Effiziente Algorithmen und Komplexitätstheorie

(Algebraische Interpolations- und Zählalgorithmen)

Marek Karpinski

Kai Werther

4. Version vom 28.04.98

Neue Auflage 1998.

Zusammenfassung

Dieses Skript ist aus den Vorlesungen *Effiziente Algorithmen und Komplexitätstheorie* im WS 89/90 und *Schnelle parallele und randomisierte Algorithmen* im SS 90 an der Universität Bonn hervorgegangen.

Inhaltsverzeichnis

Kapitel 1

Maschinenmodelle und grundlegende Algorithmen

Die Untersuchung von Algorithmen erfordert ein genau definiertes Maschinenmodell. Ein Maschinenmodell stellt Resourcen und einen Zugriff auf diese Resourcen zur Verfügung. Sie können sich in der Anzahl und der Art der elementaren Operationen unterscheiden. Die ersten Maschinenmodelle sind in den 30-er Jahren eingeführt worden, um den Begriff der Berechenbarkeit zu präzisieren. Die wichtigsten Ansätze waren Semi-Thue Systeme, Typ 0-Grammatiken und Turing Maschinen. Wir werden später nur auf die Turing Maschine eingehen, da sich diese drei Modelle als äquivalent erwiesen haben und die Turing Maschine dem intuitiven Berechenbarkeitsbegriff am nächsten kommt.

Wir können den Berechenbarkeitsbegriff auf die Erkennung von Teilmengen A von $\{0,1\}^*$ reduzieren, d.h. auf Funktionen $\{0,1\}^* \to \{0,1\}$. Beliebige Funktionen $f : I\!N \to I\!N$ können durch Funktionen $f' : \{0,1\}^* \to \{0,1\}^*$ und diese entweder durch Funktionenfamilien $g_i : \{0,1\}^* \to \{0,1\}$ oder durch Funktionen $g : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ ersetzt werden. Wir definieren eine Menge $A \subset \{0,1\}^*$ als berechenbar, falls es eine TM M gibt, die A erkennt, d.h. $\forall\, x \in A$ nach einer endlichen Anzahl von Berechnungsschritten hält und akzeptiert. Für $x \notin A$ wird die Endlichkeit der Berechnung nicht gefordert, jedoch darf die Berechnung in diesem Fall nicht mit einem akzeptierenden Halten enden. Eine solche Menge A heißt auch die von M akzeptierte Sprache $L(M)$.

Durch Einschränkung der Resourcen, bei der Turing Maschine z.B. Zeit oder Band, entsteht eine Struktur auf der Menge der erkennbaren Sprachen. Die Analyse dieser Struktur ist Gegenstand der klassischen theoretischen Informatik und soll hier nicht behandelt werden. Ein Standardwerk hierzu ist das Buch von Hopcroft und Ullman ([HU 79]).

Wir wollen auf neuere Berechnungsmodelle eingehen und neue Resourcen eröffnen. Das Turing Maschinenmodell ermöglicht nur sequentiellen Zugriff auf die gespeicherten Daten. Moderne Rechenanlagen können jedoch auf ihren Speicher beliebig zugreifen (random access). Diese Rechnerarchitektur führte zur Entwicklung eines theoretischen Random Access Maschinenmodells (RAM) mit einer beliebigen Anzahl von beliebig großen Speicherzellen. Dieses Rechnermodell ist nicht so träge wie die Turing Maschine, ermöglicht jedoch — wie wir später sehen werden — nur eine polynomielle Beschleunigung von Algorithmen. Eine direkte Folgerung aus dieser Tatsache ist, daß die Menge der in polynomieller Zeit erkennbaren Sprachen dieselbe ist. Anders gesehen ist diese Menge, aus historischen Gründen $\mathcal{P}$ genannt, sehr robust, da sie invariant gegenüber dem Maschinenmodell ist.

Der Engpaß moderner Rechenanlagen ist der Mangel an Recheneinheiten. Während in den 50-er und 60-er Jahren aus Kostengründen die Computer nur eine Recheneinheit besaßen, ist man heute bemüht, viele CPU's in einem Rechner unterzubringen, die dann parallel an der Lösung eines Problems arbeiten. Auch diese technologische Entwicklung inspirierte die theoretische Informatik und als Folge wurde das Rechnermodell der PRAM (parallele RAM) entwickelt. Es besteht aus vielen RAM's, die über einen gemeinsamen Speicher miteinander kommunizieren können. Die Untersuchung schneller paralleler Algorithmen und der Vergleich mit sequentiellen Algorithmen ist zentraler Gegenstand der aktuellen Forschung.

Der Fortschritt im Design und der Herstellung hochintegrierter Schaltkreise (VLSI) hat sich in dem Berechnungsmodell der Schaltkreise niedergeschlagen. Wir werden später sehen, daß dieses Berechnungsmodell von den anderen Modellen abweicht. Um überhaupt einen Vergleich zu den anderen Modellen zu schaffen, ist es nötig Schaltkreise mit gewissen einschränkenden Eigenschaften zu betrachten. Diese Einschränkung ist die Forderung, daß ein Schaltplan für den Schaltkreis schnell erzeugbar sein soll. Eine detaillierte Beschreibung dieser Bedingung wird in Abschnitt 1.4 gegeben.

Unser Ziel ist die Entwicklung effizienter Algorithmen, d.h. Algorithmen, die zu ihrer Durchführung nur wenig Resourcen verbrauchen. Insbesondere sind wir an der Reduzierung der benötigten Zeit interessiert. Ein klassisches Konzept ist hier die Einführung von Nichtdeterminismus. Nichtdeterminismus bedeutet, daß Aktionen der Maschinen nicht eindeutig durch ihren Zustand festgelegt sind, sondern die Maschinen eine endliche Anzahl von Wahlmöglichkeiten besitzt. Dieser Entscheidungsprozeß ist nicht bestimmt (nicht deterministisch). Eine Eingabe wird akzeptiert, wenn es eine Folge von Wahlen gibt, so daß die Maschine in einem Endzustand landet. Es ist nicht bekannt ob die Hinzufügung von Nichtdeterminismus die Menge der erkennbaren Sprachen — bei sonst gleichen Einschränkungen der Resourcen — vergrößert. In einigen Fällen ist diese Frage zu bejahen, in anderen zu verneinen, wieder andere Fragen sind noch offen. Die bekannteste offene Frage hierzu ist, ob $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ gilt. Es wird jedoch die Ungleichheit dieser beiden Komplexitätsklassen vermutet. Die beste bekannte Simulation des Nichtdeterminismus ist die vollständige Durchsuchung des Berechnungsbaumes. Dies resultiert in einer exponentiellen Laufzeit.

Nichtdeterministische Berechnungsmodelle existieren nur in der theoretischen Informatik und es ist nicht bekannt, ob man je einen realistischen Computer bauen kann, der nichtdeterministisch arbeitet. Ein realitätsnäherer Ansatz ist die Verwendung von Randomisierung anstatt von Nichtdeterminismus. Dabei wählt die Maschine die nächste Aktion in Abhängigkeit ihres Zustandes und eines Zufallsexperiments. Dieses Zufallsexperiment kann durch Zufallszahlengeneratoren erzeugt werden. Dies sind entweder Pseudozufallszahlengeneratoren, die eine scheinbar zufällige Folge von Zahlen erzeugen, oder Zufallszahlengeneratoren, die eine natürliche Zufallsquelle, wie z.B. thermisches Rauschen, verwenden. Der Bau von randomisierten Maschinen ist daher wesentlich realistischer als der Bau nichtdeterministischer Maschinen. Die Untersuchung der Beziehung zwischen randomisierten Maschinen und deterministischen Maschinen steckt noch in den Kinderschuhen. Da jedoch auch hier die beste bekannte Simulation die vollständige Durchsuchung des Berechnungsbaumes ist, erhofft man, durch Verwendung von Randomisierung schnellere Algorithmen zu bekommen und eine größere Anzahl von Problemen effizient lösen zu können.

Notationen

Zum vereinfachten Umgang mit dem asymptotischen Verhalten von Funktionen führen wir Notationen ein, die auch Landausche Symbole genannt werden. Solche Funktionen sind für unsere Zwecke meistens Funktionen, die die **Rechenzeit**, den **Speicherbedarf**, den **Prozessorenverbrauch** und den Verbrauch von **Zufallsinformation** widerspiegeln, also den Verbrauch von Resourcen, die zur Lösung eines Problems notwendig sind.

Definition 1.0.1 Sei f eine Funktion $f : I\!N \to I\!N$. Dann definieren wir die folgenden Klassen von Funktionen.

$$O(f) := \{g : I\!N \to I\!N | (\exists c > 0)(\exists n_0 \in I\!N)(\forall n > n_0) \quad g(n) \leq cf(n)\}.$$

Für Funktionen $g \in O(f)$ gibt f also bis auf einen konstanten Faktor eine obere Schranke für g an, so z.B für die Laufzeit und den Speicherverbrauch eines Algorithmus.

$$\Omega(f) := \{g : I\!N \to I\!N | (\exists c > 0) \quad g(n) \geq cf(n) \text{ für unendlich viele } n\}.$$

Für Funktionen $g \in \Omega(f)$ gibt f eine untere Schranke für g an. Die Struktur der Definition unterscheidet sich jedoch von der von $O(f)$. Dadurch ist es möglich, enge untere Schranken anzugeben. Das Beispiel 1.0.2 wird dies verdeutlichen. Außerdem sei

$$\Theta(f) := \Omega(f) \cap O(f).$$

Beispiel 1.0.2 Sei f die Laufzeit des Algorithmus der Primzahlen testet, indem die Eingabe n durch den Teilbarkeitstest mit jeder Zahl $\leq \sqrt{n}$ auf Primalität getestet wird. Damit gilt für die Laufzeit f:

$$f(n) = \begin{cases} c & \text{für } n \text{ gerade} \\ c' \cdot \sqrt{n} & \text{für } n \text{ prim.} \end{cases}$$

Würde Ω analog zu O definiert sein, so wäre $f \in \Omega(1)$, was jedoch nicht dem wahren Verhalten der Laufzeit entspricht. Durch unsere Definition gilt $f \in \Omega(\sqrt{n})$ und $f \in O(\sqrt{n})$. Dies entspricht dem wahren Laufzeitverhalten des Algorithmus.

Mit Hilfe dieser Definition haben wir Klassen von Funktionen charakterisiert, deren Wachstum durch f majorisiert bzw. minorisiert wird, bzw. deren Wachstum gleich dem Wachstum von f ist. Wir führen folgende Konvention ein. Wir schreiben

$$g = O(f) \text{ statt } g \in O(f) \text{ und}$$

$$g = \Omega(f) \text{ statt } g \in \Omega(f).$$

Zur Erläuterung sei folgendes Beispiel gedacht:

Beispiel 1.0.3 Sei

$$P(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_m n^m$$

mit positiven Koeffizienten. Dann gilt

$$P(n) = O(n^m).$$

BEWEIS: Der Beweis ist einfach :

$$\begin{aligned} P(n) &= (a_0 n^{-m} + \cdots + a_m) n^m \\ &\leq (\textstyle\sum_{i=0}^m a_i) n^m \\ &= cn^m \quad \text{für } c = \textstyle\sum a_i \text{ und alle } n \geq n_0 = 1. \end{aligned}$$

□

Der Vollständigkeit halber geben wir auch folgende

Definition 1.0.4 Sei f wie oben. Dann definieren wir

$$o(f) := \{g | \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\}$$

$$\omega(f) := \{g | \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}.$$

Damit gilt offensichtlich: $f \in o(g) \Leftrightarrow g \in \omega(f)$.

Wir wollen nun auch mit unseren Klassen von Funktionen rechnen.

Satz 1.0.5 Es gelten die folgenden Gleichungen.

$$\begin{gathered} O(f) = O(O(f)) \\ o(O(f)) = O(o(f)) = o(f) \\ O(f) \cdot O(g) = O(f \cdot g) \\ O(f) + O(g) = O(\max\{f, g\}), \end{gathered}$$

wobei die Summe und das Produkt zwischen den Klassen kanonisch definiert ist.

1.1 Turing Maschine

Zunächst wollen wir uns dem Berechnungsmodell der Turing Maschine (TM) zuwenden. Operationell besteht eine TM aus einer Zentraleinheit und einer endlichen Anzahl von Bändern. Auf diese Bänder kann die TM mittels Schreib/-Leseköpfe zugreifen. In der Zentraleinheit befindet sich ein endliches Programm, das die Bewegung der Köpfe und die Schreiboperationen in Abhängigkeit von den Bandinhalten an den aktuellen Bandpositionen und dem inneren Zustand der endlichen Kontrolle bewirkt. Je nach Anzahl und Art der Bänder, der erlaubten Bewegungsrichtungen der Köpfe und der Fähigkeit zu Schreib- bzw. Leseoperationen ergeben sich eine Vielzahl verschiedener TM Modelle. Wir betrachten hier nur einfache Modelle, die wir wie folgt formal definieren wollen.

Definition 1.1.1 Eine (k-Band) TM M ist ein 5-Tupel $M =< Q, \delta, q_0, F, \Gamma >$ wobei

- Q die endliche Menge der Zustände der endlichen Kontrolle,
- $q_0 \in Q$ der Startzustand,
- $F \subset Q$ die Menge der akzeptierenden Zustände,

- Γ das Bandalphabet, d.h. jede Zelle der Bänder beinhaltet ein Element aus Γ ($|\Gamma| \geq 2$), und
- δ die Zustandsüberführungsfunktion (das Programm)

$$\delta : (Q, \Gamma^k) \to (Q, \Gamma^k, \{L, N, R\}^k)$$

  sind.

M akzeptiert $x \in \Gamma^*$, falls M hält, d.h. nach einer endlichen Anzahl von Zustandsübergängen in einem Endzustand $q' \in F$ übergeht. Die Menge aller akzeptierten Wörter bildet die von M erkannte Sprache $L(M)$.

$$L(M) := \{x \in \Gamma^* \mid M \text{ akzeptiert } x\}.$$

Nun müssen wir den Verbrauch der Resourcen näher betrachten. Die Zeit für die Berechnung an $x \in L(M)$ sei mit $T(x)$ bezeichnet und ist die Zahl der Zustandsübergänge. Die Anzahl der benutzten Bandzellen ist der Speicherverbrauch und werde mit $S(x)$ bezeichnet. Dann

Definition 1.1.2

$$T(n) := \max_{|x|=n, x \in L(M)} T(x).$$
$$S(n) := \max_{|x|=n, x \in L(M)} S(x).$$

Falls die Berechnung der TM M immer von der ganzen Eingabe abhängt, so gilt $T(n) \geq n$ und $S(n) \geq n$, obwohl zur eigentlichen Berechnung weniger als n Speicher benötigt wird. Daher betrachten wir im weiteren ein spezielles Turing Maschinenmodell, das die Eingabe, die Ausgabe und den Arbeitsspeicher voneinander trennt.

Definition 1.1.3 Offline TM
Eine Offline TM ist eine Turing Maschine mit einem Eingabe-, einem Ausgabe- und einem Arbeitsband. Von dem Eingabeband darf nur gelesen werden und der Lesekopf darf in beide Richtungen bewegt werden. Das Ausgabeband ist eine reines Schreibband, hier darf der Schreibkopf jedoch nur nach rechts bewegt werden, d.h. eine einmal gemachte Ausgabe kann <u>nicht</u> überschrieben und damit korrigiert werden. Das Arbeitsband ist ein normales Band, auf ihm sind sowohl Schreib- als auch Lesezugriffe erlaubt, der Schreib/Lesekopf kann in beide Richtungen bewegt werden.

Abbildung 1.1 zeigt ein Diagramm einer Offline TM.

Definition 1.1.4 Online TM
Eine Online TM ist eine Offline TM, bei der auch der Eingabekopf nur nach rechts bewegt werden darf, d.h. das Modell der Online TM ist eine Einschränkung des Modells der Offline TM.

Für Offline TM's definieren wir den Speicherverbrauch $S(x)$ als die Anzahl der benutzten Zellen auf den Arbeitsbändern. Durch diese Definition ist es möglich TM's mit sublinearen Speicherverbrauch zu betrachten.

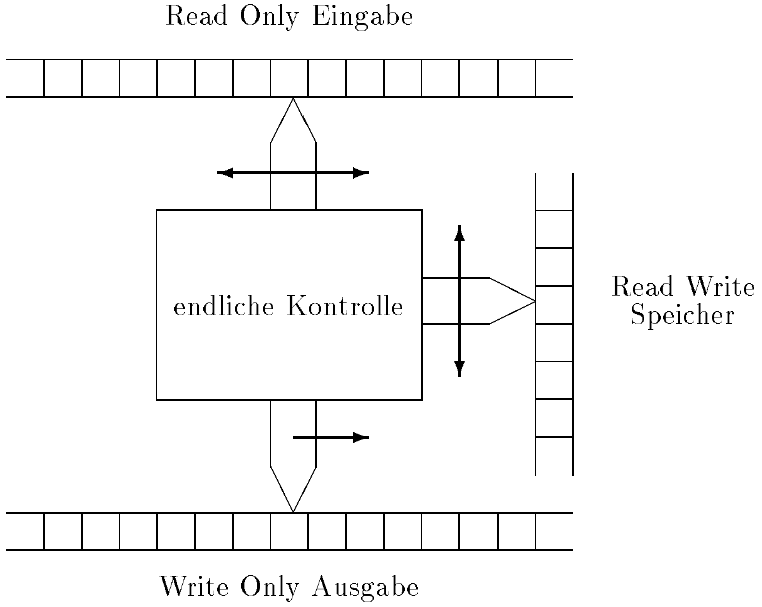

Abbildung 1.1: Offline Turing Maschine

Beispiel 1.1.5 Auf dem Eingabeband unserer TM M stehen getrennt durch ein spezielles Zeichen zwei unär dargestellte Zahlen. Wir wollen das Produkt dieser beiden Zahlen berechnen und unär dargestellt ausgeben. Dies geschieht folgendermaßen: Wir lesen die erste der beiden Zahlen von dem Eingabeband, indem wir die Anzahl der Striche binär auf dem Arbeitsband zählen. Nun kopieren wir die zweite Zahl auf das Ausgabeband und zählen danach die binäre Zahl um eins herunter. Dies wiederholen wir sooft, bis die Zahl auf dem Speicherband 0 ist. Auf dem Ausgabeband steht nun das Ergebnis. Der verbrauchte Speicher auf dem Arbeitsband ist nur logarithmisch (und somit sublinear) in der Eingabelänge unseres Algorithmus.

Im folgenden werden wir nur Offline TM's betrachten und diese einfach mit dem Begriff TM bezeichnen.

Wir können nun die Menge der berechenbaren Sprachen strukturieren, indem wir sogenannte Komplexitätsklassen einführen. Dies sind Mengen von Sprachen, die von TM mit beschränkten Resourcen erkannt werden können.

Definition 1.1.6 Beschränken wir den Zeitverbrauch zur Erkennung der Sprache A durch eine Funktion f, so erhalten wir die Sprachenklasse TIME(f).

$$\mathrm{TIME}(f) := \{A \subseteq \{0,1\}^* | \exists TMM \text{ mit } L(M) = A \text{ und } T(n) = O(f(n))\}$$

In gleicher Weise können wir auch den Speicherverbrauch beschränken.

Definition 1.1.7

$$\mathrm{SPACE}(f) := \{A \subseteq \{0,1\}^* | \exists TMM \text{ mit } L(M) = A \text{ und } S(n) = O(f(n))\}$$

Bezüglich dieser Komplexitätsklassen sind die folgenden Relationen bekannt. Zum Beweis verweisen wir auf [HU 79].

Definition 1.1.8 $s(n)$ heißt band-konstruierbar, falls es eine TM M gibt, die für $|x| = n$ höchstens $s(n)$ Bandspeicher benötigt und es mindestens ein x mit $|x| = n$ gibt, dessen Berechnung genau $s(n)$ Speicher benötigt. $s(n)$ heißt voll band-konstruierbar, falls jedes x genau $s(|x|)$ Speicher verbraucht. $t(n)$ heißt zeit-konstruierbar, falls die analoge Bedingung für den Zeitverbrauch erfüllt ist (analog auch der Begriff voll zeit-konstruierbar).

Theorem 1.1.9 Es gelten die folgenden Relationen für $f \leq g$

$$\mathrm{TIME}(f) \subseteq \mathrm{TIME}(g).$$

$$\mathrm{SPACE}(f) \subseteq \mathrm{SPACE}(g).$$

$$\mathrm{SPACE}(f) \subset \mathrm{TIME}(c^f), \quad \text{für ein } c \in I\!N \text{ falls } f(n) \geq \log n.$$

Seien s_1 und s_2 voll band-konstruierbar mit $\lim_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$. Dann gilt

$$\mathrm{SPACE}(s_1) \subsetneq \mathrm{SPACE}(s_2).$$

Sei t_2 voll zeit-konstruierbar und für t_1 gelte $\lim_{n\to\infty} \frac{t_1(n)\log t_1(n)}{t_2(n)} = 0$. Dann gilt

$$\mathrm{TIME}(t_1) \subsetneq \mathrm{TIME}(t_2).$$

Nun können wir die Menge der in polynomieller Zeit erkennbaren Sprachen definieren.

Definition 1.1.10 Die Klasse $\mathcal{P}$ aller polynomiell Zeit-berechenbaren Sprachen ist

$$\mathcal{P} := \bigcup_{f \in n^{O(1)}} \mathrm{DTIME}(f).$$

1.2 RAM - Maschinen

Nun wollen wir die grundlegenden traditionellen Rechnermodelle verlassen und uns den modernen Berechnungsmodellen zuwenden. Das grundlegende Modell ist das der Random Access Machine, das in [SS 63] eingeführt wurde. Es ist stark an das J.v.Neumann'sche Rechnermodell angelehnt.

Das Modell von v. Neumann besteht aus einer Rechen- und Steuereinheit, einem Hauptspeicher, der sowohl die Instruktionen als auch die Daten beinhaltet, zusätzlich Ein- und Ausgabeeinheiten und einem Hintergrundspeicher. Die Steuereinheit kann wahlfrei auf die Speicherzellen im Hauptspeicher zugreifen, diese Daten manipulieren und wieder speichern. Dabei wird die Zeit für eine elementare Operation durch die Zeit, die für die Kommunikation zwischen Speicher und Steuereinheit notwendig ist, majorisiert. Das v. Neumann'sche Rechnermodell ist in fast allen gängigen sequentiell arbeitenden Computern verwirklicht.

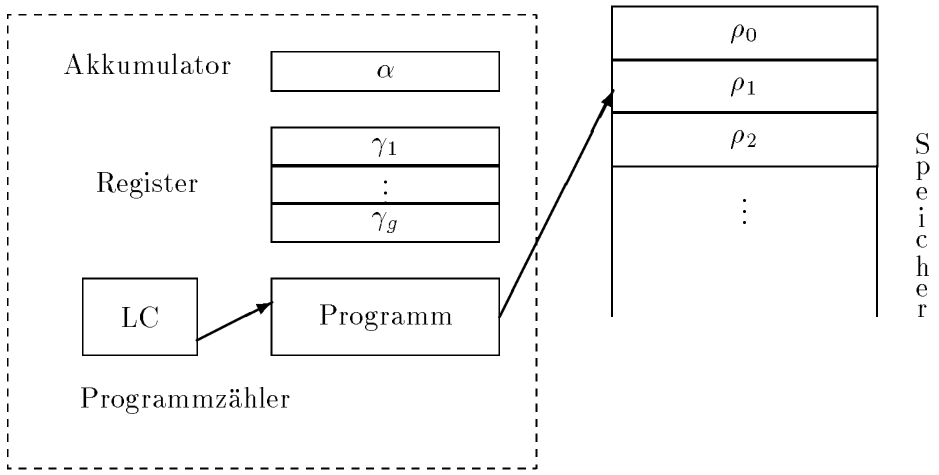

Abbildung 1.2: Random Access Maschine

Definition 1.2.1 Random Access Maschine
Eine Random Access Maschine, kurz auch RAM genannt, ist ein theoretisches Rechnermodell. Eine RAM besteht aus einem Akkumulator α und einer beschränkten Anzahl von Indexregistern $\gamma_1, \gamma_2, \ldots, \gamma_g$, einer Menge von Speicherzellen $\rho_i,\ i \in \mathbb{N}$, auf die wahlfrei zugegriffen werden kann, und einem Programmzähler LC. Der Inhalt der Speicherzellen, der Register und des Programmzählers sind beliebig große ganze Zahlen. Somit kann der Speicher auch als eine Funktion $\rho : \mathbb{N} \to \mathbb{Z}$ aufgefaßt werden. Die Abbildung 1.2 zeigt ein Schaubild der RAM.

Eine Random Access Maschine kann als eine Art 'springende' TM angesehen werden, da der wahlweise Zugriff auf den Speicher einem Sprung des Lese/Schreibkopfes an die entspechende Bandzelle entspricht (anstatt einer Folge von Kopfbewegungen).

Nun wollen wir eine genaue Beschreibung des Maschinenmodells RAM geben. Um die syntaktische Definition zu geben, benutzen wir die Backus Naur Form. Auf eine genaue semantische Definition verzichten wir, da sie für die RAM offensichtlich ist und auch nicht in den Rahmen dieser Vorlesung gehört. Genaueres kann im Skript Informatik I,II (WS 86/87, SS 87) nachgelesen werden.

Zunächst definieren wir die syntaktischen Klassen:

Definition 1.2.2 Syntaktische Kategorien der RAM

- $< \underline{\text{REG}} > ::= \alpha \mid \gamma_j \quad 1 \leq j \leq g$
- $< \underline{\text{LOP}} > ::= \rho_i \mid < \underline{\text{REG}} >$
- $< \underline{\text{OP}} > ::= i \mid \rho_i \mid < \underline{\text{REG}} >$
- $< \underline{\text{MOP}} > ::= \rho_{i+\gamma_j}$ mit $i \in \mathbb{N}$ und $1 \leq j \leq g$
- $< \underline{\text{G}} > ::= \gamma_j$ mit $1 \leq j \leq g$

- $< \underline{\text{NUM}} > ::= \mathbb{N}$

Dabei bedeutet ρ_i den Inhalt der Speicherzelle mit der Nummer i. Wie oben gesagt kann ρ auch als Funktion $\rho : \mathbb{N} \to \mathbb{Z} \supset \mathbb{N}$ interpretiert werden. Eine Adressenrechnung wird durch die Verwendung der indirekten Adressierung möglich. Sie erfolgt über die Benutzung modifizierter Operanden $< \underline{\text{MOP}} >$.

Wir können die Befehle der RAM in vier verschiedene Kategorien einteilen:

- Transportbefehle,
- Sprungbefehle,
- arithmetische Befehle und
- Indexbefehle.

Die Transportbefehle dienen zum Laden und Speichern des Akkumulatorinhaltes und der Registerinhalte von bzw. in die Speicherzellen. Für diese Aufgabe stehen 4 Befehle zur Verfügung.

Definition 1.2.3 Transportbefehle

1. $< \underline{\text{TB}} > ::= < \underline{\text{REG}} > \leftarrow < \underline{\text{OP}} >$
2. $< \underline{\text{TB}} > ::= \alpha \leftarrow < \underline{\text{MOP}} >$
3. $< \underline{\text{TB}} > ::= < \underline{\text{LOP}} > \leftarrow < \underline{\text{REG}} >$
4. $< \underline{\text{TB}} > ::= < \underline{\text{MOP}} > \leftarrow \alpha$

Die Sprungbefehle ermöglichen die Steuerung des Programmflußes.

Definition 1.2.4 Sprungbefehle

1. $< \underline{\text{SB}} > ::= \underline{\text{GOTO}} < \underline{\text{NUM}} >$
2. $< \underline{\text{SB}} > ::= \underline{\text{IF}} < \underline{\text{REG}} > < \underline{\text{VO}} > 0\ \underline{\text{THEN}}\ \underline{\text{GOTO}} < \underline{\text{NUM}} >$

wobei $< \underline{\text{VO}} >$ die Menge der Vergleichsoperatoren darstellt: $< \underline{\text{VO}} > ::= \; = \;|\neq\;|\leq\;|<\;|>\;|\geq$.

Die arithmetischen Befehle ermöglichen der RAM das Ausführen elementarer Rechenoperationen.

Definition 1.2.5 Arithmetische Befehle

1. $< \underline{\text{AB}} > ::= \alpha \leftarrow \alpha < \underline{\text{AO}} > < \underline{\text{OP}} >$

2. $< \underline{AB} > ::= \alpha \leftarrow \alpha < \underline{AO} > < \underline{MOP} >$

mit den elementaren arithmetische Operationen $< \underline{AO} > ::= \; | + \; | - \; | * \; | \underline{\text{DIV}} \; | \underline{\text{MOD}}$.

Durch Simulation von anderen Vergleichs- und arithmetischen Operationen können die Klassen $< \underline{AO} >$ und $< \underline{VO} >$ vergrößert werden. Die Kosten für diese neuen Operationen entsprechen dabei den Kosten für die Simulation.

Definition 1.2.6 Indexbefehle

1. $< \underline{IB} > ::= < \underline{G} > \leftarrow < \underline{G} > + i \quad i$ fest
2. $< \underline{IB} > ::= < \underline{G} > \leftarrow < \underline{G} > - i \quad i$ fest

Damit sind alle Klassen von Befehlen definiert. Ein RAM - Programm ist nun eine (numerierte) Folge von RAM - Befehlen. Genauer:

Definition 1.2.7 RAM - Programm
Sei $< \underline{B} > ::= < \underline{TB} > \; | < \underline{SB} > \; | < \underline{AB} > \; | < \underline{IB} >$ die Menge aller RAM - Befehle. Dann ist die Kategorie der RAM - Programme als Menge von numerierten RAM - Befehlen wie folgt definiert:
$< \underline{RP} > ::= \{ < \underline{NUM} > \;\; < \underline{B} > \}$

Als Konvention nehmen wir an, daß die Befehle eines RAM-Programms fortlaufend von 0 an numeriert sind. Zum Start eines RAM-Programmes nehmen wir zusätzlich an, daß alle Speicherzellen mit 0 initialisiert sind und nur die Speicherzelle ρ_0 die Eingabe enthält. Nach Beendigung des RAM-Programms wird die Ausgabe in der Speicherzelle ρ_1 zur Verfügung gestellt.

Unsere Definition der RAM läßt folgende Vergleiche zu Turing Maschinen und v.Neumann Rechnern zu:

- Sowohl bei der RAM als auch bei dem v.Neumann Rechner ist wahlfreier Zugriff auf den Speicherraum möglich.
- Bei RAM und Turing Maschine ist das Programm fest vorgegeben und getrennt von den zu verarbeitenden Daten; beim v.Neumann Rechner ist dies nicht so.
- Die RAM kann im Gegensatz zur Turing Maschine und v.Neumann Rechner in einer Speicherzelle beliebig große Zahlen (und damit beliebig viel Information) speichern.
- Der Speicherraum von RAM und Turing Maschine ist abzählbar unendlich, der Speicherraum des v. Neumann Rechners ist endlich.

Da eine RAM in jeder Speicherzelle beliebig große Zahlen speichern kann, und damit in einer Operation auch Daten beliebiger Größe verarbeiten kann, benötigen wir ein geeignetes Komplexitätsmaß, welches die Größe der Operanden berücksichtigt. Diese Notwendigkeit tritt bei den Turing Maschinen nicht auf, da in einer Zeiteinheit nur eine Bandzelle pro Band bearbeitet werden kann, deren Größe nur von der (festen) Kardinalität des Bandalphabetes abhängt. Daher definieren wir nun für die RAM zwei Kostenmaße.

Definition 1.2.8 Kostenmaße für die RAM
Für die RAM können wir in natürlicher Weise zwei Kostenmaße definieren:

- Einheitskostenmaß, d.h. jeder Speicherzugriff und jeder Befehl kostet eine Zeiteinheit
- Logarithmisches Kostenmaß (auch Bit-Kostenmaß genannt), d.h. die Kosten jedes Speicherzugriffs und jedes Befehls sind proportional zur Länge der Operanden (in Binärdarstellung).

Wir wollen diese Definition noch ein wenig konkretisieren:

Definition 1.2.9 Kosten für die Bereitstellung von Operanden
Mit $L(n)$ sei die Länge der Dualdarstellung von n gemeint,

$$L(n) := \begin{cases} 1 & \text{falls } n = 0 \\ \lfloor \log n \rfloor + 1 & \text{sonst.} \end{cases}$$

Die folgenden Tabellen zeigen die Kosten für die Bereitstellung von Operanden,

Operand	Einheitskostenmaß	Log. Kostenmaß
i	0	0
REG	0	0
ρ_i	1	$L(i)$
$\rho_{i+\gamma_j}$	1	$L(i) + L(\gamma_j)$

und für die Kosten der Befehle.

Befehlsart	Einheitskostenmaß	Log. Kostenmaß
Transportbefehl	1	$1 + L(m)$
Sprungbefehl	1	$1 + L(k)$
Arithm. Befehl	1	$1 + L(m_1) + L(m_2)$
Indexbefehl	1	$1 + L(i) + L(\gamma_j)$

wobei m, m_1 und m_2 Operanden und k der Sprunglabel sind.

Mit Hilfe dieser verschiedenen Kostenmaße können wir einem RAM-Programm auch verschiedene Komplexitäten zu ordnen.

Definition 1.2.10 Einheitskomplexität eines RAM-Programms
Gegeben sei ein RAM-Programm P. Dann ist die Einheitszeitkomplexität von P

$$\tilde{T}_P(n) := \max_{|x|=n} \{\text{Anzahl der ausgeführten Operationen bei Eingabe } x\}$$

und die Einheitsspeicherkomplexität von P

$$\tilde{S}_P(n) := \max_{|x|=n} \{ \text{ maximale Anzahl der benutzten Register und Zellen während der Berechnung bei Eingabe } x\}.$$

Definition 1.2.11 Log.-Komplexität eines RAM-Programms
Gegeben sei ein RAM-Programm P. Dann ist die log.-Zeitkomplexität von P

$$T_P(n) := \max_{|x|=n}\{ \text{ Summe der log.-Kosten der Operationen während der Berechnung bei Eingabe } x\}$$

und die log.-Speicherkomplexität von P

$$S_P(n) := \max_{|x|=n}\{\text{max. Summe der Längen der Binärdarst. der Zelleninhalte bei Eingabe } x\}.$$

Zur Erläuterung dieser Definition sei ein kleines Beispiel gedacht.

Beispiel 1.2.12 Das folgende RAM-Programm P berechnet 2^n.

0 $\alpha \leftarrow 1$
1 $\gamma_1 \leftarrow \rho_0$
2 $\underline{\text{IF}}\ \gamma_1 = 0\ \underline{\text{THEN}}\ \underline{\text{GOTO}}\ 6$
3 $\alpha \leftarrow \alpha * 2$
4 $\gamma_1 \leftarrow \gamma_1 - 1$
5 $\underline{\text{GOTO}}\ 2$
6 $\rho_1 \leftarrow \alpha$

Eine kleine Analyse zeigt, daß dieses Programm eine Einheitszeitkomplexität von $\tilde{T}_P(n) = O(n)$ jedoch eine log. Zeitkomplexität von $T_P(n) = O(n^2)$ besitzt.

Bemerkung 1.2.13 Die Verwendung der Technik *Square and Multiply* führt zu einem Algorithmus zur Berechnung von 2^n, der eine Einheitszeitkomplexität von $\tilde{T}_P(n) = O(\log n)$ und eine log. Zeitkomplexität von $T_P(n) = O(n \log n)$ besitzt.

Ähnlich zu den Turingmaschinen kann auch für ein RAM-Programm P die von P akzeptierte Sprache definiert werden.

Definition 1.2.14 Gegeben sei ein RAM-Programm P, so daß $f_P : \Sigma^* \to \{0,1\}$ die totale Funktion ist, die durch P berechnet wird. Dann ist die von P akzeptierte Sprache

$$L(P) := \{v \in \Sigma^* \mid f_P(v) = 1\}.$$

Nun führen wir Sprachklassen ein, die durch RAM-Programme definiert werden.

Definition 1.2.15 Sprachklassen für RAM
Sei $T : I\!N \to I\!N$. Dann gilt:

$$\text{RAM-TIME}(T(n)) := \{A \mid \exists\, P \text{ mit } T_P(n) = O(T(n)) \text{ und } L(P) = A\}.$$

Sei $S : I\!N \rightarrow I\!N$. Dann gilt:

$$\text{RAM-SPACE}(S(n)) := \{A | \exists\, P \text{ mit } S_P(n) = O(S(n)) \text{ und } L(P) = A\}.$$

Desweiteren ist die folgende Sprachklasse der simultan beschränkt berechenbaren Funktionen definiert:

$$\begin{aligned}\text{RAM-TIME-SPACE}(T(n), S(n)) := \;& \{A | \exists\, P \text{ mit } T_P(n) = O(T(n)), \\ & S_P(n) = O(S(n)) \text{ und } L(P) = A\}\end{aligned}$$

Eine besonders wichtige Klasse der simultan beschränkbaren Sprachen ist die 'Steve's Class', benannt nach Steven Cook.

Definition 1.2.16

$$\text{SC}^k := \text{RAM-TIME-SPACE}(\log^k(n), n^{O(1)})$$
$$\text{SC} := \bigcup_{k \in I\!N} \text{SC}^k$$

Später werden wir analog definierte Klassen von Sprachen kennenlernen, deren Studium einer der Schwerpunkte dieser Vorlesung sein wird.

Nun wollen wir die verschiedenen Berechnungsmodelle miteinander vergleichen. Hierzu hilft die folgende

Definition 1.2.17 Seien $T_1, T_2 : I\!N \rightarrow I\!N$ zwei Funktionen. Sie heißen polynomiell verknüpft, falls es ein Polynom P gibt, so daß

$$T_1(n) = O(P(T_2(n))) \text{ und } T_2(n) = O(P(T_1(n)))$$

gelten. Man kann dann auch $T_1^{O(1)} = T_2^{O(1)}$ schreiben.
Zwei (abstrakte) Maschinen M_1 und M_2 heißen polynomiell verknüpft, wenn T_{M_1} und T_{M_2} polynomiell verknüpft sind.

Die bisher eingeführten Modelle sind polynomiell verknüpft:

Satz 1.2.18 Das Modell der Turing Maschine ist polynomiell verknüpft zum Modell der RAM-Maschine. Damit gilt

$$\text{RAM-TIME}(n^{O(1)}) = \text{TIME}(n^{O(1)})$$

Es gilt auch:

$$\text{RAM-SPACE}(n^{O(1)}) = \text{SPACE}(n^{O(1)})$$

Beweis: Sei M eine Turing Maschine mit Laufzeit $T_M(n)$. Dann kann M leicht in (log.) Zeit $T_M(n)\log(T_M(n))$ auf einer RAM simuliert werden, indem jede Speicherzelle für ein Bandzelle der Turing Maschine steht.
Sei andererseits P ein RAM Programm mit Laufzeit $T_P(n)$. Dann kann P auf einer Mehrband Turing Maschine in Zeit $T_P(n)^5$ simuliert werden. Ein ausführlicher Beweis hierzu steht z.B. in [Me 84]. Diese Mehrband TM kann dann mit quadratischem Zeitverlust durch eine Offline TM oder eine Einband TM simuliert werden. □

Damit ist das Berechnungsmodell der Turing Maschine gleich stark zum Berechnungsmodell der RAM. Darüberhinaus kann die Turing Maschine die RAM auch noch effizient (d.h in polynomieller Zeit) simulieren.

1.3 Parallele RAM-Maschinen

Zunächst werden wir unser Modell der RAM erweitern. Dies führt zum parallelen Berechnungsmodell der PRAM.

Definition 1.3.1 Parallele RAM Maschine
Eine Menge von Tupeln $(\alpha, \gamma_1, \ldots, \gamma_g, LC)$ und eine Menge von Speicherzellen ρ_i, $i \in I\!N$, heißt PRAM, falls

- jedes Tupel $(\alpha, \gamma_1, \ldots, \gamma_g, LC)$ mit der Menge der Speicherzellen eine RAM bildet; (es gelten die im vorigen Kapitel definierten Befehle.)
- die Menge der Speicherzellen für alle RAM's dieselbe ist, d.h. die RAM's keinen lokalen Speicher besitzen;
- jede RAM durch eine für sie spezifische Nummer $\text{R}< \underline{\text{NUM}} >$ ausgezeichnet ist;
- alle RAM's dasselbe RAM-Programm bearbeiten.
- die RAM's sind synchronisiert, d. h. alle RAM's beginnen ihre Anweisung zeitgleich. Die nächste Anweisung wird erst gemeinsam von allen RAM's begonnen, nachdem **alle** aktiven RAM's die vorherige Anweisung ausgeführt haben.

Eine RAM als Teil einer PRAM unterscheidet sich also grundsätzlich von einer normalen RAM dadurch, daß sie keinen unendlich großen privaten Speicher hat. Daher können wir eine RAM, die Bestandteil einer PRAM ist, auch als Prozessor also als reines Rechen- und Steuerwerk ansehen. Es gilt jedoch folgende

Bemerkung 1.3.2 Es ist keine Einschränkung, daß die RAM's keinen lokalen Speicher haben. Der globale Speicher kann mittels einer geeigneten Adreßrechnung so aufgeteilt werden, daß er in unendlich große lokale Speicher für jede RAM und einem unendlich großen gemeinsamen Speicher zerfällt. Diese Speicheraufteilung kann z.B. mittels Methoden ähnlich zum Cantorschen Diagonalverfahren konstruiert werden.

Definition 1.3.3 Erweiterung des Befehlssatzes
Jede RAM erhält eine eigene Nummer. Diese Nummern bilden eine neue Kategorie:

$$< \underline{\text{RNUM}} > ::= \text{R} \in I\!N \; .$$

Zu den Sprungbefehlen wird der folgende Befehl hinzu addiert:

$$< \underline{\text{SB}} > ::= \underline{\text{IF}} < \underline{\text{RNUM}} > < \underline{\text{VO}} > < \underline{\text{NUM}} > \underline{\text{THEN}}\ \underline{\text{GOTO}} < \underline{\text{NUM}} > .$$

Außerdem wird eine weitere Klasse von Befehlen definiert, die für die Parallelität sorgen:

$$< \underline{\text{PB}} > ::= \underline{\text{FORK}} < \underline{\text{RNUM}} > .$$

Während die Semantik der bisher definierten Befehle offensichtlich war, müssen über diese neuen Befehle einige Worte verloren werden:

- Erreicht ein Prozessor einen bedingten Sprung, der in Abhängigkeit von der Prozessornummer steht, so verzweigt er in Abhängigkeit **seiner** (festen) Nummer. Dadurch kann der Kontrollfluß so beeinflußt werden, daß nicht alle Prozessoren das gleiche berechnen.

- Erreicht ein Prozessor einen FORK Befehl mit seiner Prozessornummer, so wird ein neuer Prozessor aktiviert. Dabei wird der lokale Inhalt des Prozessors $(\alpha, \gamma_1, \ldots, \gamma_g, LC)$ in den neu aktivierten Prozessor kopiert.

Es treten aber noch mehr Probleme auf. Zunächst muß definiert werden, wie eine Berechnung gestartet wird und wann eine Berechnung beendet ist.

Definition 1.3.4 Die Berechnung einer PRAM wird dadurch gestartet, daß der Prozessor mit der Prozessornummer 0 die Berechnung beginnt. Die Berechnung einer PRAM ist beendet, wenn der Prozessor mit der Prozessornummer 0 stoppt. Man kann also den Prozessor P_0 als *Master*-Prozessor ansehen.

Definition 1.3.5 Die Laufzeit TP_P eines PRAM Programmes P ist die Laufzeit des Prozessors mit der Prozessornummer 0.
SP_P bezeichne die maximale Anzahl von Prozessoren, die während der Berechnung aktiviert werden.

Bemerkung 1.3.6 Damit hängt die Laufzeit eines PRAM Programms von den Komplexitätsmaßen ab, die wir auf die aktivierten Prozessoren anwenden. Es gibt hier also auch ein Einheits- und ein logarithmisches Kostenmaß.

Durch die Definition der Laufzeit und des Prozessorenverbrauchs können jetzt auch Sprachklassen eingeführt werden.

Definition 1.3.7 Sprachklassen für PRAM
Sei $T : I\!N \to I\!N$. Dann gilt:

$$\text{PRAM-TIME}(T(n)) := \{A | \exists\, P \text{ mit } TP_P(n) = O(T(n)) \text{ und } L(P) = A\}$$

Sei $S : I\!N \to I\!N$. Dann gilt:

$$\text{PRAM-SIZE}(S(n)) := \{A | \exists\, P \text{ mit } SP_P(n) = O(S(n)) \text{ und } L(P) = A\}$$

Desweiteren ist die folgende Sprachenklasse, die Klasse der simultan beschränkt berechenbaren Funktionen.

$$\begin{aligned}\text{PRAM-TIME-SIZE}(T(n), S(n)) := \;& \{A | \exists\, P \text{ mit } TP_P(n) = O(T(n)),\\ & SP_P(n) = O(S(n)) \text{ und } L(P) = A\}\end{aligned}$$

In dieser Vorlesung wollen wir effiziente Algorithmen betrachten. Für ein paralleles Berechnungsmodell bedeutet dies, daß wir bei **polynomiell beschränkten Prozessorenaufwand** nur **polylogarithmisch viel Zeit** aufwenden wollen.

Definition 1.3.8 Spezielle Sprachklassen

$$\mathrm{PRAM}^k(S(n)) := \text{PRAM-TIME-SIZE}(\log^k n, S(n))$$

$$\mathrm{PRAM}^k := \bigcup_{S(n)=n^{O(1)}} \mathrm{PRAM}^k(S(n))$$

Die Tatsache, daß viele Prozessoren gleichzeitig auf einen gemeinsamen Speicher zugreifen, kann zu Konflikten führen.

1. Zwei Prozessoren wollen gleichzeitig aus einer Zelle lesen.
2. Ein Prozessor will aus einer Zelle lesen, in die ein anderer Prozessor schreiben will.
3. Zwei Prozessoren wollen in die gleiche Zelle schreiben.

Der Fall 2 kann leicht mit Hilfe der folgenden Definition gelöst werden.

Definition 1.3.9 Berechnungsphasen
Der Berechnungsvorgang läuft in drei Phasen ab.

1. Lesephase: Jeder Prozessor liest die Daten, die für den nächsten elementaren Befehl nötig sind, aus dem gemeinsamen Speicher.
2. Berechnungsphase: Jeder Prozessor führt eine Operation aus, die nur von den lokalen und gerade gelesenen Daten abhängig ist.
3. Schreibphase: Jeder Prozessor schreibt das Ergebnis seiner Berechnung in den gemeinsamen Speicher.

Somit findet der Lese- und der Schreibvorgang zu verschiedenen Zeiten statt; dadurch sind Lese- und Schreibkonflikte ausgeschlossen. Die anderen beiden Fälle lassen sich nicht allgemein lösen. Daher definieren wir je nach Lösung dieses Problems geeignete Modelle.

Definition 1.3.10 PRAM Modelle
Wir unterscheiden die folgenden Modelle von PRAM's.

- EREW-PRAM (Exclusive Read Exclusive Write): gemeinsames Lesen und Schreiben sind verboten.
- CREW-PRAM (Concurrent Read Exclusive Write): nur gemeinsames Lesen ist erlaubt, Schreibzugriffe auf dieselbe Speicherzelle sind verboten.
- CRCW-PRAM (Concurrent Read Concurrent Write): sowohl gemeinsames Lesen als auch gemeinsames Schreiben ist erlaubt.

Im Falle der CRCW-PRAM müssen wir noch folgende Fälle betrachten:

- Common CRCW-PRAM: gemeinsames Schreiben ist nur erlaubt, falls *alle* Prozessoren, die in eine Speicherzelle schreiben wollen, den **gleichen Wert** schreiben.

- Arbitrary CRCW-PRAM: beim gemeinsamen Schreiben setzt sich einer der Prozessoren **willkürlich** durch.
- Priority CRCW-PRAM: beim gemeinsamen Schreiben setzt sich der Prozessor mit der **kleinsten Prozessornummer** durch.

Das Modell der PRAM erweitert die Menge der berechenbaren Funktionen nicht, da PRAM's leicht durch RAM's simuliert werden können. Dabei simuliert die RAM in $SP(n)$ Schritten je einen Schritt der $SP(n)$ aktiven Prozessoren der PRAM. Dadurch erhalten wir einen RAM-Programm mit Laufzeit $SP(n) \cdot TP(n)$.
Ein PRAM-Programm ist somit optimal, wenn das Prozessor-Zeit-Produkt $SP(n) \cdot TP(n)$ in der gleichen Größenordnung wie die Zeit des besten sequentiellen RAM-Programms liegt.

Satz 1.3.11 Zwischen den durch die verschiedenen Typen von PRAM's definierten Sprachklassen gelten die folgenden Beziehungen im Einheitskostenmaß:

1. Priority-CRCW-PRAM$(T(n), S(n)) \subseteq$ EREW-PRAM$(T(n) \cdot \log S(n), S(n))$
2. Priority-CRCW-PRAM$(T(n), S(n)) \subseteq$ Common-CRCW-PRAM$(T(n), S(n)^2)$
3. EREW-PRAM$(T(n), S(n)) \subseteq$ Common-CRCW-PRAM$(T(n), S(n))$
   $\subseteq$ Priority-CRCW-PRAM$(T(n), S(n))$

BEWEIS:

zu 1 Wir wollen ein Priority-CRCW-PRAM Programm durch ein EREW-PRAM Programm simulieren. Dabei kann es sowohl beim Schreiben als auch beim Lesen zu Schwierigkeiten kommen.

- Schreiben
  Wir nehmen an, daß die Prozessoren $P_1 \ldots P_k$ in die Speicherzellen $\rho_1 \ldots \rho_r$ schreiben wollen. Jedem Prozessor P_i ordnen wir ein Tupel (j_i, i, x_i) zu, wobei x_i der Wert ist, den der Prozessor P_i in die Speicherzelle ρ_{j_i} schreiben möchte. Nun sortieren wir die Folge der Tupel $\{(j_i, i, x_i)\}_{1 \leq i \leq k}$ nach der lexikographischen Ordnung. Verwenden wir hierzu den parallelen Mergesort von Cole ([Co 86] und [GR 88]), so benötigen wir hierzu nur k Prozessoren und $O(\log k)$ parallele Zeit. Mit Hilfe der sortierten Folge können wir jetzt leicht festlegen, welcher Prozessor schreiben darf. Für alle Elemente der Folge gilt: Schreibe x_i nach ρ_{j_i},
  - falls das Tupel (j_i, i, x_i) das erste Tupel in der sortierten Folge ist oder
  - falls für den Vorgänger (j_l, l, x_l) des Tupels (j_i, i, x_i) $\quad j_l < j_i$ gilt,

  da in diesem Fall i die kleinste Nummer eines Prozessors ist, der nach ρ_{j_i} schreiben möchte.
- Lesen
  Sei $S := \{(j_i, i) | P_i \text{ liest aus } \rho_{j_i}\}$. Sortiere S aufsteigend nach der lexikographischen Ordnung mit $|S|$ Prozessoren und in $O(\log |S|)$ paralleler Zeit. Bezeichne $S = \{s_1, \ldots, s_k\}$ jetzt die sortierte Folge. Falls für ein i $\; s_i = (j_i, i)$ und $s_{i+1} = (j_{i+1}, i+1)$ mit $j_i \neq j_{i+1}$ gilt, so setze

$$\max(j_i) = i \text{ und } \min(j_{i+1}) = i + 1.$$

Für jedes j fertige $\max(j) - \min(j)$ Kopien von ρ_j an und verteile diese an die entsprechenden Prozessoren.

