S. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd STOC*, Seite 151–157, 1971.

[Die95] V. Diekert. Komplexitätstheorie. In Günter Grosche, V. Ziegler, D. Ziegler, E. Zeidler (Herausgeber), *Teubner-Taschenbuch der Mathematik Teil II*, Kapitel 8.4. B. G. Teubner, Stuttgart-Leipzig, 7. Auflage, 1995.

[HS66] F.C. Hennie, R.E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the Association of Computing Machinery*, 13:533–546, 1966.

[HU79] J.E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, Mass., 1979.

[Imm88] N. Immerman. Nondeterministic space is closed under complement. *SIAM Journal of Computing*, 17:935–938, 1988.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, Reading, Mass., 1994.

[Pau78] W.J. Paul. *Komplexitätstheorie.* B. G. Teubner, Stuttgart, 1978.

[Rei90] K.R. Reischuk. *Einführung in die Komplexitätstheorie.* B. G. Teubner, Stuttgart, 1990.

[Sav70] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[Sch87] U. Schöning. Graph isomorphism is in the low hierarchy. In F. J. Brandenburg (Herausgeber), *Proceedings of the 4th STACS*, Nummer 247 der Lecture Notes in Computer Science, Seite 114–124, Berlin-Heidelberg-New York, 1987. Springer-Verlag.

[Sch95] U. Schöning. *Perlen der Theoretischen Informatik.* BI-Wissenschafts-Verlag, Mannheim-Leipzig-Wien-Zürich, 1995.

[Sip80] M. Sipser. Halting space-bounded computations. *Theoretical Computer Science*, 10:335–338, 1980.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Weg93] I. Wegener. *Theoretische Informatik.* B. G. Teubner, Stuttgart, 1993.

Index