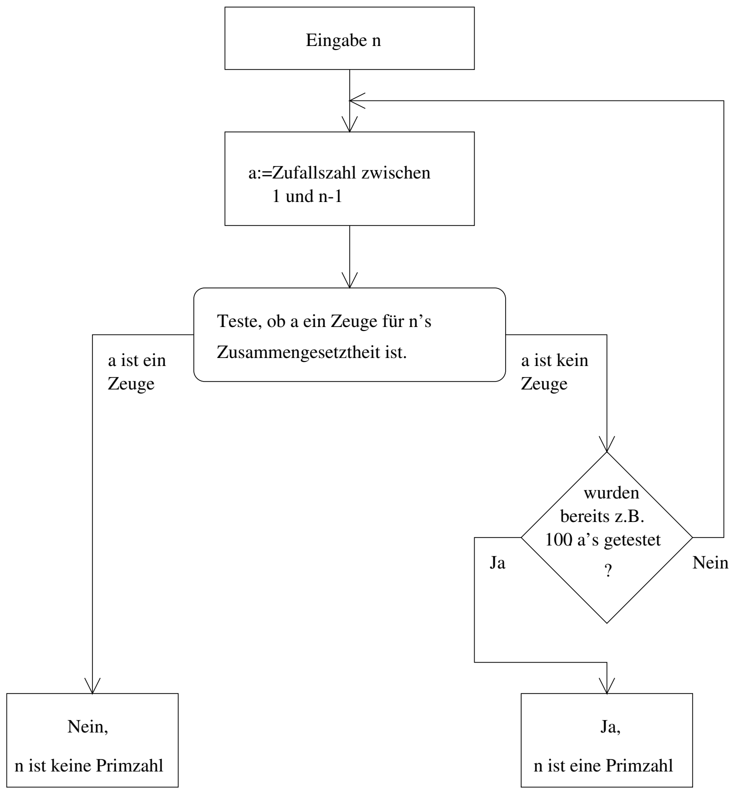
Randomisierte Algorithmen *

Rolf Niedermeier
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen
niedermr@informatik.uni-tuebingen.de



*Skript zu einer zweistündigen Vorlesung im Sommersemester 1997 für den Bereich „Theoretische Informatik“ an der Universität Tübingen. Das Skript ist erhältlich über http://www-fs.informatik.uni-tuebingen.de/~niedermr/teaching/
LaTeX-Bearbeitung von Patrick Meißner, Version vom 18. Juli 1997.

Inhaltsverzeichnis

0 Vorwort, Übersicht und Literatur

Nehmen wir an, daß es jedes Jahrtausend mindestens einen Meteoriteneinschlag gibt, der mehr als 100 Quadratmeter der Erdoberfläche verwüstet. Dann läßt sich folgern, daß ein Rechner während einer Mikrosekunde seines Daseins mit einer Wahrscheinlichkeit von mindestens 2^{-100} zerstört wird. Von diesem Standpunkt aus erscheint ein Algorithmus, der mit Fehlerwahrscheinlichkeit kleiner 2^{-100} in Sekundenschnelle $2^{400} - 593$ als die größte Primzahl kleiner 2^{400} identifiziert, als sehr praktisch (insbesondere in Bezug auf dahinterstehende, kryptologische Anwendungen). Ein *randomisierter Algorithmus* ist ein Algorithmus, dessen Verhalten in jedem Schritt von einem Zufallsexperiment („Münzwurf") abhängen kann. Da sich randomisierte Algorithmen nicht mehr deterministisch verhalten, können die Laufzeit und/oder die Korrektheit nur noch mit gewissen Wahrscheinlichkeiten garantiert werden. Es gibt zwei grundsätzliche Vorteile randomisierter Algorithmen gegenüber „normalen" (deterministischen) Algorithmen: Effizienz und Einfachheit (letzteres führt oft auch zu besserer Implementierbarkeit). In der Vorlesung werden grundlegende randomisierte Algorithmen für verschiedenste Anwendungen vorgestellt. Zudem werden randomisierte Komplexitätsklassen betrachtet und einige Methoden und Werkzeuge für die Analyse randomisierter Algorithmen bereitgestellt.

In Kapitel 1 lernen wir gleich zu Beginn eine wichtige Eigenschaft von randomisierten Algorithmen kennen: Fast alle gehorchen einer kleinen Anzahl von grundlegenden Paradigmen. Die nachfolgenden Beispiele dienen dazu, einige dieser Paradigmen zu veranschaulichen und erste Analysemethoden kennenzulernen. Anschließend begegnen wir in Kapitel 2 einem der berühmtesten randomisierten Algorithmen, dem Primzahltest von Miller/Rabin. Neben der Kryptographie ist das Erzeugen von zufälligen Primzahlen auch für einen bekannten randomisierten Pattern-Matching-Algorithmus sehr nützlich. Wir stellen den Karp/Rabin-Algorithmus als Anwendung von Primzahltests vor. Danach erhöhen wir das Abstraktionsniveau und gehen von konkreten algorithmischen Problemen über zu Problem- beziehungsweise Komplexitätsklassen (Kapitel 3). Hier begegnen uns Fragen nach der Mächtigkeit verschiedener randomisierter Algorithmen und Konzepte, ihren Zusammenhängen und insbesondere die (weitgehend offene) Frage nach den Grenzen der Randomisierung. Zum Abschluß beschäftigen wir uns in Kapitel 4 mit einem der wichtigsten Anwendungsfelder für Randomisierung: Parallele Algorithmen. Hier beweisen wir das erstaunliche „Isolations-Lemma" und wenden es auf das Problem „Perfektes Matching" an. Zuletzt studieren wir Randomisierung im Zusammenhang mit Routing-Verfahren für Parallel-Rechner. In den Anhängen A, B und C werden einfache Grundlagen aus der Wahrscheinlichkeitstheorie, der Asymptotik („O-Notation") und der Komplexitätstheorie bereitgestellt.

Literatur: Die Hauptquelle für die Vorlesung ist das umfangreiche Buch

R. Motwani und P. Raghavan:
Randomized Algorithms,
Cambridge University Press, 1995.

Desweiteren wurde aus einigen Übersichtsartikeln geschöpft:

- R. Gupta, S. A. Smolka, S. Bhaskar: On Randomization in Sequential and Distributed Algorithms, *ACM Computing Surveys*, Vol. 26(1), 7–86, March 1994.
- D. S. Johnson: The NP-Completness Column: An Ongoing Guide, *Journal of Algorithms*, Vol. 5, 433–447, 1984.
- R. M. Karp: An introduction to randomized algorithms, *Discrete Applied Mathematics*, Vol 34, 165–201, 1991.
- R. Motwani, P. Raghavan: Randomized Algorithms, *ACM Computing Surveys*, Vol. 28(1), 33–37, CRC Press, 1996.
- R. Motwani, P. Raghavan: Randomized Algorithms, *The Computer Science and Engineering Handbook*, 141–161, Ed. A. B. Tucker Jr., CRC Press, 1997.
- D. J. A. Welsh: Randomized Algorithms, *Discrete Applied Mathematics*, Vol. 5, 133–145, 1983.