Damit ist eine Simulation innerhalb der behaupteten Schranken erreicht, da maximal $S(n)$ Prozessoren parallel lesen und schreiben. □

zu 2 Wir wollen nun eine Priority-CRCW-PRAM durch eine Common-CRCW-PRAM simulieren. Dazu müssen wir unser Programm so abändern, daß sichergestellt ist, daß beim parallelen Schreiben in eine Speicherzelle nur die gleichen Werte geschrieben werden. Dazu nehmen wir an, daß die Prozessoren $P_1 \dots P_k$ in die Speicherzellen $\rho_1 \dots \rho_r$ schreiben wollen. Jedem Prozessor P_i ordnen wir jetzt ein Tupel (j_i, i) zu, wobei ρ_{j_i} die Speicherzelle ist, in die P_i schreiben will. N_i seien Hilfsspeicherzellen, die mit 0 initialisiert sind. Nun führe mit den Hilfsprozessoren $P_{i',i}$ $\quad 1 \leq i' < i \leq k$ folgende Zuweisung parallel aus:

$$N_i := 1 \text{ falls } j_i = j_{i'} \ .$$

Damit enthält N_i genau dann eine 1, falls der Prozessor P_i nicht schreiben darf. Dies ist genau dann der Fall, wenn es einen Prozessor mit kleinerer Nummer gibt, der auf die gleiche Zelle schreiben will. Da immer nur der Wert 1 geschrieben wird, ist dies ein Common-CRCW Algorithmus. Nun überprüft jeder Prozessor mit Hilfe von N_i, ob er schreibberechtigt ist und schreibt gegebenenfalls. □

zu 3 Die verschiedenen Berechnungsmodelle sind Verallgemeinerungen bzw. Spezialfälle voneinander. Daher ist die Inklusionskette klar. □

Als Abschluß dieses Abschnittes wollen wir nun noch einige einfache Beispiele von parallelen Algorithmen geben.

Beispiel 1.3.12 Skalarprodukt
Sei M ein Ring und $v, u \in M^n$ zwei Vektoren. Dann liegt die Berechnung des Skalarproduktes bzgl. des Einheitskostenmaßes in

$$\text{EREW-PRAM}(\log n, n/\log n).$$

Zum Beweis des Beispiels benötigen wir den folgenden wichtigen

Satz 1.3.13 Brent's Theorem
Sei P eine Algorithmus mit paralleler Laufzeit t, der insgesamt m Operationen benötigt. Dann kann P so implementiert werden, daß er mit p Prozessoren $O(t + m/p)$ parallele Zeit benötigt.

Beweis: Bezeiche $m(i)$ die Anzahl der parallel ausgeführten Operationen zum Zeitpunkt $1 \leq i \leq t$. Diese können mit p Prozessoren in Zeit $\frac{m(i)}{p} + 1$ Zeit ausgeführt werden. Wenn wir nun noch über t summieren erhalten wir für die Gesamtzeit

$$T_P(n) = \sum_{i=1}^{t} (\frac{m(i)}{p} + 1) = m/p + t.$$

□

BEWEIS: des Beispiels
Der erste Schritt unseres Algorithmus A_1 aktiviert in $O(\log n)$ Zeit n Prozessoren. Diese Prozessoren berechnen dann parallel die Produkte $v_i \cdot u_i$ für $1 \leq i \leq n$. Nun wird mit Hilfe eines binären Baumes der Tiefe $\log n$ die Summe der n Produkte berechnet. Damit ist unser Algorithmus A_1 in EREW-PRAM$(\log n, n)$.
Der Satz 1.3.13 besagt jedoch, daß wir unseren Algorithmus noch verbessern können. In unserem Fall ist $t = \log n$ und $m = 2n$. Bei der Wahl von $p = n/\log n$ Prozessoren erreichen wir eine parallele Berechnungszeit von $O(\log n)$. Unser modifizierter Algorithmus sieht daher folgendermaßen aus:

- aktiviere $n/\log n$ Prozessoren;
- jeder Prozessor berechnet eine Teilsumme der Länge $\log n$ in $O(\log n)$ sequentieller Zeit;
- die $n/\log n$ Teilergebnisse werden durch einen Baum der Tiefe $O(\log n)$ zusammengefaßt.

□

Damit gilt der folgende

Satz 1.3.14 Bezeichne SKP die Berechnung des Skalarproduktes, VMAT die Vektor - Matrixmultiplikation und MMAT die Matrix-Matrixmultiplikation. Dann gelten mit Hilfe des Beispiels 1.3.12 folgende Aussagen in Bezug auf das Einheitskostenmaß:

$$\begin{array}{ll} \text{SKP} & \in \text{EREW-PRAM}(\log n, n/\log n) \\ \text{VMAT} & \in \text{EREW-PRAM}(\log n, n^2/\log n) \\ \text{MMAT} & \in \text{EREW-PRAM}(\log n, n^3/\log n). \end{array}$$

Beispiel 1.3.15 Sieb des Erastosthenes
Wir wollen alle Primzahlen zwischen 1 und n berechnen. Dazu benutzen wir das Sieb des Erastosthenes: Ein Feld $a(i)_{1\leq i\leq n}$ wird ausgegeben mit: $a(i) = 1 \Leftrightarrow i$ prim. Der sequentielle Algorithmus sieht folgendermaßen aus ($a(i)$ sei mit 1 initialisiert):

```
for i := 2 to √n do
    for k := 2 to n/i do
        a(k · i) := 0
```

Für diesen sequentiellen Algorithmus gilt nun die folgende Laufzeitabschätzung im Einheitskostenmaß:

$$T_A = \sum_{i=2}^{\sqrt{n}} \frac{n}{i} = n \cdot \sum_{i=2}^{\sqrt{n}} \frac{1}{i} = O(n \log n).$$

Nun wollen wir diesen Algorithmus parallelisieren. Dies geschieht analog zu Beispiel 1.3.12, indem zunächst $\sqrt{n}$ Prozessoren für die äußere Schleife aktiviert werden. Der i-te Prozessor aktiviert nun $\frac{n}{i \log n}$ Prozessoren, die in $\log n$ paralleler Zeit die innere Schleife realisieren. Insgesamt läuft der parallele Algorithmus also in $\log n$ paralleler Zeit. Eine Abschätzung für den Prozessorenaufwand ist

$$\sum_{i=2}^{\sqrt{n}} \frac{n}{i \log n} = O(\frac{n \log n}{\log n}) = O(n).$$

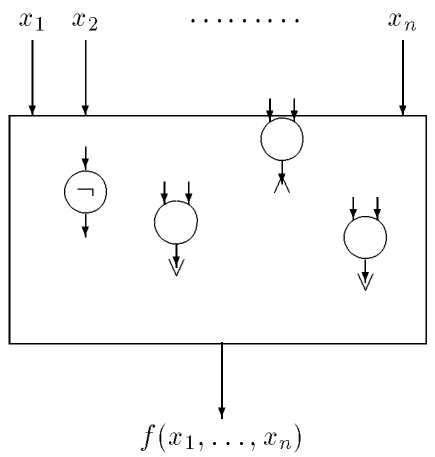

Abbildung 1.3: Schaltkreis

Bei einer parallelen Berechnungszeit von $O(\log n)$ werden also nur $O(n)$ Prozessoren verwendet. Daher ist dieser Algorithmus eine optimale Parallelisierung der sequentiellen Implementierung des Sieb des Erastosthenes.

1.4 Uniforme Schaltkreise

Nun wenden wir uns den Schaltkreisen als Berechnungsmodell zu.

Definition 1.4.1 Schaltkreis
Ein Schaltkreis $\mathcal{C}$ ist ein azyklischer, gerichteter Graph $G = (V, E)$. Die Knoten aus V sind die Gatter des Schaltkreises $\mathcal{C}$, die elementare boolesche Funktionen oder Eingaben darstellen. Die Kanten von G sind die Verbindungen zwischen den Gattern. Der Fan-In eines Gatters ist der Eingangsgrad des entsprechenden Knoten in G, der Fan-Out ist der Ausgangsgrad. Der Schaltkreis besitzt n Eingabeknoten mit Eingangsgrad 0 und einen Ausgabeknoten mit Ausgangsgrad 0. Ein Schaltkreis ist in Abbildung 1.3 dargestellt. Wir unterscheiden folgende Arten von Schaltkreisen:

- Schaltkreise mit beschränktem Fan-In und unbeschränktem Fan-Out und
- Schaltkreise mit unbeschränktem Fan-In und Fan-Out.

Die Gatter des Schaltkreises sind mit Elementen einer vollständigen Basis gelabelt, d.h. einer Menge, aus der alle booleschen Funktionen aufgebaut werden können. Normalerweise ist diese Basis die Menge $\{\vee, \wedge, \neg\}$. Es können jedoch auch die Basen $\{\neg, \vee\}$ und $\{\wedge, \oplus\}$ verwendet werden. Im letzteren Fall stellt der Schaltkreis ein Polynom über dem Körper mit zwei Elementen GF(2) dar.

Im folgenden werden wir zunächst nur Schaltkreise mit beschränktem Fan-In betrachten. Die folgenden Definitionen gelten jedoch zum Teil auch für Schaltkreise mit unbeschränktem Fan-In.

Definition 1.4.2 Größe und Tiefe von Schaltkreisen
Für einen Schaltkreis $\mathcal{C}$ können wir die Größe und die Tiefe folgendermaßen definieren.

$$\mathrm{Size}(\mathcal{C}) = \text{Anzahl der Gatter in } \mathcal{C},$$

$$\mathrm{Depth}(\mathcal{C}) = \max\{|p| \mid p \text{ ist (gerichteter) Weg in } \mathcal{C}\}.$$

Bemerkung 1.4.3 Schaltkreise mit beschränktem Fan-In und unbeschränktem Fan-Out können in Schaltkreise mit beschränktem Fan-In und Fan-Out transformiert werden, wobei sowohl die Größe als auch die Tiefe nur um einen konstanten Faktor anwächst. Näheres kann in [HKP 84] nachgelesen werden.

Schaltkreise berechnen boolesche Funktionen $f : \mathbb{B}^n \to \mathbb{B}$ in kanonischer Weise. Um Schaltkreise näher zu untersuchen, müssen also auch boolesche Funktionen betrachtet werden.

Definition 1.4.4 Sei $A \subset \{0,1\}^*$. Dann ist $A^n := A \cap \{0,1\}^n$. Mit A und A^n seien sowohl die Mengen als auch ihre charakteristischen Funktionen gemeint, d.h.

$$A(x) = \begin{cases} 1 & \text{für } x \in A \\ 0 & \text{für } x \notin A. \end{cases}$$

Außerdem sei

$$\mathbb{B}_n := \{f | f : \mathbb{B}^n \to \mathbb{B}\}.$$

Definition 1.4.5 Sei x ein boolescher Wert oder eine boolesche Variable. Dann setzen wir $x^1 = x$ und $x^0 = \neg x$. Mit diesen Bezeichnungen sei ein Minterm m_a für $a = (a(1), \ldots, a(n)) \in \{0,1\}^n$ definiert durch

$$m_a = x_1^{a(1)} \wedge \cdots \wedge x_n^{a(n)}.$$

Analog ist der Maxterm s_a definiert durch

$$s_a = x_1^{\neg a(1)} \vee \cdots \vee x_n^{\neg a(n)}.$$

Damit gilt folgender Satz über die Darstellung von booleschen Funktionen.

Satz 1.4.6 Sei $f : \mathbb{B}^n \to \mathbb{B}$ eine boolesche Funktion. Dann hat f die Darstellung als *Disjunktive Normalform (DNF)*

$$f = \bigvee_{a \in f^{-1}(1)} m_a$$

und als *Konjunktive Normalform (KNF)*

$$f = \bigwedge_{a \in f^{-1}(0)} s_a.$$

Neben diesen wohl schon bekannten kanonischen Darstellungsformen für boolesche Funktionen gibt es noch die *RSE-Ausdrücke*, die boolesche Funktionen als Polynome über GF(2) darstellen.

Satz 1.4.7 Ring Sum Expansion Ausdrücke
Für jede boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ gibt es genau einen eindeutig bestimmten 0-1 Vektor $a = (a_A)_{A \subseteq \{1,\ldots,n\}}$, so daß

$$f = \bigoplus_{A \subseteq \{1,\ldots,n\}} a_A \wedge \bigwedge_{i \in A} x_i.$$

Diese Darstellung von f heißt RSE-Ausdruck.

Beweis: Wir zeigen, daß die Abbildung $\{0,1\}^{2^n} \to \mathbb{B}_n$, $a_A \mapsto f$ bijektiv ist. Sie ist injektiv, da zwei verschiedene Vektoren a_A und a_B offensichtlich zwei verschiedene Funktionen repräsentieren. Da die Mengen $\mathbb{B}_n$ und $\{0,1\}^{2^n}$ die gleiche Kardinalität besitzen, folgt aus der Injektivität die Bijektivität. Damit ist jedem Vektor a_A eineindeutig eine boolesche Funktion $f \in \mathbb{B}_n$ zugeordnet und jedes $f \in \mathbb{B}_n$ besitzt eine Darstellung als RSE-Ausdruck. □

In Schaltkreisen mit unbeschränktem Fan-In können die DNF, KNF und die RSE-Ausdrücke in Tiefe 2 realisiert werden. Die Anzahl der Gatter kann jedoch exponentiell hoch sein. Ein Schaltkreis mit exponentiell vielen Schaltgattern ist aber aufgrund seiner unpraktikablen Größe uninteressant. Außerdem ist es technisch schwierig, Gatter mit unbeschränktem Fan-In zu realisieren. Daher betrachten wir im folgenden nur Schaltkreise mit beschränktem Fan-In. Dabei sei der Fan-In ohne Einschränkung $= 2$. Die Schwierigkeit in der technischen Realisierung gilt auch (mit gewissen Abstrichen) für den Fan-Out. Nach Bemerkung 1.4.3 können wir aber auch äquivalente Schaltkreise mit beschränktem Fan-Out betrachten.

Zwischen Schaltkreisen und den bisher betrachteten Berechnungsmodellen tritt nun ein wesentlicher Unterschied auf. Während sowohl die Turing Maschine als auch die RAM eine Funktion $f : \mathbb{B}^* \to \mathbb{B}$ berechnen, so berechnet ein Schaltkreis eine Funktion $g : \mathbb{B}^n \to \mathbb{B}$ für ein festes, durch den Schaltkreis definiertes n. Ein Schaltkreis ist ein fester Hardware Chip und berechnet somit **genau eine** boolesche Funktion und **nicht eine Familie** von booleschen Funktionen wie die RAM oder die Turing Maschine. Wenn wir also Schaltkreise mit den anderen Berechnungsmodellen vergleichen wollen, so müssen wir Familien von Schaltkreisen betrachten. Dabei ist für jede Eingabegröße ein Schaltkreis in der Schaltkreisfamilie.

Definition 1.4.8 Berechnung einer Schaltkreisfamilie
Sei $\{\mathcal{C}_n\}_{n \in \mathbb{N}}$ eine Familie von Schaltkreisen. $\{\mathcal{C}_n\}$ berechnet eine Menge $A \subseteq \{0,1\}^*$ bzw. die dazugehörige charakteristische Funktion genau dann, wenn für jedes $n \in \mathbb{N}$ der Schaltkreis $\mathcal{C}_n$ die Menge A^n berechnet. Wir schreiben dann

$$L(\{\mathcal{C}_n\}) = A.$$

Damit können wir wieder Komplexitätsklassen definieren.

Definition 1.4.9 Seien $S, T : \mathbb{N} \to \mathbb{N}$. Dann sind

$$\mathrm{SIZE}(S(n)) = \{A \mid \exists \{\mathcal{C}_n\} \text{ mit } L(\{\mathcal{C}_n\}) = A \text{ und } \mathrm{Size}(\mathcal{C}_n) = O(S(n))\}$$

und

$$\mathrm{DEPTH}(T(n)) = \{A \mid \exists \{\mathcal{C}_n\} \text{ mit } L(\{\mathcal{C}_n\}) = A \text{ und } \mathrm{Depth}(\mathcal{C}_n) = O(T(n))\}.$$

Wir möchten nun zu einer gegebenen booleschen Funktion einen entsprechenden Schaltkreis konstruieren. Dieser sollte möglichst kleine Größe und möglichst kleine Tiefe besitzen. Es gilt der folgende Satz.

Satz 1.4.10 Folgende Resultate sind bekannt.

1. SIZE$(2^n/n) = I\!B_n$, d.h. jede boolesche Funktion ist mit $O(2^n/n)$ Schaltelementen berechenbar.
2. DEPTH$(n) = I\!B_n$, d.h. jede boolesche Funktion ist in Tiefe $O(n)$ berechenbar.

Beweis:

1. Ein Beweis dieses Resultats von Lupanov kann z.B. in [Sa 76] nachgelesen werden.
2. Dieses Ergebnis ist trivial, da die DNF eine Tiefe $n + \lceil \log n \rceil$ besitzt. Es gilt sogar: jede boolesche Funktion kann in Tiefe $n + 1$ berechnet werden (siehe [MP 77]). □

Da jeder Schaltkreis eine feste Eingabegröße hat, müssen wir also für jede Eingabegröße einen Schaltkreis konstruieren. Dabei gilt ein Schaltkreis als konstruiert, falls die Beschreibung des zugrundeliegenden, azyklischen Graphen vorliegt. Was nutzt jedoch ein effizienter, kleiner und flacher Schaltkreis, wenn die Konstruktion sehr aufwendig ist? Daher betrachten wir nur Schaltkreise, die einfach zu konstruieren sind. Solche Schaltkreise nennen wir uniforme Schaltkreise. Ein weiterer Grund zur Betrachtung von uniformen Schaltkreisen liegt in der Tatsache, daß es Schaltkreisfamilien gibt, die Sprachen erkennen, die nicht rekursiv aufzählbar sind.

Definition 1.4.11 uniforme Schaltkreise
Bezeichne $\bar{\mathcal{C}}$ eine Beschreibung des Schaltkreises $\mathcal{C}$. Diese Beschreibung ist z.B. die Listendarstellung für einen gerichteten gelabelten Graphen. Dann ist die Familie $\{\mathcal{C}_n\}$ uniform, falls die Compiler-Funktion $1^n \to \bar{\mathcal{C}}_n$ Turing berechenbar ist. Wir nennen die Familie $\{\mathcal{C}_n\}$ $S(n)$-space uniform bzw. $T(n)$-Zeit uniform, falls es eine $S(n)$-space bzw. $T(n)$-Zeit beschränkte Turing Maschine gibt, die die Compiler Funktion berechnet. Insbesondere bezeichnet der Begriff P-uniform die Familien, die $p(n)$-Zeit uniform für ein Polynom p sind, und der Begriff log-space uniform die Familien, die mit einer log-space beschränkten Turing Maschine berechnet werden können. Falls Zweideutigkeiten ausgeschlossen sind, werden wir den Begriff *uniform* anstatt des Begriffes *log-space uniform* verwenden.

Eine uniforme Schaltkreisfamilie nennen wir auch UC-Algorithmus bzw. UC$_P$-Algorithmus. Dabei ist jeder UC-Algorithmus auch UC$_P$-Algorithmus, da jede log-space berechenbare Funktion auch in polynomieller Zeit berechnet werden kann.

Im folgenden werden wir nur uniforme Schaltkreise betrachten. Zu diesen Schaltkreisfamilien können wir wieder geeignet Sprachklassen definieren.

Definition 1.4.12 Sei $f : I\!N \to I\!N$. Damit sei

$$\begin{aligned}
\text{U-DEPTH}(f(n)) = \;& \{A \mid \exists\, \text{UC-Algorithmus } \{\mathcal{C}_n\} \text{ mit} \\
& L(\{\mathcal{C}_n\}) = A \text{ und Depth}(\mathcal{C}_n) = O(f(n))\},
\end{aligned}$$

$$\begin{aligned}
\text{U}_P\text{-DEPTH}(f(n)) = \;& \{A \mid \exists\, \text{UC}_P\text{-Algorithmus } \{\mathcal{C}_n\} \text{ mit} \\
& L(\{\mathcal{C}_n\}) = A \text{ und Depth}(\mathcal{C}_n) = O(f(n))\},
\end{aligned}$$

$$\begin{aligned}\text{U-SIZE}(f(n)) = \;& \{A \mid \exists\, \text{UC-Algorithmus } \{\mathcal{C}_n\} \text{ mit} \\ & L(\{\mathcal{C}_n\}) = A \text{ und } \text{Size}(\mathcal{C}_n) = O(f(n))\},\end{aligned}$$

$$\begin{aligned}\text{U}_{\mathcal{P}}\text{-SIZE}(f(n)) = \;& \{A \mid \exists\, \text{UC}_{\mathcal{P}}\text{-Algorithmus } \{\mathcal{C}_n\} \text{ mit} \\ & L(\{\mathcal{C}_n\}) = A \text{ und } \text{Size}(\mathcal{C}_n) = O(f(n))\},\end{aligned}$$

Nun werden wir die wichtigsten Komplexitätsklassen der simultan beschränkbaren Sprachen kennenlernen.

Definition 1.4.13 Nick's Class
Nach Nick Pippenger sind die folgenden Komplexitätsklassen benannt.

$$\begin{aligned}\text{NC}^k = \;& \{A \mid \exists\, \text{UC-Algorithmus } \{\mathcal{C}_n\} \text{ mit beschränktem Fan-In und mit} \\ & L(\{\mathcal{C}_n\}) = A, \text{Size}(\mathcal{C}_n) = n^{O(1)} \text{ und } \text{Depth}(\mathcal{C}_n) = O(\log^k n)\},\end{aligned}$$

$$\text{NC} = \bigcup_{k \in I\!N} \text{NC}^k.$$

Wir können die gleichen Definitionen auch auf Schaltkreise mit unbeschränktem Fan-In anwenden. Dabei entstehen neue Komplexitätsklassen. Besonders wichtig ist das Analogon zu den NC^k Klassen. Diese Klassen bezeichnen wir mit AC^k.

1.5 Beziehungen zwischen den Maschinenmodellen

1.5.1 PRAMs und uniforme Schaltkreise

In den vorherigen Abschnitten haben wir Sprachklassen in Abhängigkeit von dem betrachteten Modell eingeführt. Nun wollen wir diese Sprachklassen miteinander in Beziehung setzen.

Satz 1.5.1 Es gilt

$$\text{NC} \subseteq \mathcal{P}.$$

Beweis: Ein Schaltkreis mit einer polynomiellen Anzahl von Gattern kann leicht in polynomieller Zeit ausgewertet werden. Dies ist das sogenannte **Circuit Value Problem**, das wir in anderem Zusammenhang noch betrachten werden. □

Die allgemeine Vermutung ist, daß NC eine echte Teilmenge von $\mathcal{P}$ ist, d.h. nicht alle polynomiell lösbaren Probleme effizient parallelisierbar sind. Später werden wir einige dieser schwierigen Probleme kennenlernen.

Satz 1.5.2 Es gilt

$$\text{NC}^k \subseteq \text{AC}^k \subseteq \text{NC}^{k+1}.$$

BEWEIS: Ein Schaltkreis mit beschränktem Fan-In ist nur ein Spezialfall eines Schaltkreises mit unbeschränktem Fan-In. Daher gilt

$$\mathrm{NC}^k \subseteq \mathrm{AC}^k.$$

Sei nun ein AC^k-Schaltkreis $\mathcal{C}$ gegeben. Dann kann der Fan-In eines Gatters von $\mathcal{C}$ höchstens polynomiell groß sein. Das bedeutet, daß wir die Eingabe jedes Gatters durch einen binären Baum der Tiefe höchstens $\log(\mathrm{Size}(\mathcal{C}))$ ersetzen. Dabei hat jeder Knoten des binären Baums Fan-In 2. Durch diese Konstruktion erhalten wir einen neuen Schaltkreis $\mathcal{C}'$ mit beschränktem Fan-In, für den folgendes gilt:

$$\mathrm{Size}(\mathcal{C}') \leq \mathrm{Size}(\mathcal{C})^2,$$

$$\mathrm{Depth}(\mathcal{C}') \leq \mathrm{Depth}(\mathcal{C}) \cdot \log(\mathrm{Size}(\mathcal{C}))$$

und damit

$$\mathrm{Depth}(\mathcal{C}') \leq \mathrm{Depth}(\mathcal{C}) \cdot O(\log n),$$

da $\mathrm{Size}(\mathcal{C})$ polynomiell in n ist. □

Als letztes sei der folgende Satz gegeben.

Satz 1.5.3 Es gilt die folgende Kette von Inklusionen.

$$\mathrm{NC}^k \subseteq \text{EREW-PRAM}^k \subseteq \text{CREW-PRAM}^k \subseteq \text{CRCW-PRAM}^k = \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}.$$

BEWEIS: Die erste Ungleichung basiert auf der Arbeit [HKP 84], d.h. wir können den Schaltkreis mit unbeschränktem Fan-Out durch einen Schaltkreis mit beschränktem Fan-Out ersetzen, dessen Größe und Tiefe nur um einen konstanten Faktor anwächst. Somit brauchen wir nur einen Schaltkreis mit beschränktem Fan-In und beschränktem Fan-Out auf der EREW - PRAM zu simulieren. Dies kann leicht geschehen, indem jedes Gatter durch einen Prozessor simuliert wird.
Die übrigen Ungleichungen sind bereits in den Sätzen 1.3.11 und 1.5.2 bewiesen worden.
Die Gleichheit $\text{CRCW-PRAM}^k = \mathrm{AC}^k$ beruht auf Resultaten von Stockmeyer und Vishkin. Näheres steht in [KR 89] und [SV 84]. □

1.5.2 Sätze von Borodin

In diesem Kapitel beweisen wir zwei sehr wichtige Sätze, die Relationen zwischen den parallelen UC-Sprachklassen und den sequentiellen Sprachklassen aufzeigen. Sie stammen aus [Bo 77].

Definition 1.5.4 Die Klasse $\mathrm{NSPACE}(S(n))$ bezeichne die Klasse von Sprachen A, für die es eine nichtdeterministische Turing Maschine M mit $L(M) = A$ gibt, die höchstens $S(n)$ Speicher benötigt.

Satz 1.5.5 1.Satz von Borodin
Sei $S(n) \geq \log n$ und sei $A \in \mathrm{NSPACE}\,(S(n))$. Dann gibt es eine $S(n)$-space uniforme Familie von Schaltkreisen $\{\mathcal{C}_n\}$ der Tiefe $\mathrm{Depth}\,(\mathcal{C}_n) = O(S(n)^2)$, die A berechnet. Insbesondere gilt

$$\mathrm{NSPACE}(\log n) \subseteq \text{U-DEPTH}(\log^2 n).$$

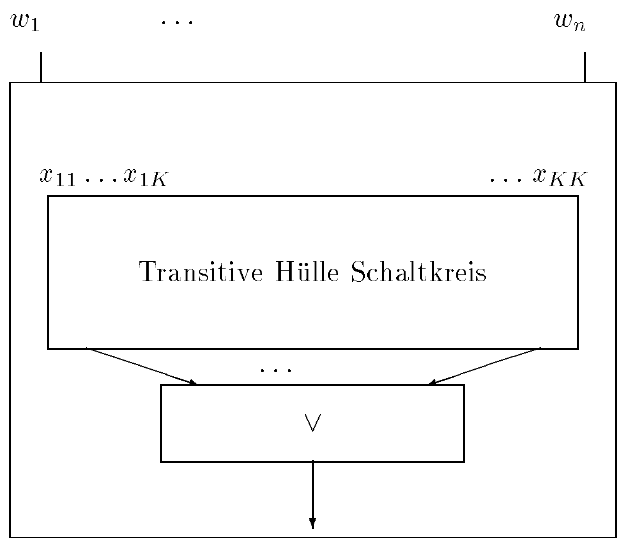

Abbildung 1.4: Simulationsschaltkreis

BEWEIS: Um diese Beziehung zwischen den Sprachklassen zu zeigen, müssen wir eine nichtdeterministische TM M auf einer Eingabe w durch einen entsprechenden uniformen Schaltkreis $\mathcal{C}$ simulieren. Die Idee ist die folgende:
Wir konstruieren einen Graphen, der die Zustandsüberführung der TM M simuliert. Jede Konfiguration von M kann dann durch einen Knoten in diesem Graphen G repräsentiert werden. Eine mögliche Konfigurationsänderung von M entspricht einer (gerichteten) Kante in G zwischen den entsprechenden Knoten. Nun wird die transitive Hülle dieses Graphen berechnet. Falls der Knoten der Anfangskonfiguration in der transitiven Hülle mit einem Knoten verbunden ist, der einer akzeptierenden Konfiguration entspricht, so akzeptiert der Schaltkreis, andernfalls akzeptiert der Schaltkreis nicht. Der so konstruierte Schaltkreis ist in Abbildung 1.4 dargestellt.

Es muß also im wesentlichen ein uniformer Schaltkreis konstruiert werden, der die transitive Hülle eines Graphen berechnet. Es sei also eine nichtdeterministische TM M gegeben.
Zunächst wollen wir die Anzahl möglicher Konfigurationen abschätzen. Die endliche Kontrolle habe q Zustände, die Länge $|w|$ der Eingabe sei n. Die Kardinalität des Bandalphabets sei s und $S(n)$ eine obere Schranke für den Platzverbrauch. Damit gibt es $s^{S(n)}$ verschiedene Konfigurationen des Arbeitsbandes. Somit gilt für die Anzahl aller möglichen Konfigurationen:

$$K \leq n \cdot q \cdot s^{S(n)} \cdot S(n).$$

Als nächstes konstruieren wir die Adjazenzmatrix des Graphen G. In G sind zwei Knoten v_1 und v_2 genau dann durch eine gerichtete Kante miteinander verbunden, wenn die entsprechende Konfiguration K_2 eine Nachfolgekonfiguration von K_1 ist. Dazu konstruieren wir die Adjazenzmatrix $X = (x_{ij})$ von G. Diese wird mit Hilfe der Eingabe $w_1, \dots, w_n$ und der Zustandsüberführungsfunktion von M wie folgt aufgebaut:

Sei i die Nummer einer Konfiguration, in der der Lesekopf über dem k-ten Zeichen des Eingabebandes steht. Dann gilt:

- x_{ij} erhält den Wert von w_k, falls j nur dann eine Nachfolgekonfiguration von i ist, wenn $w_k = 1$ ist.
- x_{ij} erhält den negierten Wert von w_k, falls j nur dann eine Nachfolgekonfiguration von i ist, wenn $w_k = 0$ ist.
- x_{ij} erhält den Wert 1, falls j in *jedem* Fall (unabhängig von w_k) eine Nachfolgekonfiguration von i ist.
- x_{ij} erhält den Wert 0, falls j in *keinem* Fall (unabhängig von w_k) eine Nachfolgekonfiguration von i ist.

Diese Verdrahtungen werden für jede Konfiguration i (und damit festgelegtem k) mit Hilfe der Zustandsüberführungstabelle durchgeführt.

Das nachstehende Lemma 1.5.6 sichert uns zu, daß die Berechnung der transitiven Hülle einer $K \times K$ Matrix in Tiefe $\log^2 K$ möglich ist.
Nun muß festgestellt werden, ob die Anfangskonfiguration mit einer akzeptierenden Konfiguration verbunden ist. Dazu werden die Knoten, die akzeptierenden Konfigurationen entsprechen, durch ein großes $\vee$ zur Ausgabe geleitet. Dieses $\vee$ hat — aufgebaut durch einen binären Baum — eine Tiefe von $\lceil \log r \rceil \leq \lceil \log K \rceil$, da die Anzahl der akzeptierenden Konfigurationen $\leq K$ ist. Die Ausgabe des Schaltkreises ist genau dann 1, wenn wenigstens eine mögliche Berechnung der TM M in einer akzeptierenden Konfiguration endet. Damit sichert die obige Konstruktion, daß die simulierte TM und die Schaltkreisfamilie die gleiche Sprache erkennen.

Die Tiefe des benötigten Schaltkreises ist somit $O(\log^2 K)$. Da die Dimension der Adjazenzmatrix $K \leq qns^{S(n)}S(n)$ ist, gilt $\log K \leq cS(n)$ und somit ist die Tiefe des Schaltkreises $O(S^2(n))$.
□

Lemma 1.5.6 Die Berechnung der transitiven Hülle eines Graphen G liegt in $NC^2(n^3)$.

Beweis: Sei $X = (x_{ij})$ die Adjazenzmatrix für G und E_n die $n \times n$ Einheitsmatrix. Dann gilt für die Adjazenzmatrix X^* der transitiven Hülle des Graphen:

$$\begin{aligned} X^* &= \bigvee_{k=0}^{\infty} X^k \\ &= \bigvee_{k=0}^{n} X^k \quad \text{da der längste Weg} \leq n \text{ ist} \\ &= (E_n \vee X)^n \quad \text{leicht induktiv zu zeigen} \\ &= (E_n \vee X)^{2^{\lceil log n \rceil}} \end{aligned}$$

Mit Hilfe dieser Darstellung kann X^* durch $\log n$ boolesche Matrixmultiplikationen berechnet werden. Nach Satz 1.3.14 kann eine boolesche Matrixmultiplikation in $NC^1(n^3)$ realisiert werden. Somit liegt die Berechnung von X^* in $NC^2(n^3)$. □

Nun beweisen wir eine Inklusion in die andere Richtung.

Satz 1.5.7 2.Satz von Borodin
Sei $S(n) \geq \log n$ und $c \in I\!N$. Dann gilt:

$$\text{U-DEPTH}(S(n)) \cap \{\{\mathcal{C}_n\} \mid \text{Fan-In}(\mathcal{C}_n) \leq c\} \subseteq \text{DSPACE}(S(n)).$$

Beweis: Wir müssen einen gegebenen uniformen Schaltkreis mit einer gegebenen Eingabe auswerten. Dieses Problem nennt man das Circuit Value Problem. Die zugehörige Sprache ist wie folgt definiert:

$$CV := \{x_1 \dots x_n \# \bar{\mathcal{C}} | \mathcal{C}(x_1, \dots, x_n) = 1\},$$

wobei $\bar{\mathcal{C}}$ die Beschreibung des Schaltkreises $\mathcal{C}$.

Es gilt folgendes

Lemma 1.5.8

$$CV \in \text{DSPACE}(\text{Depth}(\mathcal{C}) \cdot \log(\text{Size}(\mathcal{C}))).$$

Beweis: Zur Lösung dieses Problems wird ein Kellerspeicher benutzt. Bei der rekursiven Auswertung des Schaltkreises wird der Kellerspeicher nur mit höchstens Depth($\mathcal{C}$) Werten belegt. Jeder dieser Werte ist die Nummer eines Gatters und der bisher errechnete Wert für dieses Gatter. Dieses Wertepaar benötigt Speicher von der Größe $O(\log(\text{Size}(\mathcal{C})))$. Somit wird insgesamt Speicher von der behaupteten Größe verbraucht. □

Das Ergebnis dieses Lemmas befriedigt uns noch nicht, da die Größe eines Schaltkreises exponentiell zur Tiefe stehen kann. Somit liefert Lemma 1.5.8 nur:

$$\text{U-DEPTH}(S(n)) \subseteq \text{DSPACE}(S(n)^2).$$

Wir müssen also bei der Auswertung des Schaltkreises geschickter vorgehen. Dies wird durch die Uniformität des Schaltkreises ermöglicht, d.h. dadurch, daß $\bar{\mathcal{C}}$ immer wieder in log-space berechnet werden kann. Außerdem verwenden wir bei der Auswertung des Schaltkreises einen Trick, der nach Steve Cook benannt ist.
Bei der Auswertung des Schaltkreises gehen wir wieder rekursiv vor. Anstatt von jedem Knoten auf dem gerade betrachteten Pfad die Adresse zu behalten, wird nur die Adresse des aktuellen Knotens gespeichert. Da der Fan-In jedes Knoten in $\mathcal{C}$ beschränkt ist, kann ein Weg in $\mathcal{C}$ vom Ausgabeknoten zu den Eingabeknoten dadurch gespeichert werden, daß an jedem Knoten des Weges nur die Richtung des Weges, die Art des Knotens und der bisher errechnete Wert gespeichert wird. Dies benötigt für jeden Knoten nur konstanten Speicheraufwand. Durch diesen Trick ist eine Auswertung des Schaltkreises in $O(\text{Depth}(\mathcal{C}) + \log(\text{Size}(\mathcal{C})))$ Speicher möglich. Der aktuelle Knoten wird nun durch die Berechnung des ganzen Schaltkreises bestimmt. Diese Berechnung kann jedoch aufgrund der Uniformität in $O(\log n)$ Speicher geschehen. Insgesamt gilt also für den benutzten Speicher:

$$O(\text{Depth}(\mathcal{C}) + \log(\text{Size}(\mathcal{C})) + \log n) = O(\text{Depth}(\mathcal{C})),$$

da nach Voraussetzung $\text{Depth}(\mathcal{C}) = S(n) \geq \log n$ gilt. Also kann der uniforme Schaltkreis in $S(n)$ Speicher simuliert werden. □

1.6 Randomisierte Maschinenmodelle und Algorithmen

In diesem Kapitel beschäftigen wir uns mit den Grundlagen für randomisierte Berechnungen. Dabei hängt die Berechnung und damit unter Umständen auch das Ergebnis der Berechnung von zufälligen Ereignissen ab. Solche Ereignisse werden durch Zufallszahlengeneratoren erzeugt.

Definition 1.6.1 Idealer Münzwurf
Unter einem idealen Münzwurf verstehen wir eine gleichverteilte Zufallsgröße X mit Wertebereich $\{0,1\}$. Die Werte einer Folge von Auswertungen der Zufallsgröße X seien **gleichverteilt** und voneinander **unabhängig**.

Ein für uns gültiger Zufallszahlengenerator erzeugt einen idealen Münzwurf. Nun können wir unsere bisherigen Modelle geeignet modifizieren, indem wir die Verwendung von Zufallsinformation erlauben, die durch einen gültigen Zufallsgenerator erzeugt wird.
Im folgenden werden die elementaren Definitionen und Rechenregeln mit Zufallsgrößen vorausgesetzt.

1.6.1 Probabilistische Turing Maschinen

Definition 1.6.2 Probabilistische Turing Maschine (PTM)
Eine probabilistische TM ist eine Offline TM im bisherigen Sinne mit den folgenden Modifikationen:

- Es gibt ein zusätzliches Band, welches eine Folge von **Zufallsbits** $\in \{0,1\}^*$ enthält, die von einem gültigen Zufallszahlengenerator erzeugt worden sind. Der Kopf des Zufallsbandes darf nur **in eine Richtung** bewegt werden. Somit kann eine Zufallsinformation **nur einmal** verwendet werden.
- Die Zustandsüberführungsfunktion hängt auch von dem Inhalt der Zelle unter dem Lesekopf des Zufallsbandes ab. Sie bleibt aber deterministisch.
- Eine Berechnung ist nur **gültig**, falls die Länge der Zufallsfolge mindestens so groß ist wie die Länge der Berechnung.

Abbildung 1.5 zeigt eine probabilistische Turing Maschine.

Bemerkung 1.6.3 Man kann eine probabilistische TM als eine deterministische TM ansehen, falls man das Zufallsband als ein zusätzliches Eingabeband interpretiert.

Definition 1.6.4 Sei M eine PTM über dem Alphabet Σ. Dann induziert M eine partielle Funktion

$$\Phi_M : \Sigma^* \times \underbrace{\{0,1\}^*}_{\text{Zufallsbits}} \to \Sigma^*.$$

Diese Funktion ist nicht total, da die Maschine nicht bei jeder Eingabe anhalten muß. So ist $\Phi_M(x,\rho)$ undefiniert, falls die Berechnung länger ist als $|\rho|$.

Wir betrachten eine PTM nicht als deterministische TM, sondern gehen davon aus, daß der Inhalt des Zufallsbandes nicht bekannt ist. Durch die Hinzunahme einer weiteren Resource — dem **Zufall** — ergibt sich die Notwendigkeit den Berechenbarkeitsbegriff für dieses neue Modell zu definieren.

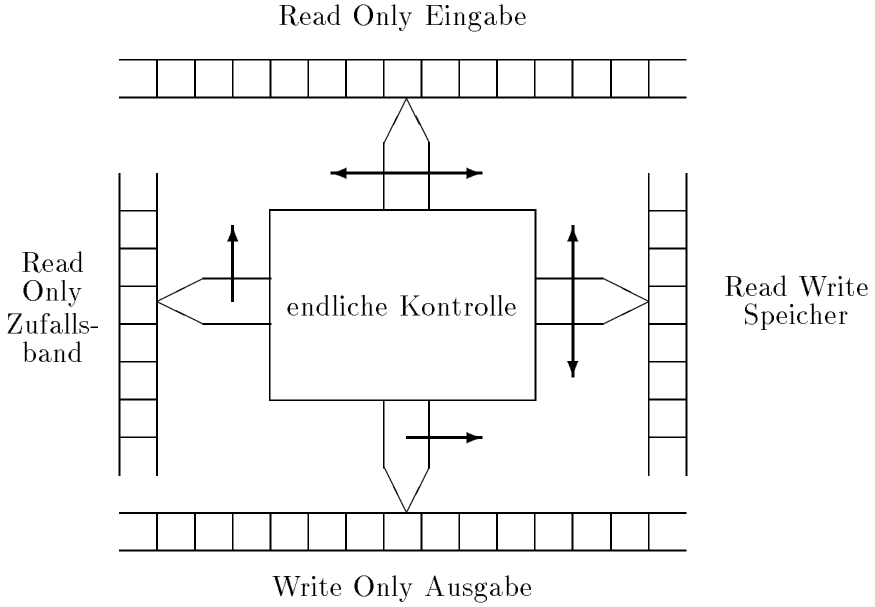

Abbildung 1.5: Probabilistische Turing Maschine

Definition 1.6.5 Berechenbarkeitsbegriff für PTM
Sei M eine PTM. M berechnet eine partielle Funktion

$$\phi_M : \Sigma^* \to \Sigma^*$$

mit

$$\phi_M(x) = \begin{cases} y & \text{wenn } Pr\{M(x) = y\} > 1/2 \\ \text{undefiniert} & \text{wenn es kein solches } y \text{ gibt.} \end{cases}$$

Dabei ist

$$Pr\{M(x) = y\} = \lim_{n\to\infty} \frac{|\{\rho \in \{0,1\}^n | \Phi_M(x,\rho) = y\}|}{2^n}$$

Der Definitionsbereich von M sei mit $\mathcal{D}(\phi_M)$ bezeichnet.

Definition 1.6.6 probabilistisch berechenbar
Eine Funktion

$$f : \Sigma^* \to \Sigma^*$$

ist probabilistisch berechenbar ($f \in$ PBF), wenn es eine PTM M mit $f \equiv \phi_M$ gibt.

Lemma 1.6.7 Jede probabilistisch berechenbare Funktion f ist berechenbar, d.h. f ist auch mit einer deterministischen Turing Maschine berechenbar. Anders ausgedrückt: Die **probabilistisch berechenbaren** Funktionen sind genau die **partiell rekursiven** Funktionen.

Beweis: Sei M eine PTM. Für jedes n und y können mit einer deterministischen TM M' die Werte

$$\alpha_{n,y} := Pr\{M(x) \text{ hält in } n \text{ Schritten mit Ausgabe } y\},$$

berechnet werden. Dies geschieht dadurch, daß für jede mögliche Zufallsfolge, die bei der probabilistischen Berechnung von M auftreten kann, die zugehörige deterministische Berechnung durchgeführt wird. Nach n Schritten dieser Simulation wird überprüft, ob und mit welchem Ergebnis die Maschine ihre Berechnung beendet hat. Somit berechnet die TM M' einen Baum mit 2^n Blättern. Falls m dieser Berechnungen zum Ergebnis y führen, so gilt

$$\alpha_{n,y} = m \cdot 2^{-n}.$$

Da

$$Pr\{M(x) = y\} = \sum_{i=0}^{\infty} Pr\{M(x) = y \text{ in genau } i \text{ Schritten}\} > 1/2$$

gilt, gibt es ein N, so daß für alle $n > N$

$$\sum_{i=0}^{n} Pr\{M(x) = y \text{ in genau } i \text{ Schritten}\} = Pr\{M(x) = y \text{ in Zeit } \leq n\} > 1/2$$

gilt, falls ein solches y mit $Pr\{M(x) = y\} > 1/2$ existiert. Somit ist auch

$$\alpha_{n,y} > 1/2 \quad \text{für alle } n > N$$

und die Simulation der PTM terminiert in diesem Fall. □

Bemerkung 1.6.8 Wie man sieht ist die Simulation einer PTM durch eine deterministische TM nicht effizient, da der Zeitaufwand exponentiell steigt. Dies gibt uns Hoffnung durch die Benutzung von Randomisierung schnellere Algorithmen zu bekommen.

Definition 1.6.9 Fehlerwahrscheinlichkeit
Sei M eine PTM. Die **Fehlerwahrscheinlichkeit** (engl. error probability) von M ist eine Funktion

$$e_M(x) = \begin{cases} Pr\{M(x) \neq \phi_M(x)\} & \text{wenn } \phi_M(x) \text{ definiert} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

M berechnet ϕ_M mit beschränkter Fehlerwahrscheinlichkeit (engl. bounded error probability), wenn es ein $\epsilon > 0$ gibt, so daß

$$\forall x \in \mathcal{D}(\phi_M) : \; e_M(x) \leq \frac{1}{2} - \epsilon.$$

Definition 1.6.10 Monte-Carlo PTM
Eine PTM M, die ϕ_M mit beschränkter Fehlerwahrscheinlichkeit berechnet, nennt man **randomisierte Turing Maschine** (RTM) oder **Monte-Carlo TM**.

Lemma 1.6.11 Zu jeder Sprache A, die von einer RTM M erkannt werden kann, gibt es für jedes $\delta > 0$ eine RTM M_δ, die A mit Fehlerwahrscheinlichkeit $e_{M_\delta} \leq \delta$ erkennt. Dabei erhöht sich die Anzahl der Schritte nur um einen Faktor, der allein von δ abhängt.

Beweis: Die RTM M_δ wiederholt den Lauf der RTM M mehrmals. Dabei gibt sie das Ergebnis aus, das am häufigsten erzielt wurde. Wir wollen nun die Fehlerwahrscheinlichkeit hierfür abschätzen. Nach der Tschebyscheffschen Ungleichung zweiter Art gilt allgemein

$$Pr\{|X - E(X)| \geq t\} \leq \mathrm{Var}(X)/t^2.$$

Wir müssen also den Erwartungswert und die Varianz für unsere Zufallsvariable $M' = \sum_{i=1}^{k} M_i$ berechnen, wobei M_i die Zufallsvariable für den i-ten Lauf ist. Da die Ereignisse der verschiedenen Läufe voneinander unabhängig sind, gilt die Additivität der Varianz und die Additivität des Erwartungswertes. Wir müssen also nur den Erwartungswert für einen Lauf i berechnen. Sei nun $x \in L(M)$ (für $x \notin L(M)$ analog).

$$E(M_i(x)) = 1 \cdot (1/2 + \epsilon) + 0 \cdot (1/2 - \epsilon) = 1/2 + \epsilon$$

$$\mathrm{Var}(M_i(x)) = (1 - (1/2 + \epsilon))^2 \cdot (1/2 + \epsilon) + (0 - (1/2 + \epsilon))^2 \cdot (1/2 - \epsilon) = (1/4 - \epsilon^2)$$

Damit ergibt sich für die Varianz bzw. den Erwartungswert von M'

$$E(M'(x)) = k(1/2 + \epsilon) \quad \text{und} \quad \mathrm{Var}(M'(x)) = k(1/4 - \epsilon^2).$$

Nach der Tschebyscheffschen Ungleichung folgt nun

$$Pr\{M'(x) \leq k/2\} \leq Pr\{|M'(x) - E(M'(x))| \geq k\epsilon\} \leq \frac{k(1/4 - \epsilon^2)}{k^2\epsilon^2} = O(1/k).$$

Daher kann die Wahrscheinlichkeit durch Iterieren beliebig verbessert werden. □

Bemerkung 1.6.12 Unter Verwendung von schärferen Ungleichungen kann man sogar zeigen, daß die Fehlerwahrscheinlichkeit exponentiell (in k) klein ist.

Mit Hilfe dieses Lemmas können wir die **Monte-Carlo TM** auch dadurch charakterisieren, daß sie eine **Fehlerwahrscheinlichkeit** $e_M(x) < 1/4$ besitzen.

Auf der anderen Seite reicht es auch aus, fehlerbeschränkte Turing-Maschinen dadurch zu kennzeichnen, daß die Fehlerwahrscheinlichkeit kleiner ist als $1/2 - 1/p(n)$ für ein festes Polynom $p(n)$. Durch eine polynomielle Anzahl von Iterationen kann auch hier die Fehlerwahrscheinlichkeit beliebig klein gehalten werden.

Definition 1.6.13 Eine Monte-Carlo PTM M heißt starke (strenge) Monte-Carlo PTM (R_STM), wenn folgendes gilt:

1. $e_M(x) = 0$ für $x \notin L(M)$,
2. $e_M(x) < 1/4$ für $x \in L(M)$.

Bemerkung 1.6.14 Für starke Monte-Carlo PTM's reicht es aus, die Fehlerwahrscheinlichkeit für $x \in L(M)$ durch ein beliebiges $\epsilon < 1$ zu beschränken. Durch mehrfache Iteration wird dann eine beliebig geringe Fehlerwahrscheinlichkeit erzielt.

Damit haben wir geklärt, was eine Berechnung einer probabilistischen Turing Maschine ist. Nun wollen wir wieder Komplexitätsklassen einführen. Hierzu muß jedoch erst die Laufzeit und der Speicherverbrauch einer probabilistischen Turing Maschine definiert werden. Für die Laufzeitdefinition haben wir zwei Möglichkeiten.

Definition 1.6.15 Laufzeit einer PTM (nach [Bl 67])
Die **probabilistische Laufzeit** T_M einer PTM M ist definiert durch:

$$T_M(x) = \begin{cases} \min\{n \mid Pr\{M(x) = \phi_M(x) \text{ in } n \text{ Schritten}\} > 1/2\} & \text{falls } \phi_M(x) \text{ definiert} \\ \infty & \text{falls } \phi_M(x) \text{ undefiniert.} \end{cases}$$

Definition 1.6.16 Mittellaufzeit einer PTM
Die **probabilistische Mittellaufzeit** (engl. average run time) $\bar{T}_M$ einer PTM M ist der **Erwartungswert** der Schrittanzahl:

$$\bar{T}_M(x) = \sum_{n=1}^{\infty} n \cdot Pr\{\text{Berechnung über } x \text{ benötigt genau } n \text{ Schritte}\}.$$

Die Mittellaufzeit ist eine ungeeignete Definition, wenn wir die Klasse der PTM's betrachten. Das anormale Verhalten wird im folgenden Lemma sichtbar.

Lemma 1.6.17 Jede Turing berechenbare Funktion ist probabilistisch in endlicher Mittellaufzeit berechenbar.

Beweis: Sei M eine TM und $A = L(M)$. Die Simulation durch die PTM M' sieht folgendermaßen aus.

```
Repeat
   Simuliere einen Schritt von M
   Falls M akzeptiert, so akzeptiere
Until CoinToss = Heads
Falls CoinToss = Heads
   so akzeptiere
   sonst verwerfe
```

Es sei dabei CoinToss eine Zufallsvariable mit Werten in { Heads, Number }.

Wir müssen zeigen, daß diese Simulation korrekt ist. Dazu betrachten wir zwei Fälle.

$x \notin A$ Falls $x \notin A$, so kann x nicht in der Schleife akzeptiert werden. Daher ist die Wahrscheinlichkeit, daß x von M' akzeptiert wird genau gleich $1/2$. Damit ist $x \notin L(M')$.

$x \in A$ Falls $x \in A$, so kann x auch in der Schleife akzeptiert werden. Die Wahrscheinlichkeit hierfür ist $2^{-T_M(x)} > 0$. Da x auch außerhalb der Schleife mit Wahrscheinlichkeit $1/2 \cdot (1 - 2^{-T_M(x)})$ erkannt wird, ergibt sich eine Gesamtwahrscheinlichkeit von

$$1/2 - 2^{-T_M(x)-1} + 2^{-T_M(x)} = 1/2 + 2^{-T_M(x)-1} > 1/2.$$

Daher gilt $x \in L(M')$.

Mit Hilfe des Berechnungsbaum in Abbildung 1.6 kann die probabilistische Mittellaufzeit von M' folgendermaßen abgeschätzt werden. Für $x \notin A$ gilt

$$\bar{T}_{M'}(x) = \sum_{n=2}^{\infty} (n+1)/2^n + 1,$$

für $x \in A$ ist dies eine obere Schranke. Es läßt sich leicht zeigen, daß $\bar{T}_{M'} < c$ für eine Konstante c ist. Daher gilt die Behauptung. □

Definition 1.6.18 Blum'sches Komplexitätsmaß ([Bl 67])
Eine Funktion T_M ist ein Komplexitätsmaß, falls T_M die folgenden beiden Axiome erfüllt sind.

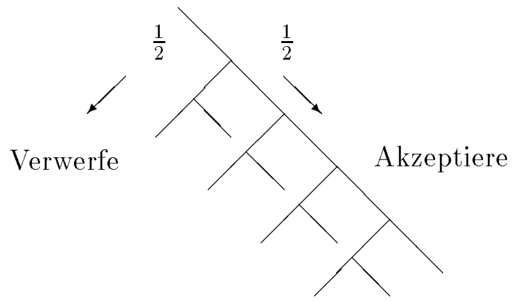

Abbildung 1.6: Berechnungsbaum

- $T_M(x)$ ist genau dann definiert, wenn $M(x)$ definiert ist,
- Das Prädikat $T_M(x) = n$ ist entscheidbar.

Die probabilistische Laufzeit T_M ist ein Komplexitätsmaß für die ganze Klasse der PTM's. Die Mittellaufzeit ist jedoch kein Komplexitätsmaß für diese Klasse wie der folgende Satz beweist.

Satz 1.6.19 $\bar{T}_M$ ist kein Komplexitätsmaß für die Klasse der probabilistischen Turing Maschinen.

Beweis: Wir zeigen, daß $\bar{T}_M$ das zweite Axiom nicht erfüllt. Dazu sei M_1 eine (deterministische) Turing Maschine, die eine nicht entscheidbare Menge erkennt. Wir simulieren M_1 durch M auf die folgende Weise:

```
Repeat
   Simuliere einen Schritt von M1
   Falls M1 akzeptiert, so gehe in eine Endlosschleife
Until CoinToss = Heads
Falls CoinToss = Heads
   so akzeptiere
   sonst verwerfe
```

M hat jetzt die Eigenschaft, daß $\bar{T}_M(x) = \infty$ für $x \in L(M_1)$ und $\bar{T}_M(x) < c$ für $x \notin L(M_1)$. Wäre $\bar{T}_M$ ein Komplexitätsmaß, so wäre $L(M_1)$ entscheidbar im Gegensatz zur Annahme. □

Der nachfolgende Satz zeigt, daß für die Klasse der randomisierten Turing Maschinen, d.h. derjenigen probabilistischen Turing Maschinen, die eine beschränkte Fehlerwahrscheinlichkeit besitzen, die beiden Laufzeitdefinitionen jedoch äquivalent sind. Daher ist die Mittellaufzeit in diesem Fall auch ein Komplexitätsmaß.

Satz 1.6.20 Sei M eine Monte-Carlo TM (RTM). Dann gibt es ein c, so daß

$$T_M(x) \leq c \cdot \bar{T}_M(x) \quad \text{falls } \phi_M \text{ definiert ist.}$$

Beweis: Da M eine Monte-Carlo TM ist, hat M eine mit $\epsilon < 1/2$ beschränkte Fehlerwahrscheinlichkeit. Sei $c = 1/(1/2 - \epsilon)$. Wenn $\bar{T}_M(x) = \infty$ ist, gibt es nichts zu beweisen. Sei also

$\bar{T}_M(x) < \infty$. Dann gilt nach der Tschebyscheff'schen Ungleichung 1.Art

$$Pr\{\text{Zeit von } M(x) > c\bar{T}_M(x)\} < \bar{T}_M(x)/c\bar{T}_M(x) = 1/c = 1/2 - \epsilon.$$

Diese Gleichung ist äquivalent zu

$$Pr\{\text{Zeit von } M(x) \leq c\bar{T}_M(x)\} > 1/2 + \epsilon.$$

Da die Fehlerwahrscheinlichkeit von M beschränkt ist, gilt

$$Pr\{M(x) \neq \phi_M(x) \text{ in Zeit } c\bar{T}_M(x)\} \leq Pr\{M(x) \neq \phi_M(x)\} < \epsilon.$$

Somit folgt

$$Pr\{M(x) = \phi_M(x) \text{ in Zeit } c\bar{T}_M(x)\} > 1/2.$$

Also ist

$$T_M(x) \leq c\bar{T}_M(x).$$

□

Nun wollen wir mit Hilfe der neuen Maschinenmodelle auch neue Komplexitätsklassen einführen.

Definition 1.6.21 Probabilistische Komplexitätsklassen

$$\text{PrTIME}(T(n)) := \{A|\exists\text{PTM } M \text{ mit } L(M) = A \text{ und } T_M(n) = O(T(n))\}$$

$$\text{RTIME}(T(n)) := \{A|\exists\text{RTM } M \text{ mit } L(M) = A \text{ und } T_M(n) = O(T(n))\}$$

$$\text{R}_S\text{TIME}(T(n)) := \{A|\exists\text{R}_S\text{TM } M \text{ mit } L(M) = A \text{ und } T_M(n) = O(T(n))\}$$

$$\mathcal{PP} := \{A|\exists\text{PTM } M \text{ mit } L(M) = A \text{ und } T_M(n) = n^{O(1)})\}$$

$$\mathcal{BPP} := \{A|\exists\text{RTM } M \text{ mit } L(M) = A \text{ und } T_M(n) = n^{O(1)})\}$$

Eine weitere Klasse ist Δ^P. Man nennt sie Las-Vegas Klasse. Eine PTM, die einen Las-Vegas Algorithmus darstellt, hat als Ausgabe 0, 1 und ?. Das Fragezeichen bedeutet, daß die PTM zu keinem korrekten Ergebnis gekommen ist. Die Wahrscheinlichkeit für ? ist beschränkt.

Für diese Komplexitätsklassen gelten nun die folgenden Inklusionen

Satz 1.6.22 Es gilt

$$\mathcal{P} \subseteq \Delta^P \begin{array}{c} \subseteq \mathcal{BPP} \subseteq \\ \subseteq \mathcal{NP} \subseteq \end{array} \mathcal{PP}.$$

Beweis: Die obere Inklusionskette ist klar, da die Modelle, mit Hilfe derer die Klassen definiert sind, Spezialfälle bzw. Verallgemeinerungen sind. Es müssen also noch die Inklusionen

$$\Delta^P \subseteq \mathcal{NP} \subseteq \mathcal{PP}$$

gezeigt werden.

Dazu sei M eine Δ^P TM. Falls $x \in L(M)$ ist, so endet jede Berechnung mit der Ausgabe 1 oder ?, falls $x \notin L(M)$ ist, so erfolgt die Ausgabe 0 oder ?. Dabei endet wenigstens ein

Berechnungszweig mit einer Ausgabe ungleich ?. M wird jetzt durch die NTM M' simuliert, indem die Aktionen, die von dem Zufallsband abhängen, nichtdeterministisch gemacht werden und die Ausgabe ? durch die Ausgabe 0 ersetzt wird. Damit gilt: $L(M) = L(M')$.
Sei jetzt M eine nichtdeterministische TM. Wir simulieren diese mit Hilfe der PTM M' analog zum Beweis von Lemma 1.6.17, wobei auch die Auswahl der nächsten Aktion von M probabilistisch geschieht. Analog zum Beweis von Lemma 1.6.17 kann man auch hier zeigen, daß $L(M) = L(M')$ gilt. □

Nachdem nun der Laufzeitbegriff für probabilistische Turing Maschinen geklärt ist, kommen wir zum Speicherverbrauch einer probabilistischen Turing Maschine.

Definition 1.6.23 Speicherverbrauch
Sei M eine probabilistische Turing Maschine. Dann ist der **Speicherverbrauch** von M

$$S_M(x) = \begin{cases} \min\{n | Pr\{M(x) = \phi_M(x) \text{ in Speicher } n\ \} > 1/2\} & \text{falls } x \in \mathcal{D}(\phi_M) \\ \infty & \text{sonst.} \end{cases}$$

Nun können analog zu Definition 1.6.21 auch Klassen mit Hilfe des Speicherverbrauches definiert werden.

Definition 1.6.24 weitere Komplexitätsklassen

$$\text{PrSPACE}(S(n)) := \{A | \exists \text{PTM } M \text{ mit } L(M) = A \text{ und } S_M(n) = O(S(n))\}$$

$$\text{RSPACE}(S(n)) := \{A | \exists \text{RTM } M \text{ mit } L(M) = A \text{ und } S_M(n) = O(S(n))\}$$

$$\text{R}_S\text{SPACE}(S(n)) := \{A | \exists \text{R}_S\text{TM } M \text{ mit } L(M) = A \text{ und } S_M(n) = O(S(n))\}$$

Damit gilt der folgende

Satz 1.6.25 Sei $S(n) \geq \log n$. Dann gilt

$$\text{PrSPACE}(S(n)) \supseteq \text{NSPACE}(S(n)).$$

Beweis: siehe [Gi 77].

1.6.2 Randomisierte Schaltkreise

Nun betrachten wir randomisierte Schaltkreise. Diese Schaltkreise sind Schaltkreise mit den weiteren folgenden Eigenschaften.

Definition 1.6.26 Probabilistische Schaltkreise
Ein probabilistischer Schaltkreis $\mathcal{C}$ ist die Realisierung einer booleschen Funktion

$$f' : I\!B^n \times I\!B^{\rho(n)} \to I\!B$$

$$f' : (x, \rho) \mapsto f'(x, \rho).$$

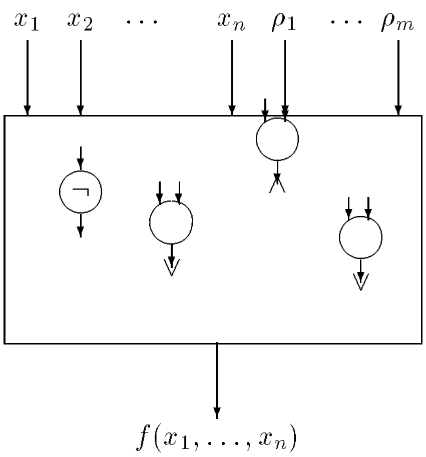

Abbildung 1.7: Probabilistischer Schaltkreis

Diese Funktion induziert eine Funktion $f : I\!B^n \to I\!B$ mit

$$f(x) = \begin{cases} 1 & Pr\{f'(x,\rho) = 1\} > 1/2 \\ 0 & Pr\{f'(x,\rho) = 0\} > 1/2 \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Ein solcher Schaltkreis ist in Abbildung 1.7 dargestellt.

Definition 1.6.27 Randomisierter Schaltkreis
Ein probabilistischer Schaltkreis $\mathcal{C}$ ist ein **randomisierter Schaltkreis**, falls eine Funktion g existiert mit

$$Pr\{f'(x,\rho) = g(x)\} > 3/4.$$

Mit Hilfe von Lemma 1.6.11 reicht es auch aus

$$Pr\{f'(x,\rho) = g(x)\} > 1/2 + 1/p(n) \qquad (*)$$

zu fordern. Ein Schaltkreis mit Fehlerwahrscheinlichkeit $< 1/4$ besteht dann aus einer polynomiellen Anzahl von Schaltkreisen, die $(*)$ erfüllen und parallel arbeiten. Die Ausgabe ist dann ein Mehrheitsentscheid wie nach Lemma 1.6.11.

Nun interessieren uns jedoch nur uniforme Schaltkreise, d.h. Schaltkreise, die log-space Turing berechenbar sind. Dies definiert die RUC Schaltkreisfamilien.

Definition 1.6.28 Randomisierte uniforme Schaltkreise
Ein randomisierte Schaltkreisfamilie $\{\overline{\mathcal{C}}_n\}$ ist uniform, falls die Compilerabbildung $1^n \to \mathcal{C}_n$ log-space Turing berechenbar ist, wobei n die Anzahl der Eingänge ohne die Eingänge der Zufallsbits ist. Eine solche Schaltkreisfamilie nennen wir dann RUC-Algorithmus.

Dadurch können wir analog zu den NC-Klassen auch hier Komplexitätsklassen einführen.

Definition 1.6.29 Sei $f : I\!N \to I\!N$. Damit sei

$$\begin{aligned}\text{U-RDEPTH}(f(n)) := \ & \{A \mid \exists \text{RUC-Algorithmus}\{\mathcal{C}_n\} \text{ mit} \\ & L(\{\mathcal{C}_n\}) = A \text{ und } \text{Depth}(\mathcal{C}_n) = O(f(n))\}\end{aligned}$$

$$\begin{aligned}\text{U-RSIZE}(f(n)) := \ & \{A \mid \exists \text{RUC-Algorithmus}\{\mathcal{C}_n\} \text{ mit} \\ & L(\{\mathcal{C}_n\}) = A \text{ und } \text{Size}(\mathcal{C}_n) = O(f(n))\}\end{aligned}$$

$$\begin{aligned}\text{RNC}^k := \ & \{A \mid \exists \text{RUC-Algorithmus}\{\mathcal{C}_n\} \text{ mit } L(\{\mathcal{C}_n\}) = A \\ & \text{und Size}(\mathcal{C}_n) = n^{O(1)} \text{ und Depth}(\mathcal{C}_n) = O(\log^k n)\}\end{aligned}$$

$$\text{RNC} = \bigcup_{k \in I\!N} \text{RNC}^k.$$

Man beachte, daß bei randomisierten Schaltkreisen die Zufallsbits während der Berechnung öfters benutzt werden können. Bei randomisierten Turing Maschinen ist dies nur möglich, falls die benutzten Zufallbits auf dem Speicherband zwischengespeichert werden. Das Berechnungsmodell der randomisierten Schaltkreise kann daher eher mit randomisierten Turing Maschinen verglichen werden, die eine Kopfbewegung über das Randomband in beide Richtungen erlauben. Solche Maschinen nennt man Two-Way Random Tape Turing Maschinen. Für diese Maschinen gilt der folgende interessante Satz.

Satz 1.6.30 ([KV 85])
Eine Two-Way Random Tape Turing Maschine sei mit $\text{R}^{(2)}\text{TM}$ bezeichnet. Damit gilt

$$\text{R}^{(2)}\text{SPACE}(\log n) = \text{PSPACE}.$$

1.6.3 Probabilistische Testalgorithmen

In diesem Kapitel werden wir einige Testalgorithmen vorstellen, mit deren Hilfe algebraische Gleichheiten überprüft werden können. Diese Algorithmen sind probabilistisch und haben den gleichen einfachen Aufbau: Falls eine algebraische Gleichung nicht allgemeingültig erfüllt ist, so gibt es nur *relativ wenige* Belegungen der Variablen, die diese Gleichung zufällig erfüllen. Ist diese algebraische Gleichung z.B. eine Polynomgleichung, so sind diese ungünstigen Belegungen der Variablen gerade die Nullstellen des Polynoms.

Unser erster Algorithmus testet, ob ein gegebenes **multivariates Polynom** identisch dem Nullpolynom ist. Dabei ist das Polynom natürlich nicht bekannt, sondern durch eine Black-Box gegeben, d.h. es sind Auswertungen an beliebigen Stellen möglich.

Algorithmus 1.6.31 Nulltest für Polynome

Eingabe: Eine Black-Box für ein Polynom $P(x_1, \ldots, x_n)$ über $Z\!\!Z$ und eine Schranke D für den Grad des Polynoms.

Schritt 1: Konstruiere die Menge $E = \{1, 2, \ldots, 4D\}$

Schritt 2: Wähle zufälliges $\bar{x} = (x_1, \ldots, x_n) \in E^n$

Ausgabe: $\begin{cases} P \equiv 0 & \text{falls } P(\bar{x}) = 0 \\ P \not\equiv 0 & \text{sonst} \end{cases}$

Satz 1.6.32 (Schwartzes Lemma [Sc 80])
Sei $P = P_1(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$ und $P \not\equiv 0$. Sei d_1 der Grad von x_1 in P_1 und P_2 das Koeffizientenpolynom von $x_1^{d_1}$:

$$P_1(x_1, \ldots, x_n) = x_1^{d_1} \cdot P_2(x_2, \ldots, x_n) + q_1(x_1, \ldots, x_n).$$

Die P_i und d_i für $i > 1$ seien entsprechend induktiv definiert. Dann hat P höchstens

$$|I_1 \times \cdots \times I_n| \left(\frac{d_1}{|I_1|} + \cdots + \frac{d_n}{|I_n|}\right)$$

Nullstellen in $I_1 \times \cdots \times I_n$.

Beweis: Wir führen den Beweis über Induktion nach der Anzahl der Variablen.

- Für den Fall eines univariaten Polynoms P gilt, daß die Anzahl der Nullstellen beschränkt ist durch
$$\deg(P) = d_1 = |I_1| \frac{d_1}{|I_1|}.$$

- Induktionsschritt:
  P_1 hat die Darstellung
$$P_1(x_1, \ldots, x_n) = x_1^{d_1} \cdot P_2(x_2, \ldots, x_n) + q_1(x_1, \ldots, x_n).$$
  Nun gibt es zwei Möglichkeiten:
  1. $(z_2, \ldots, z_n)$ ist Nullstelle von P_2. Dann kann eventuell $P(x_1, z_2, \ldots, z_n) = 0$ für alle $x_1 \in I_1$ gelten. Da die Anzahl der Nullstellen von P_2 nach Induktionsannahme durch
$$|I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|}\right)$$
     beschränkt ist, kann die Anzahl der Nullstellen von P für diesen Fall durch
$$|I_1| \cdot |I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|}\right)$$
     abgeschätzt werden.
  2. $(z_2, \ldots, z_n)$ ist keine Nullstelle von P_2. Dann ist $P(x_1, z_2, \ldots, z_n)$ ein nicht verschwindendes Polynom in x_1 vom Grade d_1. Damit ist eine Schranke für die Anzahl der Nullstellen von P in diesem Fall
$$d_1 \cdot |I_2 \times \cdots \times I_n|.$$

  Damit kann die Gesamtanzahl von Nullstellen von P durch
$$|I_1| \cdot |I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|}\right) + d_1 |I_2 \times \cdots \times I_n|$$
$$= |I_1 \times \cdots \times I_n| \left(\frac{d_1}{|I_1|} + \ldots + \frac{d_n}{|I_n|}\right)$$
  beschränkt werden. □

Korollar 1.6.33 Sei $P = P(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$, $P \not\equiv 0$ und $I = I_1 = \cdots = I_n$ mit $|I| \geq c \cdot \deg(P)$. Dann ist die Anzahl der Nullstellen in I^n durch $|I|^n/c$ beschränkt.

BEWEIS: Wir wenden den Satz 1.6.32 an. In diesem Fall gilt

$$\begin{aligned} |I|^n \cdot \textstyle\sum_{j=1}^n d_j/|I| &= |I|^{n-1} \cdot (\textstyle\sum_{j=1}^n d_j) \\ &\leq |I|^{n-1} \cdot \deg(P) \leq |I|^n/c \end{aligned}$$

und somit ist die Anzahl der Nullstellen durch $|I|^n/c$ beschränkt. □

Satz 1.6.34 Korrektheit von Algorithmus 1.6.31
Der Algorithmus 1.6.31 ist ein korrekter starker Monte-Carlo Algorithmus.

BEWEIS: Mit Hilfe von Lemma 1.6.33 gilt, daß die Anzahl der Nullstellen des Polynoms P durch $|E|^n/4$ beschränkt ist. Für einen zufällig gewählten Vektor $\bar{x}$ aus E^n und $P \not\equiv 0$ gilt damit

$$Pr\{P(\bar{x}) = 0\} \leq \frac{|E|^n/4}{|E^n|} = 1/4.$$

Damit ist die Ausgabe $P \equiv 0$ mit Wahrscheinlichkeit $\geq 3/4$ korrekt. Die Ausgabe $P \not\equiv 0$ ist immer korrekt, da für $P \equiv 0$ kein $\bar{x}$ mit $P(\bar{x}) \neq 0$ existiert. □

Als nächstes wenden wir uns dem Problem der **Matrizenmultiplikation** zu. Dabei ist zu entscheiden, ob das Produkt zweier Matrizen eine dritte gegebene Matrix ergibt. Ein möglicher deterministischer Algorithmus zur Lösung dieses Problems besteht darin, die Matrizen A und B miteinander zu multiplizieren und das Ergebnis mit C zu vergleichen. Dies liefert auf naive Weise einen Algorithmus, der in TIME(n^3) und in $NC^1(n^3)$ liegt. Mit Hilfe der neuesten Algorithmen im Bereich der Matrizenmultiplikation — aufbauend auf den Ideen von Strassen — ist dies sogar in TIME($n^{2.376}$) und $NC^1(n^{2.376})$ möglich. Eine untere Schranken für einen solchen Algorithmus wäre in jedem Fall $\Omega(n^2)$.
Der nachfolgende Algorithmus ist ein optimaler R_STM Algorithmus zur Lösung dieses Problems. Er liegt in R_STIME(n^2) bzw. in $RNC^1(n^2)$.

Algorithmus 1.6.35 Test für Matrizenmultiplikation

Eingabe: Reellwertige $n \times n$ Matrizen A,B und C.

Schritt 1: Erzeuge zufällig 2 Vektoren $X_1, X_2 \in \{-1, 1\}^n$:

$$X_1 = (x_{11}, \ldots, x_{1n}), \quad X_2 = (x_{21}, \ldots, x_{2n}).$$

Schritt 2: Berechne: $A(BX_1), A(BX_2), CX_1$ und CX_2.

Ausgabe: $\begin{cases} 1 & \text{falls } ABX_1 = CX_1 \text{ und } ABX_2 = CX_2 \\ 0 & \text{sonst} \end{cases}$

Satz 1.6.36 Der Algorithmus 1.6.35 arbeitet korrekt und liegt in R_STIME(n^2) bzw. in $RNC^1(n^2)$.

Beweis: Sei $D = (d_{ij}) = A \cdot B$ und $C = (c_{ij})$. Sei nun $\bar{x} = (x_1, \ldots, x_n)$ mit $x_i \in \{-1, 1\}$. Falls nun für $\bar{x}$

$$(A \cdot B - C) \cdot \bar{x} = 0$$

gilt, ist dies äquivalent zu

$$\forall i \quad (d_{i1} - c_{i1}, \ldots, d_{in} - c_{in}) \perp (x_1, \ldots, x_n). \tag{1.1}$$

Falls $A \cdot B \neq C$, könnten wir bei unserer zufälligen Wahl der Vektoren X_1 und X_2 schlechte Vektoren bekommen haben, für die trotzdem die Orthogonalitätsbeziehung (1.1) gilt. Es können nach Lemma 1.6.37 jedoch höchstens 2^{n-1} der 2^n vielen $\{-1, 1\}$ Vektoren orthogonal zu den n Vektoren $(d_{i1} - c_{i1}, \ldots, d_{in} - c_{in})$ sein, da wenigstens einer dieser Vektoren ungleich dem Nullvektor ist. Somit ist die Wahrscheinlichkeit, zweimal einen schlechten Vektor zu wählen, kleiner als $1/4$.
Der Algorithmus ist ein starker Monte-Carlo Algorithmus, da die Ausgabe $\not\equiv$ mit Fehlerwahrscheinlichkeit 0 erfolgt. Der Algorithmus arbeitet auch innerhalb der geforderten Zeit- und Prozessorschranken, da nur Produkte zwischen Matrizen und Vektoren berechnet werden. □

Lemma 1.6.37 Sei $v \neq (0, \ldots 0)$. Dann sind höchstens 2^{n-1} viele $\{-1, 1\}$ Vektoren orthogonal zu v.

Beweis: Da $v = (v_1, \ldots, v_n)$ ungleich dem Nullvektor ist, ist wenigstens ein $v_{i_0} \neq 0$. Sei i_0 o.B.d.A gleich n, und seien $x = (x_1, \ldots, x_{n-1}, -1)$ und $y = (x_1, \ldots, x_{n-1}, 1)$. Somit gilt

$$< x, v >= \sum_{i=1}^{n-1} x_i \cdot v_i - v_n \neq < x, v > +2v_n =< y, v > .$$

Daher können nicht sowohl $< x, v >$ als auch $< y, v >$ gleich 0 sein. Die Zuordnung $x \mapsto y$ ordnet also einem zu v orthogonalen Vektor eineindeutig einen nicht orthogonalen Vektor zu. Somit können höchstens die Hälfte der 2^n Vektoren orthogonal zu v sein. □

Nun kommen wir zu dem Problem, zu testen, ob das Produkt zweier Polynome einem dritten Polynom entspricht.

Algorithmus 1.6.38 Test für Polynommultiplikation

Eingabe: P_1, P_2 und $P_3 \in \mathbb{Z}[x]$ mit $\deg(P_i) \leq n$.

Schritt 1: Wähle zufällig $\bar{x} \in \{-4n + 1, \ldots, 0, 1, \ldots, 4n\}$.

Ausgabe: $\begin{cases} 1 & P_1(\bar{x}) \cdot P_2(\bar{x}) = P_3(\bar{x}) \\ 0 & \text{sonst.} \end{cases}$

Satz 1.6.39 Der Algorithmus 1.6.38 ist ein korrekter starker Monte-Carlo Algorithmus.

Beweis: Das Polynom $P_1 \cdot P_2 - P_3$ hat höchstens den Grad $2n$. Damit hat dieses Polynom höchstens $2n$ Nullstellen in $\mathbb{Z}$ und damit höchstens $2n$ Nullstellen in $\{-4n + 1, \ldots, 4n\}$, d.h. nur $1/4$ der möglichen Werte sind Nullstellen. Daher ist die Fehlerwahrscheinlichkeit $\leq 1/4$. Der Algorithmus ist auch ein starker Monte-Carlo Algorithmus, da ein Polynom, das an einer Stelle ungleich 0 ist, nicht identisch dem Nullpolynom sein kann. □

1.6.4 Pseudo-Zufallszahlengeneratoren

Im den vorigen Abschnitten haben wir randomisierte Maschinenmodelle eingeführt und einige probabilistische Algorithmen kennengelernt. Dabei haben wir implizit vorausgesetzt, daß immer genügend Zufallsinformation zur Verfügung steht. Es ist jedoch teuer, wenn nicht sogar unmöglich, echte Zufallsinformation in kurzer Zeit zu beschaffen. Daher werden wir jetzt deterministische Zufallszahlengeneratoren betrachten, die wir im Zusammenhang mit starken Monte-Carlo Algorithmen verwenden werden.

Definition 1.6.40 Expandierender Pseudo-Zufallszahlengenerator
Eine Funktion $\rho_E : \{0,1\}^* \to \{0,1\}^*$ ist ein $E(m)$-expandierender PNG (Pseudorandom Number Generator) wenn

1. Für alle Eingabelängen m ist ρ_E eine Abbildung
$$\rho_E : \{0,1\}^m \to \{0,1\}^{E(m)}.$$
2. Die Funktion ρ_E ist für jedes m in Speicher m und in polynomieller Zeit berechenbar.
3. Wenn A in Zeit $T(n)$ auf einer R_STM M berechenbar ist, d.h. $A = L(M)$ gilt, dann muß für jedes $x \in A$ mit $|x| > n_0$ und für alle m mit $E(m) \geq T(|x|)$, ein Seed (Zeuge) $s \in \{0,1\}^m$ mit
$$\Phi_M(x, \rho_E(s)) = 1$$
existieren, d.h. s ist ein Zeuge dafür, daß x in A ist.

Es ist nicht geklärt, ob solche expandierenden Pseudo-Zufallsgeneratoren für jedes $E(m)$ existieren.

Definition 1.6.41 PNG Hypothese
Die **PNG Hypothese** besagt, daß für jedes Polynom $E(m)$ ein $E(m)$ expandierender Pseudo-Zufallsgenerator existiert.

Falls die PNG Hypothese gilt, so können wir starke Monte-Carlo Algorithmen deterministisch in subexponentieller Zeit simulieren.

Satz 1.6.42 Falls die PNG Hypothese gilt, so kann für jedes $\epsilon > 0$ jede R_STM M in der Zeit $O(2^{T(n)^\epsilon})$ durch eine deterministische TM simuliert werden.

Beweis: Sei M eine R_STM mit $A = L(M)$, die in Zeit $T(n)$ arbeitet. Wähle $E(m) = m^d$ so, daß der Grad d von $E(m)$ größer als $1/\epsilon$ ist und m möglichst klein, so daß $E(m) \geq T(n)$ gilt. Dann ist $m = O(T(n)^\epsilon)$. Nun überprüfe für alle $s \in \{0,1\}^m$, ob $\Phi_M(x, \rho_E(s)) = 1$ ist und akzeptiere, falls es ein solches s gibt. Diese Simulation ist in Zeit $O(2^m \cdot T(n)) = O(2^{T(n)^\epsilon})$ möglich. Falls $T(n) = n^{O(1)}$ ist, so ist die Simulation in $O(2^{n^\epsilon})$ möglich. □

Für den Speicherverbrauch gilt ein ähnlicher Satz.

Satz 1.6.43 Sei M eine polynomiell Zeit beschränkte R_STM mit $A = L(M)$. Sei die Funktion $\Phi_M : \Sigma^* \times \{0,1\}^* \to \Sigma^*$ in $\bigcap_{\epsilon>0} \mathrm{DSPACE}(n^\epsilon)$ berechenbar. Dann folgt unter der PNG Hypothese, daß

$$A \in \bigcap_{\epsilon>0} \mathrm{DSPACE}(n^\epsilon).$$

BEWEIS: Sei $A \in \mathrm{R}_S\mathrm{TIME}(n^k)$. Für $\epsilon > 0$ definiere $\alpha \geq k/\epsilon$ und nehme n^α expandierenden PNG. Nun können für jedes $s \in \{0,1\}^{n^\epsilon}$ sowohl $\rho_{n^\alpha}(s)$ und gleichzeitig auch $\Phi_M(x, \rho_{n^\alpha}(s))$ in $\mathrm{DSPACE}(n^\epsilon)$ berechnet werden. Dabei wird die berechnete Zufallsfolge sofort für die Berechnung von $\Phi_M(x, \rho_{n^\alpha}(s))$ verwendet. Da diese Konstruktion für jedes ϵ gültig ist, folgt die Behauptung. □

1.7 Reduzierbarkeit und Hierarchie

Um Probleme miteinander vergleichen zu können, brauchen wir eine teilweise Ordnung auf der Menge der Probleme. Eine solche teilweise Ordnung wird durch den Reduzierbarkeitsbegriff geliefert. Für unsere Problemklassen — schnelle parallele Algorithmen für Schaltkreise — benötigen wir einen Reduzierbarkeitsbegriff, der sich an das Schaltkreismodell anlehnt.

Definition 1.7.1 Black-Box-Schaltkreis
Ein Black-Box-Schaltkreis ist ein uniformer Schaltkreis, der außer den logischen Gattern noch spezielle **Orakelknoten** enthalten darf. Für die Orakelknoten ist auch unbeschränkter Fan-In erlaubt, da das Orakel für jede beliebige Eingabegröße befragt werden darf. Dabei wird die Tiefe eines Orakelknotens (Black-Box) mit n Eingängen und m Ausgängen mit $\log(n+m)$ und die Größe polynomiell in $n+m$ veranschlagt. Die Abbildung 1.8 zeigt ein Bild eines solchen Schaltkreises mit Orakel $S(x,y)$.

Definition 1.7.2 Eine Funktion $f : \mathbb{B}^* \to \mathbb{B}^*$ ist in NC^k falls $|f(x)| = |x|^{O(1)}$ ist und die Sprache

$$A_f := \{(x,i) |\ i\text{-tes Bit von } f(x) \text{ ist } 1\}$$

in NC^k liegt.

Damit sind die Klassen Black-Box NC^k, Black-Box NC, Black-Box RNC etc. analog zu den entsprechenden Klassen definiert, nur daß statt uniformer Schaltkreise uniforme Black-Box-Schaltkreise verwendet werden.

Definition 1.7.3 NC^1 Reduzierbarkeit
X ist NC^1 reduzierbar auf Y (in Zeichen $X \leq_{NC^1} Y$), genau dann wenn es eine Familie uniformer Black-Box NC^1 Schaltkreise $\{\mathcal{C}_n\}$ für X gibt, deren Orakelknoten S_n das Problem Y lösen. Die Tiefe der Orakelknoten n ist dabei $\log(n+m)$, wie in Definition 1.7.1 vorgegeben.

Nachdem wir den Reduzierbarkeitsbegriff präzisiert haben, befassen wir uns nun mit dem Begriff des

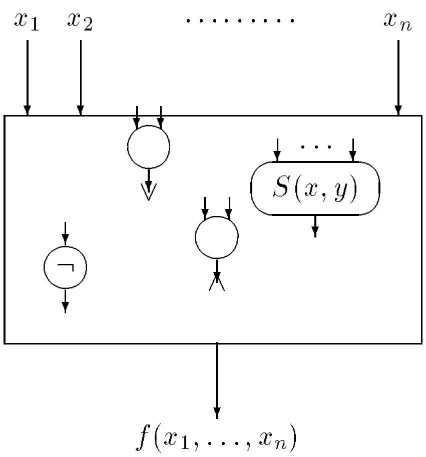

Abbildung 1.8: Black-Box Schaltkreis

Definition 1.7.4 NC ^{1} Abschluß
Der NC ^{1} Abschluß eines Problems Y besteht aus allen Problemen X, die sich auf Y mittels NC ^{1} Reduktionen zurückführen lassen:

$$Y^* = \{X | X \leq_{NC^1} Y\}.$$

Definition 1.7.5 NC ^{1}-abgeschlossen
Sei $\mathcal{Y}$ eine Klasse von Problemen. $\mathcal{Y}$ ist NC ^{1}-abgeschlossen, genau dann wenn

$$\mathcal{Y} = \mathcal{Y}^* = \{Z | Z \leq_{NC^1} Y \text{ für ein } Y \in \mathcal{Y}\}.$$

Nun gilt das folgende wichtige

Lemma 1.7.6 Die Klassen NC ^{k} sind NC ^{1} abgeschlossen für alle $k \geq 1$.

Beweis: Sei $X \leq_{NC^1} Y$ und $Y \in$ NC ^{k}. Sei $\{\mathcal{C}_n\}$ die Black-Box NC^1 Schaltkreisfamilie, die X auf Y reduziert und $\{\mathcal{D}_n\}$ die NC ^{k} Schaltkreisfamilie für Y. Wir konstruieren nun eine Klasse von uniformen NC ^{k} Schaltkreisen $\{\mathcal{R}_n\}$, die X lösen. Dazu betrachten wir den Black-Box NC ^{1} Schaltkreis der Reduktion und ersetzen die Orakelknoten durch die NC ^{k} Schaltkreise für Y. Nun haben wir einen Schaltkreis für X gewonnen. Wir müssen nun noch zeigen, daß dies ein NC ^{k} Schaltkreis ist. Es ist klar, daß der so gewonnene Schaltkreis uniform ist, da sowohl $\mathcal{C}_n$ als auch $\mathcal{D}_n$ uniform sind, und daß die Größe polynomiell ist. Wir müssen daher nur die Tiefe von $\{\mathcal{R}_n\}$ abschätzen. Dazu betrachten wir einen Weg in $\mathcal{C}_n$. Dieser Weg enthält höchstens $O(\log n)$ Gatter und Orakelknoten, die durch die Schaltkreise $\mathcal{D}_{m_i}$ ersetzt sind. Damit gilt

$$\begin{aligned}\text{Depth}(\mathcal{R}_n) &= \sum \text{Depth}(\mathcal{D}_{m_i}) + O(\log n) \\ &= O(\sum \log^k m_i) + O(\log n).\end{aligned}$$

Da die Tiefe eines Orakelknoten mit l Ein- und Ausgängen in $\{\mathcal{C}_n\}$ mit $\log l$ veranschlagt wird und $\mathcal{C}_n$ Tiefe $O(\log n)$ hat, muß gelten:

$$\sum_{i=1}^{\log n} \log m_i = O(\log n).$$

Da $\sum \log^k m_i \leq \log^{k-1} n \cdot \sum \log m_i = O(\log^k n)$ gilt, folgt

$$\mathrm{Depth}(\mathcal{R}_n) = O(\log^k n).$$

$\square$

Mit Hilfe des Reduzierbarkeitsbegriffs ist auf der Menge der Probleme eine partielle Ordnung definiert. Wir können nun fragen, wie die größten Elemente bzgl. dieser Ordnung aussehen. Dazu hilft die folgende

Definition 1.7.7 NC ^{1}-hart, NC ^{1}-vollständig
Ein Problem Y ist NC ^{1}-hart für eine Klasse $\mathcal{C}$, wenn

$$\forall X \in \mathcal{C} \quad X \leq_{NC^1} Y.$$

Ein Problem Y ist NC ^{1}-vollständig für eine Klasse $\mathcal{C}$, wenn Y NC ^{1}-hart für $\mathcal{C}$ und $Y \in \mathcal{C}$ ist.

Korollar 1.7.8 Sei $\mathcal{C}$ NC ^{1}-abgeschlossen, $\mathcal{C} \subseteq \mathcal{D}$ und Y NC ^{1}-vollständig für $\mathcal{D}$. Dann gilt

$$Y \in \mathcal{C} \;\Leftrightarrow\; \mathcal{C} = \mathcal{D}.$$

Angewandt auf unsere Situation können wir damit unseren *'Parallelitätentraum'* formulieren:

$$\mathrm{NC} = \mathcal{P}.$$

Um diesen Traum zu beweisen, reicht es zu zeigen, daß ein NC ^{1}-vollständiges Problem Y für $\mathcal{P}$ in NC liegt. Man nimmt jedoch allgemein an, daß NC $\neq \mathcal{P}$ gilt.

Neben der NC ^{1}-Reduktion können wir mit Hilfe der randomisierten Schaltkreise auch eine RNC ^{1}-Reduktion definieren. Die Definitionen laufen analog zu den obigen Definitionen und seien daher weggelassen. Wichtig für uns ist später der Abschluß bzgl. der RNC ^{1}-Reduktion.

Definition 1.7.9 RNC ^{1} Abschluß
Der RNC ^{1} Abschluß eines Problems Y besteht aus allen Problemen X, die sich auf Y mittels RNC ^{1} Reduktionen zurückführen lassen:

$$RY := \{X | X \leq_{RNC^1} Y\}.$$

Wir werden insbesondere später die Klassen DET* und RDET betrachten.

Nun stellen wir einige Probleme aus $\mathcal{P}$ vor, die vollständig für $\mathcal{P}$ bzgl. der NC ^{1}-Reduktion sind. Um Zweideutigkeiten aus dem Wege zu gehen, nennen wir diese Probleme nicht $\mathcal{P}$-vollständig, da dieser Begriff mit der logspace-Reduktion verbunden wird.

Dabei gehen wir folgendermaßen vor: Zuerst zeigen wir explizit, daß ein Problem C NC ^{1}-vollständig ist. Um nun zu zeigen, daß ein weiteres Problem D auch NC ^{1}-vollständig ist, brauchen wir dann nur noch ein schon bekanntes NC ^{1}-vollständiges Problem auf D zu reduzieren, da die NC ^{1} Reduktion transitiv ist:

$$[B \leq_{NC^1} C \,\wedge\, C \leq_{NC^1} D] \Rightarrow B \leq_{NC^1} D.$$

Unser erstes Problem ist das

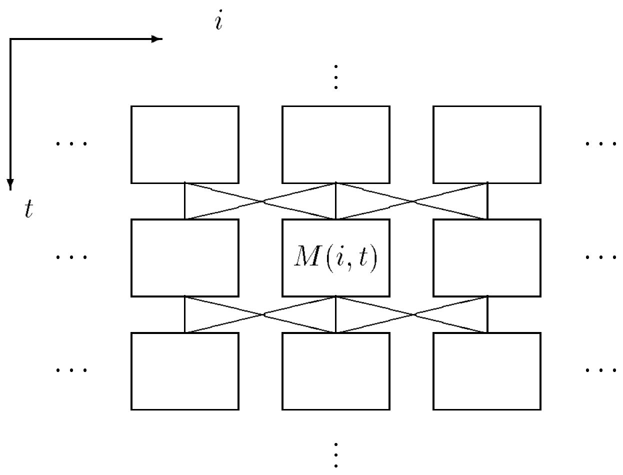

Abbildung 1.9: Simulation der Turing Maschine durch einen Schaltkreis

Definition 1.7.10 Circuit Value Problem (CVP)
Gegeben sei eine Beschreibung eines Schaltkreises $\mathcal{C}$ über der Basis $B = \{\vee, \wedge, \neg\}$, der eine boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ realisiert, und die Werte für die Eingabevariablen $x_1, \ldots, x_n$. Das Problem ist es, den Schaltkreis auszuwerten. Die Eingabegröße für dieses Problem ist die Länge der Beschreibung des Schaltkreises. Diese ist polynomiell verknüpft mit der Größe des Schaltkreises.

Satz 1.7.11 Das Circuit Value Problem ist NC^1-vollständig für $\mathcal{P}$.

BEWEIS:

- Das CVP ist in $\mathcal{P}$ enthalten. $\surd$
- Das CVP ist NC^1-hart für $\mathcal{P}$.
  Wir müssen eine Einband TM M, die ein Problem in $\mathcal{P}$ löst, durch eine Schaltkreisfamilie simulieren. Dazu sei w die Eingabe mit $n = |w|$. Dann beträgt die Laufzeit höchstens $p(n)$ für ein Polynom p und somit kann auch der Bandverbrauch durch $p(n)$ abgeschätzt werden. Wir konstruieren jetzt für dieses n einen Schaltkreis $\mathcal{C}_n$, der die Turing Maschine simuliert.
  Die Bausteine dieses Schaltkreises sind Module $M(i,t)$, in denen die endliche Kontrolle der TM verdrahtet ist. Der Schaltkreis $\mathcal{C}_n$ besteht aus einem $p(n) \times p(n)$ Quadrat dieser Module. Er ist in Abbildung 1.9 dargestellt. Das Modul $M(i,t)$ ist genau dann aktiv, wenn zur Zeit t die TM M den Kopf auf der Bandzelle i stehen hat. Die Module $M(1,1), \ldots M(p(n),1)$ enthalten die Anfangskonfiguration des Bandes; die aktuelle Konfiguration wird Schritt für Schritt an die darunterliegenden Module weitergereicht. Ein Schritt in der Berechnung der TM M wird jetzt wie folgt simuliert:
  Das aktive Modul $M(i,t)$ berechnet mit Hilfe der Information in der i-ten Bandzelle die weiteren Konfigurationen: es gibt den eventuell geänderten Inhalt der i-ten Bandzelle an das Modul $M(i,t+1)$ weiter und aktiviert je nach Kopfbewegung das Modul $M(i-1,t+1)$, $M(i,t+1)$ oder $M(i+1,t+1)$. Die nicht aktiven Module $M(k,t)$ reichen

den Inhalt der Bandzelle k an ihren Nachfolger $M(k,t+1)$ weiter. Insgesamt muß ein Modul also nur **lokal** mit 4 Leitungen verdrahtet werden. Der gesamte Schaltkreis hat eine Größe $s(\mathcal{C}_n) = c \cdot p(n)^2$, und somit ist eine Beschreibung dieses Schaltkreises in polynomieller Länge in n möglich. Die Konstruktion dieses Schaltkreises ist mit Hilfe der Cook'schen Technik in Tiefe $O(\log n)$ möglich. $\surd$

$\square$

Nun folgt eine Sammlung weiterer NC^1-vollständiger Probleme. Das erste ist sehr eng verwandt mit dem CVP, nämlich das

Definition 1.7.12 Monotones Circuit Value Problem (MCVP)
Das MCVP ist das CVP mit der Einschränkung, daß der Schaltkreis $\mathcal{D}$ keine Negationen enthalten darf, d.h. die Basis $B_M = \{\vee, \wedge\}$.

Lemma 1.7.13 Das MCVP ist NC^1-vollständig für $\mathcal{P}$.

Beweis: Gegeben sei eine Beschreibung des Schaltkreises $\mathcal{C}$ und die Eingabe $x = (x_1, \ldots, x_n)$ für das CVP. Wir werden aus diesem Schaltkreis einen monotonen Schaltkreis $\mathcal{D}$ und eine Eingabe $y = (y_1, \ldots, y_{2n})$ konstruieren, so daß

$$\forall x \; \mathcal{C}(x) = \mathcal{D}(y(x)).$$

$\mathcal{D}$ wird so aus $\mathcal{C}$ konstruiert, daß von jedem Gatter sowohl das Ergebnis als auch die Negation dieses Ergebnisses geliefert wird. Ein $\neg$ Gatter kann nun einfach durch Vertauschen dieser beiden Werte realisiert werden. Induktiv wird $\mathcal{D}$ wie folgt konstruiert:
Die Eingabe y geht aus der Eingabe x wie folgt hervor

$$y_{2i-1} = x_i \quad y_{2i} = \neg x_i.$$

Ein $\wedge$ Gatter g sei mit den Gattern h_1 und h_2 verbunden. Jetzt muß neben $h_1 \wedge h_2$ auch $\neg(h_1 \wedge h_2)$ berechnet werden. Es gilt aber das Gesetz von de Morgan

$$\neg(h_1 \wedge h_2) \;\equiv\; (\neg h_1 \vee \neg h_2).$$

Somit kann der negierte Wert ebenfalls berechnet werden, da auch $\neg h_1$ und $\neg h_2$ zur Verfügung stehen. Ein $\vee$ Gatter wird entsprechend behandelt und ein $\neg$ Gatter wird dadurch realisiert, daß die beiden Werte vertauscht werden, d.h. die Leitungen kreuzen sich.
Eine Beschreibung für $\mathcal{D}$ kann leicht in logarithmischer Tiefe aus der Beschreibung von $\mathcal{C}$ berechnet werden. Damit ist MCVP NC^1-hart und auch NC^1-vollständig für $\mathcal{P}$. $\square$

Für die weiteren NC^1-vollständigen Probleme benötigen wir zunächst einige Definitionen.

Definition 1.7.14 Clique
Sei $G = (V, E)$ eine Graph. Eine Teilmenge $C \subseteq V$ heißt **Clique** von G, falls

$$\forall u, v \in C : \quad (u, v) \in E.$$

Eine Clique C heißt **maximal**, falls C keine Teilmenge einer anderen Clique ist.
Eine Clique C heißt **Maximum Clique**, falls C eine Clique maximaler Kardinalität ist.

Zu dem Begriff der Clique gibt es den analogen Begriff der

Definition 1.7.15 Unabhängige Menge (independent set)
Sei $G = (V, E)$ ein Graph. Eine Menge $M \subseteq V$ heißt unabhängige Menge, falls M im Komplementärgraphen $\bar{G} = (V, \bar{E})$ eine Clique ist, bzw. $\forall\, u, v \in M : \ (u, v) \notin E$ gilt.

Definition 1.7.16 Lexikographische Ordnung
Sei M eine geordnete Menge und $s_i, t_j \in M$. Dann gilt

$$s_1 s_2 \ldots s_p \preceq t_1 t_2 \ldots t_q$$

genau dann, wenn

$$\exists j \text{ mit } s_j < t_j \text{ und } s_i = t_i \text{ für } i < j \text{ oder}$$
$$p \leq q \text{ und } s_i = t_i \text{ für } 1 \leq i \leq p.$$

Definition 1.7.17 <u>F</u>irst <u>L</u>exicographically <u>M</u>aximal <u>C</u>lique
Sei $G = (V, E)$ ein Graph mit geordneter Knotenmenge $V = (v_1 < \ldots < v_n)$. Eine Clique $C = (v_{i_1}, \ldots, v_{i_k})$ heißt FLMC, wenn C eine maximale Clique von G ist und für alle maximalen Cliquen $C' = ({v'}_{i_1}, \ldots, {v'}_{i_m})$ von G

$$v_{i_1} \ldots v_{i_k} \preceq {v'}_{i_1} \ldots {v'}_{i_m}$$

gilt.

Definition 1.7.18 <u>F</u>irst <u>L</u>exicographically <u>M</u>aximal <u>I</u>ndependet <u>S</u>et
Sei $G = (V, E)$ ein Graph. Eine Menge M ist ein FLMIS, falls M im Komplementärgraphen eine FLMC ist.

Lemma 1.7.19 FLMISP
Das Problem, eine FLMIS zu finden, ist NC^1 vollständig für $\mathcal{P}$.

Beweis:

- FLMISP ist in $\mathcal{P}$ enthalten. $\surd$
- FLMISP ist NC^1-hart für $\mathcal{P}$
  Wir reduzieren das MCVP auf FLMISP. Sei also der Schaltkreis $\mathcal{C}$ mit k Gattern und die Eingabe y gegeben. Die Gatter $w_1, \ldots w_k$ seien **topologisch** geordnet. Wir konstruieren einen Graphen $G = (V, E)$ mit $2k$ Knoten. Dabei werden jedem Gatter v die Knoten v_1 und v_2 zugeordnet. v_1 soll im FLMIS liegen, falls das Gatter v den Wert 1 hat, sonst soll v_2 im FLMIS sein. Wir konstruieren den Graphen induktiv analog zum Beweis von Lemma 1.7.13.
  Zunächst müssen wir auf der Knotenmenge von G eine Ordnung definieren. Dazu benutzen wir die topologische Ordnung des Schaltkreises C. Falls das Gatter v ein $\vee$ Gatter oder ein Eingabegatter mit Eingabewert 0 ist, so soll $v_2 < v_1$ gelten, andernfalls (v ist ein $\wedge$ Gatter oder ein Eingabegatter mit Eingabewert 1) gilt $v_1 < v_2$. Somit ist eine eindeutige Ordnung auf V definiert.
  Nun muß noch die Kantenmenge induktiv bestimmt werden.

- v ist ein Eingabeknoten mit Eingabewert 1:
  v_1 wird mit keinem kleineren Knoten verbunden, aber v_2 mit jedem kleineren Knoten. Damit gehört v_1 zum FLMIS, aber v_2 nicht, da er mit v_1 verbunden ist.
- v ist ein Eingabeknoten mit Eingabewert 0:
  v_2 wird mit keinem kleineren Knoten verbunden, aber v_1 mit jedem kleineren Knoten. Damit gehört v_2 zum FLMIS, aber v_1 nicht, da er mit v_2 verbunden ist ($v_2 < v_1$).
- v ist ein $\wedge$ Gatter: ($v = u \wedge y$)
  v_1 wird nur mit u_2 und y_2 verbunden, v_2 nur mit v_1. Falls u_2 oder y_2 im FLMIS liegen, so liegt auch v_2 in der FLMIS. Das heißt v_1 liegt nur im FLMIS, falls sowohl u_1 als auch y_1 im FLMIS sind. Dies entspricht genau der Funktion des $\wedge$ Gatters.
- v ist ein $\vee$ Gatter:
  v_2 wird nur mit u_1 und y_1 verbunden, v_1 nur mit v_2 ($v_2 < v_1$). Die Richtigkeit dieser Konstruktion folgt mit den de Morgan'schen Gesetzen aus der Konstruktion für das $\wedge$ Gatter. $\surd$

G ist nun so konstruiert, daß von jedem Knotenpaar (v_1, v_2) der Knoten v_1 in der $FLMIS$ liegt, falls v den Wert 1 hat, sonst liegt v_2 in der $FLMIS$.

Man beachte hierbei die Dualität zwischen $\vee$ und $\wedge$ Gattern, die durch die de Morgan'schen Gesetze gegeben ist. Die Konstruktion des Graphen kann wieder mit Hilfe der Cook'schen Technik in NC^1 durchgeführt werden. Damit gilt

$$MCVP \leq_{NC^1} FLMISP$$

und somit ist FLMISP NC^1-vollständig für $\mathcal{P}$. $\square$

Als Korollar ergibt sich

Korollar 1.7.20 Das Problem, eine FLMC zu finden, ist NC^1-vollständig für $\mathcal{P}$.

Beweis: Dieses Problem ist dual zum FLMISP. $\square$

Weitere Beispiele von NC^1 vollständigen Problemen sind

Definition 1.7.21 Context Free Emptiness
Gegeben sei eine **kontextfreie Grammatik** G. Zu entscheiden ist, ob das leere Wort Λ aus dieser Grammatik ableitbar ist, also ob $\Lambda \in L(G)$ gilt.

und

Definition 1.7.22 Max Flow Problem
Gegeben sei ein Transportnetzwerk $T = (G, C, s, t)$, wobei die Kapazitäten C **binär** dargestellt seien. Das Problem ist es, den **maximalen Fluß** in T zu finden.

Die Beweise der Vollständigkeit dieser beiden Probleme können in [GR 88] nachgelesen werden. Hier findet man auch noch weitere NC^1 vollständige Probleme für $\mathcal{P}$.

1.8 Paralleles Sortieren

In diesem Kapitel werden wir parallele Sortierverfahren kennenlernen.