Hilfreich sind auch viele Bücher und Skripten zum Thema Algorithmen und Komplexität, die meist auch einen Abschnitt randomisierten Algorithmen und Komplexitätsklassen widmen, zum Beispiel:

- G. Brassard, P. Bratley: *Fundamentals of Algorithmics*, Prentice Hall, 1996.
- D. P. Bovet, P. Crescenzi: *Introduction to the Theory of Complexity*, Prentice Hall International Series in Computer Science, 1994.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Introduction to Algorithms*, The MIT Press, 1994.
- T. Hagerup: *Vorlesungsskript Komplexitätstheorie*, Research Report, MPI-I-96-005, Max-Planck-Institut für Informatik, Saarbrücken, March 1996.
- D. Harel: *Algorithmics, The Spirit of Computing*, Addison Wesley Publishing Company, 1987.

- D. S. Hochbaum (Ed.): *Approximation Algorithms for NP-hard problems*, PWS Publishing Company, 1997.
- C. H. Papadimitriou: *Computational Complexity*, Addison Wesley Publishing Company, 1994.

Einen „randomisierten Zugang“ zum Thema „Algorithmische Geometrie“ bietet das Buch von K. Mulmuley: *Computational Geometry, An Introduction Through Randomized Algorithms*, Prentice Hall, 1994.

Bezüge zu anderen Vorlesungen:

Allgemein: Einordnung in das Gebiet Algorithmen und Komplexität.

Speziell:

- Komplexitätstheorie,
- Algorithmen und Komplexität,
- Algorithmische Geometrie,
- Parallele Algorithmen,
- Kolmogorov-Komplexität,
- Kryptologie und Komplexität,
- Datenstrukturen,
- Wahrscheinlichkeitstheorie,
- Kombinatorik,
- Algorithmische Zahlentheorie.

Nicht in dieser Vorlesung:

- (Philosophische) Diskussion des Begriffs „Zufall“,
- Erzeugung von Zufallszahlen: Pseudo Random Generators; siehe dazu unter anderem D. E. Knuth: *The Art of Computer Programming II (Seminumerial Algorithms)*, Addison Wesley Publishing Company, 1981,
- probablistische Analyse deterministischer Algorithmen,
- Average-Case-Analyse,
- und vieles mehr, insbesondere auch wahrscheinlichkeitstheoretisch anspruchsvollere Themenstellungen.

1 Einführung

Ein *randomisierter Algorithmus* (viele Autoren sprechen auch von *probabilistischen Algorithmen*) ist ein Algorithmus, dessen Verhalten in jedem Schritt von einem Zufallsexperiment („Münzwurf“) abhängen kann.

Folgendes Beispiel, bei dem wir noch vieles offen lassen, soll lediglich einen ersten Eindruck vermitteln.

Beispiel 1.1
Gegeben: Ein Polynom $f(x_1, ..., x_n)$ in n Variablen.
Frage: Ist f identisch gleich dem Nullpolynom (in Zeichen: $f \equiv 0$)?
Eine analytische Überprüfung dieser Eigenschaft kann einen extrem hohen Rechenaufwand bedeuten. Ein randomisierter Algorithmus hierfür ist aber ganz einfach:

1. Erzeuge einen Zufallsvektor $(r_1, ..., r_n)$ und bestimme $f(r_1, ..., r_n)$.
2. Falls $f(r_1, ..., r_n) \neq 0$, so wissen wir, $f \not\equiv 0$.
3. Falls $f(r_1, ..., r_n) = 0$, dann ist entweder $f \equiv 0$, oder wir hatten das ungewöhnliche Glück, eine Nullstelle von f geraten zu haben. In diesem Falle wiederholen wir den Algorithmus einige Male. Falls sich dabei immer $f(r_1, ..., r_n) = 0$ ergibt, so schließen wir $f \equiv 0$, andernfalls $f \not\equiv 0$.

Durch genügend häufiges Wiederholen wird die Wahrscheinlichkeit für eine Fehlentscheidung „vernachlässigbar“.

Das Verhalten eines randomisierten Algorithmus (insbesondere seine Laufzeit oder die Qualität (Korrektheit) seiner Ausgabe) variiert im allgemeinen von einer Ausführung zur nächsten, selbst bei gleicher Eingabe. In der Analyse randomisierter Algorithmen befassen wir uns mit „*erwarteten*“ Werten eines Leistungsmaßes (wie zum Beispiel die Laufzeit), welche für *jede* Eingabe gültig sind. Insbesondere gilt, daß die Laufzeiten und/oder Korrektheit randomisierter Algorithmen nur noch mit hoher Wahrscheinlichkeit garantiert werden können.

1.1 Ursprünge und Motivation

- Randomisierte Algorithmen in „primitiven“ Kulturen. Beispiel: Zufallswahl (mittels „Orakel“...) des Platzes/Gebietes für die Jagd. Siehe dazu J. Shallit: Randomized Algorithms in “Primitive” Cultures or what ist the Oracle Comlexity of a Dead Chicken?, *ACM SIGACT News* Vol. 23(4), 77-80, 1992.

- Wurzeln randomisierter Algorithmen liegen bei Monte Carlo Methoden in der numerischen Analyse, statistischen Physik und Simulation.
- Randomisierte Algorithmen traten in das Rampenlicht der Informatik mit den Primzahltests von Rabin (Probabilistic Algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, Academic Press 21-39, 1976) und Solovay/Strassen (A fast Monte-Carlo Test for primality, *SIAM Journal on Computing* Vol. 6, 84-86, 1977) Mitte der 70er Jahre.

Es gibt grundsätzlich zwei Arten von Vorteilen, die randomisierte Algorithmen haben:

1. Oft ist die Laufzeit/der Speicherplatzbedarf geringer als bei den besten deterministischen Algorithmen: *Effizienz.*
2. Sie sind oft sehr einfach zu verstehen und zu implementieren: *Einfachheit.*

Besondere Bedeutung haben randomisierte Algorithmen im Bereich des parallelen und verteilten Rechnens. Hier gibt es sogar Probleme, die nachweislich *keine* Lösung durch deterministische Algorithmen (Protokolle) haben, sehr wohl aber durch randomisierte.

1.2 Paradigmen für randomisierte Algorithmen

Einige wichtige Grundprinzipien unterliegen fast allen randomisierten Algorithmen. Einige davon werden wir im Laufe der Vorlesung näher kennenlernen:

- *Vereitelung der „Angriffe“ eines Gegner* (foiling an adversary):
  Betrachte Problem als ein Spiel zwischen Algorithmusentwerfer und (bösartigem) Gegenspieler. Dank Randomisierung kann der Gegenspieler nicht die Schritte des Algorithmus vorhersagen und somit keine „schlimme“ (Worst-Case) Eingabe konstruieren.
- *Zufallsstichproben* (random sampling):
  Eine kleine, zufällige Stichprobe repräsentiert eine sehr große Menge.

- *Überfluß an Zeugen* (abundance of witnesses):
  In einem großen Suchraum liegen relativ viele Elemente („Zeugen") mit einer bestimmten Eigenschaft, so daß ein zufällig gewähltes Element mit gewisser Wahrscheinlichkeit Zeuge ist.

- *Signaturen („Fingerabdrücke") und Hashing* (fingerprinting and hashing):
  Repräsentation eines großen Objektes durch einen kleinen „Fingerabdruck" mittels einer „zufälligen" Abbildung.

- *Zufällige Umordnung* (random reordering):
  Durch zufällige Umordnung der Eingabe werden „schlimme" Anordnungen mit hoher Wahrscheinlichkeit ausgeschlossen und ein relativ naiver Algorithmus erzielt gute Leistung.

- *Lastbalancierung* (load balancing):
  Auswahl zwischen verschiedenen Ressourcen (zum Beispiel Verbindungskanälen) mittels Randomisierung zwecks gleichmäßiger Verteilung der Last.

- *Schnell konvergierende Markov-Ketten* (rapidly mixing Markov chains):
  Stochastischer Prozeß zur Erzeugung einer „guten" (gleichförmigen) Stichprobe aus einem großen Raum.

- *Isolierung und Aufbrechen von Symmetrien* (isolation and symmetry breaking):
  Beim verteilten oder parallelen Rechnen müssen mehrere Prozessoren oft eine gemeinsame Entscheidung fällen. Randomisierung hilft zur Vermeidung von Deadlocks oder zum Aufbrechen von Symmetrien.

- *Probabilistische Methoden und Existenzbeweise* (probabilistic methods and existence proofs):
  Beweis der Existenz eines spezifischen kombinatorischen Objekts (zum Beispiel eines Graphen mit einer bestimmten Eigenschaft), indem gezeigt wird, daß ein zufällig gewähltes Objekt aus einem vernünftig definierten Universum die gewünschte Eigenschaft mit von Null verschiedener Wahrscheinlichkeit besitzt.
  Literaturhinweis: N. Alon, J. H. Spencer and P. Erdös: *The Probabilistic Method*, Wiley 1992.

1.3 Einige Beispiele und Begriffe

Nachfolgend betrachten wir fünf verschiedene randomisierte Algorithmen, die deren Vielfalt veranschaulichen und einige wichtige Begriffe einführen sollen. Zudem bieten sie zugleich Beispiele für einige der in Abschnitt 1.2 diskutierten Paradigmen.

1.3.1 Matrixmultiplikation

Gegeben: Drei $n \times n$-Matrizen A, B, C.
Frage: Ist $A \cdot B = C$?

Freivalds gab 1979 folgenden randomisierten Algorithmus mit Laufzeit $O(n^2)$ und beschränkter Fehlerwahrscheinlichkeit an:

1. Wähle einen Vektor $r \in \{0,1\}^n$ per Zufall, das heißt, jede Komponente von r wird mit Wahrscheinlichkeit $\frac{1}{2}$ gleich 0 und mit Wahrscheinlichkeit $\frac{1}{2}$ gleich 1 gesetzt.
2. Berechne
   (a) $x := B \cdot r$,
   (b) $y := A \cdot x$,
   (c) $z := C \cdot r$.
3. Falls $y = z$, so antworte ja, sonst nein.

Falls $A \cdot B = C$ gilt, so liefert obiger Algorithmus immer ein richtiges Ergebnis. Falls $A \cdot B \neq C$, so zeigt folgender Satz, daß mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ gilt, daß $y \neq z$. Durch k-maliges Wiederholen des Algorithmus läßt sich die Fehlerwahrscheinlichkeit auf $(\frac{1}{2})^k$ drücken.

Satz 1.2 *Seien A, B und C drei $n \times n$-Matrizen über einem Körper K mit $A \cdot B \neq C$. Dann gilt für einen per Gleichverteilung gegebenen Zufallsvektor $r \in \{0,1\}^n$, daß $\mathbf{Pr}[A \cdot B \cdot r = C \cdot r] \leq \frac{1}{2}$.*

Beweis: Sei $D := A \cdot B - C$. Nach Voraussetzung ist D ungleich der Nullmatrix. Wir schätzen die Wahrscheinlichkeit für $D \cdot r = \vec{0}$ ab. OBdA (warum?) können wir annehmen, daß die erste Reihe von D einen von Null verschiedenen Eintrag hat. Zudem können wir auch annehmen, daß alle von Null verschiedenen Einträge in dieser Reihe den Nulleinträgen vorausgehen. Sei d dieser Zeilenvektor von D, wobei die ersten $k \geq 1$ Einträge von d ungleich Null seien. Das Skalarprodukt $d \cdot r$ ergibt den ersten Eintrag von $D \cdot r$. Somit liefert $\mathbf{Pr}[d \cdot r = 0]$ eine obere Schranke für $\mathbf{Pr}[D \cdot r = 0]$. Seien r_i und d_j mit $1 \leq i, j \leq n$ die Komponenten von r und d. Dann gilt:

$$d \cdot r = 0 \iff r_1 = \frac{-\sum_{i=2}^{k} d_i \cdot r_i}{d_1}$$

Wir können ohne Einschränkung annehmen, daß $r_2, ..., r_n$ vor r_1 gewählt wurden (Prinzip der „verzögerten Entscheidung“). Dann aber nimmt der Bruch einen festen Wert w an. Da r_1 mit jeweils Wahrscheinlichkeit $\frac{1}{2}$ die Werte 0 oder 1 annimmt, kann r_1 höchstens mit Wahrscheinlichkeit $\frac{1}{2}$ gleich w sein. Also $\mathbf{Pr}[D \cdot r = 0] \leq \frac{1}{2}$.

□

Bemerkung 1.3

- Freivalds Algorithmus ist ein Beispiel für die *Signaturmethode (Fingerabdrücke).*
- Es handelt sich um einen *Monte-Carlo-Algorithmus*, (das heißt, der Algorithmus produziert manchmal eine falsche Lösung) mit einem *einseitigen Fehler* (das heißt, er irrt sich nur im Falle $A \cdot B \neq C$).
- Literatur: R. Freivalds: „Fast probablistic algorithms“, Springer LNCS 74, 57-69, 1979.

1.3.2 Randomisiertes Quicksort

Gegeben: Eine Menge S von Zahlen, die alle verschieden sein sollen.
Aufgabe: Sortiere die Zahlen S in aufsteigender Ordnung.

Beim randomisierten Quicksort-Algorithmus wird im Gegensatz zum deterministischen Quicksort-Algorithmus das Splitelement per Zufall gewählt:

1. Wähle per Zufall ein Element y aus S, so daß jedes Element mit gleicher Wahrscheinlichkeit gewählt wird („Gleichverteilung“).
2. $S_1 := \{x \in S \mid x < y\}$,
   $S_2 := \{x \in S \mid x > y\}$.
3. Sortiere rekursiv S_1 und S_2. Ausgabe: Sortierte Version von S_1 gefolgt von y gefolgt von der sortierten Version von S_2.

Wieviele Vergleiche benötigt der Algorithmus? Was ist die *erwartete* Laufzeit? Dazu definieren wir:

$S_{(i)}$:= i-t kleinstes Element in S und $n := |S|$.

$$\text{Zufallsvariable } X_{ij} := \begin{cases} 1: & \text{falls } S_{(i)} \text{ und } S_{(j)} \text{ im Laufe des Algorithmus} \\ & \text{miteinander verglichen werden,} \\ 0: & \text{sonst.} \end{cases}$$

Somit ergibt sich die Gesamtzahl der Vergleiche durch $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$ und für die erwartete Anzahl von Vergleichen

$$E[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}], \tag{1}$$

wobei die Linearität des Erwartungswertes ausgenutzt wird. Setzen wir

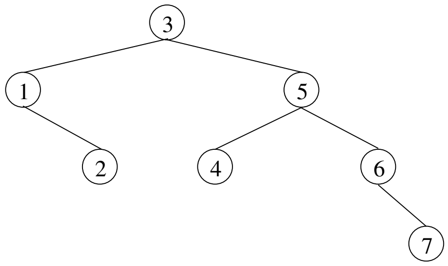
$$p_{ij} := \mathbf{Pr}[X_{ij} = 1],$$

dann gilt:

$$E[X_{ij}] := 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) = p_{ij}.$$

Zur leichteren Bestimmung von p_{ij} betrachten wir den zur Ausführung des Algorithmus gehörigen Binärbaum, bei welchem die inneren Knoten die jeweils gewählten Splitelemente repräsentieren. Zur Ermittlung von p_{ij} ist es nützlich, einen schichtweisen (in der Schicht von links nach rechts) Durchlauf des Binärbaums zu betrachten.