Definition 1.8.1 Sortierproblem
Gegeben sei eine Folge $(a_1, \ldots, a_n)$ mit $a_i \in M$. Auf M sei eine totale Ordnung definiert. Zu berechnen ist eine Permutation dieser Folge $(a_{\pi(1)}, \ldots, a_{\pi(n)})$, so daß $a_{\pi(j)} \leq a_{\pi(j)+1}$ für alle j gilt.

Als Grundoperationen für das Sortieren betrachten wir Vergleichs/Austauschoperationen, d.h. zwei beliebige Elemente der Folge können miteinander verglichen werden und in Abhängigkeit des Ergebnis dieses Vergleiches ihre Plätze tauschen.

Bemerkung 1.8.2 Wir können immer ohne Einschränkung annehmen, daß die Elemente der zu sortierenden Folge paarweise verschieden sind, da man jedem Element a_i das Paar (a_i, i) zuordnen und diese neue Folge (die aus verschiedenen Elementen besteht) bezüglich der lexikographischen Ordnung sortieren kann.

Definition 1.8.3 Sortierbaum
Ein binärer Baum T heißt **Sortierbaum**, falls

- die Blätter von T den Permutationen von $\{1, \ldots, n\}$ entsprechen,
- jeder innere Knoten von der Form (a_i, a_j) ist,
- eine Berechnung des Sortierbaums ein Wurzel-Blatt Weg $(a_{i_1}, a_{j_1}), (a_{i_2}, a_{j_2}), \ldots, \pi$ ist. Für diesen Weg muß zusätzlich gelten:
  - der Nachfolger von (a_{i_k}, a_{j_k}) ist der linke Sohn, falls $a_{i_k} < a_{j_k}$ gilt, sonst der rechte Sohn,
  - die dem Blatt in dieser Folge entsprechende Permutation **löst** das Sortierproblem.

Damit gilt der folgende Satz über die untere Laufzeitschranke für das Sortierproblem. Dabei zählen wir nur die Anzahl der benötigten Vergleichs/Austauschoperationen.

Satz 1.8.4 Für das allgemeine Sortierproblem ist $\Omega(n \log n)$ eine untere Schranke für die schlechtest mögliche Laufzeit.

Beweis: Wir können nach Bemerkung 1.8.2 annehmen, daß alle Elemente verschieden sind, d.h. nur eine der $n!$ möglichen Permutationen der Folgenelementen eine gültige Berechnung ist. Diese $n!$ möglichen Permutationen bilden die Blätter eines Sortierbaumes. Daher gibt es wenigstens einen Wurzel-Blatt Weg der Länge $\Omega(\log n!) = \Omega(n \log n)$. Wenn wir die Folge betrachten, die nur von der Permutation sortiert wird, die dem Blatt des Weges entspricht, so ist die Länge der Berechnung an dieser Folge $\Omega(n \log n)$. □

Nun wollen wir ein sequentielles Sortierverfahren vorstellen, welches sich gut parallelisieren läßt.

Algorithmus 1.8.5 Mergesort

```
Merge(a(1),...,a(n),b(1),...,b(n));
   begin
      i := j := 1
      WHILE i ≤ n OR j ≤ n DO
         IF a(i) < b(j)
            THEN c(i+j) := a(i); i := i+1
            ELSE c(i+j) := b(j); j := j+1;
      OD;
      Merge:=(c(1),...,c(2n));
   end;

Mergesort(a(1),...,a(n));
   begin
      IF n = 1
         THEN a(1)
         ELSE Merge (Mergesort(a(1),...,a(n/2)),
                     Mergesort(a(n/2+1),...,a(n)));
   end.
```

Satz 1.8.6 Der Mergesort sortiert eine Folge $a_1 \dots a_n$ in Zeit $O(n \log n)$.

Nun wollen wir den Mergesort mit Hilfe von Sortiernetzwerken parallelisieren.

Definition 1.8.7 Sortiernetzwerk
Ein **Sortiernetzwerk** ist ein gerichteter Graph $G = (V, E)$, wobei die Menge der Kanten aus **Leitungskanten** und **Vergleichskanten** besteht. Die Grundbausteine des Sortiernetzwerkes sind Compare/Exchange Bausteine, die die Vergleichs/Austausch Operationen realisieren. Ein Compare/Exchange Baustein ist in Abbildung 1.10 dargestellt. Es müssen nun die folgenden Regeln gelten:

1. Je zwei Vergleichskanten besitzen keinen gemeinsamen Knoten, bzw. jeder Knoten ist höchstens mit einer Vergleichskante inzident.
2. In jeden Knoten geht höchstens eine Leitungskante hinein.
3. Es gibt keinen Kreis $(a_1, \dots, a_n, a_1)$, so daß (a_i, a_{i+1}) Leitungskante oder (a_i, a_{i+1}) bzw. (a_{i+1}, a_i) Vergleichskanten sind.

Die Größe eines Sortiernetzwerkes wird mit der Anzahl von Vergleichskanten gleichgesetzt. Die Tiefe eines Sortiernetzwerkes ist die Länge eines längsten bzgl. der Leitungskanten gerichteten Weges.

Bemerkung 1.8.8 Falls man alle Vergleichskanten mit den beiden zugehörigen Knoten in einen Knoten zusammenzieht, erhält man einen gerichteten, azyklischen Graphen.

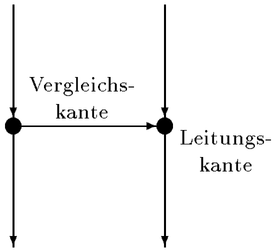

Abbildung 1.10: Compare/Exchange Baustein

Definition 1.8.9 Semantik der Sortiernetzwerke
Mit $w(l)$ sei der Wert an der Leitungskante l bezeichnet. Die Eingabe $(a_1, \ldots, a_n)$ liege an den Eingabeknoten $(x_1, \ldots, x_n)$ an. Dann ist der Wert $w(x_i, v)$ der Leitungskante $(x_i, v) = a_i$. Seien nun (y_1, v_1), (y_2, v_2), (v_1, z_1) und (v_2, z_2) Leitungskanten und (v_1, v_2) die dazwischen liegende Vergleichskante. Dann gilt

$$\begin{aligned} w(v_1, z_1) &= \min(w(y_1, v_1), w(y_2, v_2)) \\ w(v_2, z_2) &= \max(w(y_1, v_1), w(y_2, v_2)) \end{aligned}$$

Ein Sortiernetzwerk heißt korrekt, falls es für jede Eingabefolge eine sortierte Permutation der Folge ausgibt.

Im folgenden benutzen wir den Begriff Sortiernetzwerk nur noch für korrekte Sortiernetzwerke.

Der folgende Satz vereinfacht die Korrektheitsbeweise für Sortiernetzwerke.

Satz 1.8.10 Knuthsches 0-1 Gesetz
Ein Sortiernetzwerk N arbeitet genau dann korrekt, wenn es für jede 0-1 Folge korrekt arbeitet.

BEWEIS: "$\Rightarrow$" Diese ist trivial.
"$\Leftarrow$" Nehmen wir an N arbeite für alle 0-1 Folgen korrekt, versage jedoch für die Folge $(a_1, \ldots, a_n)$. Bezeichne j die Position, für die $a_{j-1} > a_j$ in der ausgegebenen Folge gilt. Zu dieser Folge und beliebigem $x \in I\!R$ definieren wir uns nun Folge $(s_1^x, \ldots, s_n^x)$ mit

$$s_i^x := \begin{cases} 0 & a_i \leq x \\ 1 & a_i > x \end{cases}$$

Nach Voraussetzung wird jede Folge (s_i^x) richtig sortiert. Betrachten wir nun die Folge $(s_i^{a_j})$. Da die Transformation von (a_i) auf (s_i^x) monoton ist, ist das Verhalten aller Compare/Exchange Bausteine für beide Folgen gleich. Daher werden die gleichen Permutationen der Folgen ausgegeben. Daraus folgt, daß $1 = s_{j-1}^{a_j} > s_j^{a_j} = 0$ für die ausgegebene 0-1 Folge $(s_i^{a_j})$ gilt. Diese Folge ist jedoch nicht sortiert, was in Widerspruch zur Voraussetzung steht. □

Nun wollen wir mit den Sortiernetzwerken unseren Mergesort parallel realisieren. Ein erstes Resultat liefert der folgende

Satz 1.8.11 (Batcher, 1968)
Es existiert eine (logspace) uniforme Folge von Sortiernetzwerken mit Tiefe $O(\log^2 n)$ und Größe $O(n \log^2 n)$.

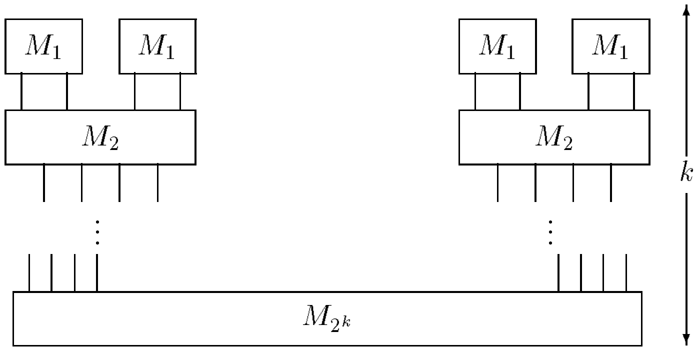

Abbildung 1.11: Aufbau eines Mischsortierers

Wir werden jetzt eine Familie dieser Schaltkreise näher betrachten. Dies sind die Odd-Even Merge Sortiernetzwerke von Batcher. Man beachte, daß diese Netzwerke zwar NC-Schaltkreise sind, jedoch keine optimale parallelen Lösungen sind.

Den allgemeinen Aufbau eines Mischsortiernetzwerkes zeigt Abbildung 1.11. Die Höhe des Mischbaumes ist dabei $\log n$.
Wir gehen im folgenden davon aus, daß die Eingabe jedes Merge-Moduls zwei (gleich lange) sortierte Folgen sind. Also sei die Länge der zu sortierenden Folge immer eine Zweierpotenz $n = 2^m$.

Wir werden jetzt den Aufbau der Merge Module des Odd-Even Merge Netzwerkes genauer analysieren.

Definition 1.8.12 Odd-Even Merge Modul
Ein Odd-Even Merge Modul der Eingabelänge n hat als Eingabe zwei sortierte Folgen der Länge n und produziert daraus eine sortierte Folge der Länge $2n$. Wir werden das Odd-Even Merge Modul rekursiv definieren.

- Ein Odd-Even Merge Modul der Eingabelänge 1 ist ein Compare/Exchange Baustein.
- Ein Odd-Even Merge Modul M der Eingabelänge n besteht aus zwei Odd-Even Merge Moduln M_1 und M_2 der Eingabelänge $n/2$. Die Eingabe von M sei $(a_1, \ldots, a_n, b_1, \ldots, b_n)$. Daraus werden die Eingaben für M_1 und M_2 so konstruiert, daß M_1 die Folgen, bestehend aus den Elementen von (a_i) bzw. (b_i) mit ungeraden Indizes, als Eingabe erhält und M_2 die Teilfolgen bestehend aus den geraden Indizes. Die Ausgabefolgen der beiden Merge Module M_1 und M_2 seien c_i bzw. d_i. Die Ausgabe e_i des Merge Moduls M werden nun folgendermaßen bestimmt:

$$f_{2i-1} = c_i \quad f_{2i} = d_i$$
$$e_1 = f_1 \quad e_{2n} = f_{2n}$$
$$e_{2i} = \min\{f_{2i}, f_{2i+1}\} \quad e_{2i+1} = \max\{f_{2i}, f_{2i+1}\} \quad \text{für } 1 \leq i < n,$$

  d.h. e_{2i}, e_{2i+1} geht durch Anwenden eines Compare/Exchange Bausteins auf f_{2i}, f_{2i+1} hervor.

Die Abbildung 1.12 zeigt den rekursiven Aufbau eines solchen Merge Moduls.

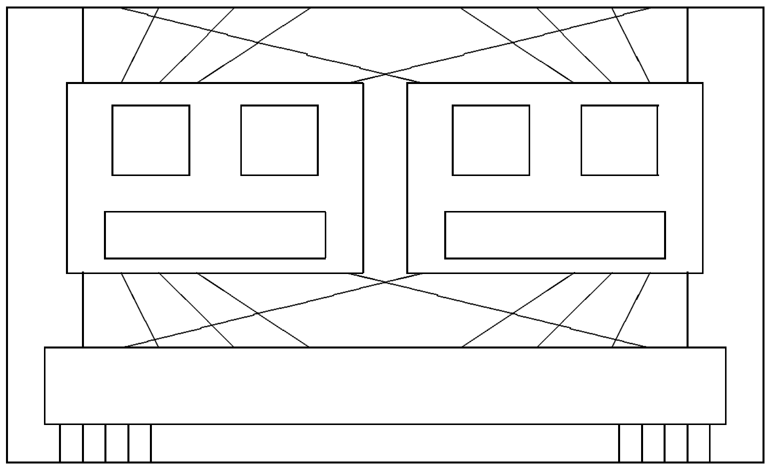

Abbildung 1.12: Aufbau eines Odd Even Merge Moduls

Lemma 1.8.13 Ein Odd-Even Merge Modul der Eingabelänge n hat eine Größe von $O(n \log n)$ und eine Tiefe von $O(\log n)$.

BEWEIS: Bezeichne M_n das Merge Modul der Eingabelänge n. Dann gelten die folgenden Rekursionsgleichungen:

$$\mathrm{Size}(M_{2n}) = 2 \cdot \mathrm{Size}(M_n) + n - 1$$

$$\mathrm{Depth}(M_{2n}) = \mathrm{Depth}(M_n) + 1$$

und

$$\mathrm{Size}(M_1) = 1 \quad \mathrm{Depth}(M_1) = 1$$

Daraus ergeben sich leicht die angegebenen Schranken. □

Korollar 1.8.14 Das Odd-Even Merge Netzwerk OE_n für Eingaben der Länge n hat eine Größe von $O(n \log^2 n)$ und eine Tiefe von $O(\log^2 n)$.

BEWEIS: Das Odd-Even Merge Netzwerk OE_n besteht aus $n/2^{i+1}$ Odd-Even Merge Moduln der Eingabelänge 2^i für $0 \leq i < \log n$. Damit ergibt sich für die Größe von OE_n

$$\begin{aligned} \mathrm{Size}(\mathrm{OE}_n) &= \textstyle\sum_{i=0}^{\log n - 1} \frac{n}{2^{i+1}} \cdot \mathrm{Size}(M_{2^i}) \\ &= \textstyle\sum_{i=0}^{\log n - 1} \frac{n}{2^{i+1}} \cdot O(2^i \log 2^i) \\ &= O(\textstyle\sum_{i=0}^{\log n - 1} n \cdot i) \\ &= O(n \log^2 n) \end{aligned}$$

und für die Tiefe von OE_n

$$\mathrm{Depth}(\mathrm{OE}_n) = \sum_{i=1}^{\log n} \log 2^i = \sum_{i=1}^{\log n} i = O(\log^2 n).$$

□

Nun wollen wir die Gültigkeit des Odd-Even Mergesort beweisen. Dazu reicht es zu zeigen, daß jedes Merge Modul korrekt arbeitet.

Satz 1.8.15 Jedes Merge Modul des Odd-Even Merge Netzwerkes produziert bei Eingabe von sortierten Folgen eine sortierte Folge.

BEWEIS: Nach Satz 1.8.10 reicht es 0-1 Folgen zu betrachten. Wir führen den Beweis nun induktiv nach der Eingabelänge n des Merge Moduls. Für $n = 1$ gibt es nichts zu beweisen, da in diesem Fall das Merge Modul ein Compare/Exchange Baustein ist.
Sei also $n = 2^m$. Da die Eingabe des Merge Moduls M sortierte Folgen (a_i) und (b_i) sind, haben sie die Form $0^p 1^{n-p}$ bzw. $0^q 1^{n-q}$. Daraus werden die Eingaben für die Merge Module M_1 und M_2 konstruiert. Nach Induktionsvoraussetzung ist die Ausgabe von M_1 also die Folge

$$0^{\lceil p/2\rceil + \lceil q/2\rceil} 1^{n - \lceil p/2\rceil + \lceil q/2\rceil}.$$

Analog ist die Ausgabe von M_2 die Folge

$$0^{\lfloor p/2\rfloor + \lfloor q/2\rfloor} 1^{n - \lfloor p/2\rfloor + \lfloor q/2\rfloor}.$$

Sei

$$d = (\lceil p/2\rceil + \lceil q/2\rceil) - (\lfloor p/2\rfloor + \lfloor q/2\rfloor).$$

Wir betrachten nun die drei möglichen Fälle

$d = 0$ (p und q sind gerade) In diesem Fall enthält die Ausgabefolge von M_1 genauso viele 0-en wie die Ausgabefolge von M_2. Somit ist die Folge $(f_i) = 0^{p+q} 1^{2n-p-q}$ bereits sortiert und damit auch $(e_i) = (f_i)$.

$d = 1$ (p gerade, q ungerade oder umgekehrt) In diesem Fall enthält die Ausgabefolge von M_1 eine 0 mehr als die Ausgabefolge von M_2. Damit ist aber die Folge $(f_i) = 0^{p+q} 1^{2n-p-q}$ bereits sortiert und somit auch (e_i).

$d = 2$ (p und q ungerade) Die Ausgabefolge von M_1 enthält zwei 0-en mehr als die Ausgabefolge von M_2. Damit hat (f_i) die folgende Form: $(f_i) = 0^{p+q-1} 1 0 1^{2n-p-q-1}$. Da $p + q - 1$ ungerade ist, werden die beiden falsch postierten 0 und 1 in der Folge (e_i) richtig gestellt. Somit ist auch in diesem Fall die Ausgabe korrekt. □

Es gibt neben den Sortiernetzwerken von Batcher auch ein Sortiernetzwerk, mit optimaler Größe und Tiefe.

Satz 1.8.16 Es gibt ein Sortiernetzwerk der Größe $O(n \log n)$ und der Tiefe $O(\log n)$.

BEWEIS: siehe [GR 88] und [AKS 83]. □

Der Nachteil dieses Sortiernetzwerkes ist, daß die Konstanten sehr groß sind. Wenn man nicht darauf angewiesen ist, Sortiernetzwerke zu benutzen ist der Sortieralgorithmus von Cole ([Co 86]) vorzuziehen. Der Algorithmus ist ein Merge-Algorithmus und hat somit mindestens die Tiefe $\Omega(\log n)$, die durch die Höhe des Merge-Baumes bedingt ist. Es gelingt Cole jedoch, das Mischen in **konstantem Zeitaufwand** zu realisieren. Dadurch erhält er einen Algorithmus mit $O(\log n)$ paralleler Zeit. Eine gute Beschreibung dieses Algorithmus wird in [GR 88] gegeben.

Kapitel 2

Algebraische Interpolation

2.1 Arithmetische Berechnungsmodelle

Oft tritt das Problem auf, in algebraischen Objekten arithmetische Berechnungen durchzuführen. Bei geeigneter Repräsentation dieser algebraischen Objekte durch Datenstrukturen kann eine solche arithmetische Berechnung durch Funktionen $f : \mathbb{B}^n \to \mathbb{B}^m$ ausgedrückt werden. Damit sind die booleschen Schaltkreise, die ein solches Problem lösen, **abhängig** von der **Datenstruktur** und **Größe** des algebraischen Objektes und nicht allein von der Art der arithmetischen Berechnung, die ausgeführt werden soll. In einem solchen Modell vermischen sich außerdem die Lösung für das eigentliche Problem mit der speziellen Arithmetik. Der Aufwand des Algorithmus setzt sich zusammen aus dem eigentlichen Aufwand zur Lösung des Problems und dem Berechnungsaufwand für die Arithmetik. Dies ist ein unerwünschter Effekt, da von der Hauptsache ablenkt wird und auch Vergleiche verschiedener Algorithmen erschwert werden. Daher führen wir jetzt Schaltkreise ein, die **arithmetische Berechnungen abstrahiert vom algebraischen Objekt** ausführen.

Beispiel 2.1.1 Die Berechnung des Produktes zweier Matrizen über GF(2) ist mit Hilfe des Satzes 1.3.14 in EREW-PRAM($\log n, n^3/\log n$) möglich. Betrachten wir das Matrizenprodukt über $\mathbb{Z}$, so erhalten wir einen Schaltkreis, in dem die Arithmetik für die Multiplikation ganzer Zahlen enthalten sein muß. Wenn wir annehmen, daß die Zahlen durch n Bits darstellbar sind, ist jede Zahlenmultiplikation mit Hilfe des Schönhage-Strassen Algorithmus ([SS 71]) in $O(\log n)$ Tiefe und $O(n \log n \log\log n)$ Prozessoren durchführbar. Somit haben wir nur einen EREW-PRAM($\log^2 n, n^4 \log\log n$) Algorithmus. In dieser Betrachtung vermischen sich also die Kosten für das eigentlichen Berechnungsproblem und die Kosten für die Arithmetik.

Die algebraischen Objekte, die wir im folgenden betrachten sind **Körper**. Wir wollen jetzt ein Berechnungsmodell einführen, das die booleschen Berechnungen von den Berechnungen in dem Körper trennt. Es stammt von J. von zur Gathen ([Ga 86]).

Definition 2.1.2 Sei F ein Körper. Dann bezeichne Ω eine Menge von Operationen über $(F \cup \mathbb{B})^*$. Zu jeder Operation $\omega \in \Omega$ sei eine sogenannte *Arity*-Funktion σ gegeben, die die Anzahl der Variablen aus F und $\mathbb{B}$ angibt:

$$\sigma = (\sigma_1, \sigma_2) : \Omega \to \mathbb{N} \times \mathbb{N}$$

mit

$$\sigma_1(\omega) = \text{Anzahl der Argumente von } \omega \text{ aus } F,$$
$$\sigma_2(\omega) = \text{Anzahl der Argumente von } \omega \text{ aus } \mathbb{B}.$$

Eine *Type* Funktion τ gibt den Wertebereich der Operation ω an:

$$\tau : \Omega \to \{F, \mathbb{B}\}.$$

Ein (Ω, σ, τ) - Programm ist ein 4-Tupel

$$P = (G, \lambda, l, r),$$

wobei

- $G = (V, E)$ ein endlicher, gerichteter, azyklischer Graph mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$ ist,
- $\lambda : V \to \Omega$ ein Labelling der Knoten ist,
- $l : V \to \{1, \ldots, |V|\}$ eine injektive Numerierung der Knoten ist und
- $r = (v_1, \ldots, v_k)$ eine Folge von Ausgabeknoten.

Dabei müssen die *Arity*-Funktionen beachtet werden und l konsistent mit der topologischen Ordnung von G sein.

Für jedes $v \in V$ gilt:
Es gibt genau $\sigma_1(\lambda(v))$ eingehende Kanten (w, v) mit $\tau(\lambda(w)) = F$ und genau $\sigma_2(\lambda(v))$ eingehende Kanten (w, v) mit $\tau(\lambda(w)) = \mathbb{B}$. Jedes solche w muß bezüglich der topologischen Ordnung kleiner sein, d.h. es muß $l(w) < l(v)$ gelten.

Ein einfaches Modell benutzt nur Operationen in F und hat die folgende Menge Ω^0 von Operationen.

Definition 2.1.3 Sei F ein Körper, $n \in \mathbb{N}$, $N = \{1, \ldots, n\}$ und

$$\Omega^0 = \Omega_F^0(n) = F \cup \{+, -, *, /\} \cup (\{in_F\} \times N)$$

mit *Arity*-Funktion $\sigma^0 = \sigma_F^0(n)$

$$\forall \omega \in \Omega^0 \quad \sigma^0(\omega) = \begin{cases} (0,0) & \text{falls } \omega \in F \cup (\{in_F\} \times N) \\ (2,0) & \text{falls } \omega \in \{+, -, *, /\} \end{cases}$$

und *Type*-Funktion $\tau^0 = \tau_F^0(n)$

$$\forall \omega \in \Omega^0 \quad \tau^0(\omega) = F.$$

Damit können wir jetzt unser erstes arithmetisches Berechnungsmodell definieren.

Definition 2.1.4 Arithmetischer Schaltkreis
Ein **Arithmetischer Schaltkreis** (ACR) über F mit n Eingängen und ν Ausgängen ist ein $(\Omega_F^0(n), \sigma_F^0(n), \tau_F^0(n))$ - Programm (G, λ, l, r) mit $|r| = \nu$.

Bemerkung 2.1.5 Arithmetische Schaltkreise berechnen rationale Funktionen über F. Die Menge der so berechenbaren rationalen Funktionen über F bezeichnen wir mit RAT_F.

Wir werden jetzt unser arithmetisches Berechnungsmodell erweitern, indem wir auch einen booleschen Kontrollfluß zulassen.

Definition 2.1.6 Interface
Der boolesche Kontrollfluß und die arithmetischen Berechnungen in F können nur über

- *Sign*-Knoten
- *Selection*-Knoten

miteinander in Verbindung treten. Dabei stellt

- *Sign* eine Funktion $F \to I\!B$ dar, mit
$$\operatorname{sign}(a) = \begin{cases} \underline{\text{TRUE}} & \text{falls } a \neq 0 \\ \underline{\text{FALSE}} & \text{falls } a = 0; \end{cases}$$
- *Selection* eine Funktion $F^2 \times I\!B \to F$ dar, mit
$$\operatorname{sel}(a_1, a_2, b) = \begin{cases} a_1 & \text{falls } b = \underline{\text{TRUE}} \\ a_2 & \text{falls } b = \underline{\text{FALSE}}; \end{cases}$$

Nun können wir ein weiteres arithmetisches Berechnungsmodell einführen.

Definition 2.1.7 Sei F ein Körper, $n, m, \nu, \mu \in I\!N$, $N = \{1, \ldots, n\}$, $M = \{1, \ldots, m\}$ und
$$\Omega^1 = \Omega^1_F(n,m) = \Omega^0_F(n) \cup I\!B \cup \{\operatorname{sign}, \operatorname{sel}, \neg, \vee, \wedge\} \cup (\{in_{I\!B}\} \times M)$$
mit erweiterter *Arity*-Funktion $\sigma^1 = \sigma^1_F(n,m)$
$$\forall \omega \in \Omega^1 \quad \sigma^1(\omega) = \begin{cases} \sigma^0(\omega) & \text{falls } \omega \in \Omega^0 \\ (1,0) & \text{falls } \omega = \operatorname{sign} \\ (2,1) & \text{falls } \omega = \operatorname{sel} \\ (0,1) & \text{falls } \omega = \neg \\ (0,0) & \text{falls } \omega \in I\!B \cup (\{in_{I\!B}\} \times M) \\ (0,2) & \text{falls } \omega \in \{\wedge, \vee\} \end{cases}$$
und erweiterter *Type*-Funktion $\tau^1 = \tau^1_F(n,m)$
$$\forall \omega \in \Omega^1 \quad \tau^1(\omega) = \begin{cases} F & \text{falls } \omega \in \Omega^0 \cup \{\operatorname{sel}\} \\ I\!B & \text{sonst.} \end{cases}$$

Definition 2.1.8 Arithmetisches Netzwerk
Ein **Arithmetisches Netzwerk** (ANT) über F mit (n,m) Eingängen und (ν,μ) Ausgängen ist ein $(\Omega^1_F(n,m), \sigma^1_F(n,m), \tau^1_F(n,m))$ - Programm (G, λ, l, r) mit ν Ausgabeknoten aus r vom Type F und μ Ausgabeknoten aus r vom Type $I\!B$.

Satz 2.1.9 Simulation von booleschen Funktionen
Falls die *Conditional Inverse*

$$\omega^-(x) = \begin{cases} x^{-1} & \text{falls } x \neq 0 \\ 0 & \text{falls } x = 0 \end{cases}$$

für $x \in F$ definiert ist, können die booleschen Funktionen durch arithmetische Operationen in F simuliert werden. Dabei entspricht der Wert TRUE der $1 \in F$ und der Wert FALSE der $0 \in F$.

BEWEIS: Wir müssen die Funktionen mit booleschen Argumenten und booleschen Werten simulieren.

- $\mathrm{sign}(x) \rightsquigarrow x \cdot \omega^-(x)$
- $\mathrm{sel}(x_1, x_2, y) \rightsquigarrow x_1 \cdot y + x_2 \cdot (1 - y)$
- $\neg x \rightsquigarrow 1 - x$
- $x \wedge y \rightsquigarrow x \cdot y$
- $x \vee y \rightsquigarrow x + y - x \cdot y$ $\square$

Durch diese Simulation können wir uns nun auf die Betrachtung von arithmetischen Schaltkreisen beschränken. Analog zu booleschen Schaltkreisen möchten wir auch hier nur uniforme Schaltkreise behandeln.

Definition 2.1.10 Sei F ein Körper. Dann sei

$$\begin{aligned} \mathrm{NC}^k_{A[F]} = \; & \{f \in RAT_F \mid \exists \text{ Folge von uniformen arithmetischen Schaltkreisen} \\ & \{\mathcal{C}_n\} \text{ mit beschränkten Fan-In und } \mathrm{Size}(\mathcal{C}_n) = n^{O(1)} \\ & \text{und } \mathrm{Depth}(\mathcal{C}_n) = O(\log^k n)\text{, die } f \text{ berechnet}\} \end{aligned}$$

$$\mathrm{NC}_{A[F]} = \bigcup_{k \in I\!N} \mathrm{NC}^k_{A[F]}.$$

Mit Hilfe dieser Schaltkreise können wir jetzt unsere Algorithmen **unabhängig** von der Körperarithmetik studieren.

2.2 Parallele Lineare Algebra

In diesem Abschnitt werden die parallele Berechnung von Determinanten einer Matrix über beliebigen Ringen und verwandte Probleme behandelt.

2.2.1 Parallele Berechnung von Determinanten

Csanky konnte 1976 für Ringe der Charakteristik 0 zeigen, daß die Determinantenberechnung in $NC^2_{A[F]}$ liegt ([Cs 76]). Dieses Ergebnis wurde 1982 durch Borodin, von zur Gathen und Hopcropft auf beliebige Ringe erweitert ([BGH 82]). Ihr $NC^2_{A[F]}$ Algorithmus erfordert jedoch $O(n^{15})$ Prozessoren. In diesem Abschnitt wird der Algorithmus von Berkowitz vorgestellt ([Be 84]). Dieser Algorithmus benötigt lediglich $O(n^{3.5})$ Prozessoren bei gleicher paralleler Zeit. Es sei bemerkt, daß die Prozessorenzahl bei der $NC^2_{A[F]}$ Determinantenberechnung 1987 durch Pan noch weiter auf $O(n^{2.5})$ verbessert werden konnte ([Pa 87]).

Sei $A = (a_{ij})$ eine $n \times n$ Matrix über einem beliebigen Körper F. Betrachte die folgende Zerlegung von A

$$A = \begin{pmatrix} a_{11} & R \\ S & M \end{pmatrix},$$

wobei R, S und M $1 \times (n-1)$, $(n-1) \times 1$ und $(n-1) \times (n-1)$ Untermatrizen von A sind.

Der Algorithmus von Berkowitz führt die Berechnung der Determinanten von A rekursiv auf die Berechnung der Determinanten von M zurück. Sei

$$R_i = (a_{i,i+1}, ..., a_{i,n}), \qquad S_i^{\mathrm{t}} = (a_{i+1,i}, ..., a_{n,i})$$

und

$$M_i = \begin{pmatrix} a_{i+1,i+1} & \dots & a_{i+1,n} \\ \vdots & & \vdots \\ a_{n,i+1} & \dots & a_{n,n} \end{pmatrix}.$$

Seien $p(\lambda)$ und $q(\lambda)$ die charakteristischen Polynome von A und M, also

$$p(\lambda) := \det(A - \lambda \cdot E_n) = \sum_{i=0}^{n} p_{n-i}\lambda^i \quad \text{und}$$

$$q(\lambda) := \det(M - \lambda \cdot E_{n-1}) = \sum_{i=0}^{n-1} q_{n-1-i}\lambda^i.$$

Es gilt $\det(A) = p(0) = p_n$, also läßt sich die Berechnung der Determinanten von A auf die Berechnung des charakteristischen Polynoms von A zurückführen. Im folgenden werden wir zeigen, wie sich das charakteristische Polynom von A mit Hilfe des charakteristischen Polynoms von M darstellen läßt.

Dazu benötigen wir den

Satz 2.2.1 Entwicklungssatz nach Laplace
Entwicklung der Determinanten nach der j-ten Spalte von A:

$$\det A = \sum_{i=1}^{n} (-1)^{i+j} \cdot a_{ij} \cdot \det(A(i|j)),$$

und Entwicklung der Determinanten nach der i-ten Zeile von A:

$$\det A = \sum_{j=1}^{n} (-1)^{i+j} \cdot a_{ij} \cdot \det(A(i|j)),$$

wobei $A(i|j)$ eine $(n-1) \times (n-1)$ Matrix ist, die aus A durch Streichung der i-ten Zeile und der j-ten Spalte hervorgeht.

Sei $\operatorname{adj}(A) \in (n \times n; R)$ die Adjungierte von A. Für $1 \leq i, j \leq n$ ist

$$\operatorname{adj}(A)_{i,j} = (-1)^{i+j} \cdot \det(A(i|j)).$$

Die Adjungierte besitzt die folgende Eigenschaft:

Lemma 2.2.2 Sei $\det(A) \neq 0$. Dann ist $B \cdot A = A \cdot B = \det(A) \cdot E_n$ eindeutig lösbar mit $B = \operatorname{adj}(A)$.

Das charakteristische Polynom von A läßt sich mit Hilfe der Adjungierten von A ausdrücken:

Lemma 2.2.3

$$p(\lambda) = (a_{11} - \lambda) \cdot \det(M - \lambda \cdot E_{n-1}) + R \cdot \operatorname{adj}(M - \lambda \cdot E_{n-1}) \cdot S,$$

wobei R, S, M wie oben definiert sind.

Beweis: Man entwickelt $\det(A - \lambda \cdot E_n)$ nach der ersten Zeile und dann nach der ersten Spalte.

$$p(\lambda) = \det(A - \lambda \cdot E_n) = \det \begin{pmatrix} a_{11} - \lambda & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} - \lambda & \dots & \dots & \vdots \\ \vdots & & & & \vdots \\ \vdots & & & a_{n-1,n-1} - \lambda & a_{n-1,n} \\ a_{n1} & \dots & \dots & a_{n,n-1} & a_{n,n} - \lambda \end{pmatrix}.$$

Die Entwicklung nach der ersten Zeile liefert

$$p(\lambda) = (a_{11} - \lambda) \cdot \det(M - \lambda \cdot E_{n-1}) + \sum_{j=2}^{n} (-1)^{1+j} \cdot a_{1j} \cdot \det(A - \lambda \cdot E_n)(1|j),$$

Entwicklung von $\det(A - \lambda \cdot E_n)(1|j)$ nach der ersten Spalte ergibt weiter

$$\begin{aligned} p(\lambda) \quad = \quad & (a_{11} - \lambda) \cdot \det(M - \lambda \cdot E_{n-1}) + \\ & \underbrace{\sum_{j=2}^{n} (-1)^{1+j} \cdot a_{1j}}_{\text{1. Zeile}} \cdot \underbrace{\sum_{i=2}^{n} (-1)^{i+1} \cdot a_{i1} \cdot \det((M - \lambda \cdot E_{n-1})(i|j))}_{\text{1.Spalte}} \end{aligned}$$

Sei

$$m_{ij} = (-1)^{i+j} \cdot \det(M - \lambda \cdot E_{n-1})(i|j).$$

Dann ist

$$p(\lambda) = (a_{11} - \lambda) \cdot \det(M - \lambda \cdot E_{n-1}) + \underbrace{\begin{pmatrix} a_{12} \\ \vdots \\ a_{1n} \end{pmatrix}^t}_{R} \cdot \underbrace{\begin{pmatrix} m_{22} & \dots & \dots & m_{n2} \\ \vdots & & & \vdots \\ m_{2n} & \dots & \dots & m_{nn} \end{pmatrix}}_{\operatorname{adj}(M - \lambda \cdot E_{n-1})} \cdot \underbrace{\begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}}_{S}$$

Also ergibt sich

$$p(\lambda) = (a_{11} - \lambda) \cdot \det (M - \lambda \cdot E_{n-1}) + R \cdot \operatorname{adj} (M - \lambda \cdot E_{n-1}) \cdot S.$$

□

Die Adjungierte von $(M - \lambda \cdot E_{n-1})$ läßt sich schreiben als

Lemma 2.2.4

$$\operatorname{adj} (M - \lambda \cdot E_{n-1}) = -\sum_{k=2}^{n} (q_0 M^{k-2} + q_1 M^{k-3} + \cdots + q_{k-2} E_{n-1}) \cdot \lambda^{n-k},$$

wobei $q(\lambda)$ das charakteristische Polynom von M ist.

Beweis: Die beiden Seiten der Gleichung werden mit $(M - \lambda \cdot E_{n-1})$ multipliziert. Für die linke Seite ergibt sich nach Lemma 2.2.2:

$$\operatorname{adj} (M - \lambda \cdot E_{n-1}) \cdot (M - \lambda \cdot E_{n-1}) = \det (M - \lambda \cdot E_{n-1}) \cdot E_{n-1} = q(\lambda) \cdot E_{n-1}.$$

Für die rechte Seite ergibt sich:

$$\begin{aligned}
&(-\sum_{k=2}^{n} (q_0 M^{k-2} + q_1 M^{k-3} + \cdots + q_{k-2} E_{n-1}) \cdot \lambda^{n-k}) \cdot (M - \lambda \cdot E_{n-1}) \\
&= \; -\sum_{k=2}^{n} (q_0 M^{k-1} + q_1 M^{k-2} + \cdots + q_{k-2} M) \cdot \lambda^{n-k} \\
&\quad + \sum_{k=1}^{n-1} (q_0 M^{k-1} + q_1 M^{k-2} + \cdots + q_{k-2} M + q_{k-1} E_{n-1}) \cdot \lambda^{n-k}. \qquad (*)
\end{aligned}$$

Der Satz von Caley-Hamilton besagt, daß die Einsetzung eines Endomorphismus M in sein charakteristisches Polynom q_M die Nullabbildung ergibt, d.h.

$$q(M) = q_0 M^{n-1} + q_1 M^{n-2} + \cdots + q_{n-1} E_{n-1} = 0.$$

Damit kann man den Index des zweiten Summanden in $(*)$ bis n laufen lassen. Die beiden Summanden lassen sich zusammenfassen und es ergibt sich

$$\begin{aligned}
(*) &= q_0 \lambda^{n-1} E_{n-1} + q_1 \lambda^{n-2} E_{n-1} + \cdots + q_{n-1} \lambda^0 E_{n-1} \\
&= q(\lambda) \cdot E_{n-1}.
\end{aligned}$$

□

Unser Ziel wird es sein, die Koeffizienten des charakteristischen Polynoms von A durch ein Gleichungssystem in Koeffizienten des charakteristischen Polynoms von M auszudrücken. Hierzu sind einige Vorüberlegungen nötig.

Eine Matrix $B \in (n \times m;\, R)$ heißt Toeplitz-Matrix genau dann wenn:

$$b_{ij} = b_{i - \min(i,j)+1,\, j - \min(i,j)+1}$$

d.h. in jeder Diagonalen von B stehen die gleichen Elemente.

Lemma 2.2.5 Seien $A = (a_{ij}) \in (n \times m;\, R)$, $B = (b_{ij}) \in (m \times p;\, R)$ untere Dreiecks-Toeplitz-Matrizen. Dann ist $C = (c_{ij}) = A \cdot B \in (n \times p)$ eine untere Dreiecks-Toeplitz-Matrix und kann mit $O(n^2)$ Prozessoren in $O(\log n)$ Tiefe berechnet werden.

Beweis: Sei $a_l = a_{l1}$ und $b_l = b_{l1}$. Dann ist

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} = (a_i, a_{i-1}, \dots, a_1, 0, \dots, 0) \cdot \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b_1 \\ \vdots \\ b_{m+1-j} \end{pmatrix}.$$

Die i-te Zeile von A enthält i Koeffizienten und $m-i$ Nullen, die j-te Spalte von B $j-1$ Nullen und $m-(j-1)$ Koeffizienten. Falls $i < j$ folgt also $c_{ij} = 0$, damit ist C untere Dreiecksmatrix.

Weiterhin gilt

$$c_{kl} = \sum_{i+j=k+l} a_i \cdot b_j.$$

Also ist $c_{k1} = c_{k+l,1+l}$ für alle $1 \leq l \leq \min(n,p) - 1$ und C ist eine Toeplitz-Matrix.

C ergibt sich demnach aus der Konvolution

$$(a_1, ..., a_n) * (b_1, ..., b_m).$$

Falls der Ring R die Fast Fourier Transformation erlaubt, so kann C durch $O(n \log n)$ Prozessoren in $O(\log n)$ Tiefe berechnet werden, anderenfalls ist ein Aufwand von $O(n^2)$ Prozessoren und $O(\log n)$ Tiefe erforderlich. $\square$

Durch Lemma 2.2.4 und Lemma 2.2.3 werden die charakteristischen Polynome von A und M in Beziehung gesetzt:

$$\begin{aligned} p(\lambda) &= (a_{11} - \lambda) \cdot \det(M - \lambda \cdot E_{n-1}) - R \cdot \left(\sum_{k=2}^{n} (M^{k-2} q_0 + \cdots + E_{n-1} q_{k-2}) \cdot \lambda^{n-k}\right) \cdot S \\ &= (a_{11} - \lambda) \cdot \underbrace{(q_0 \lambda^{n-1} + q_1 \lambda^{n-2} + q_2 \lambda^{n-3} + \cdots + q_{n-1} \lambda^0)}_{\det(M - \lambda \cdot E_{n-1})} \\ &\quad \underbrace{- R \cdot \left(\sum_{k=2}^{n} (q_0 M^{k-2} + q_1 M^{k-3} + \cdots + q_{k-3} M^1 + q_{k-2} M^0) \cdot \lambda^{n-k}\right)}_{-\operatorname{adj}(M - \lambda \cdot E_{n-1})} \cdot S \end{aligned}$$

Koeffizientenvergleich ergibt:

$$\underbrace{\begin{pmatrix} -1 & & & & & \\ a_{11} & -1 & & & 0 & \\ -RM^0S & a_{11} & -1 & & & \\ -RM^1S & -RM^0S & a_{11} & -1 & & \\ \vdots & & \ddots & & & \\ \vdots & & & & & -1 \\ -RM^{n-2}S & -RM^{n-3}S & & \dots & -RM^0S & a_{11} \end{pmatrix}}_{=:C^{(1)} \in (n+1 \times n;\, R)} \cdot \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{n-1} \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_n \end{pmatrix}$$

Rekursive Anwendung dieses Prinzips ergibt für die t-te ($t = 1, \ldots, n$) Rekursion die Matrix $C^{(t)} = (c_{ij}^{(t)}) \in (n+2-t \times n+1-t;\ R)$:

$$c_{ij}^{(t)} = \begin{cases} -1 & \text{falls} \quad i = 1 \\ a_{tt} & \text{falls} \quad i = 2 \\ -R_t M_t^{i-3} S_t & \text{falls} \quad i \geq 3 \end{cases}$$

und es gilt

$$\prod_{i=1}^{n} C^{(i)} = \begin{pmatrix} p_0 \\ \vdots \\ p_n \end{pmatrix}$$

Die Berechnung der Koeffizienten des charakteristischen Polynoms von A ist somit auf die Berechnung des obigen Produkts zurückgeführt worden. Der nächste Schritt muß also eine effiziente Berechnung der Einträge $R_t \cdot M_t^{i-3} \cdot S_t$ von $C^{(t)}$ sein.

Lemma 2.2.6 Seien $R \in (1 \times m;\ R)$, $S \in (m \times 1;\ R)$, $M \in (m \times m;\ R)$, $m < n$ und sei $T = \{R \cdot M^i \cdot S\}_{i=0}^{m}$. Dann liegt die Berechnung von T in $NC^2(n^{\alpha+\epsilon})$ für jedes $\epsilon > 0$, wobei n^α die Anzahl der benötigten Prozessoren ist, um die Matrizenmultiplikation in $O(\log n)$ Tiefe durchzuführen (zur Zeit ist $\alpha < 2.376$).

Beweis: Sei

$$U = \{R \cdot M^i\}_{i=0}^{n^{0.5}}, \qquad V = \{M^{j \cdot n^{0.5}} \cdot S\}_{j=0}^{n^{0.5}}$$

Jedes Element von T kann eindeutig als Skalarprodukt von Vektoren aus U und V berechnet werden, denn der Exponent k des Termes M in einem beliebigen Element von T kann eindeutig durch $k = i + j \cdot n^{0.5}$ mit $i, j < n^{0.5}$ dargestellt werden. T kann aus U und V durch n Skalarprodukte der Größe $n-1$ berechnet. Der Aufwand zur Berechnung eines solchen Skalarproduktes liegt in $\mathrm{NC}^1_{A[F]}(n)$, also kann T durch U und V in $NC^1_{A[F]}(n^2)$ berechnet werden.

Da die Berechnung von U und V äquivalent ist, geben wir nur die Konstruktion von U an.

Sei

$$Z_\beta = \{R \cdot M^i\}_{i=0}^{n^\beta}. \qquad (U = Z_{0.5})$$

Wir zeigen durch Induktion über β, daß sich Z_β mit $O(n^{\alpha+\epsilon})$ Prozessoren in $O(\log^2 n)$ Tiefe berechnen läßt.

Für $\beta - \epsilon = 0$ ist $Z_\epsilon = \{R \cdot M^i\}_{i=0}^{n^\epsilon}$ zu berechnen. Die Menge $\{M^i\}_{i=0}^{n^\epsilon}$ läßt sich durch binäres Potenzieren durch einen Baum der Tiefe $\epsilon \cdot \log n$ berechnen und damit sind zur Berechnung von Z_ϵ $O(n^\epsilon n^\alpha)$ Prozessoren und Tiefe $O(\log^2 n)$ ausreichend.

Falls die Aussage für $Z_{\beta-\epsilon}$ richtig ist, so folgt die Aussage für Z_β durch folgende Überlegung:

Z_β läßt sich aus $Z_{\beta-\epsilon}$ und $Y_\beta = \{M^{j \cdot n^{\beta-\epsilon}}\}_{j=0}^{n^\epsilon}$ berechnen, denn $Z_\beta = \{Z_{\beta-\epsilon} \cdot y \mid y \in Y_\beta\}$. Nach Induktionsanfang kann Y_β durch $O(n^{\alpha+\epsilon})$ Prozessoren in $O(\log^2 n)$ Tiefe berechnet, die gleiche Aussage gilt auch für $Z_{\beta-\epsilon}$ nach Induktionsvoraussetzung. Damit ergibt sich als Aufwand zur Berechnung von Z_β auch $O(n^{\alpha+\epsilon})$ Prozessoren und $O(\log^2 n)$ Tiefe. □

Mit Hilfe von Lemma 2.2.6 können wir nun das Hauptresultat beweisen:

Theorem 2.2.7 Sei $A \in (n \times n; R)$ und $p(\lambda)$ das charakteristische Polynom von A. Für jedes $\epsilon > 0$ können die Koeffizienten des charakteristischen Polynoms mit $O(n^{\alpha+1+\epsilon})$ Prozessoren in $O(\log^2 n)$ Tiefe berechnet werden.

Beweis: Aus den obigen Betrachtungen ergab sich

$$\prod_{i=1}^{n} C^{(i)} = \begin{pmatrix} p_0 \\ \vdots \\ p_n \end{pmatrix}.$$

Nach Lemma 2.2.6 kann jedes $C^{(i)}$, $1 \leq i \leq n$ in $\mathrm{NC}^2_{A[F]}(n^{\alpha+\epsilon})$ berechnet werden. Damit liegt die Berechnung der Menge $\{C^{(i)}\}_{i=1}^n$ in $\mathrm{NC}^2_{A[F]}(n^{\alpha+1+\epsilon})$. Durch binäres Multiplizieren und mit Hilfe von Lemma 2.2.5 läßt sich die Berechnung des Produktes $\prod_{i=1}^n C^{(i)}$ in $\mathrm{NC}^2_{A[F]}(n^3)$ durchführen. □

Korollar 2.2.8

$$\begin{aligned} \mathrm{DET} &\in \mathrm{NC}^2_{A[F]}(n^{\alpha+1+\epsilon}), \\ \mathrm{ADJ} &\in \mathrm{NC}^2_{A[F]}(n^{\alpha+1+\epsilon}). \end{aligned}$$

Beweis: Nach Lemma 2.2.4 gilt

$$\mathrm{adj}\,(A) = -(p_0 A^{n-1} + \cdots + p_{n-2}A + p_{n-1}E_n).$$

Die Menge $\{A^i\}$ läßt sich in $\mathrm{NC}^2_{A[F]}(n^{\alpha+1})$ berechnen. □

2.2.2 Paralleler GCD-Algorithmen für Polynome

Borodin, von zur Gathen und Hopcroft ([BGH 82]) geben einen effizienten, parallelen Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier Polynome über einem beliebigen Körper an.

Theorem 2.2.9 Sei F ein beliebiger Körper. Dann läßt sich $\gcd(f,g)$ für $f,g \in F[x]$ mit $\deg f = m \leq n = \deg g$ mit $O(n^{4.5})$ Prozessoren in $O(\log^2 n)$ Tiefe berechnen.

Beweis: Die Berechnung von $\gcd(f,g)$ wird auf die Berechnung einer Determinante zurückgeführt. Sei $h = \gcd(f,g)$ der normierte größte gemeinsame Teiler von f und g und $\deg h = d$. Dann existieren Polynome $u, v \in F[x]$ mit $h = uf + vg$ und $\deg u < n - d$, $\deg v < m - d$.

Für jedes $k \geq 0$ und alle Polynome $s = \sum s_i x^i$ und $t = \sum t_i x^i$, läßt sich die Bedingung

“$sf + tg$ ist normiert vom Grad k und $\deg s < n - k$, $\deg t < m - k$”

in ein $(n+m-2k) \times (n+m-2k)$ Gleichungssystem S_k übersetzen:

$$\begin{pmatrix}
f_m &&&&&& g_n &&&&& \\
f_{m-1} & f_m &&&&& g_{n-1} & g_n &&&& \\
&&&&&&&&&& \ddots & \\
\vdots &&& \ddots &&& \vdots &&&&& g_n \\
&&&&& f_m & \vdots &&&&& \\
f_0 &&&&&&&&&&& \vdots \\
&&&&& \vdots & g_0 &&&&& \vdots \\
&& \ddots &&&&&& \ddots &&& \\
&&& f_0 & \cdots & f_k &&&& g_0 & \cdots & g_k
\end{pmatrix} \cdot \begin{pmatrix} s_{n-k-1} \\ \\ \vdots \\ \\ s_0 \\ t_{m-k-1} \\ \\ \vdots \\ \\ t_0 \end{pmatrix} = \begin{pmatrix} 0 \\ \\ \vdots \\ \vdots \\ \\ \\ \\ 0 \\ 1 \end{pmatrix}$$

Sei A_k die Koeffizientenmatrix zu S_k. Es ergibt sich der folgende Algorithmus zur Berechnung von $\gcd(f,g)$:

Algorithmus 2.2.10 GCD-Algorithmus für Polynome

Eingabe:	Koeffizientenvektoren der Polynome $f(x)$ und $g(x)$.
Schritt 1:	Berechne parallel $a_0, \ldots, a_m$, wobei $a_k = \det(A_k)$ und A_k wie oben definiert.
Schritt 2:	Setze $d := \min\{k \mid a_k \neq 0\}$.
Schritt 3:	Berechne die Lösung (s,t) von S_k (A_k ist nichtsingulär).
Ausgabe:	$\gcd(f,g) := sf + tg$.

Damit ist die Berechnung von $\gcd(f,g)$ auf die parallele Berechnung von $O(n)$ Determinanten von Matrizen der Größe maximal $2n$ zurückgeführt. Mit Korollar 2.2.8 ergibt sich GCD $\in NC^2_{A[F]}(n^{\alpha+2+\epsilon})$. □

2.2.3 Parallele Berechnung des Ranges

In diesem Abschnitt soll der Algorithmus von Mulmuley ([Mu 87]) zur parallelen Berechnung des Ranges einer Matrix über einem *beliebigen* Körper F vorgestellt werden.