Beispiel 1.4
$S = \{3, 4, 2, 1, 5, 7, 6\}$. Ein möglicher Binärbaum:



Zwei Beobachtungen:

1. Die Elemente $S_{(i)}$ und $S_{(j)}$ mit $i < j$ werden verglichen genau dann, wenn es beim schichtweisen Durchlauf des Baumes kein Element $S_{(l)}$

mit $i < l < j$ gibt derart, daß $S_{(l)}$ sowohl vor $S_{(i)}$ als auch vor $S_{(j)}$ durchlaufen wird.
Denn: Betrachte ein $S_{(k)}$ mit $i \leq k \leq j$ derart, daß $S_{(k)}$ das erste der Elemente $S_{(i)}$, $S_{(i+1)}$, ..., $S_{(j)}$ ist, das beim schichtweisen Durchlauf erreicht wird. Fallunterscheidung:

(a) $k \notin \{i, j\}$: Dann gehört $S_{(i)}$ zum linken und $S_{(j)}$ zum rechten Unterbaum von $S_{(k)}$, also gibt es keinen Vergleich zwischen $S_{(i)}$ und $S_{(j)}$.

(b) $k \in \{i, j\}$: Dann gibt es eine Vorfahr-Nachfahr-Beziehung zwischen $S_{(i)}$ und $S_{(j)}$, sie werden also miteinander verglichen.

2. Jedes der Elemente $S_{(i)}$, $S_{(i+1)}$, ..., $S_{(j)}$ spielt mit gleicher Wahrscheinlichkeit ($= \frac{1}{j-i+1}$) die Rolle des $S_{(k)}$. Die Wahrscheinlichkeit, daß $k \in \{i, j\}$, ist daher $\frac{2}{j-i+1}$, sprich $p_{ij} = \frac{2}{j-i+1}$.

Somit erhalten wir mit (1) für die erwartete Laufzeit ($\hat{=}$ Anzahl von Vergleichen)

$$\sum_{i=1}^{n} \sum_{j>i} p_{ij} = \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \leq \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \leq 2 \cdot \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} = 2n \cdot H_n,$$

wobei $H_n := \sum_{k=1}^{n} \frac{1}{k}$ die *n-te harmonische Zahl* ist.

Wir haben also gezeigt:

Satz 1.5 *Die erwartete Anzahl von Vergleichen bei einer Ausführung von randomisiertem Quicksort ist höchstens* $2n \cdot H_n$ $(= O(n \cdot \log(n)))$.

Bemerkung 1.6

- Randomisiertes Quicksort ist ein Beispiel für das Paradigma *Zufallsstichproben.*
- Satz 1.5 gilt für jede Eingabe. Es werden keine Annahmen über die Eingabeverteilung gemacht. Die Laufzeit des Algorithmus ist eine Zufallsvariable. Randomisiertes Quicksort ist also ein *Las-Vegas-Algorithmus*, denn er liefert immer ein korrektes Ergebnis, nur die Laufzeit ist eine Zufallsvariable.
- Es läßt sich sogar zeigen, daß der Algorithmus „mit sehr hoher Wahrscheinlichkeit“ nicht viel mehr als die erwartete Laufzeit benötigt.
- $H_n = \ln(n) + \Theta(1)$.

1.3.3 Spielbaumauswertung

Gegeben: Ein vollständiger Binärbaum T_k, bei welchem alle Blätter eine Distanz von $2k$ von der Wurzel haben. Innere Knoten mit ungerader Distanz sind mit ODER beschriftet, solche mit gerader Distanz tragen die Aufschrift UND. Jedes Blatt hat einen Booleschen Wert, „wahr" oder „falsch".
Frage: Wieviele Blätter muß ein Algorithmus zur Bestimmung des Wertes der Wurzel bei einer beliebig vorgegebenen Eingabe ansehen.

Betrachte folgenden randomisierten, rekursiven Algorithmus zur Ermittlung des Wertes des Wurzelknotens. Dabei repräsentieren 0 und 1 wie üblich die Booleschen Werte falsch und wahr.

1. Auswertung eines UND-Knoten:
   (a) Wähle per Zufall eines der beiden Kinder und werte rekursiv dieses Kind aus.
   (b) Falls das ausgewertete Kind den Wert 1 hat, so werte auch das zweite Kind aus. Falls es aber den Wert 0 hat, so gib 0 zurück.
2. Auswertung eines ODER-Knoten analog mit vertauschten Rollen von 0 und 1.

Satz 1.7 *Die erwartete Anzahl von Blättern, die der Algorithmus zur Bestimmung des Wertes der Wurzel eines Baumes T_k betrachten muß, ist höchstens 3^k.*

Beweis: Induktion über k. Der Fall $k = 0$ ist trivial.
Induktionsannahme: Für die Auswertung von T_{k-1} werden höchstens 3^{k-1} Blätter benötigt.

Betrachte zunächst einen Baum T mit Wurzelknoten ODER und zwei T_{k-1}-Unterbäumen. Hier gilt: Falls die Wurzel von T den Wert 1 hat, so muß mindestens eines seiner beiden Kinder den Wert 1 haben. Mit Wahrscheinlichkeit $\frac{1}{2}$ wird zunächst dieses Kind gewählt, wodurch per Induktionsannahme folgt, daß in diesem Falle höchstens 3^{k-1} Blätter betrachtet werden müssen. Mit Wahrscheinlichkeit $\frac{1}{2}$ müssen aber beide Unterbäume ausgewertet werden, also $2 \cdot 3^{k-1}$ Blätter betrachtet werden. Zusammen ergibt das erwartete Kosten von höchstens

$$\frac{1}{2} \cdot 3^{k-1} + \frac{1}{2} \cdot 2 \cdot 3^{k-1} = \frac{3}{2} \cdot 3^{k-1}.$$

Falls aber die Wurzel von T den Wert 0 hat, so müssen beide Unterbäume ausgewertet, sprich höchstens $2 \cdot 3^{k-1}$ Blätter betrachtet werden.

Betrachte nun die Wurzel eines T_k-Baumes, einen UND-Knoten. Falls die Wurzel den Wert 1 hat, so müssen beide Unterbäume den Wert 1 ergeben. Aufgrund der obigen Vorbetrachtung für T und unter Ausnutzung der *Linearität des Erwartungswertes* ergibt sich hiermit für die erwarteten Kosten eine obere Schranke von

$$2 \cdot \frac{3}{2} \cdot 3^{k-1} = 3^k.$$

Falls die Wurzel den Wert 0 hat, so muß mindestens einer ihrer Unterbäume den Wert 0 haben. Mit Wahrscheinlichkeit $\frac{1}{2}$ wird dieser zuerst gewählt, also ergibt sich für die erwarteten Auswertungskosten wiederum eine obere Schranke von

$$\frac{1}{2} \cdot 2 \cdot 3^{k-1} + \frac{1}{2} \cdot \left(\frac{3}{2} \cdot 3^{k-1} + 2 \cdot 3^{k-1} \right) \leq 3^k.$$

□

Wegen $n = 4^k$ für die Gesamtblattanzahl eines T_k-Baumes erhalten wir sofort:

Korollar 1.8 *Die erwartete Anzahl von betrachteten Blättern ist* $n^{\log_4 3} \approx n^{0.793}$*, wobei* n *die Gesamtblattzahl des Baumes* T_k *sei.*

Bemerkung 1.9

- Die Spielbaumauswertung ist ein Beispiel für das Paradigma *Vereitelung der Angriffe eines Gegners.*
- Der Algorithmus ist ein *Las-Vegas-Algorithmus*: Er gibt *immer* die korrekte Antwort, nur die Laufzeit hängt vom Zufall ab.
- Es ist nicht schwer zu zeigen, daß für *jeden* deterministischen Algorithmus eine Instanz von T_k existiert, so daß *alle* n Blätter betrachtet werden müssen. Der randomisierte Algorithmus ist in diesem Sinne also besser als jeder deterministische Algorithmus!
- Die Randomisierung läßt sich auch so verstehen, als ob per Zufall ein Algorithmus aus einer Familie deterministischer Algorithmen gewählt würde.
- Es läßt sich zeigen, daß obiger Algorithmus optimal ist, sprich *jeder* randomisierte Algorithmus muß im Durchschnitt $n^{0.793}$ Blätter auswerten. (M. Saks, A. Wigderson: Probabilistic Boolean decision trees and the complexity of evaluating game trees, 27. *IEEE Symposium on Foundations of Computer Science*, 29–38, 1986).

1.3.4 2SAT (Random Walks)

Gegeben: Eine aussagenlogische Formel F in 2KNF, das heißt eine Konjunktion von Disjunktionen der Länge 2.
Frage: Besitzt F eine erfüllende Belegung?

Wir betrachten folgenden randomisierten Algorithmus:

1. Starte mit einer beliebigen Belegung T der Variablen von F.
2. Wiederhole r mal:
   (a) Falls jede Klausel unter der gegebenen Belegung den Wert „wahr" hat, so gib aus „Formel ist erfüllbar" und halte.
   (b) Andernfalls nimm irgendeine Klausel mit Wert „falsch" (all ihre Literale sind unter der Belegung falsch).
   Wähle per Zufall eines der Literale der Klausel und modifiziere die gegebene Belegung so, daß das Literal nun wahr wird („Flipping-Schritt").
3. Gib aus „die Formel ist wahrscheinlich unerfüllbar" und halte.

Satz 1.10 *Für eine erfüllbare 2KNF-Formel mit n Variablen findet obiger Algorithmus mit $r := 2n^2$ Wiederholungen mit Mindestwahrscheinlichkeit $\frac{1}{2}$ eine erfüllende Belegung.*

Beweis: Sei $\hat{T}$ eine erfüllende Belegung für die gegebene Formel und bezeichne $t(i)$ die erwartete Anzahl von Wiederholungen des Flipping-Schrittes bis zum Finden einer erfüllenden Belegung unter der Annahme, daß unsere Startbelegung T sich in genau i Werten von $\hat{T}$ unterscheidet. Es gilt:

$$\begin{aligned} t(0) &= 0, \\ t(i) &\leq \frac{1}{2} \cdot (t(i-1) + t(i+1)) + 1, \\ t(n) &\leq t(n-1) + 1. \end{aligned}$$

Wir schätzen nach oben ab, indem wir Ungleichheitszeichen durch Gleichheitszeichen ersetzen. Dies führt zu:

$$\begin{aligned} x(0) &= 0, \\ x(i) &= \frac{1}{2} \cdot (x(i-1) + x(i+1)) + 1, \\ x(n) &= x(n-1) + 1, \end{aligned}$$

mit $x(i) \geq t(i)\ \forall i = 0, 1, \ldots, n$.
Die Werte der $x(i)$ lassen sich aus dem gegebenen linearen Gleichungssystem bestimmen. Wir erhalten dabei $x(1) = 2n - 1$, $x(2) = 4n - 4$ und allgemein $x(i) = 2in - i^2$, also insbesondere $x(n) = n^2$. Das heißt: $t(i) \leq x(i) \leq x(n) = n^2$, sprich die erwartete Anzahl von Schritten ist höchstens n^2. Der Beweis ergibt sich somit aus folgendem nützlichen Lemma mit der Wahl $k = 2$. □

Lemma 1.11 *(Markov-Ungleichung)*
Sei X eine Zufallsvariable mit nichtnegativen, ganzzahligen Werten. Dann gilt $\forall k > 0 : \mathbf{Pr}[X \geq k \cdot E[X]] \leq \frac{1}{k}$ *(wobei E[X] den Erwartungswert von X bezeichne).*

Beweis:

$$E[X] = \sum_i i \cdot \mathbf{Pr}[X = i] = \sum_{i < k \cdot E[X]} i \cdot \mathbf{Pr}[X = i] + \sum_{i \geq k \cdot E[X]} i \cdot \mathbf{Pr}[X = i]$$
$$\geq k \cdot E[X] \cdot \mathbf{Pr}[X \geq k \cdot E[X]].$$

□

Bemerkung 1.12

- Der betrachtete Algorithmus fällt unter das Paradigma *Markov-Ketten*. Es handelt sich um einen eindimensionalen *Random Walk* mit einem reflektierenden und einem absorbierenden Ende.
- Der Algorithmus ist vom Typ Monte-Carlo.
- Es gibt 3KNF-Formeln, für die der Random-Walk-Algorithmus sehr schlecht arbeitet, insbesondere keine erwartete polynomielle Laufzeit erzielt.
- Literatur: C. H. Papadimitriou: On selecting a satisfying truth assignment. *In Proceedings 32. IEEE Symposium on Foundations of Computer Science*, 163-169, 1991.

1.3.5 Das Problem der speisenden Philosophen

Das Problem der speisenden Philosophen ist ein klassisches Problem der Ressourcen-Allokierung.