Zunächst geben wir einige grundlegende Begriffe der linearen Algebra an (vgl. z.B. [Fi 82]).

Sei V ein Vektorraum über F und ϕ ein Endomorphismus über V

$$\phi : V \longrightarrow V.$$

Sei $\lambda \in F$ Eigenwert von ϕ, so definieren wir

$$\mathrm{Eig}(\phi;\lambda) := \{v \in V \ : \ \phi(v) = \lambda v\} = \mathrm{Ker}(\phi - \lambda \cdot \mathrm{id}_V) \subset V$$

als den *Eigenraum* von ϕ zum Eigenwert λ und

$$\mathrm{Hau}(\phi;\lambda) := \bigcup_{s=1}^{\infty} \mathrm{Ker}(\phi - \lambda \cdot \mathrm{id}_V)^s \subset V$$

als den *Hauptraum* von ϕ zum Eigenwert λ.

Sei

$$P_\phi := \det(\phi - t \cdot \mathrm{id}_V) \in F[t]$$

das *charakteristische Polynom* von ϕ.

Ein Endomorphismus ist trigonalisierbar (d.h. es gibt eine Basis $\mathcal{B}$ von V derart, daß $M_\mathcal{B}(\phi)$ eine obere Dreiecksmatrix ist) genau dann, wenn das charakteristische Polynom von ϕ über K in Linearfaktoren zerfällt. In diesem Fall kann $\mathcal{B}$ so gewählt werden, daß $M_\mathcal{B}(\phi)$ Jordansche Normalform hat. Grundlegend für den Beweis dieser Tatsache ist der Satz über die Zerlegung in Haupträume, den wir auch zur Berechnung des Ranges heranziehen werden.

Satz 2.2.11 Zerlegung in Haupträume
Sei ϕ ein Endomorphismus eines F-Vektorraumes V derart, daß das charakteristische Polynom P_ϕ in Linearfaktoren zerfällt, d.h.

$$P_\phi = \pm(t-\lambda_1)^{r_1} \cdot \dots \cdot (t-\lambda_k)^{r_k}$$

mit paarweise verschiedenen $\lambda_1, \dots, \lambda_k \in F$. Seien $W_i = \mathrm{Hau}(\phi; \lambda_i)$ die Haupträume von ϕ zum Eigenwert λ_i, $1 \le i \le k$.

Dann hat man eine direkte Summenzerlegung

$$V = W_1 \oplus \dots \oplus W_k$$

und es gilt für $i = 1, \dots, k$

a) $W_i = \mathrm{Ker}(\phi - \lambda_i \cdot \mathrm{id}_V)^{r_i}$,
b) $\dim W_i = r_i$,
c) $\phi(W_i) \subset W_i$,
d) $P_{\phi/W_i} = \pm(t-\lambda_i)^{r_i}$,
e) $(\phi/W_i - \lambda_i \cdot \mathrm{id}_{W_i})^{r_i} = 0$.

Wir wenden uns nun der Berechnung des Ranges einer Matrix $A \in (n \times n; F)$ zu. Sei $P(t)$ das charakteristische Polynom von A und $K(A)$ der Hauptraum von A zum Eigenwert 0,

$$K(A) = \bigcup_{s=1}^{\infty} \mathrm{Ker}(A^s).$$

Sei m die Vielfachheit von 0 in $P(t)$, d.h. m ist die größte natürliche Zahl, so daß t^m das Polynom $P(t)$ teilt.

Betrachte den algebraischen Abschluß von F. Sowohl die Dimension von $K(A)$ als auch das charakteristische Polynom sind invariant unter dieser Erweiterung. Im algebraischen Abschluß läßt sich demnach Theorem 2.2.11 anwenden, und es folgt, daß

$$m = \dim(K(A))$$

gilt. Die Dimension des Hauptraumes läßt sich also durch Berechnung des charakteristischen Polynoms (im Grundkörper) einfach ermitteln.

Zur Ermittlung des Ranges (oder äquivalent des Defektes) der Matrix A ist jedoch die Berechnung der Dimension des Nullraumes $\dim(\mathrm{Ker}(A))$ erforderlich. Falls die Dimension des Hauptraumes $K(A)$ und die Dimension des Nullraumes übereinstimmen, so kann der Rang von A leicht berechnet werden:

$$n - \mathrm{rg}(A) = \mathrm{defekt}(A) = \dim(\mathrm{Ker}(A)) = \dim(K(A)) = m.$$

Im Allgemeinen kann man jedoch nicht davon ausgehen, daß die Dimension des Hauptraumes $K(A)$ und die Dimension des Nullraumes übereinstimmen, allerdings sind wir in der Lage die Matrix A so zu transformieren, daß diese Aussage stets erfüllt ist und der Rang von A unter dieser Transformation erhalten bleibt.

Zunächst einmal können wir davon ausgehen, daß A quadratisch und symmetrisch ist. Anderenfalls betrachte die Matrix

$$\begin{pmatrix} 0 & A \\ A^{\mathrm{t}} & 0 \end{pmatrix},$$

deren Rang der doppelte Rang der ursprünglichen Matrix ist.

Weiterhin ist der Rang von A invariant unter Erweiterung des Grundkörpers. Man hat also die Freiheit F soweit zu erweitern, solange die Arithmetik im Erweiterungskörper nicht zu teuer wird. Wir erweitern F durch Adjunktion eines transzendenten Elements x, sei also $G = F(x)$. G ist isomorph zum Körper der rationalen Funktionen in der Unbestimmten x.

Sei $C \in (n \times n; G)$ definiert durch

$$C = X \cdot A,$$

wobei $X = (x_{ij})$ eine Diagonalmatrix mit $x_{ii} = x^{i-1}, 1 \leq i \leq n$ ist. Da X nichtsingulär ist, gilt $\mathrm{rg}(A) = \mathrm{rg}(C)$. Mit Hilfe des folgenden Lemmas 2.2.12 werden wir zeigen, daß für die Matrix C die Dimension des Hauptraumes $K(C)$ mit der Dimension des Nullraumes von C übereinstimmt. Mit den obigen Bemerkungen läßt sich dann der Rang von A aus dem charakteristischen Polynom von C berechnen.

Lemma 2.2.12 $\quad \mathrm{rg}(C \cdot C) = \mathrm{rg}(C)$.

Beweis: Es ist nur zu zeigen, daß $\mathrm{rg}(AXA) = \mathrm{rg}(A)$ ist, denn dann gilt

$$\mathrm{rg}(CC) = \mathrm{rg}(XAXA) = \mathrm{rg}(AXA) = \mathrm{rg}(A) = \mathrm{rg}(C).$$

Trivialerweise gilt $\mathrm{rg}(AXA) \leq \mathrm{rg}(A)$. Zu zeigen bleibt, daß auch $\mathrm{rg}(AXA) \geq \mathrm{rg}(A)$ gilt.

Falls $\mathrm{rg}(AXA) \geq \mathrm{rg}(A)$, so folgt aus $AXAu(x) = 0$, daß $Au(x) = 0$ gilt, wobei $u(x)$ ein Vektor mit rationalen Funktionen als Einträgen ist. Angenommen, $v(x) = Au(x) \neq 0$ für einen Vektor $u(x)$ mit Polynomen in x als Einträgen. Dann ist

$$\begin{aligned} 0 &= AXAu(x) \\ &= u^{\mathrm{t}}(z)AXAu(x) \\ &= u^{\mathrm{t}}(z)A^{\mathrm{t}}XAu(x) \\ &= v^{\mathrm{t}}(z)Xv(x) \\ &= \sum_{i=1}^{n} v_i(z)v_i(x)x^{i-1} \end{aligned}$$

Sei m_i der Grad von $v_i(x)$ und $m = \max\{m_i\}$. Sei k der maximale Index, sodaß $m_k = m$ ist. Betrachte dann das Monom $z^{m_k}x^{m_k}x^{k-1} = z^{m_k}x^{m_k+k-1}$ von $\sum_{i=1}^n v_i(z)v_i(x)x^{i-1}$. Dieses Monom kann sich nicht wegheben, da die Potenz m_k von z maximal ist, und unter allen Monomen mit Anteil z^{m_k} die Potenz $m_k + k - 1$ von x maximal und einmalig ist. Damit gilt widersprüchlich

$$\sum_{i=1}^{n} v_i(z)v_i(x)x^{i-1} \neq 0,$$

also ist $\mathrm{rg}\,(AXA) \geq \mathrm{rg}\,(A)$ und damit folgt die Aussage des Lemmas. □

Betrachtet man C als Endomorphismus von $V = G^n$, folgt aus Lemma 2.2.12, daß

$$\mathrm{Ker}\,(C) \cap C(V) = \{0\}.$$

Sei $x \in \mathrm{Ker}\,(C) \cap C(V)$ und $x = C(y)$. y existiert, da $x \in C(V)$. Angenommen, $x \neq 0$, dann ist $y \notin \mathrm{Ker}\,(C)$, aber $y \in \mathrm{Ker}\,(CC)$. Dies ist nach Lemma 2.2.12 ein Widerspruch, also ist $x = 0$.

Damit läßt sich V als direkte Summe

$$V = \mathrm{Ker}\,(C) \oplus C(V)$$

darstellen und die Restriktion $C/C(V)$ von C auf $C(V)$ ist ein Automorphismus. Das bedeutet, daß $\mathrm{Ker}\,(C^s) = \mathrm{Ker}\,(C)$ für jedes s, also ist $\mathrm{Ker}\,(C) = K(C)$.

Damit ergibt sich der folgende Algorithmus zur Berechnung des Ranges einer symmetrischen $n \times n$ Matrix A über einem beliebigen Körper F:

Algorithmus 2.2.13 Berechnung des Ranges einer Matrix

Eingabe: Matrix A über einem beliebigem Körper F.

Schritt 1: Bilde $X = (x_{ij})$ mit $x_{ii} = x^{i-1}$ und $x_{ij} = 0$ für $i \neq j$.

Schritt 2: Berechne $Q(t) = \det\,(XA - tE_n)$ über $F(x)$.

Ausgabe: $n - m$, wobei m die Vielfachheit von 0 in $Q(t)$ ist.

Borodin, Cook und Pippenger zeigen, daß sich die Determinante aus Schritt 2) in $O(\log^2(n))$ Tiefe mit $O(n^{4.5})$ Prozessoren berechnen läßt.

Zusammenfassend ergibt sich also

Satz 2.2.14 Die Berechnung des Ranges einer Matrix über einem beliebigen Körper liegt in $\mathrm{NC}^2(n^{4.5})$. □

2.3 Perfekt Matching Probleme

Nun wollen wir uns dem zweiten wichtigen Problemkreis zuwenden. Dies sind Algorithmen zur Konstruktion von **Perfekten Matchings** und der **Interpolation von Polynomen**. Wir werden sehen, daß diese beiden Probleme sehr eng miteinander verwandt sind und auf die

Berechnung von Determinanten zurückgeführt werden können. Da viele Probleme insbesondere in der Graphentheorie auf das Perfekt Matching Problem reduziert werden können, ist dies ein Engpaß im Entwurf paralleler Algorithmen.

Definition 2.3.1 Matching
Gegeben sei ein Graph $G = (V, E)$. Eine Teilmenge $M \subseteq E$ heißt

- Matching, falls keine zwei Kanten in M inzident sind;
- nichterweiterbares oder auch maximales Matching, falls es kein Matching gibt, das M echt enthält;
- Maximum Matching, falls M ein Matching maximaler Kardinalität ist;
- Perfektes Matching, falls jeder Knoten aus V mit genau einer Kante aus M adjazent ist.

Definition 2.3.2 Determinante, Permanente
Gegeben sei eine Matrix $A = (a_{i,j})_{1 \leq i,j \leq n}$. Dann sind die **Determinante** und **Permanente** von A wie folgt definiert:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$\operatorname{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

Falls $A \in \mathbb{B}^{n \times n}$ kann auch die **logische Permanente** von A definiert werden:

$$\operatorname{perm}^L(A) = \bigvee_{\sigma \in S_n} \bigwedge_{i=1}^{n} a_{i,\sigma(i)}.$$

Nun können wir die verschiedenen Probleme, die mit dem Perfekt Matching Problem in Zusammenhang stehen, aufzählen. Sei $G = (V, E)$ ein Graph.

1. **Entscheidungsproblem**: Besitzt G ein Perfektes Matching ?
2. **Konstruktionsproblem**: Konstruiere ein (beliebiges) Perfektes Matching von G.
3. **Berechnungsproblem**: Berechne die Anzahl verschiedener Perfekter Matchings von G.
4. **Aufzählungsproblem**: Konstruiere alle Perfekten Matchings von G.

Das einfachste dieser Probleme ist das Entscheidungsproblem. Um dieses Problem zu lösen hilft der folgende Satz von Tutte. Zunächst jedoch eine

Definition 2.3.3 Tutte'sche Matrix
Sei $G = (V, E)$ ein Graph mit Adjazenzmatrix A. Zu G sei nun die folgende schiefsymmetrische Matrix (Tutte'sche Matrix) C in den Variablen x_{ij} definiert:

$$c_{ij} = \begin{cases} x_{ij} & i < j \text{ und } a_{ij} = 1 \\ -x_{ij} & i > j \text{ und } a_{ij} = 1 \\ 0 & i = j \text{ oder } a_{ij} = 0. \end{cases}$$

Man beachte, daß $\det(C)$ ein Polynom in den Variablen x_{ij} ist.

Satz 2.3.4 (Tutte, 1952)
Ein ungerichteter Graph $G = (V, E)$ mit Adjazenzmatrix A besitzt genau dann ein Perfektes Matching, wenn $\det(C) \not\equiv 0$ ist, d.h. $\det(C)$ enthält mindestens ein Monom.

BEWEIS: Falls G ein Perfektes Matching besitzt, so induziert die zugehörige Permutation ein Monom in der Determinante der Tutte'schen Matrix, welches sich nicht wegheben kann, da die Permutation aus 2-Zyklen besteht. Somit ist die Determinante nicht identisch 0 und die Tutte'sche Matrix regulär.
Welche weiteren Permutationen können nun Monome in der Tutte'schen Matrix induzieren? Eine Permutation, die einen ungeraden Zyklus enthält, wird durch die Permutation, die diesen Zyklus in entgegengesetzter Richtung durchläuft und sonst gleich ist, wieder gelöscht. Damit können keine Permutationen mit ungeraden Zyklen Monome erzeugen. Falls es Permutationen gibt, die nur gerade Zyklen enthalten, so gibt es auch Permutationen die nur 2-Zyklen enthalten. Wenn die Tutte'sche Matrix also regulär ist, so gibt es in der Determinante der Tutte'schen Matrix ein Monom, das von einem Perfekten Matching induziert ist, und damit auch ein Perfektes Matching. □

Mit Hilfe dieses Satzes können wir das Entscheidungsproblem auf den Nulltest für ein multivariates Polynom zurückführen.

Definition 2.3.5 Das Interpolationsproblem, d.h. der Test, ob ein als Black-Box gegebenes n-variates Polynom $P(x_1, \ldots, x_n)$ mit $\deg(P) \leq m$ identisch 0 ist, sei mit $\mathrm{IP}(n, m)$ bezeichnet.

Damit gilt mit Algorithmus 1.6.31:

$$\mathrm{IP}(n, m) \in \mathrm{R}_S\mathrm{TIME}((n+m)^{O(1)})$$

$$\mathrm{IP}(n, m) \in \mathrm{R}_S\mathrm{SPACE}(\log(n+m))$$

$$\mathrm{IP}(n, m) \in \text{Black-Box } \mathrm{RNC}^1$$

Wir können nun den effizienten parallelen Algorithmus von Berkowitz aus dem vorigen Abschnitt dazu verwenden, die Determinante der Tutte'schen Matrix in NC^2 zu berechnen. Daraus ergibt sich, daß das Entscheidungsproblem in RNC^2 liegt.

Leider ist kein nicht probabilistischer Algorithmus bekannt, der das Perfekt Matching Problem in NC löst.

Aus diesem Grund werden wir im folgenden nur eine spezielle Klassen von Graphen (bipartite Graphen mit polynomiell beschränkter Permanente) betrachten, und für diese das Perfekte Matching Problem in NC lösen. Desweiteren werden wir eine Lösung des Perfekt Matching Problems für beliebige Graphen entwickeln, die in RNC liegt. Die erste Arbeit ist von Grigoriev und Karpinski (siehe [GK 87]), die zweite von Mulmeley, U.Vazirani und V.Vazirani (siehe [MVV 87]).

2.3.1 Perfektes Matching in speziellen bipartiten Graphen

Sei $G = (V, E)$ ein bipartiter Graph, d.h. $V = V_1 \cup V_2$, so daß alle Kanten Knoten aus V_1 und V_2 verbinden. Wir wollen in solchen Graphen Perfekte Matchings betrachten. Perfekte Matchings können nur existieren, falls $|V_1| = |V_2|$ gilt. In diesem Fall ist die sogenannte bipartite Adjazenzmatrix eine quadratische Matrix. Sie ist eine Teilmatrix der vollständigen Adjazenzmatrix und ist wie folgt definiert:

$$A = (a_{ij}) \text{ mit } a_{ij} = 1 \;\Leftrightarrow\; (v_{1,i}, v_{2,j}) \in E.$$

Im Zusammenhang mit bipartiten Graphen verwenden wir im folgenden nur bipartite Adjazenzmatrizen. Es gilt nun das folgende

Lemma 2.3.6 Sei $G = (V, E)$ ein bipartiter Graph. Dann ist der Wert der Permanente der Adjazenzmatrix von G gleich der Anzahl der (verschiedenen) Perfekten Matchings in G. Insbesondere gilt

$$\operatorname{perm}^L(A) = 1 \;\Leftrightarrow\; G \text{ besitzt Perfektes Matching.}$$

BEWEIS: Sei σ eine Permutation mit $a_{i,\sigma(i)} = 1$ für alle $1 \le i \le n$. Dann induziert diese Permutation sowohl ein Perfektes Matching in G als auch einen positiven Beitrag für den Wert der Permanente von A. □

Definition 2.3.7 Zu der Adjazenzmatrix A definieren wir die **Variablenmatrix** C mit

$$c_{ij} := a_{ij} x_{ij}$$

in den Variablen x_{ij}.

Nun können wir analog zum Beweis von Lemma 2.3.6 auch einsehen, daß die Anzahl von Termen in $\det(C)$ genau dem Wert der Permanente $\operatorname{perm}(A)$ entspricht. Insbesondere gilt:

$$\det(C) \equiv 0 \iff G \text{ besitzt kein Perfektes Matching.}$$

Definition 2.3.8 Sparsity
Sei P ein Polynom. Die **Sparsity** von P ist definiert als die Anzahl von Termen in P.
Ein Polynom Q heißt **t-sparse**, falls die Anzahl von Termen in Q kleiner gleich t ist.

Hiermit gilt:

$$\operatorname{perm}(A) = \text{Sparsity}(\det(C)) = \#\text{ Perfekter Matchings.}$$

Diese Tatsache hat eine wichtige Auswirkung auf unser Problem. Falls die Permanente polynomiell beschränkt ist, so ist auch die Anzahl der Terme in $\det(C)$ polynomiell beschränkt. Damit können wir das Entscheidungsproblem auf das folgende Problem transformieren.

Definition 2.3.9 Sparse Interpolationsproblem
Gegeben sei eine Black-Box für ein t-sparses Polynom $P(x_1, \ldots, x_n)$ über $\mathbb{Z}$. Ist $P \equiv 0$?

Für unsere spätere Anwendung wird das t-sparse Polynom die t-sparse Determinante der Variablenmatrix sein. Von Grigoriev und Karpinski stammt der folgende Algorithmus, der das **Sparse Interpolationsproblem** löst.

Algorithmus 2.3.10 ([GK 87])

Eingabe: Black-Box für t-sparses Polynom $q(x_1, \ldots, x_n)$.

Schritt 1: Konstruiere die ersten n Primzahlen $p_1, \ldots, p_n$.

Schritt 2: Berechne $\alpha_i = q(p_1^i, \ldots, p_n^i)$ für alle $i = 0, \ldots, t-1$.

Ausgabe: $\begin{cases} q \equiv 0 & \text{falls } \alpha_i = 0 \text{ für alle } 0 \leq i < t \\ q \not\equiv 0 & \text{sonst.} \end{cases}$

Satz 2.3.11 Der Algorithmus 2.3.10 arbeitet korrekt.

Beweis: Sei das t-sparse Polynom q folgendermaßen dargestellt:

$$q = \sum_{k=1}^{t} c_k T_k(x_1, \ldots, x_n)$$

wobei T_k die Monome und c_k die zugehörigen Koeffizienten seien. Da sich verschiedene Monome nicht gegenseitig wegheben können, ist q genau dann $\equiv 0$, wenn $c_k = 0$ für $1 \leq k \leq t$. Sei nun

$$\Omega_k := T_k(p_1, \ldots, p_n)$$

der Wert des k-ten Monoms ausgewertet an der Stelle $(p_1, \ldots, p_n)$, wobei p_i die i-te Primzahl sei. Aufgrund der Eindeutigkeit der Primzahlenzerlegung gilt

$$\Omega_k = \Omega_{k'}$$

nur falls $k = k'$ ist. Somit ist der Wert Ω_k eine eindeutige Kodierung des k-ten Monoms von q. Da Ω_k die Kodierung *eines* Monoms ist, gilt zusätzlich für alle $i \geq 1$

$$\Omega_k^i = T_k(p_1^i, \ldots, p_n^i).$$

Sei nun α_i der Funktionswert von q ausgewertet an der Stelle $(p_1^i, \ldots, p_n^i)$, also

$$\alpha_i = \sum_{k=1}^{t} c_k \Omega_k^i \text{ für } 0 \leq i < t.$$

Dieses lineare Gleichungssystem läßt sich auch in Matrixform schreiben:

$$\left(\Omega_k^i\right) \cdot (c_k) = (\alpha_i) \quad 1 \leq k \leq t,\ 0 \leq i < t.$$

Die Matrix (Ω_k^i) ist eine sogenannte *Vandermonde'sche* Matrix. Eine Matrix dieser Art ist genau dann regulär, wenn alle Ω_k paarweise verschieden sind. Genauer gilt:

$$\det\left(\Omega_k^i\right) = \prod_{j > j'} (\Omega_j - \Omega_{j'}).$$

Somit ist das Gleichungssystem eindeutig lösbar. Falls für alle i $\alpha_i = 0$ gilt, so ist die eindeutige Lösung

$$c_k = 0 \text{ für alle } k.$$

Damit ist auch $q \equiv 0$. Es gilt also:

$$q \equiv 0 \;\Leftrightarrow\; \alpha_i = 0 \text{ für alle } 0 \le i < t.$$

□

Da die Berechnung von Determinanten in polynomieller Zeit und sogar in $NC^2(n^{4.5})$ möglich ist, haben wir somit einen NC-Algorithmus für das Entscheidungsproblem auf der Menge der bipartiten Graphen mit polynomiell beschränkter Permanente.

Satz 2.3.12 Sei $G = (V, E)$ ein bipartiter Graph mit $\operatorname{perm}(A) \le cn^k$, wobei A die bipartite Adjazenzmatrix ist. Dann liegt das **Entscheidungsproblem**, d.h. der Test, ob G ein Perfektes Matching besitzt, in $NC^2(n^{k+4.5}\log n)$.

Nun möchten wir uns dem **Konstruktionsproblem** zuwenden. Zunächst eine vorbereitende Definition.

Definition 2.3.13 Erzeuger
Das Paar (i, j) heißt **Erzeuger**, falls $a_{i,j} = 1$ und

$$\operatorname{perm}(A(i|j)) \neq 0$$

gilt. In diesem Fall gibt es *mindestens ein* Perfektes Matching, welches die Kante (i, j) benutzt.

Definition 2.3.14 Erzeugerzeile
Eine Zeile i, die mindestens zwei Erzeuger (i, j_1) und (i, j_2) enthält, heißt Erzeugerzeile (row generator).

Mit Hilfe dieser Definitionen können wir einen Konstruktionsalgorithmus mittels der folgenden Rekursion aufbauen:

- Falls keine Erzeugerzeile existiert, ist das Perfekte Matching in jeder Zeile eindeutig und ist somit eindeutig bestimmt. Dieses Perfekte Matching kann dann leicht mit Hilfe von Lemma 2.3.16 berechnet werden.
- Falls es mindestens eine Erzeugerzeile mit Erzeugern (i, j_1) und (i, j_2) gibt, so können parallel die Matrizen

  $$A(i|j_1) \text{ und } A(i|j_2)$$

  rekursiv weiteruntersucht werden. Diese Matrizen spiegeln den Graphen G'_j nach Entfernen des entsprechenden Knotenpaares wieder. Ein gefundenes Perfektes Matching in G'_j wird zu einem Perfekten Matching von G erweitert, indem die beiden Knoten i und j_1 bzw. j_2 mit der sie verbindenden Kante (i, j_1) bzw. (i, j_2) hinzugenommen werden. Nun ist die Summe der Perfekten Matchings in G'_{j_1} und G'_{j_2} höchstens so groß wie die Anzahl der Perfekten Matchings in G. Daher enthält einer der beiden Graphen höchstens die Hälfte der Perfekten Matchings von G.

Es genügt also eine logarithmische Rekursionstiefe bis ein eindeutiges Perfektes Matching gefunden ist. Da es höchstens $O(n^k)$ Perfekte Matchings gibt müssen insgesamt auch höchstens $O(n^k)$ Entscheidungsprobleme gelöst werden. Daher ergibt sich der folgende

Satz 2.3.15 Falls $\text{perm}\,(A) \leq cn^k$ ist, dann arbeitet der Konstruktionsalgorithmus korrekt in $\text{NC}^3(n^{2k+4.5} \log n)$

Lemma 2.3.16 Falls das Perfekte Matching eindeutig ist, kann es in NC^2 konstruiert werden.

BEWEIS: Falls das Perfekte Matching eindeutig ist, so enthält die Determinante der Variablenmatrix C nur einen einzigen Term. Falls wir die Variablen mit Primzahlen belegen, ist der Wert der Determinante ein Produkt von n dieser Primzahlen. Wir müssen also nur die Determinante auf Teilbarkeit durch die eingesetzten Primzahlen überprüfen, um die richtigen Kanten auszuwählen, d.h.

$$p_i | \det\,(C(p_1, \ldots, p_m)) \;\Leftrightarrow\; \text{zugehörige Kante gehört zum Perfekten Matching.}$$

□

Nun wollen wir als letztes noch das **Aufzählungsproblem** angehen, d.h wir wollen alle Perfekten Matchings berechnen. Damit erschlagen wir auch das **Berechnungsproblem**.

Dazu verallgemeinern wir zunächst den Begriff des Erzeugers.

Definition 2.3.17 Aktive Menge
Eine Menge $M = \{a_{i_1 j_1}, a_{i_2 j_2}, \ldots, a_{i_r j_r}\}$ mit $a_{i_k, j_k} = 1$ für $1 \leq k \leq r$ heißt **aktiv**, falls es ein Perfektes Matching gibt, welches **alle** Kanten aus M enthält, d.h.

$$\text{perm}\,(A(i_1|j_1, i_2, |j_2, \ldots, i_r|j_r)) \neq 0$$

ist. Aktive Mengen sind also Teillösungen zu Perfekten Matchings.

Damit sind die einelementigen aktiven Mengen gerade Erzeuger und die n-elementigen aktiven Mengen die Perfekten Matchings.

Wir wollen nun aus kleineren aktiven Mengen größere aktive Mengen induktiv aufbauen. Der Induktionsanfang besteht aus unseren Erzeugermengen. Diese sind mit Hilfe des Entscheidungstests konstruierbar.

Der Induktionsschritt sieht wie folgt aus: Aktive Mengen der Kardinalität 2^m wollen wir zu aktiven Mengen der Kardinalität 2^{m+1} zusammenfassen. Dadurch erhalten wir eine logarithmische Induktionstiefe. Der einfacheren Beschreibung halber sei die folgende Definition gedacht.

Definition 2.3.18 Mit $S_{\alpha,\beta}$ bezeichen wir die Menge aller Teillösungen der Größe $2^{\alpha-1}$, deren Elemente (dies sind Erzeuger) aus den Zeilen $\beta 2^{\alpha-1}+1$ bis $(\beta+1)2^{\alpha-1}$ der Adjazenzmatrix stammen, d.h.

$$S_{\alpha,\beta} := \{(j_1, \ldots, j_{2^{\alpha-1}}) | a_{\beta 2^{\alpha-1}+1, j_1}, \ldots, a_{(\beta+1)2^{\alpha-1}, j_{2^{\alpha-1}}} \text{ ist aktiv}\}.$$

Da die Anzahl von Teillösungen nicht größer als die Anzahl von Perfekten Matchings sein kann, folgt für alle α und β

$$|S_{\alpha,\beta}| \leq cn^k.$$

Satz 2.3.19 Falls $\text{perm}(A) \leq cn^k$ ist, dann liegt das Aufzählungsproblem in $\text{NC}^3(n^{3k+5.5}\log n)$.

Beweis: Wir können aus zwei benachbarten Teillösungsmengen $S_{\alpha,\beta}$ und $S_{\alpha,\beta+1}$ die Teillösungsmenge $S_{\alpha+1,\beta}$ wie folgt konstruieren. Zunächst ist

$$T_{\alpha+1,\beta} = \{(u,v)|u \in S_{\alpha,\beta},\ v \in S_{\alpha,\beta+1}\}$$

eine Obermenge von $S_{\alpha,\beta+1}$ mit Kardinalität $\leq c'n^{2k}$. Die Menge $T_{\alpha,\beta+1}$ ist also eine Menge möglicher Elemente für $S_{\alpha,\beta+1}$. Ob ein Element aus $T_{\alpha,\beta+1}$ wirklich zu $S_{\alpha,\beta+1}$ gehört, kann leicht mit Hilfe des Entscheidungstest für die gestrichene Adjazenzmatrix überprüft werden. Also werden zur Konstruktion der Menge $S_{\alpha,\beta}$ höchstens $O(n^{2k})$ Entscheidungstests benötigt. Der Gesamtaufwand zur Konstruktion von $S_{\alpha,\beta+1}$ beträgt daher $O(n^{3k+4.5}\log n)$ Prozessoren und eine Parallelzeit von $O(\log^2 n)$. Da höchstens n dieser Mengen $S_{\alpha,\beta}$ parallel in einem Rekursionsschritt erzeugt werden müssen und die Rekursionstiefe logarithmisch ist, liegt der gesamte Algorithmus in $NC^3(n^{3k+5.5}\log n)$. □

2.3.2 Randomisiertes Perfektes Matching

In diesem Abschnitt werden wir nun versuchen, randomisierte Lösungen für das allgemeine Perfekt Matching Problem, d.h. ein allgemeingültiges Verfahren für beliebige Graphen G, zu geben.

Das **Entscheidungsproblem** haben wir bereits gelöst, indem wir den randomisierten Nulltest für multivariate Polynome aus Algorithmus 1.6.31 angewandt auf die Determinante der Tutte'schen Matrix verwenden. Mittels Lemma 2.3.6 gilt ja, daß die Determinante der Tutte'schen Matrix genau dann identisch 0 ist, falls G kein Perfektes Matching besitzt.

Nun wollen wir das **Konstruktionsproblem** betrachten. Dazu brauchen wir jedoch einige Vorbereitungen.

Definition 2.3.20 Gewichtssystem
Eine **Gewichtssystem** GS ist ein Tripel (Q, F, w), wobei

- Q eine endliche Menge,
- F eine Familie von Teilmenge aus Q, d.h $F \subseteq \mathcal{P}(Q)$ und
- w eine Gewichtsfunktion $w : Q \to \mathbb{Z}$

sind.

Definition 2.3.21 Die Funktion $\bar{w} : \mathcal{P}(Q) \to \mathbb{Z}$ ist wie folgt definiert:

$$\bar{w}(S) := \sum_{x \in S} w(x).$$

Außerdem sei

$$\min(Q, F, w) := \{S \in F | \bar{w}(S) = \min\{\bar{w}(S') | S' \in F\}\}.$$

Wir nennen ein Gewichtssystem eindeutig, falls es unter den Mengen aus F eine eindeutige Menge S mit kleinstem Gewicht $\bar{w}(S)$ gibt, d.h.

$$\min(Q, F, w) = \{S\}.$$

Wir schreiben dann (falls S eindeutig)

$$S = \min(Q, F, w).$$

Das folgende wichtige Lemma ermöglicht uns, einen randomisierten Algorithmus für das Perfekt Matching Problem anzugeben.

Lemma 2.3.22 Gewichtslemma, Isolationslemma
Sei (Q, F, w) ein Gewichtssystem mit $|Q| = n$ und $w : Q \to [1, \ldots, 2n]$ eine randomisierte Funktion, die eine unabhängige Gleichverteilung induziert. Dann ist

$$Pr\{(Q, F, w) \text{ ist eindeutig}\} \geq 1/2$$

Beweis: Sei $Q = \{x_1, \ldots, x_n\}$ und seien $w(x_1) \ldots w(x_n)$ unabhängige und gleichverteilte Zufallsvariablen mit Wertebereich $[1, \ldots, 2n]$.

Seien für $1 \leq i \leq n$

$$W_i := \min\{\bar{w}(S) | S \in F, x_i \notin S\} \quad \text{und}$$
$$W_i' := \min\{\bar{w}(S \setminus \{x_i\}) | S \in F, x_i \in S\}.$$

Damit sei

$$\Delta_i := W_i - W_i'.$$

Man beachte, daß die Zufallsvariablen Δ_i stochastisch unabhängig von den Zufallsvariablen $w(x_i)$ sind. Es können nun die folgenden Fälle auftreten:

- falls $\forall S \in \min(Q, F, w) : \; x_i \notin S$, dann ist $\Delta_i < w(x_i)$
- falls $\forall S \in \min(Q, F, w) : \; x_i \in S$ ist, so ist $\Delta_i > w(x_i)$
- falls $\exists S, S' \in \min(Q, F, w)$ mit $x_i \in S, x_i \notin S'$, dann ist $\Delta_i = w(x_i)$.

Im letzten Fall, nämlich falls $\Delta_i = w(x_i)$ für ein $1 \leq i \leq n$ gilt, so ist das Gewichtssystem nicht eindeutig, da zwei verschiedene gewichtsminimale Mengen sich um das Element x_i unterscheiden. Die Wahrscheinlichkeit hierfür kann folgendermaßen abgeschätzt werden:

$$Pr\{\exists i \text{ mit } \Delta_i = w(x_i)\} \leq \sum_{i=1}^{n} Pr\{\Delta_i = w(x_i)\}.$$

Da die Zufallsvariablen $w(x_i)$ unabhängig von den Δ_i sind, gilt nun

$$Pr\{w(x_i) = \Delta_i\} = \frac{1}{2n}.$$

Also ist die Wahrscheinlichkeit, daß das Gewichtssystem eindeutig ist, größer gleich $1/2$. □

Wir wollen dieses Lemma nun auf unser Perfekt Matching Problem anwenden. Es ergibt sich also folgendes

Korollar 2.3.23 Sei $G = (V, E)$ ein Graph, $|V| = n$. Wir definieren das Gewichtssystem $GS = (E, PM, w)$ mit $PM = \{M \subseteq E | M$ ist Perfektes Matching in $G\}$ und w als randomisierte Funktion $w : E \to [1, \ldots, 2|E|]$. Dann ergibt sich aus Lemma 2.3.22

$$Pr\{(E, PM, w) \text{ ist eindeutig}\} \geq 1/2.$$

Mit Hilfe der randomisierten Funktion

$$w : E \to [1, \ldots, 2|E|]$$

aus Korollar 2.3.23 können wir nun eine gewichtete Adjazenzmatrix $A = (a_{i,j})$ wie folgt konstruieren.

Definition 2.3.24 Seien w_{ij} gleichverteilte Zufallsvariablen in $[1, \ldots, 2|E|]$. Dann sei die randomisierte Adjazenzmatrix A durch Einsetzen der $2^{w_{ij}}$ in die Tutte'sche Matrix definiert:

$$a_{i,j} = \begin{cases} 2^{w_{ij}} & \text{falls } i < j \text{ und } (i,j) \in E \\ -2^{w_{ji}} & \text{falls } i > j \text{ und } (i,j) \in E \\ 0 & \text{falls } i = j \text{ oder } (i,j) \notin E. \end{cases}$$

Mit Hilfe des Korollar 2.3.23 können wir nun Aussagen über diese Matrix machen.

Lemma 2.3.25 Gegeben sei ein Graph $G = (V, E)$ mit einer Gewichtsfunktion für die Kanten. Diese Gewichtsfunktion w sei so gewählt, daß das gewichtsminimale Perfekte Matching *eindeutig* ist. Falls wir eine geeignete randomisierte Gewichtsfunktion verwenden, ist die Wahrscheinlichkeit hierfür nach Korollar 2.3.23 $\geq 1/2$. Dann ist

$$\det(A) \neq 0$$

und für die größte Zweierpotenz 2^α, die $\det(A)$ teilt, gilt

$$\alpha = 2 \cdot \min\{\bar{w}(M) | M \in PM\}.$$

Beweis: Wir erinnern uns an die Definition der Determinante einer Matrix:

$$\det(A) = \sum_{\pi \in S_n} \text{sign}(\pi) \cdot \text{value}(\pi) \quad \text{mit}$$

$$\text{value}(\pi) = \prod_{i=1}^{n} a_{i,\pi(i)}.$$

Dabei tragen nur solche Permutationen π zum Wert der Determinante bei, für die value $(\pi) \neq 0$, für die $\forall i \ (i, \pi(i)) \in E$ gilt. Wir werden also nur die Permutationen betrachten, die einen Wert zur Determinante beitragen. Ein solcher Wert ist dann immer eine Zweierpotenz, da das Produkt von Zweierpotenzen wieder eine Zweierpotenz ist. Mit $w_{\min}$ bezeichnen wir das minimale Gewicht eines Perfekten Matchings. Für eine Permutation π gibt es die folgenden Möglichkeiten.

- Die Permutation π induziert ein Perfektes Matching.
  In diesem Fall bildet $\pi : i \mapsto j, j \mapsto i$ ab, d.h. es entstehen nur 2-Zyklen. Bei der Berechnung von value(π) trägt dieser 2-Zyklus mit dem Wert $2^{w_{ij}}(-2^{w_{ij}}) = -2^{2w_{ij}}$ zu Buche. Somit ist der Wert dieser Permutation
  $$\text{value}(\pi) = (-1)^{n/2} 2^{w_\pi},$$
  wobei w_π die Summe der Gewichte der zu π gehörenden Kanten entspricht. Falls π das eindeutige gewichtsminimale Perfekte Matching darstellt, so ist $w_\pi = 2w_{\min}$, da die Kanten von π doppelt gezählt werden. Für alle anderen Permutationen σ ist $w_\sigma > w_\pi = 2w_{\min}$ und somit 2^{w_σ} immer ein **gerades** Vielfaches von $2^{2w_{\min}}$.

- Die Permutation π enthält keine ungeraden Zyklen.
  Wir können die geraden Zyklen in zwei Perfekte Matchings einteilen, indem wir in den Zyklen alternierende Kanten betrachten. Wir spalten also diese Permutation in zwei Perfekte Matchings M_1 und M_2 auf. Aufgrund der Eindeutigkeit eines minimalgewichteten Matchings, kann höchstens eines dieser beiden Perfekten Matchings minimalgewichtig sein. Damit ist
  $$2^{w_\pi} = 2^{w_{M_1} + w_{M_2}} > 2^{2w_{\min}},$$
  und somit auch ein **gerades** Vielfaches von $2^{2w_{\min}}$.

- Die Permutation enthält auch ungerade Zyklen.
  In diesem Fall betrachten wir neben π auch die Permutation σ, die einen der ungeraden Zyklen in umgekehrter Richtung durchläuft. Wir sehen leicht ein, daß value$(\pi) = -$value(σ) und sign$(\pi) =$ sign(σ) gilt, da σ aus π durch Anwenden einer geraden Anzahl von Transpositionen hervorgeht. Damit heben sich diese beiden Permutationen in der Determinante auf.

Insgesamt erhalten wir als Summanden in der Determinante nur gerade Vielfache von $2^{2w_{\min}}$ und einen Summand von der Form $(-1)^{n/2} 2^{2w_{\min}}$. Daher ist
$$\det(A) = (-1)^{n/2} 2^{2w_{\min}} + 2c2^{2w_{\min}}.$$
Also ist $\det(A) \neq 0$ und die höchste $\det(A)$ teilende Zweierpotenz ist 2^α für $\alpha = 2w_{\min}$. □

Mit Hilfe dieses Lemmas können wir nun einen randomisierten Algorithmus angeben, der zu einem gegebenen Graphen ein Perfektes Matching konstruiert, falls ein solches existiert.

Algorithmus 2.3.26 Randomisiertes Perfektes Matching

Eingabe: $G = (V, E)$

Schritt 1: Konstruiere die Matrix A wie oben angegeben.

Schritt 2: Berechne $\det(A)$ und $\mathrm{adj}(A) = \left((-1)^{i+j} \det A(j|i)\right)_{i,j}$.

Schritt 3: Finde die größte Zahl w, so daß 2^{2w} $\det(A)$ teilt. Dies ist das Gewicht des gewichtsminimalen Perfekten Matchings.

Ausgabe: $M = \{(i,j) |\, \det(A(j|i)) \cdot 2^{w_{ij}} / 2^{2w} \text{ ist ungerade}\}$.

Satz 2.3.27 Die Menge M aus Algorithmus 2.3.26 ist das (minimale gewichtete) gesuchte Perfekte Matching von G.

Beweis: Wir möchten feststellen, ob die Kante (i,j) zum gewichtsminimalen Perfekten Matching gehört. Dazu betrachten wir alle Permutationen, die diese Kante beinhalten, d.h. alle π mit $\pi(i)=j$. Diese Permutationen tragen

$$D_i^j := \sum_{\pi,\pi(i)=j} \text{sign}(\pi)\text{value}(\pi) = 2^{w_{ij}}\det(A(i|j))$$

zu $\det(A)$ bei. Offensichtlich gilt

$$\det(A) = \sum_i D_i^j = \sum_j D_i^j$$

Für jeden Knoten i ist nur eine Kante (i,j) im Perfekten Matching. Daher müssen alle Permutationen, die diese Kante nicht beinhalten, ein gerades Vielfaches von $2^{2w_{\min}}$ zu $\det(A)$ beitragen. Also ist auch D_i^k für $k \neq j$ ein gerades Vielfaches von $2^{2w_{\min}}$.
Betrachten wir nun D_i^j. Nur ein Summand dieser Summe ist ein ungerades Vielfaches von $2^{2w_{\min}}$, nämlich gerade die Permutation zum gewichtsminimalen Perfekten Matching. Alle anderen Summanden sind wie oben gerade Vielfache von $2^{w_{\min}}$. Daher ist die Summe ein ungerades Vielfaches. Die Ausdrücke D_i^j sind leicht zu berechnen, da $\text{adj}(A)$ gegeben ist. □

Damit gilt nun folgendes

Korollar 2.3.28 Perfekt Matching $\in$ RDET.

Man beachte bei unserem Algorithmus jedoch den hohen Verbrauch an Zufallsinformation. Die Anzahl der benutzten Zufallsbits beträgt $n^2 \log n$.

Als nächstes wollen wir das Problem des Maximum Matchings lösen, indem wir dieses auf das Perfekt Matching Problem zurückführen.

Satz 2.3.29 Maximum Matching $\leq_{NC^1}$ Perfekt Matching

Beweis: Gegeben sei der Graph $G=(V,E)$. In einem Maximum Matching von G ist die Anzahl der nicht saturierten Knoten minimal. Daher konstruieren wir zu dem Graphen G die Graphenfamilie $G_k=(V_k,E_k)$ mit $G_k \supset G$ und suchen unter diesen Graphen den minimalen Graphen, der ein Perfektes Matching besitzt. Dieses Perfekte Matching in G_k korrespondiert dann zu einem Maximum Matching in G. Die Konstruktion der Graphen sieht nun wie folgt aus:

$$V_k = V \cup \{w_1,\ldots,w_k\}$$
$$E_k = E \cup \{(v,w_i) | v \in V,\ 1 \leq i \leq k\}.$$

Dabei konstruieren wir nur die Graphen G_k mit gerader Knotenmenge, da sonst kein Perfektes Matching möglich ist. Wenn G_k ein Perfektes Matching besitzt, so besitzen auch alle G_i mit $i>k$ ein Perfektes Matching. Unter allen diesen Graphen G_k, die ein Perfektes Matching besitzen (Entscheidungstest), suchen wir mit Hilfe der binären Suche den kleinsten Graphen . Zu diesem kleinsten Graphen G_l konstruieren wir nun ein Perfektes Matching. Dieses Perfekte Matching eingeschränkt auf G ist nun ein Maximum Matching, da die Anzahl der Kanten, die mit Knoten der Form w_i verbunden sind, in G_l kleinstmöglich war. □

Als nächstes wollen wir das Max-Flow Problem mit unär dargestellten Gewichten auf Perfect Matching reduzieren. Dazu zunächst einige Definitionen.

Definition 2.3.30 Transport-Netzwerk
Ein Transport-Netzwerk ist eine endlicher gerichteter Graph $T = (V, E, s, t)$ mit $s \in V$ als Quelle (source) und $t \in V$ als Senke (sink). Der Eingangsgrad von s und der Ausgangsgrad von t seien ohne Einschränkung 0. Ein Fluß von T ist eine Funktion $f : E \to I\!N$, so daß

$$\forall v \in V \setminus \{s, t\} \quad \sum_u f(u, v) = \sum_u f(v, u),$$

d.h. für alle Knoten gilt das Kirchhoff'sche Gesetz.

Definition 2.3.31 Transport-Netzwerk mit Kapazitäten
Ein Transport-Netzwerk mit Kapazitäten (TNC) ist ein Tupel

$$C = (V, E, s, t, c),$$

wobei (V, E, s, t) ein Transport-Netzwerk und $c : E \to I\!N$ eine Kapazitätsfunktion sind. Ein Fluß in einem TNC ist ein Fluß in einem Transport-Netzwerk, für den zusätzlich

$$0 \leq f(u, v) \leq c(u, v)$$

gilt. Der Wert eines solchen Flußes ist dann

$$v(c, f) = \sum_v f(s, v).$$

Definition 2.3.32 Max-Flow Problem
Gesucht ist f mit $v(c, f) = \max_{f'} v(c, f')$.

Insbesondere werden wir nun das 0-1 Max-Flow Problem betrachten, in der jede Kante die Kapazität 1 oder 0 hat. (Im Falle der Kapazität 0 kann die Kante auch weggelassen werden.)

Satz 2.3.33 0-1 Max-Flow $\leq_{NC^1}$ Perfekt Matching.

Beweis: Sei C das Transport-Netzwerk. Wir gehen analog zum Beweis von Satz 2.3.29 vor. Ein System kantendisjunkter Wege von s nach t induziert einen 0-1 Fluß in C, da jeder dieser Wege zum maximalen Fluß beiträgt. Die Anzahl dieser maximalen kantendisjunkten Wege und damit auch der Wert für den Max-Flow kann durch $m = |E|$ abgeschätzt werden. Wir konstruieren jetzt eine Familie ungerichteter Hilfsgraphen $\{H_k\}$ wie folgt:

$$V(H_k) = \{s_1, \ldots, s_k\} \cup \{(e, 1) | e \in E\} \cup \{(e, 2) | e \in E\} \cup \{t_1, \ldots, t_k\}$$

$$\begin{aligned} E(H_k) \quad = \quad & \{(s_i, (e, 1)))|\exists v \text{ mit } e = (s, v) \text{ und alle } 1 \leq i \leq k\} \\ & \cup\{((e, 1), (e, 2))|e \in E\} \cup \{((e, 1), (f, 2))|e \cap f \neq \emptyset\} \\ & \cup\{((e, 2), t_i)|\exists v \text{ mit } e = (v, t) \text{ und alle } 1 \leq i \leq k\}. \end{aligned}$$

Aus einem Perfekten Matching in H_k können wir nun ein maximales System von s-t Wegen wie folgt konstruieren

In dem Perfekten Matching von H_k wird jeder Knoten der Form s_i mit einem Knoten $(e, 1)$ verbunden. Daher kann der zugehörige Zwillingsknoten $(e, 2)$ nicht mit $(e, 1)$, sondern er muß mit einem Knoten $(f, 1)$ verbunden sein. Dabei sind die Kanten e und f in C adjazent, d.h. sie haben einen gemeinsamen Knoten. Mit $(f, 1)$ fahren wir jetzt sofort, bis der letzte Zwillingsknoten mit t_j verbunden ist. Auf diese Weise ordnen wir jedem s_i einen Pfad $s_i, (e, 1), (f, 1), \ldots, t_j$ in C zu. Aufgrund der Matching-Eigenschaft sind alle diese Pfade disjunkt. Das Perfekte Matching in H_k induziert also ein k-Wegesystem in C. Daher testen wir parallel jeden Graphen H_k auf die Existenz eines Perfekten Matchings.

Nun brauchen wir mittels binärer Suche nur noch den größten Graphen zu suchen, der ein Perfektes Matching besitzt, ein Perfektes Matching zu konstruieren und dann den entsprechenden Max-Flow auszugeben. □

Als Korollar ergibt sich

Korollar 2.3.34 Unary Max-Flow $\leq_{NC^1}$ Perfekt Matching.

BEWEIS: Man betrachte den Multigraphen G', der dadurch entsteht, daß eine Kante mit Kapazität k in G durch k Kanten mit Kapazität 1 ersetzt wird. □

Man beachte jedoch, daß das Binary Max-Flow Problem $\mathcal{P}$-vollständig ist.

Damit gilt nun folgender zusammenfassender

Satz 2.3.35 Es gilt

$$\left.\begin{array}{c}\text{Unary-Max-Flow}\\ \text{Maximum Matching}\end{array}\right\} \leq_{NC^1} \text{Perfekt Matching} \leq_{RNC^1} \text{DET}.$$

Mit Hilfe der PNG-Hypothese gilt dann auch

Satz 2.3.36 Falls die PNG-Hypothese (siehe Definition 1.6.41) gilt

$$\left.\begin{array}{c}\text{Perfekt Matching}\\ \text{Maximum Matching}\\ \text{Unary Max-Flow}\end{array}\right\} \in \bigcap_{\epsilon>0} \text{DSPACE}\,(n^{\epsilon}).$$

BEWEIS: Mittels Satz 1.6.43. □

2.4 Interpolation in endlichen Körpern

Im vorigen Abschnitt haben wir einen Algorithmus kennengelernt, der sparse multivariate Polynome über $\mathbb{Z}$ auf Null testet. Nun wollen wir diesen Nulltest auf multivariate Polynome über endlichen Körpern erweitern. Dies ist ein schwierigeres Problem, da die Anzahl der Elemente und somit die relative Anzahl von Nullstellen größer ist. Daher ist die Wahrscheinlichkeit, eine Nullstelle eines Polynoms zu treffen, hoch und hat damit nicht soviel Aussagekraft. Außerdem ist jede Funktion über einem endlichen Körper durch ein Polynom darstellbar. Für den Fall GF(2) hatten wir dies mit Hilfe des Satzes 1.4.7 gesehen.

2.4.1 Interpolation in GF(2)

Im weiteren werden wir auch nur den Fall des Körpers GF(2) mit zwei Elementen betrachten.

Im Fall, daß nur Polynome mit beschränkter Anzahl von Monomen betrachtet wurden, erhielten wir im Fall $\mathbb{Z}$ einen polynomiellen Algorithmus, im Fall einer Gradbeschränkung erhielten wir jedoch nur einen probabilistischen Algorithmus. Eine Gradbeschränkung bei Polynomen über endlichen Körpern GF(2) ist jedoch nicht sinnvoll, da in GF(q) die Gleichheit

$$x^q \equiv x$$

gilt. Somit ist jedes Polynom in GF(q) als Funktion äquivalent zu einem gradbeschränkten Polynom.

Aus diesem Grunde werden wir nur den ersten Fall t-sparser Polynome betrachten. Wir bezeichen dieses Problem mit $\mathrm{IP}_{GF2}(n,t)$.

Definition 2.4.1 Mit $\mathrm{GF}(2)_t[x_1,\ldots,x_n]$ bezeichnen wir die Menge der t-sparsen Polynome in den Variablen $x_1,\ldots,x_n$, die nicht identisch 0 sind.