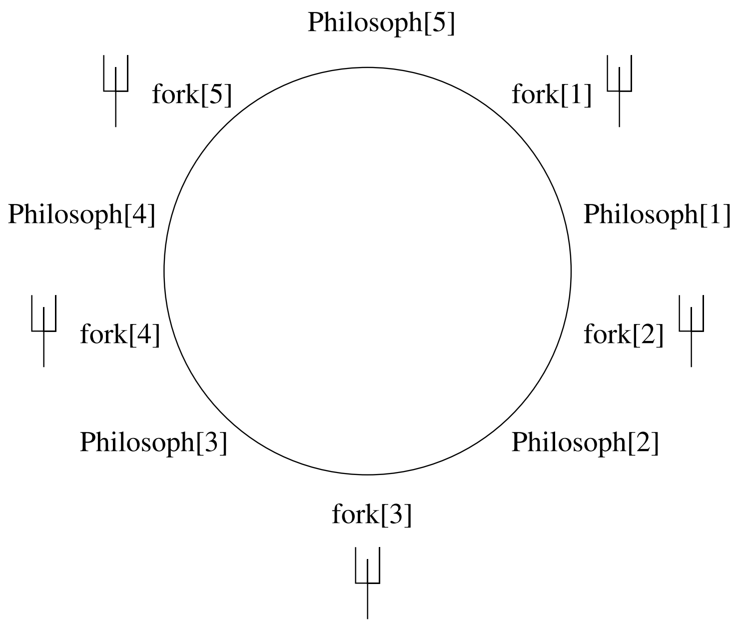
Gegeben: n in einem Kreis angeordnete Philosophen mit einer Gabel zwischen je einem Paar von benachbarten Philosophen. Um zu essen, muß ein Philosoph *beide* zu ihm benachbarten Gabeln haben.
Problem: Gesucht ist ein *deadlock-freies* (falls es irgendwann einen hungrigen Philosophen gibt, dann wird irgendwann schließlich auch einer essen) und *lockout-freies* (jeder hungrige Philosoph bekommt irgendwann etwas zu essen) Protokoll.

Für das Problem gibt es viele deterministische Lösungen basierend auf gemeinsamem Speicher oder Kommunikation durch Message-Passing. Keine dieser Lösungen ist aber

1. sowohl *voll verteilt*, das heißt ohne zentralen Speicher oder zentralen Prozeß, auf den jeder andere Prozeß zugreifen kann,
2. als auch *symmetrisch*, das heißt alle Prozesse (Philosophen) führen das selbe Programm aus, und alle lokalen und gemeinsamen Variablen sind identisch initialisiert; zudem kennen die Prozesse nicht ihre Identität, sprich können Prozeßnummern nicht vergleichen.

Lehmann und Rabin zeigten 1981, daß es keine voll verteilte und symmetrische, deterministische Lösung für das Problem geben kann. Alternativ gaben sie folgendes randomisiertes Protokoll an, welches deadlock-frei (aber noch nicht lockout-frei) ist. Deadlock-frei bedeutet in diesem Zusammenhang, daß bei Existenz eines hungrigen Philosophen mit Wahrscheinlichkeit 1 ein Philosoph schließlich essen wird.
Für das Protokoll benutzen wir für jedes Paar benachbarter Philosophen eine gemeinsame Variable fork[i], welche besagt, ob die jeweilige Gabel frei ist.

Beispiel 1.13 Für $n = 5$ ergibt sich folgendes Bild:



Protokoll für jeden Philosophen:

Wiederhole für immer folgendes:

1. Denke bis hungrig.
2. Wirf eine Münze (Zufallsexperiment) um zu entscheiden, ob zuerst die linke oder die rechte Gabel aufgenommen werden soll.
3. Warte, bis die gewählte Gabel frei ist, und nimm sie auf. (Atomare Operation!)
4. Falls die andere Gabel nicht frei ist, so lege die gewählte Gabel wieder zurück und gehe nach (2).
5. Andernfalls (das heißt, die zweite Gabel ist auch frei) nimm auch die andere Gabel auf.
6. Kritischer Bereich: Iß.
7. Lege beide Gabeln nieder.

Der Schlüssel zur Lösung des Problems ist also die randomisierte „Gabelaufnahmereihenfolge“.

Bemerkung 1.14

- Die speisenden Philosophen sind ein Beispiel für das Paradigma *Aufbrechen von Symmetrien*
- Es gibt auch zusätzlich lockout-freie Lösungen. Somit sind die speisenden Philosophen ein Problem, für das es *nachweislich* keine deterministische Lösung (unter den angegebenen Bedingungen) gibt, jedoch eine randomisierte.
- Literatur: D. Lehmann, M. O. Rabin: On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem. *In Proc. of the 8th Annual ACM Symposium on the Principles of Programming Languages*, ACM, New York, 133-138 (1981).

2 Ein Primzahltest

Mitte der 70er Jahre wurden randomisierte Primzahltests unabhängig von Solovay/Strassen und von Miller/Rabin entdeckt. Mit diesen Entdeckungen begann der Aufstieg randomisierter Algorithmen im Bereich der Algorithmik und der Komplexitätstheorie. Primzahltests sind von besonderer Bedeutung in der Kryptographie. Im Gegensatz zum Testen auf die Primzahleigenschaft ist kein effizienter (= Polynomzeit) deterministischer oder randomisierter Algorithmus für das Zerlegen einer ganzen Zahl in ihre Primfaktoren bekannt. Hierauf beruht auch die Sicherheit des RSA Kryptosystems.

2.1 Einige zahlentheoretische Grundlagen

Mitteilung 2.1 *Eine zusammengesetzte ganze Zahl a kann auf genau eine Weise als Produkt ihrer Primfaktoren in der Form $a = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_r^{e_r}$ mit $p_1 < p_2 < \ldots < p_r$, $e_i > 0$ geschrieben werden.*

Restklassen-Arithmetik: Wir interessieren uns für zwei endliche abelsche (kommutative) Gruppen — die *additive Gruppe modulo n*, das heißt ($\mathbb{Z}_n$, $+_n$), und die *multiplikative Gruppe modulo n*, das heißt ($\mathbb{Z}_n^*$, $\cdot_n$), wobei $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ und $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \mathrm{ggT}(a, n) = 1\}$.
Beispiel: $\mathbb{Z}_{15}^* := \{1, 2, 4, 7, 8, 11, 13, 14\}$, $8 \cdot_{15} 7 = 11$, $3 +_6 5 = 2$.
Falls p eine Primzahl ist, so ist $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$. Wichtige Grundlage und Ausgangspunkt für das randomisierte Primzahltesten ist folgender Satz von Fermat aus dem Jahre 1640.

Mitteilung 2.2 *(Kleiner Satz von Fermat)*
Falls p prim ist, so gilt für alle $a \in \mathbb{Z}_p^*$: $a^{p-1} \equiv 1 \pmod p$.

Beispiel: $5^6 = 15625 = 2232 \cdot 7 + 1 \equiv 1 \pmod 7$

Gemäß dem kleinen Satz von Fermat bezeugt ein $a \in \mathbb{Z}_n^*$ mit $a^{n-1} \not\equiv 1 \pmod n$, daß n nicht prim ist. Eine analoge Zeugenaussage liefert der folgende Satz. Zusammen mit dem kleinen Satz von Fermat läßt sich daraus dann der Miller/Rabin-Test konstruieren.

Mitteilung 2.3 *Falls $x \not\equiv \pm 1 \pmod n$ und $x^2 \equiv 1 \pmod n$ (in Worten: x ist nichttriviale Quadratwurzel von 1 modulo n), dann ist n nicht prim.*

Für die Anwendung des kleinen Satzes von Fermat bedarf es eines effizienten Algorithmus für die *modulare Exponentiation.* Die zugrundeliegende Technik heißt *wiederholtes Quadrieren* (repeated squaring). Dabei werden folgende Gleichheiten ausgenutzt:

$$a^{2\cdot c} \pmod n = (a^c)^2 \pmod n \ = ((a^c) \pmod n)((a^c) \pmod n) \pmod n$$
$$\text{bzw. } a^{2\cdot c+1} \pmod n = a \cdot (a^c)^2 \pmod n$$

Durch Ausnutzung der genannten Gleichheiten läßt sich insbesondere das Entstehen zu großer Zwischenwerte vermeiden. Zur Berechnung von $a^b \pmod n$ ergibt sich somit folgender Algorithmus, wobei $< b_k, b_{k-1}, ..., b_1, b_0 >$ die Binärdarstellung von b sei (mit höchstwertigem Bit b_k).

Modular-Exponentiation (a, b, n):
$d := 1$
for $i = k$ **downto** 0 **do**
  begin
  $d := (d \cdot d) \bmod n$
  if $b_i = 1$ **then**
    $d := (d \cdot a) \bmod n$
  end
return d

Für viele Anwendungen (wie Kryptographie) müssen wir große „Zufallsprimzahlen“ finden. Über die Dichte von Primzahlen gibt folgender bedeutende Satz Auskunft, wobei $\pi(n) :=$ „Anzahl der Primzahlen $\leq n$“.

Mitteilung 2.4 *(Primzahlsatz)*

$$lim_{n\to\infty} \pi(n) / \left(\frac{n}{\ln(n)}\right) = 1.$$

Aus dem Primzahlsatz folgt, daß die Wahrscheinlichkeit, daß eine zufällig gewählte Zahl zwischen 1 und n eine Primzahl ist, ungefähr gleich $1/\ln(n)$ ist. Um eine 100-stellige Primzahl zu gewinnen, muß man also im Durchschnitt etwa ln $(10^{100}) \approx 230$ zufällige Zahlen testen. Für das Erzeugen großer, zufälliger Primzahlen (wie man sie zum Beispiel beim RSA Kryptosystem benötigt) bedarf es somit noch eines effizienten Primzahltests. Dies ist Gegenstand des nächsten Abschnitts.

2.2 Der Miller/Rabin-Test

Naiver Ansatz: „Teste für alle ungeraden Zahlen 3, 5, ..., $\lfloor\sqrt{n}\rfloor$, ob sie n teilen". Diese Methode benötigt im Worst-Case $\Theta(\sqrt{n})$ Schritte, sie ist also exponentiell (!) in der Eingabegröße, sprich bezogen auf die Binärdarstellung von n.

Die grundlegende Idee für randomisierte Primzahltests ist es, für gegebenes n geeignete *„Zeugen"* zu finden, die belegen, daß n nicht prim ist. Diese Zeugen sollen schnell zu überprüfen sein und es soll viele von ihnen geben, das heißt wenigstens die Hälfte aller Zahlen zwischen 1 und $n-1$ sollen Zeugen sein.

2.2.1 Schema für einen randomisierten Primzahltest

Sowohl der Primzahltest von Solovay/Strassen als auch der von Miller/Rabin folgen einem bestimmten Schema (siehe Abbildung 1). Sie unterscheiden sich lediglich in der zu testenden Zeugeneigenschaft.

Bei dem Schema von Abbildung 1 ist das Ergebnis „n ist keine Primzahl" immer wahr, das Ergebnis „n ist eine Primzahl" kann (mit geringer Wahrscheinlichkeit, zum Beispiel 2^{-100}) falsch sein. Es verbleibt also, eine schnell zu testende Zeugeneigenschaft mit den gewünschten Eigenschaften zu finden. Dabei sind der kleine Satz von Fermat (Mitteilung 2.2) und Mitteilung 2.3 von großer Hilfe.

2.2.2 Eine Methode, die „fast" funktioniert

Der kleine Satz von Fermat besagt, daß wenn n prim ist, so gilt $a^{n-1} \equiv 1 \pmod{n}$ für alle $a \in \{1, \ldots, n-1\}$. Überraschenderweise gilt für die zusammengesetzte Zahl n und ein zufällig gewähltes $a \in \{2, \ldots, n-1\}$ auch „fast immer" $a^{n-1} \not\equiv 1 \pmod{n}$, so daß dies ein fast perfektes Kriterium (Zeuge) für Primzahltests bildet:

Pseudoprim(n):
$a :=$ Zufallszahl zwischen 2 und $n-1$;
if Modular-Exponentiation$(a, n-1, n) \not\equiv 1 \pmod{n}$
  then return „zusammengesetzt"
else return „prim"

Wann gibt Pseudoprim(n) falsche Antworten? Die Antwort „zusammengesetzt" ist immer korrekt. Die Antwort „prim" kann falsch sein. Wie häufig?

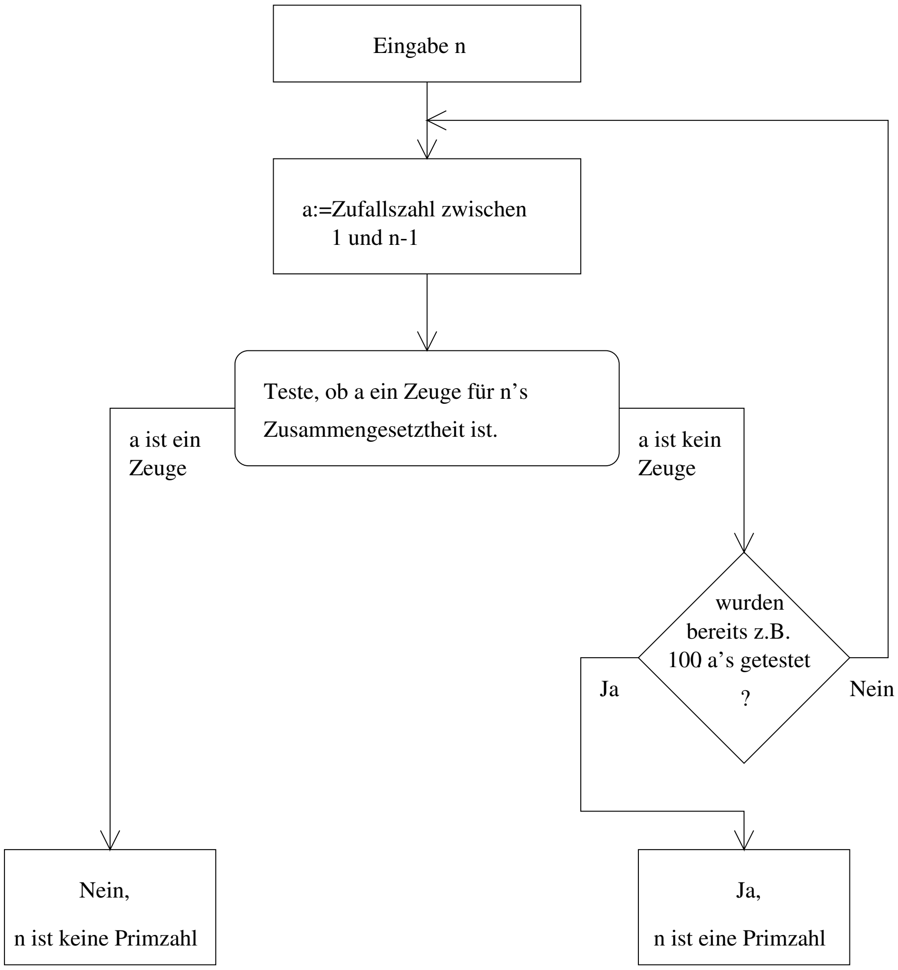


Abbildung 1: Schema für einen randomisierten Primzahltest.