Wir suchen jetzt Testmengen, mit Hilfe derer wir **alle** Polynome aus $\mathrm{GF}(2)_t[x_1,\ldots,x_n]$ vom Nullpolynom trennen können, d.h für jedes Polynom p aus $\mathrm{GF}(2)_t[x_1,\ldots,x_n]$ existiert eine Zeuge x in der Testmenge mit $p(x) \neq 0$. Die Menge aller dieser Testmengen sei mit $S(n,t)$ bezeichnet:

$$S(n,t) = \{X \subseteq \{0,1\}^n | \forall f \in \mathrm{GF}(2)_t[x_1,\ldots,x_n] \; \exists x \in X \text{ mit } f(x) \neq 0\}.$$

Um zu entscheiden, ob ein t-sparses Polynom p identisch dem Nullpolynom ist, reicht es nun aus, alle Elemente aus einer dieser Testmengen in das t-sparse Polynom p einzusetzen. Daher interessieren wir uns für eine minimale Größe einer solchen Testmenge, um eine untere Schranke für das Entscheidungsproblem zu erhalten. Diese untere Schranke sei mit $d(n,t)$ bezeichnet:

$$d(n,t) = \min\{|X| \,|X \in S(n,t)\}$$

Wir wollen nun ein solches minimales X konstruieren. Zuerst geben wir eine untere Schranke für die Größe der X.

Satz 2.4.2 Zu $N \subset \{1,\ldots,n\}$ sei $a^N = (a_1^N,\ldots,a_n^N)$ durch

$$a_i^N = 0 \;\Leftrightarrow\; i \in N$$

definiert. Dann enthält jedes $X \in S(n,t)$ alle a^N mit $|N| \leq \lfloor \log t \rfloor$, d.h. alle Vektoren, die höchstens $\lfloor \log t \rfloor$ viele 0-en enthalten. Damit kann die Größe von X durch

$$|X| \geq \sum_{i=0}^{\lfloor \log t \rfloor} \binom{n}{i}$$

abgeschätzt werden und folglich ist

$$d(n,t) \geq \sum_{i=0}^{\lfloor \log t \rfloor} \binom{n}{i} = \Omega(n^{\log t}).$$

BEWEIS: Für jede Menge $N \subset \{1, \ldots, n\}$ mit $|N| \leq \lfloor \log t \rfloor$ definieren wir das Polynom

$$P_N = \bigwedge_{i \in N} (x_i \oplus 1) \bigwedge_{i \notin N} x_i.$$

P_N hat die folgenden Eigenschaften:

1. P_N hat $2^{\lfloor \log t \rfloor} \leq t$ Monome und ist somit t-sparse.
2. $P_N(x) = 1$ genau dann wenn $x = a^N$.

Daher muß für jedes $N \subset \{1, \ldots n\}$ mit den obigen Eigenschaften, der Vektor a^N in X enthalten sein. □

Wir haben nun eine untere Schranke für die Größe von Testmengen bekommen, indem wir eine Teilmenge jeder Testmenge konstruiert haben. Wir wollen nun eine obere Schranke für die Größe der kleinsten Testmenge angeben. Als Vorbereitung dient das folgende

Lemma 2.4.3 Decomposition Lemma ([CDGK 88])
Sei $n = n_1 + n_2$ und $X(n_i, t_i) \in S(n_i, t_i)$ für $i = 1, 2$. Dann ist

$$\bigcup_{t_1 \cdot t_2 \leq t} X(n_1, t_1) \times X(n_2, t_2) \in S(n, t)$$

eine Testmenge aus $S(n, t)$.

BEWEIS: Betrachten wir $0 \not\equiv f \in \mathrm{GF}(2)_t[x_1, \ldots, x_n]$ als ein Polynom in den Unbekannten $x_{n_1+1}, \ldots, x_n$ mit Koeffizientenpolynomen in $\mathrm{GF}(2)[x_1, \ldots, x_{n_1}]$. Die Anzahl von Termen sei $t_2 \leq t$. Dann hat mindestens eines der t_2 Koeffizientenpolynome höchstens $t_1 = \lfloor t/t_2 \rfloor$ viele Terme. Dieses Koeffizientenpolynom $g \not\equiv 0$ kann nun mit Hilfe von $X(n_1, t_1)$ vom Nullpolynom unterschieden werden, d.h. $\exists a^{(1)} \in X(n_1, t_1)$ mit $g(a^{(1)}) \neq 0$. Daher ist auch $f(a^{(1)}, x_{n_1+1}, \ldots, x_n)$ ein nichtidentisch verschwindendes t_2-sparses Polynom in n_2 Unbestimmten. Daher finden wir $a^{(2)} \in X(n_2, t_2)$ mit $f(a^{(1)}, a^{(2)}) \neq 0$. □

Wiederholtes Anwenden dieses Lemmas liefert das folgende

Korollar 2.4.4 Für jede Partition $\pi = (\pi_1, \ldots, \pi_s)$ von n (d.h. $\sum \pi_i = n$) gilt

$$\bigcup_{t_1 \cdot t_2 \cdot \ldots \cdot t_s \leq t} X(\pi_1, t_1) \times \cdots \times X(\pi_s, t_s) \in S(n, t).$$

Wir wenden dieses Korollar nun für $\pi = (1, \ldots, 1)$ an. Damit $t_1 \cdot t_2 \cdot \ldots \cdot t_n \leq t$ gilt, darf $(t_1, \ldots, t_n)$ höchstens $\lfloor \log t \rfloor$ 2en enthalten, die restlichen t_i sind 1. Die Testmengen, aus denen unsere Testmenge $X(n, t)$ aufgebaut ist, sind

$$X(1, 2) = \{0, 1\} \text{ und } X(1, 1) = \{1\}.$$

Damit besteht die Menge $X(n, t)$ aus allen Vektoren $x \in \{0, 1\}^n$ mit höchstens $\lfloor \log t \rfloor$ vielen Nullen. Dies ist gerade die Menge, die in Satz 2.4.2 konstruiert wurde. Daher ist diese Teilmenge jeder Testmenge selber eine Testmenge, und daher die kleinste. Die obere und untere Schranke fallen zusammen und damit gilt

$$d(n, t) = \Theta(n^{\log t}).$$

Die Größe einer minimalen Testmenge ist superpolynomiell. Es gibt daher keinen deterministischen Algorithmus, der ein t-sparses Polynom über GF(2) mit n Variablen durch Einsetzen von Werten aus $\mathrm{GF}(2)^n$ in polynomieller Zeit vom Nullpolynom unterscheiden kann. Da das Entscheidungsproblem aber trivialerweise nichtdeterministisch in polynomieller Zeit lösbar ist ergibt sich folgendes

Korollar 2.4.5 Falls das t-sparse Polynom als Black-Box gegeben ist, so ist

$$\mathrm{IP}_{GF2}(n,t) \begin{array}{l} \in \text{Black-Box } \mathcal{NP} \\ \notin \text{Black-Box } \mathcal{P}. \end{array}$$

Also ist

$$\text{Black-Box } \mathcal{NP} \neq \text{Black-Box } \mathcal{P}.$$

Auch für t-sparse Polynome über $\mathrm{GF}(q)$ erhalten wir eine solche untere Schranke für die Größe einer minimalen Testmenge. Daher werden wir zu Körpeerweiterungen übergehen, um effiziente Algorithmen zu bekommen. Die hier verwendete Theorie kann z.B. in [LN 86] nachgelesen werden.

2.4.2 Interpolationsalgorithmus von Clausen, Grabmeier und Karpinski

In diesem Abschnitt wird der von Clausen, Grabmeier und Karpinski [CGK 87] entwickelte Interpolationsalgorithmus für ein t-sparses Polynom vorgestellt.

Dieser Algorithmus benötigt lediglich linear in t viele Auswertungspunkte, arbeitet jedoch in der *großen* Körpererweiterung $\mathrm{GF}(q^n)$ von $\mathrm{GF}(q)$. Diese Körpererweiterung ist zu mächtig, um einen Algorithmus zu erhalten, der noch in NC liegt.

Falls die Auswertungen sehr teuer sind, d.h. die Komplexität in der Anzahl der Auswertungsstellen gemessen wird, oder die Arithmetik in $\mathrm{GF}(q^n)$ gegeben ist, so ist dieser Algorithmus allen anderen bekannten Algorithmen vorzuziehen.

Um Monome zu trennen, haben Grigoriev und Karpinski [GK 87] sich die Eindeutigkeit der Primzahlzerlegung zunutze gemacht und Monome an den Punkten $(p_0^i, \ldots, p_{n-1}^i)$ ausgewertet, wobei die p_j's paarweise verschiedene Primzahlen sind. Verschiedene Monome liefern an diesen Punkten verschiedene Werte.
In [CGK 87] wird diese Idee auf den endlichen Fall (Polynome über $\mathrm{GF}(q)$ in n Variablen) übertragen, indem zum einen Monome durch die q-adische Darstellung ihres Exponenten repräsentiert werden, zum anderen die Eindeutigkeit des diskreten Logorithmus zur Basis eines primitiven Elementes ω aus $\mathrm{GF}(q^n)$ genutzt wird. Monome werden dann durch die Punkte $(\omega^{iq^0}, \omega^{iq^1}, \ldots, \omega^{iq^{n-1}})$ getrennt.

Satz 2.4.6 Sei $f \in \mathrm{GF}(q)[x_1, \ldots, x_n]$ ein t-sparses Polynom, $t \geq 2$ mit $\deg_{x_j}(f) < q$ für $1 \leq j \leq n$ und ω ein primitives Element aus $\mathrm{GF}(q^n)$. Sei $f_i := f(\omega^{iq^0}, \omega^{iq^1}, \ldots, \omega^{iq^{n-1}})$ für $0 \leq i \leq t-1$ und $q \not| \, i$ für $i > 0$.
Dann gilt:

$$f \equiv 0 \qquad \Longleftrightarrow \qquad f(0, \ldots, 0) = 0 \text{ und } f_i = 0 \text{ für } 0 \leq i \leq t-1 \text{ und } q \not| \, i \text{ für } i > 0.$$

Für $t = 1$ gilt:

$$f \equiv 0 \Longleftrightarrow f(1, \cdots, 1) = 0\,.$$

BEWEIS: Der Fall $t = 1$ ist trivial, sei also $t \geq 2$.
Sei $\mathbf{q}^n := \{0, \ldots, q-1\}^{\{1,\ldots,n\}}$ eine Menge von Multiindices. f ist dann eine Linearkombination der q^n Monome $x^\alpha := x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$:

$$f = \sum_{\alpha \in \mathbf{q}^n} c_\alpha x^\alpha$$

Sei $f(0, \ldots, 0) = c_{(0,\ldots,0)} = 0$, sonst ist $f \not\equiv 0$.
Betrachte die Abbildung $\Omega : \mathbf{q}^n \setminus \{(0, \ldots, 0)\} \to \mathrm{GF}(q^n) \setminus \{0\}$ definiert durch

$$\Omega(\alpha) := \Omega_\alpha = \prod_{\nu=0}^{n-1} \omega^{\alpha_\nu q^\nu} = \omega^{(\sum_{\nu=0}^{n-1} \alpha_\nu q^\nu)} \quad .$$

Da ω ein primitives Element aus $\mathrm{GF}(q^n)$ ist, ist aufgrund der Eindeutigkeit der q-adischen Darstellung für Elemente aus $\mathrm{GF}(q^n)$ die Abbildung Ω bijektiv. Verschiedenen Monomen werden also mittels Ω verschiedene Werte aus $\mathrm{GF}(q^n)$ zugewiesen.
f ist t-sparse, also sind von den Koeffizienten von f höchstens t von Null verschieden. Sei $\mathrm{supp}(f) := \{\alpha \mid c_\alpha \neq 0\}$ der Träger von f. Dann gilt $|\mathrm{supp}(f)| =: k \leq t$.
Sei $A = \{\alpha^{(0)}, \ldots, \alpha^{(t-1)}\}$ eine k-elementige Teilmenge von $\mathbf{q}^n \setminus \{(0, \ldots, 0)\}$ mit $\mathrm{supp}(f) \subseteq A$. Da $c_{(0,\ldots,0)} = 0$ gilt:

$$f = \sum_{\alpha \in \mathbf{q}^n} c_\alpha x^\alpha = \sum_{\alpha \in A} c_\alpha x^\alpha \quad .$$

also gilt für alle $0 \leq i < k \leq t$

$$\begin{aligned}
f_i &= f(\omega^{iq^0}, \omega^{iq^1}, \ldots, \omega^{iq^{n-1}}) \\
&= \sum_{\alpha \in A} c_\alpha \cdot \omega^{\alpha_0 i q^0} \cdot \ldots \cdot \omega^{\alpha_{n-1} i q^{n-1}} \\
&= \sum_{\alpha \in A} c_\alpha \cdot \omega^{(\sum_{\nu=0}^{n-1} \alpha_\nu q^\nu) \cdot i} \\
&= \sum_{\alpha \in A} c_\alpha \cdot \Omega_\alpha^i \quad .
\end{aligned}$$

Hieraus ergibt sich das folgende lineare Gleichungssystem:

$$\underbrace{\left(\Omega_\alpha^i\right)_{\substack{0 \leq i < k \\ \alpha \in A}}}_{=: \mathcal{V}} \cdot (c_\alpha)_{\alpha \in A} = (f_i)_{0 \leq i < k} \quad . \tag{2.1}$$

Da Ω eine bijektive Abbildung ist, gilt $\Omega_\alpha \neq \Omega_\beta$ für $\alpha \neq \beta$. Dann ist $\mathcal{V}$ eine *Vandermonde'sche* Matrix, also nichtsingulär, damit ist das lineare Gleichungssystem (2.1) eindeutig lösbar. Es gilt also:

$$f \equiv 0 \quad \Longleftrightarrow \quad (c_\alpha)_{\alpha \in A} = 0 \quad \Longleftrightarrow \quad (f_i)_{0 \leq i < k} = 0 \quad .$$

Es ist nur noch zu zeigen, daß die f_i mit $q \not| \, i$ nicht benötigt werden.
Betrachte dazu den Frobenius Automorphismus $y \to y^q$ in $\mathrm{GF}(q^n)$. Es gilt

$$f_{j \cdot q} = (f_j)^q \quad \text{für alle } i < q^n \quad \text{, denn}$$

$$
\begin{aligned}
f_{j\cdot q} &= \sum_{\alpha\in A} c_\alpha \cdot \Omega_\alpha^{j\cdot q} \\
&= \sum_{\alpha\in A} c_\alpha^q \cdot \Omega_\alpha^{j\cdot q} \quad \text{, da der Fr. Aut. Elemente aus GF}(q) \text{ fixiert} \\
&= \sum_{\alpha\in A} (c_\alpha \cdot \Omega_\alpha^j)^q \\
&= (\sum_{\alpha\in A} c_\alpha \cdot \Omega_\alpha^j)^q \quad \text{, Homomorphismus-Eigenschaft} \\
&= (f_j)^q\,.
\end{aligned}
$$

Sei nun $i = j \cdot q$ also $q|i$ und $q \not| j$. Dann ist $f_i = (f_j)^q$, also können die fehlenden f_i berechnet werden. □

Die Eigenschaft, daß mit Hilfe der Abbildung Ω die Monome eines Polynoms f separiert werden, kann auch zur Rekonstruktion von f benutzt werden. Als weitere Technik werden die Newton'sche Identitäten verwendet.

Satz 2.4.7 Sei $f \in \mathrm{GF}(q)[x_1, \ldots, x_n]$ ein t-sparses Polynom mit $\deg_{x_j}(f) < q$ für $1 \le j \le n$ und ω ein primitives Element aus $\mathrm{GF}(q^n)$. Um f zu rekonstruieren, reicht es aus die Werte $f_i := f(\omega^{iq^0}, \omega^{iq^1}, \ldots, \omega^{iq^{n-1}})$ für $0 \le i < 2t$ und $q \not| i$ für $i > 0$ und $f(0, \ldots, 0)$ zu kennen.

Beweis: Es wird die Notation aus Satz 2.4.6 benutzt.
Man kann annehmen, daß $f(0, \ldots, 0) = c_{(0,\ldots,0)} = 0$, andernfalls wird $f - f(0, \ldots, 0)$ rekonstruiert. Dann kann f dargestellt werden durch

$$f = \sum_{\alpha\in\mathbf{q}^n} c_\alpha x^\alpha = \sum_{\alpha\in\mathbf{q}^n\setminus\{(0,\ldots,0)\}} c_\alpha x^\alpha \quad .$$

Sei $A = \{\alpha_0, \ldots, \alpha_l\}$ eine beliebige Teilmenge von $\mathbf{q}^n \setminus \{(0, \ldots, 0)\}$ und $e_i(A), 0 \le i \le l$ die Auswertung des i-ten elementaren symmetrischen Polynoms in $|A|$ Unbestimmten an der Stelle $(\Omega_\alpha)_{\alpha\in A}$ (dies ist wohldefiniert, da $\{(0, \ldots, 0)\} \notin A$); d.h.

$$
\begin{aligned}
e_0(A) &= 1 \\
e_1(A) &= \Omega_{\alpha_0} + \Omega_{\alpha_1} + \ldots + \Omega_{\alpha_l} \\
e_2(A) &= \Omega_{\alpha_0} \cdot \Omega_{\alpha_1} + \Omega_{\alpha_0} \cdot \Omega_{\alpha_2} + \ldots + \Omega_{\alpha_0} \cdot \Omega_{\alpha_l} + \\
&\quad\; \Omega_{\alpha_1} \cdot \Omega_{\alpha_2} + \ldots + \Omega_{\alpha_1} \cdot \Omega_{\alpha_l} + \ldots + \Omega_{\alpha_{l-1}} \cdot \Omega_{\alpha_l} \\
&\;\vdots \\
e_l(A) &= \Omega_{\alpha_0} \cdot \Omega_{\alpha_1} \cdot \ldots \cdot \Omega_{\alpha_l} \quad .
\end{aligned}
$$

Sei

$$\Lambda(x) := \prod_{\beta\in A} (x - \Omega_\beta) \;\in \mathrm{GF}(q^n)[x]\,.$$

Nach der *Newton*'schen Formel (s. [LN 86]) gilt damit:

$$\Lambda(x) = \sum_{j=0}^{|A|} \underbrace{(-1)^{|A|-j} \cdot e_{|A|-j}(A)}_{\lambda_j(A)} \cdot x^j\,.$$

Die Nullstellen dieses Polynoms sind $(\Omega_\alpha)_{\alpha\in A}$. Das führt zu den verallgemeinerten Newton'schen Identitäten:

$$0=\sum_{j=0}^{|A|}\lambda_j(A)\cdot\Omega_\alpha^j,\ \alpha\in A\,.$$

Diese Gleichungen werden für festes $i, 0\le i<q^n$ mit $c_\alpha\cdot\Omega_\alpha^i$ multipliziert, und dann für alle $\alpha\in A$ aufaddiert:

$$\begin{aligned}\sum_{\alpha\in A}0\cdot c_\alpha\cdot\Omega_\alpha^i &= \sum_{\alpha\in A}\sum_{j=0}^{|A|}\lambda_j(A)\cdot c_\alpha\cdot\Omega_\alpha^{j+i}\\ 0 &= \sum_{j=0}^{|A|}\lambda_j(A)\cdot\sum_{\alpha\in A}c_\alpha\cdot\Omega_\alpha^{j+i}\\ 0 &= \sum_{j=0}^{|A|}\lambda_j(A)\cdot f_{i+j}\,.\end{aligned}$$

Da $\lambda_{|A|}=e_0(A)=1$ ist, gilt

$$\begin{aligned}\sum_{j=0}^{|A|-1}\lambda_j(A)\cdot f_{i+j} &= -\lambda_{|A|}\cdot f_{i+|A|}\\ &= -f_{i+|A|}\qquad\text{für alle } 0\le i<|A|\end{aligned}$$

Es ergibt sich also das folgende lineare Gleichungssystem:

$$(f_{i+j})_{0\le i,j<|A|}\cdot(\lambda_j(A))_{0\le j<|A|}=\left(-f_{i+|A|}\right)_{0\le i<|A|}\tag{2.2}$$

Betrachte die Matrix $(f_{i+j})_{0\le i,j<|A|}$:

$$f_{i+j}=\sum_{\alpha\in A}c_\alpha\cdot\Omega_\alpha^{i+j}=\sum_{\alpha\in A}\Omega_\alpha^i\cdot c_\alpha\cdot\Omega_\alpha^j$$

$$\Longrightarrow (f_{i+j})_{0\le i,j<|A|}=\left(\Omega_\alpha^i\right)^t_{\substack{0\le i<|A|\\ \alpha\in A}}\cdot D_A\cdot\left(\Omega_\alpha^j\right)_{\substack{0\le i<|A|\\ \alpha\in A}}$$

mit $D_A=\operatorname{diag}\left((c_\alpha)_{\alpha\in A}\right)$.

Damit ist der Rang von $(f_{i+j})_{0\le i,j<|A|}=\operatorname{rg}(D_A)=|\operatorname{supp}(f)|=:k\le t$.
Für $A=\operatorname{supp}(f)$ ist dann $(f_{i+j})_{0\le i,j<k}$ nichtsingulär.
Dann kann das lineare Gleichungssystem (2.2) eindeutig gelöst werden und man erhält die Koeffizienten

$$\lambda_j(\operatorname{supp}(f)),\ 0\le j<k\quad\text{des Polynoms}\quad\Lambda(x)=\prod_{\beta\in\operatorname{supp}(f)}(x-\Omega_\beta).$$

Die Bestimmung der Nullstellen dieses Polynoms liefert $(\Omega_\alpha)_{\alpha\in\operatorname{supp}(f)}$. Da Ω bijektiv ist, kann hieraus $\operatorname{supp}(f)$ bestimmt werden.

Desweiteren ist durch die Kenntnis der Nullstellen auch die Matrix $\mathcal{V}$ des Gleichungssystems (2.1) bekannt, also können die Koeffizienten $(c_\alpha)_{\alpha\in\operatorname{supp}(f)}$ von f berechnet werden.

Damit ist f vollständig rekonstruiert.

Wie im Beweis von Satz 2.4.6 gezeigt wurde, folgt aus den Eigenschaften des Frobenius-Automorphismus, daß die Werte f_i mit $q|\ i$ aus den bekannten Werten f_j mit $q \not| j$ berechnet werden können.

Insgesamt werden also $1+t-\lfloor\frac{2t-1}{q}\rfloor$ Auswertungen benötigt. □

Zur Interpolation eines t-sparsen Polynoms $f \in \mathrm{GF}(q)[x_0,\ldots,x_{n-1}]$ mit $\deg_{x_j}(f) < q$ für alle j ergibt sich aus Satz 2.4.6 und Satz 2.4.7 der folgende Algorithmus:

Algorithmus 2.4.8 Interpolationsalgorithmus [CGK 87]

Eingabe: Orakel für f

Schritt 1: Bestimme ein primitives Element ω in $\mathrm{GF}(q^n)$

Schritt 2: Befrage das Orakel für die $1+t-\lfloor\frac{2t-1}{q}\rfloor$ Werte von $f(0,\ldots,0)$ und $f_i := f(\omega^{iq^0},\omega^{iq^1},\ldots,\omega^{iq^{n-1}})$ für $0 \le i < 2t$ und $q \not| i$ für $i > 0$.
Falls das konstante Glied $f(0,\ldots,0) \neq 0$ wird $f - f(0,\ldots,0)$ interpoliert.

Schritt 3: Berechne die fehlenden f_i, also mit $i = q^s \cdot i_0, 1 \le s,\ s$ maximal, ergibt sich $f_i = f_{i_0}^{q^s}$

Schritt 4: Bestimme $k = \mathrm{rg}\,((f_{i+j})_{0\le i,j<t})$

Schritt 5: Löse das lineare Gleichungssystem

$$(f_{i+j})_{0\le i,j<k} \cdot (\lambda_j)_{0\le j<k} = (-f_{i+k})_{0\le i<k}$$

Schritt 6: Bestimme alle Wurzeln des Polynoms

$$\Lambda(x) = x^k + \sum_{j=0}^{k-1} \lambda_j \cdot x^j$$

Damit erhält man $(\Omega_\alpha)_{\alpha\in\mathrm{supp}(f)}$.

Schritt 7: Berechne α aus Ω_α um $\mathrm{supp}(f)$ zu erhalten

Schritt 8: Löse das lineare Gleichungssystem

$$\left(\Omega_\alpha^i\right)_{\substack{0\le i<k\\ \alpha\in A}} \cdot (c_\alpha)_{\alpha\in A} = (f_i)_{0\le i<k}$$

Damit erhält man $(c_\alpha)_{\alpha\in\mathrm{supp}(f)}$

Ausgabe: $(c_\alpha,\alpha)_{\alpha\in\mathrm{supp}(f)}$

Es ist natürlich nur dann sinnvoll diesen Algorithmus anzuwenden, wenn $2t < q^n$, anderenfalls benötigt man lediglich q^n Auswertungen $f(a_{(0)}),\ldots f(a_{(q^n-1)})$ um die triviale Interpolation durch die Lösung des linearen Gleichungssystems

$$\left(f(a_{(i)}\right)_{0\le i<q^n} = \left(a_{(i)}^\alpha\right)_{\substack{0\le i<q^n\\ \alpha\in\mathbf{q}^n}} \cdot (c_\alpha)_{\alpha\in\mathbf{q}^n}$$

durchzuführen.

Sei ein primitives Element $\omega \in \mathrm{GF}(q^n)$ gegeben. Nach [Mu 87] lassen sich die Schritte 4, 5, 7, 8, in denen der Rang bzw. die Inverse einer $t \times t$-Matrix berechnet wird, mit $O(t^{4.5})$ Prozessoren in $O(\log^2 t)$ Parallelzeit durchführen. Die gleichen Größenordnungen ergeben sich für die Faktorisierung des univarianten Polynoms in Schritt 6, wenn der Algorithmus verwendet wird, der in [Ga 84] vorgeschlagen wird. Damit gilt:

Satz 2.4.9 Der Interpolationsalgorithmus [CGK 87] ist NC-reduzierbar auf die Berechnung diskreter Logarithmen in $\mathrm{GF}(q^n)$.

Satz 2.4.10 Der Interpolationsalgorithmus [CGK 87] ist, bzgl. der Anzahl der Auswertungsstellen, für univariante Polynome optimal.

Beweis: Zu betrachten ist der Fall $n = 1$ und $2t < q$. Sei A eine beliebige Teilmenge von $\mathrm{GF}(q)$ mit höchstens $2t$ Elementen. Dann ist $0 \not\equiv h := \prod_{a \in A}(x-a) \in \mathrm{GF}(q)[x]$ ein Polynom vom Grad höchstens $2t < q$ und mit höchstens $2t$ Monomen, h besitzt also eine Darstellung der Form $f - g$ mit f, g t-sparse. Da h in A verschwindet, nehmen f und g in A dieselben Werte ein. Also ist A nicht geeignet, um t-sparse Polynome zu rekonstruieren. □

2.4.3 Interpolationsalgorithmus von Grigoriev, Karpinski und Singer

In diesem Abschnitt soll der erste NC-Algorithmus zur Interpolation von sparsen Polynomen in endlichen Körpern von Grigoriev, Karpinski und Singer [GKS 88] vorgestellt werden.

Der Algorithmus [CGK 87] benötigt die Berechnung diskreter Logarithmen in $\mathrm{GF}(q^n)$. Zur Zeit sind hierfür jedoch noch keine effizienten Algorithmen bekannt. Um einen NC-Algorithmus zu entwickeln, müssen also kleinere Grade der Körpererweiterung von $\mathrm{GF}(q)$ betrachtet werden, so daß die Arithmetik in dieser Körpererweiterung in NC liegt, jedoch die Anzahl der Auswertungsstellen noch polynomiell beschränkt ist.

Die erste kanonische Überlegung ist, keine Körpererweiterung von $\mathrm{GF}(q)$ zu benutzen, also in $\mathrm{GF}(q)$ zu bleiben. Es läßt sich jedoch zeigen (siehe [CDGK 88]), daß die benötigte Anzahl von Auswertungsstellen, um den Nulltest in $\mathrm{GF}(q)$ durchzuführen, nicht polynomiell beschränkt ist. Damit kann es auch keinen Interpolationsalgorithmus in $\mathrm{GF}(q)$ geben, der in polynomieller Zeit arbeitet.

In [GKS 88] wird gezeigt, daß schon eine Erweiterung von logarithmischem Grad genügt, um zum einen noch effiziente Arithmetik durchführen zu können, zum anderen diese Erweiterung noch hinreichend viele Elemente besitzt, um mit polynomiell vielen Auswertungsstellen das gesuchte Polynom zu rekonstruieren. Da NC-Interpolation in $\mathrm{GF}(q)$ nicht möglich ist, ist diese geringfügige Erweiterung die kleinstmögliche Erweiterung.

Genauer gilt:

Satz 2.4.11 Sei $f \in \mathrm{GF}(q)[x_1, \ldots, x_n]$ ein t-sparses Polynom, q beliebig. Dann existiert ein deterministischer paralleler Algorithmus (NC^3) um f über der geringfügigen Körpererweiterung $\mathrm{GF}(q^{\lceil 4\log_q(nt)+3\rceil})$ durch $(O(qn^2t^5))$ Anfragen zu interpolieren. Der Algorithmus läuft in $O(\log^3(ntq))$ Parallelzeit und benötigt $O(n^2t^6q^3\log^{5.5}(ntq)+q^{2.5}\cdot\log^2 q)$ Prozessoren.

Im Algorithmus [CGK 87] ergibt sich das Problem, die Auswertungsstellen so zu bestimmen, daß verschiedene Monome an den Auswertungsstellen unterschiedliche Werte annehmen, um dadurch die Monome unterscheiden zu können.
Dieses Problem wird durch Ausnutzung der Eindeutigkeit der q-adischen Darstellung und der Eindeutigkeit des diskreten Logarithmus gelöst. Dadurch gelangt man in den Körper $\mathrm{GF}(q^n)$; man benötigt jedoch sehr wenige Auswertungsstellen.
Die Idee in [GKS 88] besteht darin, eine größere Anzahl von Auswertungsstellen zuzulassen, die zwar nicht alle die Eigenschaft besitzen, daß sie verschiedene Monome trennen, jedoch werden sie so konstruiert, daß noch genügend viele von ihnen diese Eigenschaft besitzen, um das Polynom effizient rekonstruieren zu können. Um der Menge der Auswertungsstellen diese Eigenschaft zusichern zu können, werden sie mit Hilfe von Cauchymatrizen definiert.

Definition 2.4.12 Cauchymatrix
Seien x_i, y_i für $1 \le i,j \le N$ feste Werte. Eine $(N\times N)$-Matrix $C=(c_{ij})_{1\le i,j\le N}$ heißt *Cauchymatrix*, falls

$$c_{ij} = \frac{1}{x_i+y_j} \qquad \text{für alle } 1 \le i,j \le N$$

gilt.

Lemma 2.4.13 Determinante einer Cauchymatrix
Sei C eine Cauchymatrix. Dann gilt:

$$\det C \quad = \quad \frac{\prod_{1\le i<j\le N}(x_j-x_i)\cdot(y_j-y_i)}{\prod_{1\le i,j\le N}(x_i+y_i)}\,.$$

Eine ähnliche Formel gilt auch für jeden nichtverschwindenden Minor von C.

Im folgenden wird eine Cauchymatrix C benutzt, die durch $c_{ij} := \frac{1}{i+j} \bmod p$ für $1 \le i,j \le N$ mit p prim definiert ist. Falls $2N < p$, so sind nach Lemma 2.4.13 C und alle Minoren von C nichtsingulär.

Der Algorithmus [GKS 88] besteht aus zwei Hauptteilen:

1. Effizienter Nulltest von Polynomen aus $\mathrm{GF}(q)[x_1,\ldots,x_n]$ in der Körpererweiterung $\mathrm{GF}(q^{\lceil 2\log_q(nt)+3\rceil})$.

2. Lösung des Interpolationsproblems durch induktive Aufzählung von Teillösungen für Teilmonome und Koeffizienten durch rekursive Anwendung des obigen Nulltests.

Es wird davon ausgegangen, daß das Orakel in der Lage ist, Funktionswerte von f zu liefern, die in einer beliebigen Körpererweiterung $\mathrm{GF}(q^s)$ ausgewertet worden sind. Der Test, ob ein Polynom $f \in \mathrm{GF}(q)[x_1,\ldots,x_n]$ identisch dem Nullpolynom ist, hat dann die folgende Gestalt.

Algorithmus 2.4.14 Nulltest [GKS 88]

Eingabe: Black-Box für $f \in \mathrm{GF}(q^s)[x_1, \ldots, x_n]$, t-sparse, s beliebig

Ausgabe: $\begin{cases} \text{Yes} & \text{falls } f \equiv 0 \\ \text{No} & \text{falls } f \not\equiv 0 \end{cases}$

Schritt 1: Bestimme s so, daß $q^s - 1 > 4qn \cdot (n-1) \cdot \binom{t}{2}$ mit s ist minimal.

Schritt 2: Konstruiere den Körper $\mathrm{GF}(q^s)$ aus $\mathrm{GF}(q)$ und s. Bestimme ein primitives Element ω aus $\mathrm{GF}(q^s)$.

Schritt 3: $N = \lceil \frac{q^s-1}{4nq} \rceil$. Bestimme mit Hilfe des Siebes von Erastosthenes eine Primzahl p mit $2N < p \leq 4N$

Schritt 4: Bestimme $c_{ij} := \frac{1}{i+j} \bmod p$ für $1 \leq i, j \leq N$ durch den Euklidischen Algorithmus. Sei $C = (c_{ij})_{1 \leq i,j \leq N}$

Schritt 5: Sei $\bar{C} = (\bar{c}_{ij})$ eine beliebige $(N \times n)$-Teilmatrix von C.

Schritt 6: Werte die Black-Box in $\mathrm{GF}(q^s)$ an den Punkten

$$\omega^{l \cdot \bar{c}_i} = (\omega^{l \cdot \bar{c}_{i1}}, \omega^{l \cdot \bar{c}_{i2}}, \ldots, \omega^{l \cdot \bar{c}_{in}}) \quad \text{für } i = 1, \ldots, N,\ l = 0, \ldots, t-1$$

und bei $(0, \ldots, 0)$ aus. Wenn alle Auswertungen Null ergeben, so ist $f \equiv 0$.

Satz 2.4.15 Der Nulltest [GKS 88] ist korrekt.

Beweis: Sei $f(0, \ldots, 0) = c_{(0,\ldots,0)} = 0$, sonst ist $f \not\equiv 0$. Es ist folgendes nachzuweisen:

$$f \not\equiv 0 \quad \Longleftrightarrow \quad \exists\, 0 \leq l < t, 1 \leq i \leq N :\ f(\omega^{l \cdot \bar{c}_i}) \neq 0\,.$$

Sei $\mathbf{q}^n$ wie im Beweis von Satz 2.4.6 definiert. f ist dann eine Linearkombination der $q^n - 1$ Monome $x^\alpha := x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$:

$$f = \sum_{\alpha \in \mathbf{q}^n \setminus \{(0,\ldots,0)\}} c_\alpha x^\alpha\,.$$

Sei $\alpha' = (\alpha'_1, \ldots, \alpha'_n)$, $\alpha'' = (\alpha''_1, \ldots, \alpha''_n)$ und $\bar{c}_i = (\bar{c}_{i_1}, \ldots, \bar{c}_{i_n})$.

Zunächst soll die oben erwähnte Eigenschaft der Auswertungsstellen nachgewiesen werden, d.h. es wird gezeigt, daß es für jedes beliebige Paar von Monomen $x^{\alpha'}$ und $x^{\alpha''}$ mit $\alpha' \neq \alpha''$ einen Punkt aus der Menge der Auswertungsstellen gibt, an dem sich die beiden Monome unterscheiden. Das bedeutet, daß es eine Reihe $\bar{c}_i$ aus der Matrix $\bar{C}$ gibt, so daß die Monome $x^{\alpha'}$ und $x^{\alpha''}$ an der Stelle $\omega^{\bar{c}_i}$ unterschiedliche Werte annehmen. Sei

$$\Omega_{\alpha,i} := x^\alpha|_{\omega^{\bar{c}_i}} = \omega^{(\sum_{j=1}^{n} \alpha_j \cdot \bar{c}_{i_j})}\,.$$

Es soll gezeigt werden:

$$\exists\, 1 \leq i \leq N :\ \Omega_{\alpha',i} \neq \Omega_{\alpha'',i} \quad \text{für beliebige } \alpha' \neq \alpha''$$

Eine Reihe $\bar{c}_i$ aus der Matrix $\bar{C}$ soll *schlecht* für $\alpha' \neq \alpha''$ heißen, wenn $\bar{c}_i$ die Monome $x^{\alpha'}$ und $x^{\alpha''}$ nicht trennt, also wenn $\Omega_{\alpha',i} = \Omega_{\alpha'',i}$.

Schlechte $\bar{c}_i$ können folgendermaßen charakterisiert werden: $\Omega_{\alpha',i} = \Omega_{\alpha'',i}$ ist äquivalent zu $\omega^{\bar{c}_i \cdot \alpha'} = \omega^{\bar{c}_i \cdot \alpha''}$. Da ω ein primitives Element aus $\mathrm{GF}(q^s)$ ist, erzeugt ω die zyklische Gruppe $\mathrm{GF}(q^s) \setminus \{0\}$ der Ordnung $q^s - 1$. Das bedeutet

$$\begin{aligned}
& & \omega^{\bar{c}_i \cdot \alpha'} = \omega^{\bar{c}_i \cdot \alpha''} & \\
&\Longleftrightarrow & \bar{c}_i \cdot \alpha' \equiv \bar{c}_i \cdot \alpha'' & \pmod{q^s - 1} \\
&\Longleftrightarrow & \sum_{j=1}^{n} \bar{c}_{i_j} \cdot \alpha'_j \equiv \sum_{j=1}^{n} \bar{c}_{i_j} \cdot \alpha''_j & \pmod{q^s - 1} \\
&\Longleftrightarrow & \sum_{j=1}^{n} (\alpha'_j - \alpha''_j) \cdot \bar{c}_{i_j} \equiv \delta & \pmod{q^s - 1}
\end{aligned} \tag{2.3}$$

Es gilt $0 \leq \alpha'_j, \alpha''_j \leq q - 1$, also $|\alpha'_j - \alpha''_j| \leq q - 1$ und nach Schritt 4 $|\bar{c}_{i_j}| < p \leq 4N$. Damit gilt

$$|\sum_{j=1}^{n} (\alpha'_j - \alpha''_j) \cdot \bar{c}_{i_j}| \leq \sum_{j=1}^{n} |\alpha'_j - \alpha''_j| \cdot |\bar{c}_{i_j}| \leq n \cdot (q-1) \cdot 4N\,. \tag{2.4}$$

Nach Schritt 3 ist $N = \frac{\lceil q^s - 1 \rceil}{4nq}$, also kann man die Ungleichung (2.4) weiter abschätzen zu

$$|\sum_{j=1}^{n} (\alpha'_j - \alpha''_j) \cdot \bar{c}_{i_j}| \leq q^s - 1.$$

Also gilt (2.3) auch ohne die Modulobildung; damit hat mat eine Charakterisierung für schlechte $\bar{c}_i$ gefunden:

$$\bar{c}_i \text{ schlecht} \Longleftrightarrow \sum_{j=1}^{n} (\alpha'_j - \alpha''_j) \cdot \bar{c}_{i_j} = 0 \tag{2.5}$$

Für ein festes Paar von Monomen $x^{\alpha'}, x^{\alpha''}$ kann es höchstens $(n-1)$ schlechte Vektoren geben, denn falls $\bar{c}_i^{(1)}, \ldots, \bar{c}_i^{(n)}$ sämtlich schlecht wären, dann wäre die aus den Vektoren $\bar{c}_i^{(1)}, \ldots, \bar{c}_i^{(n)}$ gebildete Untermatrix $\mathcal{C}$ von C nichtsingulär, denn nach (2.5) gilt:

$$\underbrace{\begin{pmatrix} \bar{c}_{i,1}^{(1)} & \ldots & \bar{c}_{i,1}^{(1)} \\ \vdots & & \vdots \\ \bar{c}_{i,1}^{(n)} & \ldots & \bar{c}_{i,1}^{(n)} \end{pmatrix}}_{=:\,\mathcal{C}} \cdot \begin{pmatrix} \alpha'_1 - \alpha''_1 \\ \vdots \\ \alpha'_n - \alpha''_n \end{pmatrix} = 0$$

Damit ist der Kern von $\mathcal{C}$ ungleich dem Nullraum, also ist $\mathcal{C}$ singulär. Dies steht im Widerspruch zu Lemma 2.4.13, damit gilt die obige Behauptung.

Da f t-sparse ist, gibt es höchstens $\binom{t}{2}$ Paare von Monomen, damit also höchstens $(n-1) \cdot \binom{t}{2}$ schlechte Vektoren für beliebige $\alpha' \neq \alpha''$. Da

$$(n-1) \cdot \binom{t}{2} = \frac{4nq \cdot (n-1) \cdot \binom{t}{2}}{4nq} < \frac{\lceil q^s - 1 \rceil}{4nq} = N\,,$$

muß ein $i_0, 1 \leq i_0 \leq N$ existieren mit $\Omega_{\alpha',i_0} \neq \Omega_{\alpha'',i_0}$ für beliebige $\alpha' \neq \alpha''$.
Damit trennt c_{i_0} alle Monome.

Für dieses i_0 gilt die gewünschte Aussage, d.h. es gilt:

$$f \not\equiv 0 \quad \Longleftrightarrow \quad \exists 0 \leq l < t : \; f(\omega^{l \cdot \bar{c}_{i_0}}) \neq 0 \quad \text{denn:}$$

Sei $\Omega_\alpha = \Omega_{\alpha,i_0}$ und $f_l := f(\omega^{l\cdot \bar{c}_{i_0}})$. Dann gilt für alle $0 \leq l < |\mathrm{supp}(f)| < t$

$$\begin{aligned} f_l &= \sum_{\alpha\in\mathrm{supp}(f)} c_\alpha \cdot x^\alpha\big|_{\omega^{l\cdot\bar{c}_{i_0}}} \\ &= \sum_{\alpha\in\mathrm{supp}(f)} c_\alpha \cdot \left(\omega^{\bar{c}_{i_0}\cdot\alpha}\right)^l \\ &= \sum_{\alpha\in\mathrm{supp}(f)} c_\alpha \cdot \Omega_\alpha^l \end{aligned}$$

Hieraus ergibt sich das folgende lineare Gleichungssystem:

$$\underbrace{\left(\Omega_\alpha^l\right)_{\substack{0\leq l<|\mathrm{supp}(f)|\\ \alpha\in\mathrm{supp}(f)}}}_{=:\mathcal{V}} \cdot (c_\alpha)_{\alpha\in\mathrm{supp}(f)} = (f_l)_{0\leq l<|\mathrm{supp}(f)|} \;. \tag{2.6}$$

$\mathcal{V}$ ist eine nicht singuläre Vandermonde Matrix, denn $\alpha \neq (0.\ldots,0)$. Damit ist das lineare Gleichungssystem (2.6) eindeutig lösbar. Es gilt also:

$$f \equiv 0 \quad\Longleftrightarrow\quad (c_\alpha)_{\alpha\in\mathrm{supp}(f)} = 0 \quad\Longleftrightarrow\quad (f_l)_{0\leq l<|\mathrm{supp}(f)|} = 0\;.$$

Damit ist alles gezeigt. □

Im folgenden sollen effizient Teillösungen der Interpolationsaufgabe durch Zurückführung auf den Nulltest ermittelt werden, und aus diesen Teillösungen effizient die Lösung der eigentlichen Interpolationsaufgabe gefunden werden. Dazu sei im weiteren ohne Einschränkung angenommen, daß $n = 2^m$, um die Notation zu vereinfachen.

Die Teillösungen sollen die folgende Gestalt haben:

$$S_{\alpha,\beta} = \{(k_1,\ldots,k_{2^{\alpha-1}}) \mid x_{\beta\cdot\,2^{\alpha-1}+1}^{k_1}\cdot\ldots\cdot x_{\beta\cdot\,2^{\alpha-1}+2^{\alpha-1}}^{k_{2^{\alpha-1}}} \text{ taucht in einem Monom von } f \text{ auf}\}$$

mit $1 \leq \alpha \leq m+1$ und $0 \leq \beta < 2^{m+1-\alpha}$.
Für $\alpha = 1$ und $0 \leq \beta < n$ ergibt sich:

$$\begin{aligned} S_{\alpha,\beta} &= \{k_1 \mid x_{\beta+1}^{k_1} \text{ taucht in einem Monom von } f \text{ auf}\} \\ &= \text{Menge der in } f \text{ auftretenden Exponenten von } x_{\beta+1} \end{aligned}$$

Für $\alpha = m+1$, also $\beta = 0$ ergibt sich

$$S_{m+1,0} = \{(k_1,\ldots,k_n) \mid x_1^{k_1}\cdot\ldots\cdot x_n^{k_n} \text{ taucht in einem Monom von } f \text{ auf}\}$$

dann die gesuchte Lösung der Interpolationsaufgabe.

Ausgehend von $S_{1,\beta}$ sollen $S_{\alpha,\beta}$ für $\alpha = 1,\ldots,m+1$ rekursiv bestimmt werden (vgl. Abb. 2.1).

Zunächst soll im Basisschritt für $0 \leq \beta < n$ $S_{1,\beta}$ bestimmt werden, d.h. es sollen für alle auftretenden Variablen die jeweils auftauchenden Exponenten bestimmt werden. Dazu wird f nach Potenzen von $x_{\beta+1}$ sortiert:

$$f(x) = \sum_{l=0}^{q-1} x_{\beta+1}^l \cdot P_{l,\beta+1}(\underbrace{x_1,\ldots,x_\beta}_{=:x'}, \underbrace{x_{\beta+2},\ldots,x_n}_{=:x''})\;.$$

$$\begin{tikzcd}[column sep=small]
\substack{S_{1,0}\\ \{x_1\}} \arrow[dr] & & \substack{S_{1,1}\\ \{x_2\}} \arrow[dl] & & \substack{S_{1,2}\\ \{x_3\}} \arrow[dr] & & \substack{S_{1,3}\\ \{x_4\}} \arrow[dl] & & \substack{S_{1,4}\\ \{x_5\}} \arrow[dr] & & \substack{S_{1,5}\\ \{x_6\}} \arrow[dl] & & \substack{S_{1,6}\\ \{x_7\}} \arrow[d] \\
& \substack{S_{2,0}\\ \{x_1x_2\}} \arrow[drr] & & & & \substack{S_{2,1}\\ \{x_3x_4\}} \arrow[dll] & & & & \substack{S_{2,2}\\ \{x_5x_6\}} \arrow[dr] & & & \substack{S_{2,3}\\ \{x_7\}} \arrow[dll] \\
& & & \substack{S_{3,0}\\ \{x_1x_2x_3x_4\}} \arrow[drrr] & & & & & & & \substack{S_{3,1}\\ \{x_5x_6x_7\}} \arrow[dllll] & & \\
& & & & & & \substack{S_{4,0}\\ \{x_1x_2x_3x_4x_5x_6x_7\}} & & & & & &
\end{tikzcd}$$

Abbildung 2.1: Rekursionsschema für $n = 7, m = \lceil \log_2 n \rceil = 3$

$P_{l,\beta+1} \in \mathrm{GF}(q)[x', x'']$ ist ein Polynom in $n-1$ Variablen. Damit ist $S_{1,\beta} = \{l | P_{l,\beta+1} \not\equiv 0\}$.
Man möchte den Nulltest [GKS 88] für $P_{l,\beta+1}$ anwenden. Dazu muß aus der Black-Box für f eine Black-Box für $P_{l,\beta+1}$ konstruiert werden. Sei dazu $\mathrm{GF}(q) = \{a_1, \ldots, a_q\}$. Dann ist für $1 \leq i \leq q$:

$$f(x', a_i, x'') = \sum_{l=0}^{q-1} a_i^l \cdot P_{l,\beta+1}(x', x'') .$$

Man erhält also das folgende lineare Gleichungssystem:

$$\underbrace{\begin{pmatrix} a_1^0 & \ldots & a_1^{q-1} \\ \vdots & & \vdots \\ a_q^0 & \ldots & a_q^{q-1} \end{pmatrix}}_{=:\, \mathcal{A}} \cdot \underbrace{\begin{pmatrix} P_{0,\beta+1}(x', x'') \\ \vdots \\ P_{q-1,\beta+1}(x', x'') \end{pmatrix}}_{=:\, \mathcal{P}_l} = \underbrace{\begin{pmatrix} f(x', a_1, x'') \\ \vdots \\ f(x', a_q, x'') \end{pmatrix}}_{=:\, \mathcal{F}}$$

$\mathcal{A}$ ist eine nicht singuläre Vandermonde Matrix, also ist das Gleichungssystem eindeutig lösbar. Dann ist mit

$$\begin{aligned} P_{l,\beta+1}(x', x'') &= (0, \ldots, 0, 1, 0, \ldots, 0) \cdot \mathcal{P}_l \\ &= \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{:=\, u_l} \cdot \mathcal{A}^{-1} \cdot \mathcal{F} \end{aligned}$$

eine Black-Box für $P_{l,\beta+1}$ gegeben, wobei u_l an der l-ten Position eine Eins hat. □

Nun wollen wir rekursiv die Mengen $S_{\alpha,\beta}$ berechnen. Seien für festes $\alpha < m+1$ die Mengen $S_{\alpha,\beta}$ für $0 \leq \beta < 2^{m+1-\alpha}$ bestimmt.
Dann soll $S_{\alpha+1,\beta}$ aus $S_{\alpha,2\beta}$ und $S_{\alpha,2\beta+1}$ bestimmt werden.
Betrachte dazu den Aufbau der Teillösungen:

$$\overbrace{x_{\beta\cdot\, 2^\alpha+1}^{k_1} \cdot \ldots \cdot x_{\beta\cdot\, 2^\alpha+2^{\alpha-1}}^{k_{2^{\alpha-1}}} \cdot x_{\beta\cdot\, 2^\alpha+2^{\alpha-1}+1}^{k_{2^{\alpha-1}+1}} \cdot \ldots \cdot x_{\beta\cdot\, 2^\alpha+2^\alpha}^{k_{2^\alpha}}}^{k \,\in\, S_{\alpha+1,\beta}}$$

$$= \underbrace{x_{2\beta\cdot\ 2^{\alpha-1}+1}^{k_1} \cdot \ldots \cdot x_{2\beta\cdot\ 2^{\alpha-1}+2^{\alpha-1}}^{k_{2^{\alpha-1}}}}_{k\,\in\, S_{\alpha,2\beta}} \cdot \underbrace{x_{(2\beta+1)\cdot\ 2^{\alpha-1}+1}^{k'_1} \cdot \ldots \cdot x_{(2\beta+1)\cdot\ 2^{\alpha-1}+2^{\alpha-1}}^{k'_{2^{\alpha-1}}}}_{k'\,\in\, S_{\alpha,2\beta+1}} \quad .$$

Sei $S := S_{\alpha,2\beta} \times S_{\alpha,2\beta+1}$, also

$$S = \{u \cdot v \mid u \in S_{\alpha,2\beta}, v \in S_{\alpha,2\beta+1}\}$$

Sicherlich ist $S \supseteq S_{\alpha+1,\beta}$, im folgenden sollen diejenigen Elemente aus S bestimmt werden, die in einem Monom von f auftauchen.

$$S_{\alpha+1,\beta} = \{u \in S \mid \ u \text{ taucht in einem Monom von } f \text{ auf}\} \,.$$

Da f t-sparse ist, ist $|S_{\alpha,2\beta}| \leq t$, $|S_{\alpha,2\beta+1}| \leq t$, also ist $|S| \leq t^2$.

Das weitere Vorgehen ist dem Basisschritt ähnlich; man möchte das Polynom f nach Elementen aus S sortieren, also nach den entsprechenden Polynomen in 2^α Variablen ordnen, und dann entscheiden, ob die dazugehörigen Koeffizientenpolynome ungleich Null sind. Dies geschieht durch Lösung von Gleichungssystemen mit Matrizen, deren Einträge Potenzen von Werten von Elementen aus S, also von Monomen in 2^α Variablen, sind. Damit die sich ergebenden Gleichungssysteme gelöst werden können, müssen diese Werte zu einer nichtsingulären Matrix führen, also sind die Monome zu trennen.

Beim Basisschritt ist dies einfach, denn die Monome in einer Variablen werden durch die Elemente aus $\mathrm{GF}(q)$ getrennt. Für den Rekursionsschritt läßt sich diese Aufgabe analog zum Nulltest durchführen.

Bestimme dazu s_1 so, daß $q^{s_1} - 1 > 4qn(n-1)\binom{t^2}{2}$ mit s_1 minimal erfüllt ist, also $s_1 \leq \lceil 4\log_q(nt) + 3\rceil$.
Konstruiere den Körper $\mathrm{GF}(q^{s_1})$ aus $\mathrm{GF}(q)$ und s_1 und bestimme ein primitives Element ω_1 aus $\mathrm{GF}(q^{s_1})$.
Sei $N_1 = \frac{\lceil q^{s_1}-1\rceil}{4nq}$, damit ist $N_1 > (n-1)\binom{t^2}{2}$. Bestimme eine Primzahl p_1 mit $2N_1 < p_1 \leq 4N_1$
Bilde $(N_1 \times N_1)$ Cauchymatrix $C = (c_{ij})$ mit $c_{ij} := \frac{1}{i+j} \bmod p_1$ für $1 \leq i, j \leq N_1$ Sei V eine beliebige $(N_1 \times 2^\alpha)$-Teilmatrix von C.

Dann gilt analog zum Beweis von Satz 2.4.15:
Es gibt eine Reihe v_0 aus V, so daß sich alle Elemente aus S an der Stelle $\omega_1^{v_0}$ unterscheiden, d.h. mit $x^u = x_{\beta\cdot\ 2^\alpha+1}^{u_1} \cdot \ldots \cdot x_{\beta\cdot\ 2^\alpha+2^\alpha}^{u_{2^\alpha}}$ gilt:

$$u_1, u_2 \in S, u_1 \neq u_2 \implies x^{u_1}|_{\omega_1^{v_0}} \neq x^{u_2}|_{\omega_1^{v_0}} \tag{2.7}$$

Sei

$$\Omega_u := x^u|_{\omega_1^{v_0}} = \omega_1^{u\cdot v_0} \quad \text{für } u \in S$$

Sei $x' = x_1 \cdot \ldots \cdot x_{\beta\cdot\ 2^\alpha}$ und $x'' = x_{(\beta+1)\cdot\ 2^\alpha+1} \cdot \ldots \cdot x_n$.
Nun wird f nach Elementen aus S sortiert:

$$f(x', x, x'') = \sum_{u\in S} x^u \cdot P_u(x', x'') \tag{2.8}$$

$P_u \in \mathrm{GF}(q)[x', x'']$ ist ein Polynom in $n - 2^\alpha$ Variablen.
Damit ist

$$S_{\alpha+1,\beta} = \{u | P_u \not\equiv 0\}$$

Analog zum Basisschritt soll der Nulltest [GKS 88] für P_u angewendet werden; also muß aus der Black-Box für f eine Black-Box für P_u konstruiert werden. Dazu wird (2.8) an den Stellen $x = \omega_1^{v_0 \cdot l}$ für $0 \leq l < |S|$ ausgewertet:

$$f(x', \omega_1^{v_{01} \cdot l}, \ldots, \omega_1^{v_{02^\alpha} \cdot l}, x'') = \sum_{u \in S} \Omega_u^l \cdot P_u(x', x'') \, .$$

Damit ergibt sich das lineare Gleichungssystem:

$$\underbrace{\left(\Omega_u^l\right)_{\substack{0 \leq l < |S| \\ u \in S}}}_{=\mathcal{B}} \cdot \underbrace{(P_u(x', x''))_{u \in S}}_{=: \mathcal{P}} = \underbrace{\left(f(x', \omega_1^{v_0 \cdot l}, x'')\right)_{0 \leq l < |S|}}_{=: \mathcal{F}} \tag{2.9}$$

Wegen der Aussage (2.7) ist $\mathcal{B}$ eine nicht singuläre Vandermonde'sche Matrix, also ist das Gleichungssystem (2.9) eindeutig lösbar. Dann ist mit

$$\begin{aligned} P_u(x', x'') &= (0, \ldots, 0, 1, 0, \ldots, 0) \cdot \mathcal{P} \\ &= \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{:= s_u} \cdot \mathcal{B}^{-1} \cdot \mathcal{F} \end{aligned}$$

eine Black-Box für P_u gegeben, wobei s_u an der Position eine Eins hat, die der Position von u in S entspricht.

Für $\alpha = m+1$ hat man alle in f auftretenden Monome bestimmt. Die zugehörigen Koeffizienten ergeben sich durch die Lösung des Gleichungssystems (2.6) des Nulltests. □

Damit haben wir den

Algorithmus 2.4.16 Interpolationsalgorithmus [GKS 88]

Eingabe: Black-Box für $f \in \mathrm{GF}(q^s)[x_1, \ldots, x_n]$, t-sparse, s beliebig.

Ausgabe: Sparse Darstellung von f: $(c_u, u)_{u \in \mathrm{supp}(f)}$.

Schritt A: Führe die Schritte 1 bis 5 des Nulltestes durch und stelle die Nulltestmenge

$$\omega^{l \cdot \bar{c}_i} = (\omega^{l \cdot \bar{c}_{i,1}}, \omega^{l \cdot \bar{c}_{i,2}}, \ldots, \omega^{l \cdot \bar{c}_{i,n}}) \quad \text{für } 0 \leq i \leq N, \ 1 \leq l < t$$

bereit.

Schritt B: Berechne für den Basisschritt die Matrix

$$\mathcal{A}^{-1} = \left(a_k^j\right)^{-1}_{\substack{a_k \in \mathrm{GF}(q) \\ 0 \leq j < q}} .$$

Schritt C: Führe parallel für jedes $0 \leq \beta < n$ den Basisschritt durch.

Basisschritt

Schritt 1: Bestimme durch Anfragen an die Black-Box mit $s = \lceil \log_q(4qn(n-1)\binom{t}{2} + 2) \rceil$ die Werte

$$f_{i,l}^{(k)} := f(\omega^{l \cdot \bar{c}_{i,1}}, \ldots, \omega^{l \cdot \bar{c}_{i,\beta}}, a_k, \omega^{l \cdot \bar{c}_{i,\beta+2}}, \ldots, \omega^{l \cdot \bar{c}_{i,n}})$$

für $0 \leq l < t$, $1 \leq i \leq N$, $1 \leq k \leq q$ und

$$f_{0,t}^{(k)} := f(0, \ldots, 0, a_k, 0, \ldots, 0) \quad \text{für } 1 \leq k \leq q.$$

Schritt 2: Berechne parallel für $0 \le l \le t$, $0 \le i < N$ die Folge von Vektoren

$$P_{i,l} = \mathcal{A}^{-1} \cdot \left(f_{i,l}^{(k)}\right)_{1 \le k \le q} .$$

Schritt 3: $S_{1,\beta} = \{ \, k \mid \exists (i,l) \, : \, P_{i,l}^{(k)} \neq 0 \, \}$.

Schritt D: Bereite den Rekursionsschritt vor:

Schritt 4: Bestimme s_1 so, daß $q^{s_1} - 1 > 4qn \cdot (n-1) \cdot \binom{t^2}{2}$ mit s_1 minimal, also $s_1 = \lceil \log_q (4qn(n-1)\binom{t^2}{2} + 2) \rceil$.

Schritt 5: Konstruiere den Körper $\mathrm{GF}(q^{s_1})$ aus $\mathrm{GF}(q)$ und s_1. Bestimme ein primitives Element ω_1 aus $\mathrm{GF}(q^{s_1})$.

Schritt 6: $N_1 = \lceil \frac{q^{s_1}-1}{4nq} \rceil$. Bestimme mit Hilfe des Siebes von Erastosthenes eine Primzahl p_1 mit $2N_1 < p_1 \le 4N_1$.

Schritt 7: Bestimme $c_{ij} := \frac{1}{i+j} \bmod p_1$ für $1 \le i,j \le N_1$ durch den Euklidischen Algorithmus. Sei $C = (c_{i,j})_{1 \le i,j \le N_1}$.

Schritt 8: Sei $V = (v_{i,j})$ eine beliebige $(N_1 \times n)$-Teilmatrix von C.

Schritt 9: Stelle die Testmenge

$$\omega_1^{l \cdot v_i} = (\omega_1^{l \cdot v_{i,1}}, \omega_1^{l \cdot v_{i,2}}, \ldots, \omega_1^{l \cdot v_{i,n}}) \quad \text{für } 0 \le i \le N,\ 0 \le l < t$$

bereit.

Schritt E: Sei $m = \lceil \log_2 n \rceil$. Führe sequentiell für $\alpha = 1, \ldots, m$ Schritt F durch.

Schritt F: Führe parallel für jedes $0 \le \beta < 2^{m-\alpha}$ den Rekursionsschritt durch.

Rekursionsschritt

Schritt 10: Setze $S := S_{\alpha,2\beta} \times S_{\alpha,2\beta+1}$.

Schritt 11: Bestimme parallel für $1 \le j \le N_1$ und $u \in S$ die Summen

$$\sigma_{u,j} = \sum_{i=1}^{2^\alpha} u_i \cdot v_{j,i}.$$

Sei v_j eine Reihe aus V für die die Menge $\{\sigma_{u,j} \mid u \in S\}$ aus paarweise verschiedenen Elementen besteht. Berechne $\Omega_u = \omega_1^{\sigma_{u,j}}$, $u \in S$.

Schritt 12: Berechne die Matrix

$$\mathcal{B}^{-1} = \left(\Omega_u^k\right)^{-1}_{\substack{0 \le k < |S| \\ u \in S}} .$$

Schritt 13: Bestimme durch Anfragen an die Black-Box mit $s = s_1$ die Werte

$$f_{i,l}^{(u)} := f(\omega^{l \cdot \bar{c}_{i,1}}, \ldots, \omega^{l \cdot \bar{c}_{i,\beta \cdot 2^\alpha}}, \omega_1^{k \cdot v_{j,1}}, \ldots, \omega_1^{k \cdot v_{j,2^\alpha}}, \omega^{l \cdot \bar{c}_{i,(\beta+1) \cdot 2^\alpha + 1}}, \ldots, \omega^{l \cdot \bar{c}_{i,n}})$$

für $0 \le l < t$, $1 \le i \le N$, $0 \le k < |S|$ und

$$f_{0,t}^{(k)} := f(0, \ldots, 0, \omega_1^{k \cdot v_{j,1}}, \ldots, \omega_1^{k \cdot v_{j,2^\alpha}}, 0, \ldots, 0) \quad \text{für } 0 \le k < |S|.$$

Schritt 14: Berechne parallel für $0 \le l \le t$, $0 \le i < N$ die Folge von Vektoren

$$P_{i,l} = \mathcal{B}^{-1} \cdot \left(f_{i,l}^{(u)}\right)_{u \in S}.$$

Schritt 15: $S_{\alpha+1,\beta} := \{ \, u \mid \exists (i,l) \; : \; P_{i,l}^{(u)} \neq 0 \, \}.$

Schritt G: $\mathrm{supp}(f) = S_{m+1,0}$.
Die Koeffizienten von f ergeben sich durch die Komponenten eines Vektors $P_{i,l}$ des letzten Rekursionsschrittes.

□

Wir wollen nun den Zeit- und Prozessorenbedarf abschätzen. Wir beginnen mit der Abschätzung der Größenordnungen von s, s_1, N und N_1. Man beachte, daß $s \le \lceil 3+2\log_q(nt)\rceil$, und daher $s = O(\log_q(nt))$. Aus der Definition von N und N_1 sind $N < qnt^2$ und $N_1 < qnt^4$ abzuleiten. Ferner gilt: $|\mathrm{GF}(q^s)| \le Nnq$, sowie $|\mathrm{GF}(q^{s_1})| \le N_1 nq$.

Zunächst analysieren wir den Nulltest. Um den Körper $\mathrm{GF}(q^s)$ zu konstruieren, suchen wir nach einem irreduziblen Polynom $\Phi \in \mathrm{GF}(q)[z]$ vom Grad s. Überprüfung der Reduzierbarkeit für die q^{s+1} univarianten Polynome vom Grad s anhand des Algorithmus von Berlekamp [Be 70] ergibt ein irreduzibles Φ. Hierfür werden $q^{s+1}O(\log^{5.5}(Nnq)) = O(Nnq^2\log^{5.5}(Nnq))$ Prozessoren und $O(\log^2(Nnq))$ Parallelzeit benötigt. $\mathrm{GF}(q^s)$ ist isomorph zu $\mathrm{GF}(q)[z]/\Phi$. Die Arithmetik in $\mathrm{GF}(q^s)$ wird durch die Arithmetik für Polynome vom Grad s in $\mathrm{GF}(q)$ und Reduktion modulo Φ repräsentiert. Werden die Polynome z.B. durch ihre Begleitmatrizen dargestelt, so ergibt sich für die Arithmetik in $\mathrm{GF}(q^s)$ gegenüber der Arithmetik in $\mathrm{GF}(q)$ ein Faktor von höchstens $O(\log^{2.5}(ntq))$ Prozessoren und $O(\log(ntq))$ Parallelzeit.

Um ein primitives Element ω in $\mathrm{GF}(q^s)$ zu finden, wird $q^s - 1$ in Primfaktoren zerlegt: $q^s - 1 = \prod p_i^{\alpha_i}$. Dies kostet mittels des Siebes von Erastosthenes $O((Nnq)^{0.75}\log(Nnq))$ Prozessoren und $O(\log(Nnq))$ Parallelzeit. Anschließend wird für jedes $a \in \mathrm{GF}(q^s)$ überprüft, ob die Potenzen $a^{\frac{q^s-1}{p_i}}$ von Eins verschieden sind. In diesem Fall ist a ein primitives Element in $\mathrm{GF}(q^s)$. Berechnung dieser Potenzen für alle $a \in \mathrm{GF}(q^s)$ ergibt einen Aufwand von höchstens $O((Nnq)\log^{3.5}(Nnq))$ Prozessoren und $O(\log^2(Nnq))$ Parallelzeit.

Der nächste Hauptschritt ist die Bereitstellung der Auswertungsstellen, d.h. es müssen Nnt Potenzen von ω berechnet werden, was einen Aufwand von $O(Nnt\log^{2.5}(Nnq))$ Prozessoren und $O(\log^2(Nnqt))$ Parallelzeit bedeutet.

Insgesamt betrgt der Aufwand für den Nulltest $O(Nnqt\log^{3.5}(Nnq))$ Prozessoren und $O(\log^2(Nnqt))$ Parallelzeit.

Die Komplexität des Basisschrittes wird durch die Komplexität des Nulltests und die Komplexität für die Invertierung der $(q \times q)$-Matrix $A = (a_i^j)_{\substack{a_i \in \mathrm{GF}(q) \\ 0 \le j < q}}$ bestimmt. Dabei ergibt sich ein Aufwand von $O(q^{2.5}\log^2 q)$ Prozessoren und $O(\log^2 q)$ Parallelzeit unter Anwendung von [Mu 87]. Daher beträgt die Komplexität des Basisschrittes $O(Nnqt\log^{3.5}(Nnq) + q^{2.5}\log^2 q)$ Prozessoren und $O(\log^2(Nnqt))$ Parallelzeit. Die Black-Box wird parallel für tqN Argumente ausgewertet.

Der Rekursionsschritt wird parallel für $0 \le \beta < 2^{m-\alpha}$ durchgeführt. Um die Black-Box für P_u zu konstruieren, muß die Matrix B in (2.9) berechnet werden. Hierfür müssen wir die $N_1 \cdot t^2$

Summen $\sigma_{u,j} = \sum_{i=1}^{2^\alpha} u_i \cdot v_{j,i}$ für $u \in S$ und $1 \leq j \leq N_1$ berechnen und überprüfen, ob sie für ein festes j paarweise verschieden sind. Sei diese Reihe durch v_0 bezeichnet. Mit $\Omega_u = \omega_1^{\sigma_{u,0}}$ für $u \in S$ ergibt sich die $(t^2 \times t^2)$-Matrix B. Dies kostet $O(N_1 t^2 \log^{2.5}(N_1 nq))$ Prozessoren und $O(\log^2(N_1 nqt))$ Parallelzeit. Der Aufwand für die Invertierung der Matrix B beträgt $O(t^5 \log^{2.5}(N_1 nq))$ Prozessoren und $O(\log^2 t)$ Parallelzeit. Im Schritt 13 wird die Black-Box für $O(t^3 N)$ Auswertungen befragt. Im Schritt 14 werden tN Vektoren der Länge $O(t^2)$ über $\mathrm{GF}(q^{s_1})$ parallel berechnet. Dies benötigt $O(Nt^3 \log^2(N_1 nq))$ Prozessoren und $O(\log t)$ Parallelzeit.

Hieraus ergibt sich ein Aufwand von $2^{m-\alpha}(O(N_1 t^2 \log^{2.5}(N_1 nq)) + O(t^5 \log^{2.5}(N_1 nq)))$ Prozessoren, $O(\log^2(N_1 nqt))$ Parallelzeit und $2^{m-\alpha}(O(t^3 N))$ Anfragen pro Rekursionsschritt.

Insgesamt ergibt sich durch Summierung über $O(\log n)$ Rekursionsschritte unter Anwendung der Abschätzungen für N und N_1 eine Gesamtkomplexität des Algorithmus von

- $O(t^6 n^2 q^3 \log^{5.5}(ntq) + q^{2.5} \log^2 q)$ Prozessoren,
- $O(\log^3(ntq))$ Parallelzeit und
- $O(qn^2 t^5)$ Befragungen

und damit die Aussage des Satzes 2.4.11. □

Kapitel 3

Zählalgorithmen

3.1 Berechnungsmodell

In diesem Kapitel wollen wir uns einem ganz neuen Problemkreis zuwenden. Wir stellen die Frage nach der Anzahl von Lösungen zu gewissen Problemen. Diese Fragestellung erweitert das Entscheidungsproblem, bei dem nur nach der Existenz einer Lösung gefragt ist. Ein Beispiel haben wir bereits in Abschnitt 2.3.1 kennnengelernt. Dort haben wir sowohl das Entscheidungsproblem als auch das Zählproblem gelöst. Weitere Anwendungen von Zählalgorithmen sind die Bestimmung der Anzahl von erfüllenden Belegungen einer booleschen Formel, die Anzahl der Nullstellen eines Polynoms über endlichen Körpern und graphentheoretische Fragen wie die Anzahl der Matchings, Cliquen etc. in einem Graphen.

Zählprobleme lassen sich nicht direkt mit Entscheidungsproblemen vergleichen. So ist das Ergebnis eines Zählalgorithmus eine natürliche Zahl. In [Va 79b] wird ein geeignetes Berechnungsmodell für Zählprobleme eingeführt.

Definition 3.1.1 Counting TM
Eine Counting TM (kurz CTM) M ist eine nichtdeterministische Turing Maschine, die außerdem ohne zusätzlichen Berechnungsaufwand auf einem Extraband die Anzahl akzeptierender Berechnungsfolgen ausdruckt. Damit berechnet diese TM eine Funktion

$$f_M : \Sigma^* \to I\!N.$$

Mit Hilfe dieser Definition können wir nun Komplexitätsklassen definieren.

Definition 3.1.2 $\#\mathcal{P}$
Mit $\#\mathcal{P}$ bezeichnen wir alle Probleme, die mit einer polynomiell Zeit beschränkten Counting TM berechnet werden können.

Anhand der Definition ist klar, daß die zugehörigen Zählprobleme von Problemen aus $\mathcal{NP}$ in $\#\mathcal{P}$ liegen. Wie bei den Entscheidungsproblemen interessieren uns auch hier die schwersten Probleme der Klasse. Dazu müssen wir zunächst einen geeigneten Reduktionsbegriff einführen.

Definition 3.1.3 Orakel-Counting TM
Eine Orakel-Counting TM mit dem Orakel X ist eine deterministische TM M, die einen speziellen Orakelzustand in ihrer endlichen Kontrolle besitzt. Geht M in diesen Zustand über, so wird der Inhalt eines speziellen Arbeitsbandes, des Orakelbandes, als Eingabe für das Orakel X verwendet, welches das Ergebnis, d.h. die Anzahl der Lösungen des Problems X auf das Orakelband schreibt. Mit diesem Ergebnis kann M dann weiterarbeiten. Wir bezeichen die Klasse der so in polynomiell beschränkter Zeit berechenbaren Probleme mit $\mathcal{P}^X$.

Damit können wir eine geeignete Reduktion wie folgt definieren.

Definition 3.1.4 Ein Zählproblem Y ist auf X reduzierbar, falls $Y \in \mathcal{P}^X$. Ein Zählproblem X ist $\#\mathcal{P}$ -hart, wenn

$$\#\mathcal{P} \subseteq \mathcal{P}^X$$

Ein Zählproblem X ist $\#\mathcal{P}$ -vollständig, wenn X $\#\mathcal{P}$ -hart ist und X in $\#\mathcal{P}$ liegt.

Kandidaten für $\#\mathcal{P}$ -vollständige Zählprobleme sind Zählprobleme, die aus $\mathcal{NP}$ -vollständigen Entscheidungsproblemen hervorgehen. Ob alle diese Zählprobleme wirklich $\#\mathcal{P}$ -vollständig sind, ist ein offenes Problem. Valiant gibt in [Va 79a] ein erstes nichttriviales $\#\mathcal{P}$ -vollständiges Zählproblem an.

Satz 3.1.5 Die Berechnung der Permanente einer $n \times n$ Matrix mit Einträgen aus $\{0, 1\}$, d.h. die Berechnung der Anzahl Perfekter Matchings eines bipartiten Graphen, ist $\#\mathcal{P}$ -vollständig.

Dieser Satz ist sehr erstaunlich, da das zugehörige Entscheidungsproblem, d.h. der Test auf die Existenz von Perfekt Matching, in polynomieller Zeit möglich ist.

3.2 DNF - Zählprobleme

Ausgehend von dem $\#\mathcal{P}$ -vollständigen Problem der Permanentenberechnung hat Valiant noch eine Reihe weiterer $\#\mathcal{P}$ -vollständiger Probleme gefunden (siehe [Va 79b]). Für unsere Anwendungen wird das folgende Problem am wichtigsten sein.

Definition 3.2.1 monotone 2-DNF Zählproblem
Gegeben sei eine boolesche Formel f in disjunktiver Normalform, so daß keine Klausel mehr als zwei Literale enthält. Außerdem gebe es keine negierten Literale, d.h. die Formel ist monoton. Bezeichne $\#f$ die Anzahl der erfüllenden Belegungen

$$\#f := |\{\bar{x} \mid f(\bar{x}) = 1\}|.$$

Das monotone 2-DNF Zählproblem ist die Berechnung von $\#f$.

Satz 3.2.2 Das monotone 2-DNF Zählproblem ist $\#\mathcal{P}$ -vollständig.

Bemerkung 3.2.3 Das monotone DNF Erfüllungsproblem ist trivialerweise in $\mathcal{P}$.

Wir sehen also, daß viele wichtige und grundlegende Zählprobleme so schwer sind, daß wir keine effizienten polynomiellen Lösungen erwarten dürfen. Was kann uns aus diesem Dilemma retten?

- Einschränkungen auf spezielle Klassen von zu zählenden Objekten.
- Approximative anstatt exakter Lösungen.

Im Falle von DNF-Formeln zeigt Satz 3.2.2, daß Einschränkungen hier nicht unbedingt zum Ziel führen. Daher werden wir nun versuchen Algorithmen zu entwickeln, um die Anzahl erfüllender Lösungen einer booleschen Formel in disjunktiver Normalform approximativ zu bestimmen. Dafür müssen wir die Eigenschaften solcher approximativer Algorithmen genauer definieren.

Definition 3.2.4 ϵ-Approximationsalgorithmus
Ein Algorithmus A für das Zählproblem f ist ein ϵ-Approximationsalgorithmus, falls für die Ausgabe y von A

$$(1-\epsilon)\cdot \#f \leq y \leq (1+\epsilon)\cdot \#f$$

gilt. Die Eingabegrößen des Algorithmus sind die Eingabegröße des Zählproblems f und der Kehrwert von ϵ.

Definition 3.2.5 (ϵ, δ)-Approximationsalgorithmus
Ein Algorithmus A für das Zählproblem f ist ein (ϵ, δ)- Approximationsalgorithmus, falls für die Ausgabe y von A

$$Pr[(1-\epsilon)\cdot \#f \leq y \leq (1+\epsilon)\cdot \#f] \geq 1-\delta$$

gilt. Die Eingabegrößen des Algorithmus sind die Eingabegröße des Zählproblems f, der Kehrwert von ϵ und $\log(1/\delta)$.

Im folgenden werden wir Approximationsalgorithmen mit polynomieller Laufzeit kennenlernen. Wir beginnen im nächsten Abschnitt mit einem (ϵ, δ)-Approximationsalgorithmus für das DNF-Zählproblem. Er ist der Arbeit [KLM 87] entnommen.

3.2.1 (ϵ, δ)-Approximationsalgorithmus für das DNF Zählproblem

In diesem Abschnitt werden wir einen probabilistischen Zählalgorithmus kennenlernen. Das Prinzip dieses Algorithmus ist einfach:
Dazu sei $G = \{\bar{x} \mid f(\bar{x}) = 1\}$ und $U \supseteq G$. Der Algorithmus besteht aus N unabhängig ausgeführten Versuchen. Jeder dieser Versuche besteht aus den folgenden Schritten.

1. Wähle zufällig ein Element $\tilde{x}$ aus U. Zufällig bedeutet dabei unabhängig und gemäß einer uniformen Verteilung.
2. Teste ob $\tilde{x} \in G$, d.h. ob $f(\tilde{x}) = 1$.

3. Setze die Ausgabe

$$Y := \begin{cases} |U| & \text{falls } f(\tilde{x}) = 1 \\ 0 & \text{sonst .} \end{cases}$$

Damit ist das Ergebnis eines Schrittes eine Zufallsvariable Y. Der Erwartungswert von Y ist

$$E[Y] = \frac{1}{|U|} \sum_{x \in U} |U| \cdot f(x) = \sum_{x \in G} f(x) = |G|.$$

Damit gilt auch für die Ausgabe $\tilde{Y} = \frac{1}{N} \sum_{i=1}^{N} Y_i$ des Zählalgorithmus

$$E[\tilde{Y}] = |G|,$$

d.h. der Mittelwert der Ergebnisse der einzelnen Versuche ist eine geeignete Zufallsvariable für die Anzahl der erfüllenden Belegungen. Der folgende Satz gibt Auskunft über die minimale Anzahl von Versuchen, so daß diese Zufallsvariable eine (ϵ, δ)-Approximation darstellt.

Satz 3.2.6 (Bernstein)
Sei $\mu = |G|/|U|$. Dann ist der obige Algorithmus ein (ϵ, δ)-Approximationsalgorithmus, wenn die Anzahl der Versuche N größer gewählt wird als

$$\frac{1}{\mu} \cdot \frac{4}{\epsilon^2} \ln(\frac{2}{\delta}).$$

Der Beweis dieses Satzes benötigt einige wahrscheinlichkeitstheoretische Lemmata.

Lemma 3.2.7 Seien $Z, Z_1, Z_2, \cdots Z_k$ unabhängige Zufallsvariablen mit der gleichen Verteilung. Seien c und d Konstanten. Dann gilt

$$E[e^{d(Z_1 + \cdots + Z_k) + c}] = (E[e^{dZ}])^k \cdot e^c.$$

Lemma 3.2.8 Für jede Zufallsvariable Z und jede Konstante $d \geq 0$ gilt

$$Pr[Z \geq 0] \leq E[e^{dZ}].$$

Beweis: Sei die Zufallsvariable

$$Z' = \begin{cases} 1 & \text{falls } Z \geq 0 \\ 0 & \text{sonst .} \end{cases}$$

Dann ist $Z' \leq e^{dZ}$ und daher $E[Z'] \leq E[e^{dZ}]$. Außerdem gilt $E[Z'] = Pr[Z \geq 0]$. □

Im folgenden seien die Zufallsvariablen $Y, Y_1, \ldots, Y_N$ $\{0,1\}$-wertige unabhängige Zufallsvariablen mit der gleichen Verteilung und Erwartungswert $E[Y] = \mu$.

Lemma 3.2.9 Sei $d \leq 1$ eine Konstante. Dann gilt

$$E[e^{dY}] \leq e^{\mu d(1+d)}.$$

Beweis: Für den Erwartungswert von e^{dY} gilt

$$E[e^{dY}] = \mu \cdot e^d + (1-\mu).$$

Die Funktion e^x können wir für $x \leq 1$ durch $1 + x + x^2$ nach oben und für $x \geq 0$ durch $1 + x$ nach unten abschätzen. Dadurch ergibt sich

$$E[e^{dY}] \leq \mu \cdot (1 + d + d^2) + (1-\mu) = 1 + \mu d(1+d).$$

Benutzen wir die zweite Abschätzung, so erhalten wir

$$E[e^{dY}] \leq e^{\mu d(1+d)}.$$

□

Mit diesem Lemma können wir die Wahrscheinlichkeit für große Abweichungen der Ausgabe des Algorithmus von dem zu berechnenden Wert beschränken. Es gilt das folgende

Korollar 3.2.10 Für $\epsilon \leq 2$ gilt

$$Pr[\sum_{i=1}^{N} Y_i > (1+\epsilon)\mu N] \leq e^{-\mu\epsilon^2 N/4}.$$

Beweis: Bezeichne

$$(*) = Pr[\sum_{i=1}^{N} Y_i > (1+\epsilon)\mu N].$$

Aus Lemma 3.2.8 folgt

$$(*) \leq E[e^{d\cdot(\sum_{i=1}^{N} Y_i - (1+\epsilon)\mu N)}].$$

Nun wenden wir Lemma 3.2.7 an und es ergibt sich

$$(*) \leq E[e^{dY}]^N \cdot e^{-d(1+\epsilon)\mu N}.$$

Nun schätzen wir $E[e^{dY}]$ mittels Lemma 3.2.9 ab und setzen $d = \epsilon/2$:

$$(*) \leq e^{-\mu\epsilon^2 N/4}.$$

□

Damit haben wir die Wahrscheinlichkeit für eine Abweichung nach oben abgeschätzt. Nun müssen wir auch die Wahrscheinlichkeit für Abweichungen nach unten begrenzen. Dies geschieht analog.

Lemma 3.2.11 Sei $d \leq 1$ eine Konstante. Dann gilt

$$E[e^{-dY}] \leq e^{-\mu d(1-\frac{d}{2})}.$$

Beweis: Für den Beweis benötigen wir jetzt Abschätzungen der Funktion e^{-x}. Es gilt $1-x \leq e^{-x} \leq 1 - x + x^2/2$. Aus diesen Ungleichungen ergibt sich

$$E[e^{-dY}] = \mu e^{-d} + (1-\mu) \leq \mu(1 - d + d^2/2) + 1 - \mu = 1 - \mu d(1 - d/2) \leq e^{-\mu d(1-d/2)}.$$

Nun können wir auch die Wahrscheinlichkeit für ein Abweichen nach unten beschränken.

Korollar 3.2.12 Für $\epsilon \leq 2$ gilt

$$Pr[\sum_{i=1}^{N} Y_i < (1-\epsilon)\mu N] \leq e^{-\mu\epsilon^2 N/4}.$$

BEWEIS: Analog zum Beweis zu Korollar 3.2.10. □

Mit Hilfe der Korollare 3.2.10 und 3.2.12 ergibt sich der Satz von Bernstein.

Im folgenden sei G nun die Menge aller Einsstellen von f. Nun wollen wir versuchen eine geeignete Obermenge U zu konstruieren. Diese Obermenge U muß die folgenden Eigenschaften besitzen, damit der Satz 3.2.6 von Bernstein eine polynomielle Laufzeit garantiert.

- $|U|/|G|$ muß polynomiell in der Eingabegröße sein.
- Der Wert $|U|/|G|$ muß in polynomieller Zeit berechenbar sein.
- Es muß in polynomieller Zeit möglich sein, Elemente von U gleichverteilt zu wählen.

Die Menge $U = \{0,1\}^n$ aller Vektoren der Länge n ist nicht geeignet, da der Quotient $|U|/|G|$ für DNF-Formeln mit nur wenigen erfüllenden Belegungen nicht polynomiell beschränkt ist. Die Menge U muß also kleiner gewählt werden. Um diese Wahl durchzuführen, beachten wir, daß f in disjunktiver Normalform gegeben ist

$$f = c_1 \vee c_2 \vee \cdots \vee c_m.$$

f wird also genau dann erfüllt, wenn mindestens eine Klausel c_i erfüllt wird. Bezeichne D_i die Menge der Erfüllenden für die Klausel c_i

$$D_i = \left\{ (\bar{x}_1 \dots \bar{x}_n) \mid \bar{x}_k = \left\{ \begin{array}{ll} 1 & x_k \text{ kommt als positives Literal vor} \\ 0 & x_k \text{ kommt als negatives Literal vor} \\ 1 \text{ oder } 0 & x_k \text{ kommt nicht vor} \end{array} \right. \right\}.$$

Die Kardinalität von D_i ist $|D_i| = 2^{n-\#\text{Variablen in } c_i}$. Außerdem gilt

$$G = \bigcup_{i=1}^{m} D_i,$$

d.h. G ist die Vereinigung von bekannten Mengen. Mit Hilfe der D_i's können wir jetzt die Kardinalität von G beschränken:

$$\max_{1\leq i\leq m} |D_i| \leq |G| \leq \sum_{i=1}^{m} |D_i| \leq m \max_{1\leq i\leq m} |D_i| \leq m|G|.$$

Wir wählen also U die direkte Summe der D_i. $U = \bigoplus_{i=1}^{m} D_i$ ist eine Obermenge der Menge $\tilde{G} = \{(i,s) \mid c_i(s) = 1 \text{ und } c_j(s) = 0 \text{ für alle } j < i\}$, die offensichtlich gleichmächtig zu G ist. Die Menge U erfüllt auch die Bedingungen des Satzes von Bernstein, da das Verhältnis $|U|/|\tilde{G}|$ durch m beschränkt ist. Wir können die Menge $\tilde{G}$ auch als Menge der Einsstellen der folgenden Funktion interpretieren:

$$\varphi(s,i) = \left\{ \begin{array}{ll} 1 & i \text{ ist die kleinste Nummer einer Klausel mit } c_i(s) = 1 \\ 0 & \text{sonst.} \end{array} \right.$$

Die Definitionsmenge von φ ist gerade U und $\tilde{G}$ die Menge der Einsstellen von φ. Es sei bemerkt, das wir für die Korrektheit des Algorithmus nur die Eigenschaft von φ benötigen, daß $\sum_{i=1}^{m} \varphi(s,i) = 1$ für alle $s \in \tilde{G}$ gilt. Die ersten beiden Anforderungen für den (ϵ, δ)-Approximationsalgorithmus haben wir erfüllt. Nun müssen wir noch zeigen, wie Elemente aus U mittels einer Gleichverteilung gewählt werden können. Dies geschieht folgendermaßen:

- Zuerst wähle eine Klausel mit Wahrscheinlichkeit $|D_i|/|U|$.
- Wähle eine erfüllende Belegung dieser Klausel mit Wahrscheinlichkeit $1/|D_i|$.

Mit Hilfe der obigen Wahl wird jedes Element von U mit gleicher Wahrscheinlichkeit $1/|U|$ gewählt. Damit haben wir einen (ϵ, δ)-Approximationsalgorithmus gewonnen.

Algorithmus 3.2.13 (ϵ, δ)-Approximationsalgorithmus

Eingabe: DNF-Formel f mit m Termen, Fehlertoleranz ϵ, Fehlerwahrscheinlichkeit δ.

Schritt 1: Berechne die Anzahl der erfüllenden Belegungen $|D_k|$ der k-ten Klausel.

Schritt 2: Berechne $A_i = \sum_{k=1}^{i} |D_k|$ für $0 \leq i \leq m$

Schritt 3: Setze $\tilde{Y} := 0$.

Schritt 4: Führe die Schleife $N = m \cdot \frac{4 \ln(2/\delta)}{\epsilon^2}$ mal durch

a) Wähle i mit Wahrscheinlichkeit $|D_i|/|U|$. Dies geschieht durch Wahl von $j \in \{1 \ldots |U|\}$ mit Wahrscheinlichkeit $1/|U|$ und binäre Suche des i's mit $A_{i-1} < j \leq A_i$.

b) Wähle erfüllende Belegung s der Klausel c_i, d.h s mit $c_i(s) = 1$.

c) Setze $k := 0$.

d) Suche die erste Klausel c_k mit $c_k(s) = 1$.

e) Wenn $k = i$ ist, so setze $Y := |U|$, sonst setze $Y := 0$.

f) Setze $\tilde{Y} := \tilde{Y} + Y$.

Ausgabe: $\tilde{Y}/N$.

Bemerkung 3.2.14 Die Schritte 4 a) und 4 b) wählen ein $(i, s) \in U$ mit Wahrscheinlichkeit $1/|U|$.

Satz 3.2.15 Der obige Algorithmus ist ein (ϵ, δ)-Approximationsalgorithmus und hat eine Laufzeit von

$$O\left(\frac{nm^2 \ln(1/\delta)}{\epsilon^2}\right).$$

Beweis: Der Satz 3.2.6 von Bernstein sichert uns bei richtig gewähltem N die (ϵ, δ)-Eigenschaft, falls in Schritt 4 e) die Zufallsvariable Y den Erwartungswert $E[Y] = |G|$ hat. Sei $\text{cov}\,(s) = \{i \mid c_i(s) = 1\}$ die Menge aller Klauseln, die von s erfüllt werden. Dann gilt

$$\sum_{i \in \text{cov}\,(s)} \varphi(s,i) = 1,$$

da es für jedes $s \in G$ nur ein i mit $\varphi(i,s) = 1$ gibt. Daraus folgt, daß $E[Y] = \sum_{(s,i)} \varphi(s,i) = \sum_s \sum_{i \in \mathrm{cov}\,(s)} \varphi(s,i) = \sum_s 1 = |G|$ ist. Daher liefert der Algorithmus eine (ϵ, δ)-Approximation.

Die Laufzeit ergibt sich aus der Anzahl der Schleifendurchläufe von Schritt 4 und dem teuersten Schritt in dieser Schleife. Dies ist die Bestimmung des i^*, für welches $\varphi(i^*, s) = 1$ gilt. Dazu muß für jedes $i \leq i^*$ überprüft werden, ob s die Klausel c_i erfüllt. Da jede solche Überprüfung $O(n)$ Zeit kostet und das gesuchte i^* von der Größenordnung $O(m)$ ist, ergibt sich die obige Laufzeit des Algorithmus. □

Mit dem Algorithmus 3.2.13 haben wir einen (ϵ, δ)-Approximationsalgorithmus gewonnen. Dabei ist die Berechnung der Funktion $\varphi(i,s)$ mit einer Laufzeit von $O(nm)$ sehr teuer. Für die Korrektheit des Algorithmus benötigen wir nur die Eigenschaft

$$\sum_{i \in \mathrm{cov}\,(s)} \varphi(s,i) = 1 \qquad (*).$$

Wir werden nun eine randomisierte Funktion φ' konstruieren, die die Eigenschaft $(*)$ besitzt, deren Berechnungzeit im Erwartungswert aber wesentlich günstiger ist.

3.2.2 Selbstjustierender Zählalgorithmus

Wir werden die Funktion φ des Zählalgorithmus durch eine Zufallsvariable φ' ersetzen, die nur von der erfüllenden Belegung s abhängig ist und für jedes s den Erwartungswert $E[\varphi'(s)] = 1/|\mathrm{cov}\,(s)|$ hat. Damit ergibt sich, daß φ' die Bedingung $(*)$ erfüllt. Ein einzelner Versuch sieht dann genauso aus wie oben:

- Wähle $(i,s) \in U$.
- Berechne $\varphi'(s)$.
- Setze $Y = |U| \cdot \varphi'(s)$.

Für den Erwartungswert von Y gilt

$$E[Y] = \sum_s \sum_{i \in \mathrm{cov}\,(s)} \varphi'(s) = |G|.$$

Damit ist der obige Versuch für einen (ϵ, δ)-Approximationsalgorithmus geeignet. Die Berechnung von $\varphi'(s)$ geschieht folgendermaßen.

1.) $t(s) := 0$
2.) wiederhole bis $s \in D_j$
  a) $t(s) := t(s) + 1$
  b) wähle $j \in 1 \ldots m$ mit Wahrscheinlichkeit $1/m$.
3.) $\varphi'(s) := t(s)/m$

Die Zeit zur Berechnung der Zufallsvariable $\varphi'(s)$ ist also auch eine Zufallsvariable.

Lemma 3.2.16 Der Erwartungswert für $t(s)$ ist

$$E[t(s)] = \frac{m}{|\mathrm{cov}\,(s)|}.$$

Beweis: $t(s)$ ist eine geometrisch verteilte Zufallsvariable mit Erfolgswahrscheinlichkeit $|\mathrm{cov}\,(s)|/m$. □

Damit ergibt sich $E[\varphi'(s)] = 1/|\mathrm{cov}\,(s)|$. Also ist die obige Prozedur für einen (ϵ, δ)-Approximationsalgorithmus geeignet.

Nun wollen wir den durchschnittlichen Zeitverbrauch der obigen Prozedur untersuchen. Dazu sei eine Aktion eine Schleifeniteration der Prozedur. Gesucht ist also der Erwartungswert für die Anzahl t von ausgeführten Iterationen pro Versuch. Die Kosten für jede dieser Aktion kann durch die Kosten für die zufällige Wahl von $(i, s) \in U$, der zufälligen Wahl von $j \in \{1 \ldots m\}$ und des Testes, ob $s \in D_j$ ist, abgeschätzt werden. Im Falle des DNF-Zählproblems sind diese Kosten $O(n + \log |U| + \log m)$. Die Anzahl der Klauseln m ist durch 2^n beschränkt, die Größe von U durch $m2^n$. Im allgemeinen kann also $\log |U|$ größer sein als $O(n)$. Wir betrachten jedoch nur Eingaben, für die U klein, also $|U| = O(n)$ ist. Für die anderen Eingaben ist $|U|/|G|$ für $U = \{0, 1\}^n$ und $G = \{s \mid f(s) = 1\}$ polynomiell beschränkt. Daher sind die Kosten, die innerhalb einer Schleifeniteration auftreten, von der Größenordnung $O(n)$.

Lemma 3.2.17 Sei $\mu = |G|/|U|$. Dann ist

$$E[t] = m\mu.$$

Beweis: Es gilt

$$\begin{aligned} E[t] &= \frac{1}{|U|} \sum_{(s,i)\in U} E[t(s)] \\ &= \frac{1}{|U|} \sum_{s\in G} \sum_{i\in \mathrm{cov}\,(s)} E[t(s)] \\ &= \frac{1}{|U|} \sum_{s\in G} m = m\mu. \end{aligned}$$

□

Damit haben wir einen (ϵ, δ)-Approximationsalgorithmus. Angewandt auf das DNF-Zählproblem ist er aber leider nicht schneller als der alte Algorithmus, da wir $E[t]$ nur durch m abschätzen können. Daher beträgt auch hier die Laufzeit $O(nm^2)$. Wir können aber trotzdem Vorteil aus der neuen Berechnung von φ' ziehen.

Nehmen wir an, wir könnten μ berechnen und folglich die Anzahl der Versuche mit

$$N(\mu) = \frac{c \ln(1/\delta)}{\mu\epsilon^2}$$

angeben, wobei c eine geeignete Konstante ist. Dann ist die Anzahl der insgesamt ausgeführten Aktionen $T(\mu)$ eine Zufallsvariable mit Erwartungswert

$$E[T(\mu)] = N(\mu) \cdot E[t] = \frac{cm \log(1/\delta)}{\epsilon^2}.$$

Dieser Erwartungswert ist um den Faktor m geringer als die obere Schranke, die wir berechnet haben. Man beachte, daß $E[T(\mu)]$ von μ unabhängig ist.

Sei $T = cm \log(1/\delta)\frac{1}{\epsilon^2}$ mit Konstante c. Wir werden den Algorithmus nun T Aktionen lang ausführen. Die Anzahl der Versuche, die während dieser Aktionen beendet wurden, ist nun eine

Zufallsvariable N_T in Abängigkeit von T. Wir werden zeigen, daß für geeignet gewähltes c mit hoher Wahrscheinlichkeit genügend Versuche durchgeführt worden sind. Intuitiv wird dieses deutlich, da der Erwartungswert von N_T ungefähr gleich $T/E[t] = N(\mu)$ ist. Der Algorithmus justiert sich also selbst, da sowohl Versuche mit wenigen als auch mit vielen Aktionen ausgeführt werden. Bei einer hinreichend großen Anzahl von Versuchen wird die Anzahl von ausgeführten Aktionen mit hoher Wahrscheinlichkeit sehr nahe an diesem Erwartungswert liegen. Damit sieht unser selbstjustierender Zählalgorithmus wie folgt aus.

Algorithmus 3.2.18 Selbstjustierender Zählalgorithmus

Eingabe: DNF-Formel f mit m Termen, Fehlertoleranz ϵ, Fehlerwahrscheinlichkeit δ

Schritt 1: Setze $G := 0$, $N_T := 0$,

Schritt 2: Setze $T := 8(1+\epsilon)m\ln(2/\delta)/\epsilon^2$.

Schritt 3: Führe die Schleife durch bis $G > T$

- a) Wähle (s, i) zufällig aus U.
- b) $t(s) := 0$.
- c) Wiederhole bis $s \in D_j$
  - i) Wenn $G > T$ gehe zur Ausgabe.
  - ii) Wähle zufällig $j \in \{1 \ldots m\}$ mit Wahrscheinlichkeit $1/m$.
  - iii) $t(s) := t(s) + 1$.
  - iv) $G := G + 1$.
- d) Setze $N_T := N_T + 1$
- e) Setze $\varphi'(s) := t(s)/m$
- f) Setze $Y_{N_T} := |U|\varphi'(s)$

Schritt 4: $\tilde{Y} := \frac{1}{N_T}\sum_{i=1}^{N_T} Y_i$

Ausgabe: $\tilde{Y}$

Dieser Algorithmus hat die (ϵ, δ)-Eigenschaft und Laufzeit $O(nm)$, da eine Aktion in $O(n)$ Zeit durchführbar ist.

Satz 3.2.19 Sei

$$\tilde{Y} = \frac{T \cdot |U|}{mN_T}$$

die Schätzung des Algorithmus. Diese Schätzung ist eine (ϵ, δ)-Approximation, falls bei $\epsilon < 1$

$$T = \frac{8(1+\epsilon)m\ln(2/\delta)}{\epsilon^2}$$

gewählt wird.

Zum Beweis dieses Satzes müssen wir wieder ein wenig Vorarbeit leisten. Für $k = 1, \ldots, m$ sei

$$R_k = \{s \in G \mid |\mathrm{cov}\,(s)| = k\}$$

und $r_k = |R_k|$. Dann ist $\sum_{k=1}^m r_k = |G|$ und $\sum_{k=1}^m kr_k = |U|$. Die Wahrscheinlichkeit, daß ein zufällig gewähltes Element aus G in R_k liegt, ist damit $kr_k/|U|$, d.h. die Wahrscheinlichkeit für alle $s \in R_k$ ist gleichgroß und nur von k abhängig. Daher können wir die Zufallsvariablen τ_k definieren, die die gleiche Verteilung haben wie $t(s)$ für $s \in R_k$. Auch diese Variablen haben eine geometrische Verteilung

$$Pr[\tau_k = j] = k/m(1 - k/m)^{j-1}$$

mit Erwartungswert m/k. Für $d = \lambda/m$ mit $\lambda \leq 1/2$ ergibt sich

$$\begin{aligned} E[e^{d\tau_k}] &= \sum_{j=1}^{\infty} e^{dj} Pr[\tau_k = j] \\ &= e^d \frac{k}{m} \sum_{j=1}^{\infty} (e^d(1 - k/m))^{j-1} \\ &= \frac{e^d k}{m(1 - (1 - k/m)e^d)}. \end{aligned}$$

Die Reihe konvergiert, da $(1 - k/m)e^d < 1$ für $d \leq 1/2m$ gilt.

Im folgenden seien $t_1, t_2, \ldots$ identisch verteilte Zufallsvariablen mit der gleichen Zufallsverteilung wie t. Die Zufallsvariable t_i ist die Anzahl der Aktionen im i-ten Versuch. Sei $S_l = \sum_{i=1}^l t_i$ die Anzahl von Aktionen nach dem l-ten Versuch und N_i die Anzahl der Versuche, die nach i Aktionen beendet sind. Es gilt nach Definition

$$N_i < l \iff S_l > i.$$

Lemma 3.2.20 Sei $0 \leq \lambda \leq 1/2$ und $d = \lambda/m$. Dann gilt

$$E[e^{dt}] \leq e^{(\lambda + 2\lambda^2)\mu} = e^{(md + 2(md)^2)\mu}.$$

Beweis: Es gilt

$$E[e^{dt}] = \frac{1}{|U|} \sum_{k=1}^{m} kr_k E[e^{d\tau_k}].$$

Wir kennen den Erwartungswert von $e^{d\tau_k}$. Wir setzen ihn in die Gleichung ein und erhalten

$$E[e^{dt}] = \frac{1}{|U|} \sum_{k=1}^{m} \frac{k^2 r_k}{m(e^{-d} - 1 + k/m)}.$$

Wir können nun die Summanden nach oben abschätzen. Dazu benutzen wir die Ungleichungen $1 - d \leq e^{-d}$ und daraus folgend $k/m - d \leq e^{-d} - 1 + k/m$. Damit ergibt sich

$$\frac{k}{m(e^{-d} - 1 + k/m)} \leq \frac{k}{m(k/m - d)} = 1 + \frac{d}{k/m - d}.$$

Setzen wir diese Abschätzung in die Ungleichung ein, erhalten wir

$$E[e^{dt}] \leq 1 + \frac{1}{|U|} \sum_{k=1}^{m} \frac{kr_k d}{k/m - d}.$$

Aufgrund von $d = \lambda/m$ und $k \geq 1$ ergibt sich $k/m - d \geq \frac{k(1-\lambda)}{m}$. Also folgt

$$E[e^{dt}] \leq 1 + \frac{1}{|U|}\sum_{k=1}^{m} \frac{r_k \lambda}{1-\lambda}.$$

Nun benutzen wir die Ungleichung $1/(1-\lambda) \leq 1 + 2\lambda$ für $0 \leq \lambda \leq 1/2$. Außerdem verwenden wir die Definition der r_k mit der Gleichheit $\sum r_k = |G|$. Es folgt

$$E[e^{dt}] \leq 1 + \frac{|G|}{|U|}\lambda(1+2\lambda) = 1 + \mu(\lambda + 2\lambda^2) \leq e^{\mu(\lambda+2\lambda^2)}.$$

□

Aus diesem Lemma können wir das folgende Korollar ableiten.

Korollar 3.2.21 Sei $\epsilon \leq 2$. Dann gilt

$$Pr[S_l > (1+\epsilon)m\mu l] \leq e^{-\mu\epsilon^2 l/8}.$$

Beweis: Nach Lemma 3.2.8 gilt

$$Pr[S_l > (1+\epsilon)m\mu l] \leq E[e^{d(S_l-(1+\epsilon)m\mu l)}]$$

und mittels Lemma 3.2.7 folgt

$$= E[e^{dt}]^l e^{-d(1+\epsilon)m\mu l}.$$

Dabei bezeichne t die Zufallsvariable mit der gleichen Verteilung wie die Zufallsvariablen t_i. Aus Lemma 3.2.20 ergibt sich eine Abschätzung für $E[e^{dt}]$

$$E[e^{dt}] \leq e^{(md+2(md)^2)\mu}.$$

Setzen wir nun für $d = \epsilon/4m$ ein, so folgt

$$\begin{aligned} Pr[S_l > (1+\epsilon)m\mu l] &\leq e^{(\epsilon/4+\epsilon^2/8)\mu l - \epsilon/4(1+\epsilon)\mu l} \\ &= e^{-\mu\epsilon^2 l/8}. \end{aligned}$$

□

Eine ähnliche Abschätzung gilt auch in die andere Richtung.

Lemma 3.2.22 Sei $0 \leq \lambda \leq 1/2$ und $d = \lambda/m$. Dann gilt

$$E[e^{-dt}] \leq e^{(-\lambda-\lambda^2)\mu} = e^{(-md-(md)^2)\mu}.$$

Korollar 3.2.23 Sei $\epsilon \leq 2$. Dann gilt

$$Pr[S_l > (1-\epsilon)m\mu l] \leq e^{-\mu\epsilon^2 l/8}.$$

Nun können wir die Korrektheit des selbstjustierenden Zählalgorithmus beweisen.

Beweis: von Satz 3.2.19
Sei

$$k_1 = \frac{8(1+\epsilon)\log(2/\delta)}{\mu\epsilon^2(1+\epsilon)}$$

und
$$k_2 = \frac{8(1+\epsilon)\log(2/\delta)}{\mu\epsilon^2(1-\epsilon)}.$$
Damit ist $T = k_1 m\mu(1+\epsilon) = k_2 m\mu(1-\epsilon)$. Wenn $k_1 \leq N_T \leq k_2$ gilt, so gilt auch
$$\frac{T}{k_2} \leq \frac{T}{N_T} \leq \frac{T}{k_1}$$
und damit
$$\frac{T|U|}{mk_2} \leq \tilde{Y} \leq \frac{T|U|}{mk_1}.$$
Nach Einsetzen von k_1 und k_2 ergibt sich
$$|G|(1-\epsilon) \leq \tilde{Y} \leq |G|(1+\epsilon)$$
und somit ist die Schätzung eine ϵ-Approximation. Es reicht also zu zeigen, daß
$$Pr[N_T > k_2] + Pr[N_T < k_1] \leq \delta$$
ist. Dies ergibt sich aber unmittelbar durch Benutzen der Korollare 3.2.21 und 3.2.23:
$$Pr[N_T < k_1] = Pr[S_{k_1} > T] = Pr[S_{k_1} > k_1 m\mu(1+\epsilon)] \qquad (*).$$
Durch Anwenden von Korollar 3.2.21 und Einsetzen von k_1 ergibt sich
$$(*) \leq e^{-\mu\epsilon^2 k_1/8} = e^{\log(2/\delta)} = \delta/2.$$
Analoges gilt für den anderen Teil. □

3.2.3 Zuverlässigkeitsprobleme in Netzwerken

In diesem Abschnitt möchten wir die Fehlerwahrscheinlichkeit eines Netzwerkes bestimmen. Das Modell für unser Netzwerk ist ein ungerichteter Graph $G = (V, E)$. Eine Teilmenge $K \subseteq V$ ist die Menge der Terminals, die miteinander kommunizieren wollen. Jeder Kante $e \in E$ ist eine Fehlerwahrscheinlichkeit $p(e)$ zugeordnet. Mit Wahrscheinlichkeit $p(e)$ ist die Kante *schlecht*, d.h. eine Kommunikation über diese Kante ist nicht möglich, sonst ist die Kante *gut*.

Definition 3.2.24 Ein Netzwerk N ist fehlerhaft, falls es ein Paar von Terminals aus K gibt, das durch keinen Weg aus guten Kanten verbunden ist.

Damit können wir unsere Aufgabe präzisieren: Gegeben sei ein Netzwerk $N = (G, K, p)$, gesucht ist die Fehlerwahrscheinlichkeit von N, d.h. die Wahrscheinlichkeit, daß N fehlerhaft ist.

Definition 3.2.25 Ein Zustand des Netzwerkes N ist eine Funktion $z : E \to \{\text{gut}, \text{schlecht}\}$. Die Wahrscheinlichkeit für den Zustand z ist
$$b(z) = \prod_{\{e|z(e)=\text{schlecht}\}} p(e) \prod_{\{e|z(e)=\text{gut}\}} (1-p(e)).$$

Ein Zustand z ist ein Fehlerzustand, falls das Netzwerk N in Zustand z fehlerhaft ist. Sei F die Menge der Fehlerzustände und

$$b(F) = \sum_{z \in F} b(z).$$

Dann ist die Fehlerwahrscheinlichkeit des Netzwerks N gleich $b(F)$.

Unser Ziel ist also die Bestimmung von $b(F)$. Um auf dieses Problem einen Approximationsalgorithmus anwenden zu können, formulieren wir es ein wenig um.

Definition 3.2.26 Ein K-trennender Schnitt S ist eine Teilmenge der Kanten, so daß mindestens zwei Knoten aus K durch S getrennt werden.

Mit Hilfe dieser Definition können wir einen Fehlerzustand s auch dadurch charakterisieren, daß es einen K-trennenden Schnitt gibt, der nur aus schlechten Kanten besteht. Einen solchen Schnitt nennen wir einen Fehlerschnitt im Zustand s.

Wenn zusätzlich eine explizite Liste aller K-trennenden Schnitte vorliegt, können wir nun einen Approximationsalgorithmus für die Bestimmung der Fehlerwahrscheinlichkeit angeben.

Das Universum U sei die Menge aller Paare (c, s), wobei s ein Fehlerzustand und c ein Fehlerschnitt im Zustand s ist. Dabei wird ein Element aus U mit Wahrscheinlichkeit $a(c, s) = b(s)$ gewählt. Zu einem Zustand s sei einer der Fehlerschnitte $g(s)$ ausgezeichnet. Diesen Fehlerschnitt bezeichnen wir als kanonischen Fehlerschnitt $g(s)$ und wählen hierzu den Schnitt, der die Zusammenhangskomponente des ersten Terminals am kleinsten macht (bzgl. der lexikographischen Ordnung auf den Knoten). Sei R die Menge aller Paare $(g(s), s)$. Es gilt

$$a(R) = \sum_{s \in F} b(s) = b(F).$$

Der Test, ob ein Paar (c, s) in R liegt, besteht darin $g(s)$ zu berechnen und c mit $g(s)$ zu vergleichen.

Zu einem K-trennenden Schnitt c sei $P(c)$ die Wahrscheinlichkeit, daß ein Zustand s gewählt wird für den c ein Fehlerschnitt ist. Dafür müssen alle Kanten aus c schlecht sein, also ist $P(c) = \prod_{e \in c} p(e)$. $a(U)$ ergibt sich jetzt durch die Summierung dieser Wahrscheinlichkeiten

$$a(U) = \sum_{c} P(c).$$

Als nächstes müssen wir Elemente (c, s) aus U mit Wahrscheinlichkeit $a(c, s)/a(U) = b(s)/a(U)$ wählen. Dies geschieht durch Wahl eines K-trennenden Schnittes mit Wahrscheinlichkeit $P(c)/a(U)$ und dann durch zufälliges Wählen eines zugehörigen Fehlerzustandes. Dabei sind alle Kanten $e \in c$ schlecht und die übrigen Kanten e werden mit Wahrscheinlichkeit $p(e)$ mit dem Wert schlecht belegt.

Als letztes muß noch das Verhältnis $a(U)/a(R)$ abgeschätzt werden. Da für jeden Fehlerzustand die Zahl der zugehörigen Fehlerschnitte nicht größer als die Anzahl der K-trennenden Schnitte sein kann, folgt sofort, daß die Anzahl der K-trennenden Schnitte eine obere Schranke für $a(U)/a(R)$ ist. Falls die Fehlerwahrscheinlichkeiten klein sind, erhält man durch das folgende Lemma eine bessere obere Schranke.

Lemma 3.2.27
$$\frac{a(U)}{a(R)} \leq \prod_{e \in E} (1 + p(e)).$$

Beweis: Sei $m = |E|$ die Anzahl der Kanten von G und D die Menge aller Tupel der Form $(b(e_1), \ldots b(e_i))$ für $0 \leq i \leq m$ mit $b(e_j) \in \{\text{gut}, \text{schlecht}\}$ für $1 \leq j \leq i$. Desweiteren bezeichne λ den leeren Tupel. Wir konstruieren zwei Funktion $\varphi : D \to I\!R$ und $\psi : D \to I\!R$ mit $\varphi(\lambda) = a(R)$ und $\psi(\lambda) > a(U)$. Aus der Konstruktion der Funktionen ergibt sich auch die folgende Abschätzung über das Verhältnis:
$$\psi(b(e_1), \ldots, b(e_i))/\varphi(b(e_1), \ldots, b(e_i)) \leq \prod_{j=i+1}^{m} (1 + p(e)).$$

Damit ergibt sich
$$a(U)/a(R) \leq \psi(\lambda)/\varphi(\lambda) \leq \prod_{e} (1 + p(e)).$$

Wir definieren jetzt die Funktionen φ und ψ. $\varphi(b(e_1), \ldots, b(e_i))$ sei die bedingte Wahrscheinlichkeit, daß s ein Fehlerzustand ist, wobei $s(e_j) = b(e_j)$ für $1 \leq j \leq i$ gelte. Wir können φ auch rekursiv folgendermaßen definieren:
$$\varphi(s(e_1), \ldots, s(e_m)) = \begin{cases} 1 & \text{falls } s \text{ ein Fehlerzustand ist} \\ 0 & \text{falls } s \text{ kein Fehlerzustand ist.} \end{cases}$$

und
$\varphi(b(e_1), \ldots, b(e_{i-1})) := p(e_i)\varphi(b(e_1), \ldots, b(e_{i-1}), \text{schlecht}) + (1 - p(e_i))\varphi(b(e_1), \ldots, b(e_{i-1}), \text{gut})$.
Analog definieren wir auch ψ:
$$\psi(s(e_1), \ldots, s(e_m)) = \begin{cases} 1 & \text{falls } s \text{ ein Fehlerzustand ist} \\ 0 & \text{falls } s \text{ kein Fehlerzustand ist.} \end{cases}$$

und $\psi(b(e_1), \ldots, b(e_{i-1})) := p(e_i)\psi(b(e_1), \ldots, b(e_{i-1}), \text{schlecht}) + \psi(b(e_1), \ldots, b(e_{i-1}), \text{gut})$.

Für das Verhältnis zwischen ψ und φ erhalten wir induktiv
$$\psi(b(e_1), \ldots, b(e_{i-1}))/\varphi(b(e_1), \ldots, b(e_{i-1})) \leq \prod_{j=i}^{m} (1 + p(e_j)).$$

Bezeichne $\psi_{i-1} = \psi(b(e_1), \ldots, b(e_{i-1}))$, ψ^s_{i-1}, bzw. ψ^g_{i-1} die Werte $\psi(b(e_1), \ldots, b(e_{i-1}), \text{schlecht})$ bzw. $\psi(b(e_1), \ldots, b(e_{i-1}), \text{gut})$ und gelten analoge Bezeichnungen für φ. Es gilt
$$\frac{\psi_{i-1}}{\varphi_{i-1}} = \frac{p(e_i)\psi^s_{i-1} + \psi^g_{i-1}}{p(e_i)\varphi^s_{i-1} + (1 - p(e_i))\varphi^g_{i-1}}.$$

Wir setzen nun die Abschätzungen $\psi^*_{i-1} \leq \varphi^*_{i-1} \prod_{j=i+1}^{m} (1 + p(e_j))$ ein und erhalten
$$\frac{\psi_{i-1}}{\varphi_{i-1}} \leq \prod_{j=i+1}^{m} (1 + p(e_j)) \frac{p(e_i)\varphi^s_{i-1} + \varphi^g_{i-1}}{p(e_i)\varphi^s_{i-1} + (1 - p(e_i))\varphi^g_{i-1}}.$$

Aus der Definition von φ ergibt sich, daß $\varphi^s_{i-1} \geq \varphi^g_{i-1}$ ist. Unter Verwendung dieser Ungleichheit ergibt sich der Rest des Beweises durch leichtes Ausrechnen. Damit haben wir
$$\psi(\lambda)/\varphi(\lambda) \leq \prod_{j=1}^{m} (1 + p(e_j))$$

bewiesen.

Um den Beweis zu vervollständigen, müssen wir noch zeigen, daß $\psi(\lambda) \geq a(U)$ ist. Es ist leicht zu sehen, daß $\psi(b(e_1), \ldots, b(e_{i-1})) = \sum_s \prod_{e_j} p(e_j)$, wobei über alle Fehlerzustände s mit den Werten $b(e_j)$ an den Kanten e_j summiert wird und das Produkte über alle Kanten e_j mit $j \geq i$ geht, die in s schlecht sind. Wir können diese Summe aufteilen in einen Teil mit $s(e_i) =$ schlecht und einen Teil mit $s(e_i) =$ gut. Benutzen wir die Definition von ψ, sehen wir induktiv die Richtigkeit dieser Formel.

Somit ist $\psi(\lambda) = \sum_{s \in F} \prod_{s(e) \text{ schlecht}} p(e)$. Da jeder K-separierende Schnitt wenigstens den Fehlerzustand induziert, indem alle Kanten des Schnittes im Zustand schlecht sind, ist $\psi(\lambda)$ größer gleich $a(U) = \sum_c \prod_{e \in c} p(e)$. □

3.3 GF(q) Zählprobleme

Im vorigen Abschnitt haben wir das Problem untersucht, die Anzahl der Einsstellen einer booleschen Funktion zu berechnen, die in disjunktiver Normalform gegeben ist. Nun stellen wir die analoge Frage, was passiert, wenn die Funktion als Polynom über GF(2) gegeben ist. Diese Fragestellung läßt sich verallgemeinern auf die Frage der c-Stellen von Polynomen über beliebigen endlichen Körpern. Dies ist eine sehr wichtige Frage in der Geometrie, Algebra und algebraischen Geometrie.

3.3.1 Exaktes Zählen

Zunächst werden wir zeigen, daß das allgemeine Problem auch hier sehr schwer ist, indem wir es mit dem Satz 3.2.2 von Valiant in Verbindung bringen.

Zunächst benötigen wir einige Lemmata über Polynome über GF(2).

Seien für $1 \leq i \leq m$ die Polynome $w_i \in \mathrm{GF}(2)[x_1, \ldots, x_n]$. Zu diesen Polynomen sei

$$u(x_1, \ldots, x_n, z_1, \ldots z_m) := \bigoplus_{i=1}^{m} w_i(x_1, \ldots x_n) z_i$$

mit $z_i \notin \{x_1, \ldots x_n\}$ definiert. Dann gilt

Lemma 3.3.1 Bezeichne $s(u)$ die Nullstellenmenge des Polynoms u und $s(\{w_i\})$ die Nullstellenmenge des System $\{w_i\}$. Dann gilt

$$\#s(u) = \#s(\{w_i\})2^m + (2^n - \#s(\{w_i\}))2^{m-1}.$$

Beweis: Wir unterscheiden zwei Fälle.

$x \in s(\{w_i\})$: In diesem Fall sind für alle möglichen Einsetzungen für die Variablen z_i (x, z) Nullstellen von u. Daher trägt dieser Fall den Wert $\#s(\{w_i\})2^m$ zur Summe bei.

$x \notin s(\{w_i\})$: Dieser Fall tritt $2^n - \#s(\{w_i\})$ mal auf. Sei i_0 der kleinste Index, so daß $w_{i_0}(x) \neq 0$ ist. Wir können alle Variablen z_i mit $i \neq i_0$ beliebig wählen, da die Gleichung

$\bigoplus_{i \neq i_0} w_i(x) z_i \oplus w_{i_0}(x) z_{i_0}$ wegen $w_{i_0} \neq 0$ lösbar ist. Weiterhin ist dadurch z_{i_0} eindeutig bestimmt. Man beachte, daß dies in jedem Körper gilt. In diesem Fall ergeben sich also $(2^n - \#s(\{w_i\}))2^{m-1}$ Nullstellen von u.

Zusammen ergibt sich genau die behauptete Anzahl von Nullstellen. □

Aus diesem Lemma folgen einige einfache Korollare.

Korollar 3.3.2 Das System $\{w_i\}_{1 \leq i \leq m}$ hat eine Lösung genau dann, wenn $\#s(u) > 2^{n+m-1}$ ist. ($\#s(u) \geq 2^{n+m-1}$ gilt immer).

Korollar 3.3.3

$$\#s(\{w_i\}) = \frac{\#s(u) - 2^{n+m-1}}{2^{m-1}}.$$

Wir können die Aussagen auch auf beliebige andere endliche Körper erweitern.

Korollar 3.3.4 Seien u und w_i Polynome wie in Lemma 3.3.1 aber über $GF(q)$. Dann gilt

$$\#s(u) = \#s(\{w_i\})q^m + (q^n - \#s(\{w_i\}))q^{m-1}.$$

Beweis: Analog zum Beweis von Lemma 3.3.1 □

Nun werden wir die Berechnung der Einsstellen einer monotonen booleschen Formel in 2-DNF Form auf die Berechnung der Einsstellen eines Polynoms über GF(2) vom Grad 3 reduzieren. Es gilt

Satz 3.3.5 ([EK 90])
Die Berechnung der Einsstellen von $p \in \mathrm{GF}(2)[x_1, \ldots, x_n]$ ist $\#\mathcal{P}$ vollständig, wenn p vom Grad ≥ 3 ist.

Beweis: Sei f eine monotone 2-DNF Formel

$$f = c_1 \vee \cdots \vee c_m$$

und $c_i = a_i \wedge b_i$. Die Berechnung von $\#f$ ist nach Satz 3.2.2 $\#\mathcal{P}$ -vollständig. Zu f definieren wir das System von Polynomen $\{w_i\}$ mit $w_i = a_i b_i$. Offensichtlich ist die Anzahl von Nullstellen des Systems $\{w_i\}$ und der Nullstellen von f gleich. Sei u wie oben durch $u = \bigoplus w_i z_i$ definiert, dann gilt mit Korollar 3.3.3

$$\#f = 2^n - \#s(\{w_i\}) = 2^n - \frac{\#s(u) - 2^{n+m-1}}{2^{m-1}}.$$

Daher ist die Berechnung von $\#f$ auf die Berechnung von $\#s(u)$ (und damit auf die Berechnung von $\#u$) für Polynome u vom Grad 3 in polynomieller Zeit reduzierbar. □

Für Polynome u vom Grad 4 gilt der folgende

Satz 3.3.6 Für ein Polynom $u \in \mathrm{GF}(2)[x_1 \dots x_n]$ vom Grad 4 ist das Problem zu bestimmen, ob $\#u > 2^{n-1}$ bzw. $\#u = 2^{n-1}$ gilt, $\mathcal{NP}$ hart.

Beweis: Wir reduzieren das 3-SAT- Problem für Formeln in konjunktiver Normalform auf das obige Problem. Dazu sei $f = \bigwedge_{i=1}^{m}(a_i \vee b_i \vee c_i)$ eine solche Formel in den Variablen $x_1, \dots, x_n$ und Literalen a_i, b_i, c_i. Aus f konstruieren wir das System $\{w_i = (a_i \vee b_i \vee c_i) \oplus 1\}$ von Gleichungen über GF(2). Dies ist möglich, da

$$\neg x = x \oplus 1$$

und

$$(a_i \vee b_i \vee c_i) = a_i \oplus b_i \oplus c_i \oplus a_i b_i \oplus a_i c_i \oplus b_i c_i \oplus a_i b_i c_i$$

gelten. Nun konstruktieren wir wieder ein Polynoms u wie in Satz 3.3.5. Korollar 3.3.3 zeigt, daß das System $\{w_i\}$ genau dann eine nichtleere Nullstellenmenge besitzt, wenn $\#s(u) > 2^{n+m}$ gilt. Das heißt f ist genau dann erfüllbar, wenn $\#s(u) > 2^{n+m}$ ist. Daher ist die Entscheidung, ob $\#s(u) = 2^{n+m}$ oder $\#s(u) > 2^{n+m}$ gilt, genauso schwer wie ein $\mathcal{NP}$-vollständiges Problem. □

Im folgenden werden wir einen polynomiellen Algorithmus für Polynome vom Grad 2 entwickeln. Die Schwere des Zählproblems hängt also ganz entschieden von dem Grad des Polynoms ab. Zwischen Grad 2 und Grad 3 ist ein tiefer Schnitt zu entdecken.

Satz 3.3.7 Für Polynome f über GF(2) vom Grad 2 existiert ein Algorithmus zur Berechnung von $\#f$, der in Zeit $O(n^{\alpha+1})$ arbeitet, wobei α der Exponent der Matrixmultiplikation ist ($\alpha \leq 2.378$).

Der Algorithmus besteht im wesentlichen darin, das Polynom f auf ein sogenanntes Read-Once Polynom zu transformieren.

Definition 3.3.8 Ein Polynom g heißt Read-Once Polynom, falls jede Variable nur einmal vorkommt.

Satz 3.3.9 Die Anzahl der Einsstellen eines Read-Once Polynoms kann in polynomieller Zeit berechnet werden.

Beweis: Sei $g(x_1, \dots, x_n)$ das Read-Once Polynom mit m Monomen. Bezeichne k_i die Anzahl von Variablen im i-ten Monom. Nun kann durch Umbennenen der Variablen erreicht werden, daß g geordnet ist:

$$g = x_1 x_2 \dots x_{k_1} \oplus x_{k_1+1} \dots x_{k_1+k_2} \oplus \dots \oplus x_{n-k_m+1} \dots x_n (\oplus 1).$$

Im konstantenfreien Fall gilt für die Anzahl von Einstellen $\#g$ von g

$$\begin{aligned} \#g &= \#(g_{|\{x_1,\dots,x_{n-k_m}\}}) \cdot (2^{k_m} - 1) + (2^{n-k_m} - \#(g_{|\{x_1,\dots,x_{n-k_m}\}})) \\ &= (2^{k_m} - 2) \cdot \#(g_{|\{x_1,\dots,x_{n-k_m}\}}) + 2^{n-k_m}. \end{aligned}$$

Hieraus kann rekursiv der Wert von $\#g$ berechnet werden. □

Für spezielle Arten von Read-Once Polynomen können wir eine geschlossene Formel für die Anzahl der Einsstellen angeben.

Lemma 3.3.10 Sei $g \in \mathrm{GF}(2)[x_1, \ldots, x_n]$ vom Grad 2. Wenn n ungerade ist, so sei

$$g = x_1 x_2 \oplus \cdots \oplus x_{n-2} x_{n-1} \oplus x_n.$$

Dann ist

$$\# g = 2^{n-1}.$$

Im Falle von geradem n sei

$$g = x_1 x_2 \oplus \cdots \oplus x_{n-1} x_n.$$

Dann ist

$$\# g = 2^{n-1} - 2^{\frac{n-2}{2}}.$$

Nun können wir die Transformation von f auf ein Read-Once Polynom vornehmen.

Lemma 3.3.11 Sei f ein Polynom über $\mathrm{GF}(2)[x_1, \ldots, x_n]$ vom Grad 2. Dann gibt es ein Read-Once Polynom $g \in \mathrm{GF}(2)[y_1, \ldots y_m]$ mit $m \leq n$, eine Transformationsmatrix $T = (t_{ij})$ mit linear unabhängigen Zeilen und einen Translationsvektor $C = (c_i)^t$, so daß gilt

$$g(\bigoplus_j t_{1j} x_j + c_1, \bigoplus_j t_{2j} x_j + c_2, \ldots, \bigoplus_j t_{mj} x_j + c_m) = f(x_1, \ldots, x_n).$$

Dabei ist g von der Form

$$g = y_1 \oplus y_2 y_3 \oplus \cdots \oplus y_{m-1} y_m \qquad \text{oder}$$

$$g = y_1 y_2 \oplus \cdots \oplus y_{m-1} y_m (\oplus 1).$$

Die Berechnung von T, C und g kann in Zeit $O(n^{\alpha+1})$ durchgeführt werden.

Beweis: Der Beweis ist konstruktiv und gibt direkt einen rekursiven Algorithmus an. Die Rekursion geht über die Variablenmenge von f.

Sei $f \in \mathrm{GF}(2)[x_1, \ldots, x_n]$. Zunächst schreiben wir f als

$$f = \alpha(x_1, \ldots, x_{n-1}) \cdot x_n \oplus h(x_1, \ldots, x_{n-1}).$$

h ist eine Funktion auf der Variablenmenge $\{x_1, \ldots, x_{n-1}\}$. Wir wenden nun unseren Algorithmus rekursiv auf h an und erhalten das transformierte Read-Once Polynom β von der Form

$$\beta = y_1 \oplus y_2 y_3 \oplus \cdots \oplus y_{k-1} y_k \qquad \text{Typ I}$$

oder

$$\beta = y_1 y_2 \oplus \cdots \oplus y_{k-1} y_k (\oplus 1) \qquad \text{Typ II},$$

die zugehörige $k \times (n-1)$ Transformationsmatrix T und den zugehörigen Translationsvektor C. Die Variablen y_i sind damit Linearkombinationen der Variablen x_j und der Konstante 1. Dabei sind die einzelnen Variablen y_i linear unabhängig. Daher gilt $k \leq n-1 < n$. Nun hat f die Darstellung

$$f = \alpha x_n \oplus \beta.$$

Dabei ist α ein lineares Polynom, da f höchstens Grad 2 hat. Für α betrachten wir nun die folgenden Fälle.

1. $\alpha = 1$.

   - β ist vom Typ I. Dann fassen wir die Variablen x_n und y_1 zu einer neuen Variablen zusammen:
   $$f = \underbrace{x_n \oplus y_1}_{y_1'} \oplus y_2 y_3 \oplus \cdots \oplus y_{k-1} y_k.$$
   Durch diese Transformation wird nur die Variable y_1 geändert, d.h. es gilt
   $$y_1' := y_1 \oplus x_n.$$
   Die übrigen Variablen, der Vektor C und der Typ von β bleiben unverändert.
   - β ist vom Typ II. Dann ist f schon vom Typ I:
   $$\begin{aligned} f &= x_n \oplus y_1 y_2 \oplus \cdots \oplus y_{k-1} y_k (\oplus 1) \\ &= y_1' \oplus y_2' y_3' \oplus \cdots \oplus y_k' y_{k+1}. \end{aligned}$$
   Die Transformation hat also die Gestalt
   $$y_1' := x_n (\oplus 1) \qquad y_i' := y_{i-1} \text{ für } i \geq 2$$
   Der Translationsvektor C wird entsprechend erweitert. Allerdings ist β nun vom Typ I.

2. α ist linear unabhängig von den Variablen in β. Daher gilt $k < n - 1$. Wenn wir α in der Form
$$\alpha = \bigoplus t_\alpha^i x_i + c_\alpha$$
darstellen, so ist der Vektor t_α unabhängig von den Zeilenvektoren von T. Damit ist
$$f = \cdots y_{k-1} y_k \oplus \underbrace{x_n \alpha}_{y_{k+1} y_{k+2}}.$$
Es ergibt sich also
$$y_i' := y_i \text{ für } i \leq k \qquad y_k := x_n \quad y_{k+1} := \alpha.$$

3. α ist linear abhängig von den Variablen von β.
Dann sei α ausgedrückt in den Variablen von β von der Form
$$\alpha = y_{i_1} \oplus \cdots \oplus y_{i_t}$$
oder
$$\alpha = y_{i_1} \oplus \cdots \oplus y_{i_a} \oplus 1.$$
Diese Darstellung kann durch Matrizeninversion gewonnen werden. Wir reduzieren α nun stückchenweise. Dabei unterscheiden wir zwei Fälle.

   - Es gibt einen Index s und einen Index t, so daß y_s und y_t ein Monom von β bilden, d.h. $y_s y_t$ ist ein Term von β. Damit sieht f folgendermaßen aus:
   $$f = \ldots \oplus y_s x_n \oplus y_t x_n \oplus y_s y_t = \ldots (x_n \oplus y_s)(x_n \oplus y_t) \oplus x_n.$$
   Die Variablen y_s und y_t ergeben sich folglich zu
   $$y_s := y_s \oplus x_n \qquad y_t := y_t \oplus x_n.$$
   α haben wir zu
   $$\alpha := \alpha \oplus y_s \oplus y_t \oplus 1$$
   reduziert, d.h. α enthält nun zwei Variablen weniger.

- Es gibt keine solche Indices. Dann gilt

$$f = \ldots \oplus y_s x_n \oplus y_s y_t = \ldots y_s(x_n \oplus y_t),$$

  wobei y_t kein Term in α ist. Damit sieht die Transformation folgendermaßen aus

$$y'_t := y_t \oplus x_n \qquad y'_i := y_i \text{ für } i \neq t.$$

  $\alpha := \alpha \oplus y_s$ verkürzt sich um eine Variable.

Falls $\alpha \neq 0$ ist, so ist der Rekursionsschritt noch nicht abgeschlossen und wir müssen für dieses neue α ebenfalls wieder alle Fallunterscheidungen durchgehen. Man beachte jedoch, daß sich in diesem Fall die Anzahl der y Variablen nicht erhöht.

Damit haben wir alle Fälle diskutiert. Die Laufzeit des Algorithmus ist $O(n^{\alpha+1})$. Dies ergibt sich aus einem Maximalaufwand von $O(n^{\alpha})$ für jede Rekursionsstufe, der durch die Matrixinversion im Fall 3 bedingt ist. □

3.3.2 Approximative Zählung der Einsstellen von GF(2) Polynomen

Im vorigen Abschnitt haben wir gesehen das die Bestimmung der Einsstellen einer booleschen Funktion, die als GF(2) Polynom gegeben ist, sehr schwierig ist. In diesem Abschnitt werden wir zeigen, daß wir den Approximationsalgorithmus für das DNF Zählproblem auch für das GF(2) Zählproblem verwenden können. Dazu werden wir den folgenden Satz beweisen.

Satz 3.3.12 ([KL 90])
Sei $C = \{c_i\}$ eine beliebige Menge von m paarweise verschiedenen, nichtleeren Klauseln über n Variablen. Bezeichne G die Menge der Einsstellen der Funktion $g = \bigvee c_i$ und F die Menge der Einsstellen der Funktion $f = \bigoplus c_i$, d.h wir sehen die Menge C einmal als Monome einer Funktion in diskunktiver Normalform und einmal als Monome eines Polynoms über GF(2) an. Dann gilt

$$|G|/|F| \leq m.$$

Sei U die in Abschnitt 3.2.1 definierte Überdeckung von G. Dann gilt für das Verhältnis von $|U|$ und $|G|$

$$\frac{|U|}{|G|} \leq m.$$

Mit Hilfe von Satz 3.3.12 erhalten wir dann

$$\frac{|U|}{|F|} \leq m^2.$$

Diese Aussagen implizieren direkt den folgenden

Satz 3.3.13 Es gibt einen (ϵ, δ)-Approximationsalgoritmus für das $GF(2)$ Zählproblem, der in Zeit

$$O(nm^3 \ln(1/\delta)/\epsilon^2)$$

läuft.

Sei $X = \{x_1, \ldots, x_k\}$ eine Menge von Variablen und s eine Belegung der Variablen $\{x_1, \ldots x_n\}$. Dann definieren wir für jede Teilmenge X' von X die Belegung $s(X')$ durch

$$s(X')_i := \begin{cases} 0 & \text{falls } x_i \in X \setminus X' \\ s_i & \text{sonst} \end{cases} .$$

Lemma 3.3.14 Sei s eine Variablenbelegung mit $g(s) = 1$ und X die Variablenmenge eines maximalen Terms c mit $c(s) = 1$. Dann gibt es wenigstens eine Variablenbelegung $s' = s(X')$ für ein $X' \subseteq X$, mit $f(s') = 1$, d.h. $s' = s(X')$ erfüllt eine ungerade Anzahl von Klauseln.

Beweis: Sei $T = \{c_i \mid c_i(s) = 1\}$ die Menge von Klausen, die von s erfüllt werden.

- Wenn $|T|$ ungerade ist, dann erfüllt $X' = X$ die Bedingung.
- Wenn $|T|$ gerade ist, so dann teilen wir T in die Mengen T' und T''. T' ist die Menge aller Klauseln, so daß mindestens eine Variable aus X in der Klausel vorkommt, T'' enthält die Klauseln, die keine Variablen mit X gemeinsam haben.
  - Falls $|T''|$ ungerade ist, dann wählen wir $X' = \emptyset$. Dadurch werden alle Klauseln in T'' erfüllt aber keine in T'.
  - Falls $|T''|$ und damit auch $|T'|$ gerade ist, so benutzen wir die folgende Konstruktion. Dazu sei
  $$P(X') := \{c_i \mid c_i(s(X')) = 1\},$$
  $$Q(X') := \{c_i \mid \mathrm{Var}\,(c_i) \cap X = X'\}$$
  und $p(X')$ und $q(X')$ die Parität der zugehörigen Mengen. Aufgrund der Maximalität von c besteht $Q(X)$ nur aus dem Element c. Damit ist $q(X) = 1$. Wir sind nun interessiert an einem X' mit $p(X') = 1$. Zwischen den Mengen P und Q (und damit auch zwischen p und q) besteht der folgende, einfach zu beweisende Zusammenhang:
  $$P(X') = \bigcup_{X'' \subseteq X'} Q(X'').$$
  Dieser Zusammenhang läßt sich durch eine nicht singuläre obere Dreiecksmatrix ausdrücken. Ebenso lassen sich die Werte von p durch die Werte von q durch ein lineares invertierbares Gleichungssystem über GF(2) ausdrücken. Da $q(X) \neq 0$ ist, folgt also, daß es wenigstens ein X' mit $p(X') \neq 0$ gibt. □

Lemma 3.3.15 Es gibt eine Zuordnung $\varphi : G \to F$, so daß für alle $s \in F$ die Urbildmenge $\varphi^{-1}(s)$ höchstens m Elemente enthält.

Beweis: Zu $s \in F$ sei t_s ein maximaler Term, der durch s erfüllt wird, d.h. es gibt keinen Term t mit $t(s) = 1$, dessen Variablenmenge die Variablenmenge von t_s enthält. Für die (höchstens) $m - 1$ vielen Terme, die nicht von s erfüllt werden, sei s_t die Belegung, in der alle Variablen, die in t vorkommen, auf 1 gesetzt sind, die aber sonst mit s übereinstimmt. Damit definieren wir die Urbildmenge $M_s = \varphi^{-1}(s)$ von s als

$$M_s = \{s\} \cup \{s_t \mid t(s) = 0\}.$$

Wir behaupten nun, daß $G = \cup_{s \in F} M_s$ gilt. Sei $\bar{s} \in G$ und s' die nach Lemma 3.3.14 konstruierte erfüllende Belegung von f. Nach dieser Konstruktion gilt $\bar{s} \in M_{s'}$. Daher ist φ die gewünschte Zuordnung.

BEWEIS: von Satz 3.3.12
In Lemma 3.3.15 haben wir ein Abbildung $\varphi : G \to F$ konstruiert, so daß für alle $s \in F$ die Urbildmenge $\varphi^{-1}(s)$ höchstens m Elemente enthält. Dann kann die Menge G höchstens m mal so groß sein wie F. □

Dieses Ergebnis ist sogar optimal. Betrachten wir für eine Zweierpotenz m das Polynom

$$g_T = \prod_{i \in T} (x_i \oplus 1) \prod_{i \notin T} x_i.$$

Wenn $|T| = \log m$ ist, so hat g_T genau m Terme und eine Einsstelle. Die entsprechende Funktion

$$f_T = \bigwedge_{i \in T} (x_i \vee 1) \bigwedge_{i \notin T} x_i.$$

in disjunktiver Normalform hat jedoch m Einsstellen.

Bis jetzt haben wir GF(2)-Polynome mit Termen betrachtet, deren Variablenmenge jeweils nichtleer war. Dies sind gerade diejenigen Polynomen, die nicht die Konstante 1 (= Term mit leerer Variablenmenge) enthalten. Für GF(2)-Polynome mit Konstante 1 gilt der folgende

Satz 3.3.16 Sei p ein Polynom in den Variablen $\{x_1, \ldots x_n\}$ mit $p(0, \ldots, 0) = 1$. Sei $U = \{0, 1\}^n$ die Menge aller Einsetzungen für die Variablen und G die Menge aller Einsstellen von p. Dann gilt

$$|U|/|G| \leq m.$$

Dieser Satz impliziert direkt einen Algorithmus für diese Polynome, der in Laufzeit

$$O(nm^2 \log(1/\delta)/\epsilon^2)$$

implementierbar ist.

3.3.3 Approximative Zählung von c-Stellen multilinearer Polynome über GF(q)

In diesem Abschnitt wollen wir die Anzahl von c-Stellen von multilinearen Polynomen über GF(q) bestimmen ([KL 91]).

Definition 3.3.17 Ein multivariates Polynom $f(x_1, \ldots, x_n)$ heißt multilinear, wenn $\deg_{x_i} \leq 1$ für jedes i gilt.

Definition 3.3.18 Es bezeichne

$$\#_c g = |\{(x_1, \ldots x_n) \in \mathrm{GF}(q)^n \mid g(x_1, \ldots x_n) = c\}|$$

die Anzahl der c-Stellen von $g \in \mathrm{GF}(q)[x_1, \ldots, x_n]$.

Lemma 3.3.19 Sei $g \in \mathrm{GF}(q)[x_1, \ldots, x_n]$ ein multilineares nicht konstantes Polynom. Dann gilt für alle $c \in \mathrm{GF}(q)$:

$$\#_c g \geq (q-1)^{n-1}.$$

Beweis: Wir beweisen das Lemma durch Induktion über n der Anzahl der Variablen.

$n = 1$ Sei $g(x) = ax + b$ mit $a \neq 0$. Dann hat die Gleichung $g(x) = s$ für jedes $s \in \mathrm{GF}(q)$ genau eine Lösung. Somit gilt

$$\#_c g = 1 = (q-1)^0.$$

$n \to n+1$ Wir stellen das Polynom g folgendermaßen dar:

$$g(x_1, \ldots x_{n+1}) = x_{n+1} h(x_1, \ldots x_n) + k(x_1, \ldots x_n).$$

Dabei sind h und k multilineare Polynome in den Variablen $x_1, \ldots, x_n$. Wir unterscheiden nun drei Fälle:

- $h(x_1, \ldots, x_n) \equiv 0$
  In diesem Fall ist
  $$\#_c g = q \#_c k \geq q(q-1)^{n-1} < (q-1)^n.$$
- $h(x_1, \ldots x_n) \equiv d \neq 0$
  Das Polynom g hat also die Gestalt $g(x_1, \ldots x_{n+1}) = x_{n+1} d + k(x_1, \ldots x_n)$. Für festes $(x_1, \ldots x_n)$ haben wir ein lineares univariates Polynom, welches jeden Wert $c \in \mathrm{GF}(q)$ genau einmal annimmt unabhängig von dem Wert von $k(x_1, \ldots, x_n)$. Für jede beliebige Belegung der Variablen $x_1, \ldots, x_n$ hat $g(x_1, \ldots, x_n, x_{n+1})$ genau eine Lösung. Somit ist
  $$\#_c g = q^n > (q-1)^n.$$
- $h(x_1, \ldots, x_n)$ ist nicht konstant. Nach Induktionsannahme gibt es mindestens $(q-1)^{n-1}$ Lösungen der Gleichung $h(x_1, \ldots x_n) = d$. Für $d \neq 0$ ist gibt es für jede dieser Lösungen $(x_1, \ldots, x_n$ genau ein x_{n+1} mit $g(x_1, \ldots, x_{n+1}) = c$. Damit gilt
  $$\#_c g \geq \sum_{d \neq 0} \#_d h \geq (q-1)(q-1)^{n-1} = (q-1)^n.$$

□

Bezeichne $\mathrm{Var}(t)$ die Variablenmenge eines Polynoms oder Terms t.

Satz 3.3.20 Sei $f \in \mathrm{GF}(q)[x_1, \ldots, x_n]$ ein multilineares Polynom und $c \in \mathrm{GF}(q)$. Bezeichne m die Anzahl der Terme von $f = \sum t_i$, $D(f) := \{s \in \mathrm{GF}(q)^n \mid \exists t_i \; t_i(s) \neq 0\}$ und $S_c(f) := \{s \in \mathrm{GF}(q)^n \mid f(s) = c\}$. Falls f keinen konstanten Term enthält oder der konstante Term gleich c ist, dann gilt

$$\frac{|D(f)|}{|S_c(f)|} \leq (q-1)m.$$

BEWEIS: Wir partitionieren $D(f)$ in Mengen $D_{i,j}(f)$ und überdecken $S_c(f)$ durch eine gleiche Anzahl von Mengen $R_{i,j}(f)$. Durch eine Beziehung zwischen den Mengen $R_{i,j}(f)$ und$D_{i,j}(f)$ erhalten wir dann die gewünschte Aussage.

Zunächst überdecken wir $D(f)$ durch nicht notwendigerweise disjunkte Mengen $D_i(f)$:

$$D_i(f) := \{s \in \mathrm{GF}(q)^n \mid t_i(s) \neq 0, \neg\exists t_j > t_i \text{ mit } t_j(s) \neq 0\}.$$

Dabei bedeutet $t_i < t_j$, daß $\mathrm{Var}\,(t_i) \subset \mathrm{Var}\,(t_j)$ gilt.

$D_i(f)$ besteht also aus den Zuweisungen s, für die der Term t_i maximal unter allen Termen mit $t(s) \neq 0$ ist.

$D_i(f)$ partitionieren wir in $q^{n-\deg t_i}$ Mengen $D_{i,j}(f)$. $D_{i,j}(f)$ besteht aus all den Zuweisungen aus $D_i(f)$, die auf den Variablen, die nicht in t_i auftauchen, gleich sind. Die Größe von $D_{i,j}(f)$ ist durch die Anzahl von Zuweisungen s zu den Variablen in t_i mit $t_i(s) \neq 0$ beschränkt. Da jede der $\deg t_i$ vielen Variablen einen Wert ungleich 0 erhalten muß, sind dies $(q-1)^{\deg t_i}$ viele.

Zu den Mengen $D_{i,j}(f)$ definieren wir die Mengen $R_{i,j}(f)$ in der folgenden Weise. Die Zuweisungen zu den Variablen, die nicht in t_i auftauchen, sei so wie in $D_{i,j}(f)$, d.h. wir betrachten Polynome p , die aus einer partiellen Zuweisung der Variablen hervorgehen. Diese Polynome sind Polynome in den Variablen in $\mathrm{Var}\,(t_i)$. Wir erweitern nun die partielle Zuweisung, so daß p und damit f zu c evaluiert.

Aufgrund der Definition der Mengen $D_i(f)$, werden die Polynome, die durch partielle Zuweisung der Variablen aus f hervorgehen, nicht konstant. Daher können wir Lemma 3.3.19 auf die Polynome p anwenden und erhalten daher für alle i, j eine untere Schranke für die Kardinalität der Mengen $R_{i,j}(f)$:

$$|R_{i,j}(f)| \geq (q-1)^{\deg t_i - 1} \geq \frac{|D_{i,j}(f)|}{q-1}.$$

Da die Mengen $R_{i,j}(f)$ die Menge $S_c(f)$ überdecken haben, wir die Ungleichung

$$|S_c(f)| \leq \sum_{i,j} |R_{i,j}(f)|.$$

Aufgrund der Konstruktion sind die Mengen $R_{i,j}(f)$ und $R_{i,k}(f)$ für $j \neq k$ disjunkt, da sie den Variablen außerhalb von $\mathrm{Var}\,(t_i)$ verschiedene Werte zuweisen. Daher kann jedes Element $s \in S_c(f)$ in höchstens m verschiedenen Mengen $R_{i,j}(f)$ auftauchen. Damit haben wir

$$m \cdot |S_c(f)| \geq \sum_{i,j} |R_{i,j}(f)|.$$

Zusammenfassend ergibt sich

$$\frac{|D(f)|}{|S_c(f)|} \leq \frac{\sum_{i,j} |D_{i,j}(f)|}{1/m \sum_{i,J} |R_{i,j}(f)|} \leq \frac{\sum_{i,j} (q-1) |R_{i,j}(f)|}{1/m \sum_{i,j} |R_{i,j}(f)|} = m(q-1).$$

□

Die Schranke des Satzes 3.3.20 ist scharf. Dies zeigt das Beispiel

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} x_i.$$

Es gibt $(q-1)^n$ Belegungen, so daß (der einzige) Term von f ungleich 0 wird, aber genau $(q-1)^{n-1}$ viele Belegungen, so daß f den Wert c annimmt. Das Verhältnis ist genau $q-1$.

Wir haben nun wieder eine geeignete Überdeckung konstruiert, und können wieder dasselbe Schema für einen (ϵ, δ)-Approximationsalgorithmus anwenden.

Kürzlich wurde dieses Ergebnis von Grigoriev und Karpinski [GK 91] auf den Fall beliebige Polynome über GF(q) erweitert. Wir wollen jedoch auf eine Darstellung verzichten und verweisen auf die Literatur.

Literaturverzeichnis

[AHU 74] Aho,A.V., Hopcroft,J.E, Ullman,J.D., *The Design and Analysis of Computer Algorithms*, Addison - Wesley, 1974.

[AKS 83] Ajtai,M., Komlos,E., Szemeredi,E., *An $O(n \log n)$ sorting network*, STOC 1983, pp. 1–9, und *Sorting in $c \log n$ Parallel Steps*, Combinatica 3, 1983, pp. 1–18.

[Ba 68] Batcher,K., *Sorting Networks and Their Application*, AFIPS Spring Joint Computing Conference 32, 1968, pp. 307–314.

[Be 70] Berlekamp,E.R., *Factoring Polynomials over Large Finite Fields*, Math. Comp. 24, 1970.

[Be 84] Berkowitz,S., *On Computing the Determinant in Small Parallel Time Using a Small Number of Processors*, Inform. Process. Lett. 18, 1984, pp. 147–150.

[Bl 67] Blum,M., *A Machine-Independent Theory of the Complexity of Recursive Functions*, J. ACM 14, 1967, pp. 322–336.

[Bo 77] Borodin,A., *On Relating Time and Space to Size and Depth*, SIAM J. Comput. 6, 1977, pp. 733–744.

[BGH 82] Borodin,A., Gathen,J. von zur, Hopcroft,J.E.,*Fast Parallel Matrix and GCD Computations*, Information and Control 52, 1982, pp. 241–256.

[CDGK 88] Clausen,M., Dress,A., Grabmeier,J., Karpinski,M.,*On Zero-Testing and Interpolation of k-sparse Multivariate Polynomials over Finite Fields*, Research Report No. 8522-CS, University of Bonn, 1988.

[CGK 87] Clausen,M., Grabmeier,J., Karpinski,M., *Efficient Deterministic Interpolation of Multivariate Polynomials over Finite Fields*, Research Report No. 8519-CS, University of Bonn, 1987.

[Co 85] Cook,S.A.,*A Taxonomy of Problems with Fast Parallel Algorithms*, Information and Control 63, 1985, pp. 2–22.

[Co 86] Cole,R.,*Parallel Merge Sort*, FOCS 1986, pp. 511–516.

[Cs 76] Csanky,L., *Fast Parallel Matrix Inversions Algorithms*, SIAM J. Comput. 5, 1976, pp. 618–623.

[EK 90] Ehrenfeucht,A., Karpinski,M., *The Computational Complexity of (XOR,AND)-Counting Problems*, Research Report No. 8543-CS, University of Bonn, 1990.

[Fi 82] Fischer,G., *Lineare Algebra*, Vieweg Studium, [FW 78] Fortune,S., Wyllie,J., *Parallelism in Random Access Machines*, PROC 10th ACM STOC 78.

[Ga 84] Gathen,J. von zur, *Parallel Agorithms fom Algebraic Problems*, SIAM J. Comput. 13, 1984, pp. 172–179.

[Ga 86] Gathen,J.von zur, *Parallel Arithmetic Computations*, Lecture Notes in Computer Science 233, Springer Verlag, 1986, pp. 93–112.

[Gi 77] Gill,J.,*Computational Complexity of Probabilisitic Turing Machines*, SIAM J.Comput. 6, 1977, pp. 675–695.

[GK 87] Grigoriev,D.Y., Karpinski,M., *The Matching Problem for Bipartite Graphs with Polynomially Bounded Permanents is in NC*, FOCS 1987, pp. 166–172.

[GK 91] Grigoriev,D.Y., Karpinski,M., *An Approximation Algorithm for the Number of Zeros of Arbitrary Polynomials over $GF(q)$*, Research Report No. 8565-CS, University of Bonn, 1991.

[GKS 88] Grigoriev,D.Y., Karpinski,M., Singer,M.F., *Fast Parallel Algorithms for Sparse Multivariate Polynomial Interpolation over Finite Fields*, Research Report No. 8523-CS, University of Bonn, 1988.

[GR 88] Gibbons,A., Rytter,W., *Efficient Parallel Algorithms*, Cambridge University Press 1988.

[HKP 84] Hoover,H.J., Klawe,M.M., Pippenger,N.J., *Bounding Fan-Out in Logical Networks*, J ACM 31, 1984, pp. 13–18.

[HS 79] Horowitz,E., Sahni,S., *Fundamentals of Computer Algorithms*, Computer Science Press, 1979.

[HU 79] Hopcroft,J.E., Ullman,J.D., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

[KL 85] Karp,R.M., Luby,M., *Monte-Carlo Algorithms for the Planar Multiterminal Network Reliability Problem*, Journal of Complexity 1, 1985, pp. 45–64.

[KL 90] Karpinski,M., Luby,M., *Approximating the Number of Solutions of a GF[2] Polynomial*, Proc. of the second annual ACM-SIAM Symposium on discrete Algorithms, Nr.2, 1990, pp 300–303.

[KL 91] Karpinski,M., Lhotzky,B., *An (ϵ, δ) Approximation Algorithm of the Number of Zeros of a Multilinear Polynomial over $GF(q)$*, Research Report No. 8564-CS, University of Bonn, 1991.

[KLM 87] Karp,R.M., Luby,M., Madras,N., *Monte-Carlo Approximation Algorithms for Enumeration Problems*, draft 1987.

[KR 89] Karp,R.M., Ramachandran,V., *A Survey of Parallel Algorithms for Shared-Memory Machines*, Handbook of Theoretical Computer Science, Vol. A, 1989, pp. 869–942.

[KV 85] Karpinski,M., Verbeek,R., *There Is No Polynomial Deterministic Space Simulation of Probabilistic Space with a Two-Way Random-Tape Generator*, Information and Control 67, 1985, pp. 158–162.

[Kn 81] Knuth,D.E., *The Art of Computer Programming*, Addison Wesley, 1981.

[LN 86] Lidl, Niederreiter, *Introduction into Finite Fields and their Application*, Cambridge University Press, 1986.

[Me 84] Mehlhorn,K., *Data Structures and Algorithms*, Springer- Verlag, 1984.

[MP 77] McColl,W.F., Paterson,M.S., *The Depth of All Boolean Functions*, SIAM J.Comput. 6, 1977, pp.373–380.

[Mu 87] Mulmeley,K., *A Fast Parallel Algorithm to Compute the Rank of a Matrix over an Arbitrary Field*, Combinatorica 7, 1987.

[MVV 87] Mulmeley,K., Vazirani,U., Vazirani,V., *Matching is as Easy as Matrix Inversion*, Combinatorica 7, 1987.

[Pa 87] Pan,V., *Complexity of Parallel Matrix Computations*, Theoretical Computer Science 54, 1987, pp. 65–85.

[Pa 87] Parberry,I., *Parallel Complexity Theory*, Wiley & Sons, 1987.

[Sa 76] Savage,J.E., *The Complexity of Computing*, Wiley, 1976.

[Sc 80] Schwartz,J.T., *Fast Probabilistic Algorithms for Verification of Polynomial Identities*, J.ACM 27, 1980, pp. 701–717.

[SS 63] Shepherdson,J.C., Sturgis,H.E., *Computability of Recursive Functions*, J.ACM 10, 1963, pp. 217–255.

[SS 71] Schönhage,A., Strassen,V., *Schnelle Multiplikation großer Zahlen*, Computing 7, 1971, pp. 281–292.

[SV 84] Stockmeyer,L., Vishkin,U., *Simulation of Parallel Random Access Machines by Circuits*, SIAM J.Comput. 13, 1984, pp. 409–422.

[Va 79a] Valiant, L.G., *The Complexity of Computing the Permanent*, Theoretical Computer Science 8, 1979, pp. 189–201.

[Va 79b] Valiant, L.G., *The Complexity of Enumeration and Reliability Problems*, SIAM J.Comput. 8, 1979, pp. 410–421.

[We 87] Wegener,I., *The Complexity of Boolean Functions*, Teubner 1987.

[WSHF 81] Wulf,W.A., Shaw,M., Hilfinger,P.N., Flon,L., *Fundamental Structures of Computer Science*, Addison Wesley, 1981.