Leider existieren zusammengesetzte Zahlen n (die sogenannten *Carmichael-Zahlen*), für die $a^{n-1} \equiv 1 \pmod{n}$ für alle $a \in \mathbb{Z}_n^*$ gilt.[1] Die ersten drei Carmichael-Zahlen sind 561, 1105 und 1729. So liefert Pseudoprim(n) für eine Carmichael-Zahl n „mit sehr hoher Wahrscheinlichkeit" das falsche Ergebnis. Carmichael-Zahlen sind jedoch extrem selten: Nur 255 von ihnen sind beispielsweise kleiner als 10^8. Der Miller/Rabin-Test löst nun auch noch das Problem der Carmichael-Zahlen.

[1] Zu beachten: $\mathbb{Z}_n^* \subsetneq \mathbb{Z}_n$ falls n keine Primzahl ist.

2.2.3 Der Miller/Rabin-Test

Statt dem Test „$a^{n-1} \not\equiv 1 \pmod{n}$" benutzt der Miller/Rabin-Test eine erweiterte Zeugeneigenschaft, die aus der Kombination von Mitteilung 2.2 und Mitteilung 2.3 entsteht. Wesentliche Neuerung gegenüber dem Algorithmus Pseudoprim(n) ist, daß während der modularen Exponentiation zusätzlich überprüft wird, ob nichttriviale Quadratwurzeln von 1 modulo n entstehen (vergleiche Mitteilung 2.3). Somit erhalten wir folgendes Zeugenkriterium, wobei wir ohne Einschränkung annehmen, daß die Eingabe n ungerade ist:

Zeuge(a, n):
{Sei $< b_k, .., b_0 >$ die Binärdarstellung von $n - 1$}
$d := 1$;
for $i = k$ **downto** 0 **do**
  begin
  $x := d$;
  $d := (d \cdot d)$ **mod** n;
  if $d = 1$ **and** $x \neq 1$ **and** $x \neq n - 1$
    then return *true* ;
  if $b_i = 1$
    then $d := (d \cdot a)$ **mod** n;
  end
if $d \neq 1$ **then return** *true* ;
return *false*

Es empfiehlt sich, Zeuge(2,27) und Zeuge(7,561) einmal selbst durchzurechnen. (Zur Erinnerung: 561 ist eine Carmichael-Zahl.) Es zeigt sich, daß für $n = 27$ das Zeugenkriterium basierend auf der Existenz nichttrivialer Quadratwurzeln (Mitteilung 2.3) nicht greift (in der Tat greift es hier für kein $a < n$) und für $n = 561$ das Zeugenkriterium basierend auf dem Kleinen Satz von Fermat nicht greift (in der Tat greift es hier für kein $a \in \mathbb{Z}^*_{561}$, also mit sehr hoher Wahrscheinlichkeit für zufällig gewählte a nicht).

Proposition 2.5 *Falls Zeuge*(a, n) *für ein* $a \in \{1, \ldots, n-1\}$ *den Wert true zurückgibt, so ist* n *nicht prim.*

Beweis: Die Aussage folgt unmittelbar aus dem Kleinen Satz von Fermat (Mitteilung 2.2) und Mitteilung 2.3.

□

Folgender gewichtige Satz ermöglicht schließlich den Miller/Rabin-Test. Er zeigt, daß die Kombination beider Zeugenkriterien (Mitteilung 2.2 und Mitteilung 2.3) das gewünschte leistet, beide für sich genommen aber zu wenig sind.

Mitteilung 2.6 *Falls n eine ungerade, zusammengesetzte Zahl ist, so gibt es mindestens $(n-1)/2$ Zahlen $a \in \{1, .., n-1\}$, für die Zeuge(a,n) den Wert true liefert.*

Miller/Rabin(n, s):
for $j = 1$ **to** s **do**
  begin
  $a :=$ Zufallszahl zwischen 1 und $n-1$;
  if Zeuge(a, n)
    then return „zusammengesetzt";
  end
return „prim"

Der Miller/Rabin-Algorithmus benötigt $O(s \cdot \log n)$ arithmetische Operationen, ist also bis auf den Faktor s linear in der Eingabelänge.

Satz 2.7 *Für jedes ungerade $n > 2$ und positives s ist die Fehlerwahrscheinlichkeit von Miller/Rabin(n, s) höchstens 2^{-s}.*

Beweis: Aus Mitteilung 2.6 folgt, daß im Rumpf der for-Schleife bei Aufruf von Miller/Rabin(n, s) mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ ein Zeuge gefunden wird, falls n zusammengesetzt ist. Miller/Rabin(n, s) irrt sich nur, falls er es in jeder Iteration verpaßt, einen Zeugen für die Zusammengesetztheit von n zu finden. Diese Wahrscheinlichkeit ist aber durch 2^{-s} beschränkt. □

Bemerkung 2.8

- Der vorangehende Teil dieses Kapitels folgte (in stark gekürzter Form) dem Kapitel 33 („Number-Theoretic Algorithms") im Buch von Cormen /Leiserson /Rivest: „Introduction to Algorithms", The MIT Press 1994. Dort finden sich insbesondere auch Beweise der meisten zahlentheoretischen Grundlagen.
- Der Miller/Rabin-Test ist ein Beispiel für das Paradigma *Überfluß an Zeugen*.
- Der Miller/Rabin-Test ist ein *Monte-Carlo-Algorithmus* mit einseitigem Fehler.

- Ein *alternativer randomisierter Primzahltest* ist Solovay und Strassen zu verdanken: A fast Monte-Carlo test for primality, *SIAM Journal on Computing*, Vol. 7(1), 84-85, 1977.

- Der schnellste bekannte *deterministische Primzahltest* hat superpolynomielle Laufzeit $(\log n)^{O(\log\log\log n)}$. Siehe L. M. Adleman, C. Pomerance, and R. S. Rumley: On distinguishing prime numbers from composite numbers, *Annals of Mathematics*, Vol. 117, 173-206, 1983 und H. Cohen and H. W. Lenstra: Primality testing and Jacobi sums, *Mathematics of Computation*, Vol. 42(165), 297-330, 1984.

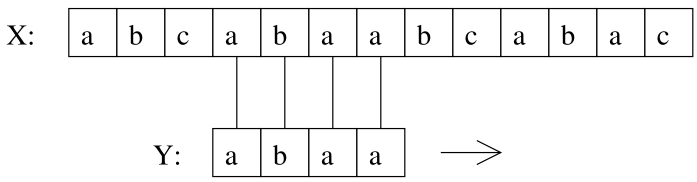
- Der Miller/Rabin-Test ist verwandt mit einem *deterministischen Primzahltest*, welcher unter Annahme, der Gültigkeit der *erweiterten Riemann-Hypothese* sogar polynomielle Laufzeit hat. Siehe G. L. Miller: Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, Vol. 13(3), 300-317, 1976.

- Einen Überblick über (offene) Fragen der *algorithmischen Zahlentheorie* bietet L. M. Adleman: Algorithmic Number Theory - The Complexity Contribution. *In Proceedings 35. IEEE Symposium on Foundations of Computer Science*, 88-113, 1994.

2.3 Anwendung: Pattern Matching

Gegeben: Eine Zeichenkette $X = x_1x_2...x_n$ und ein „Pattern" $Y = y_1y_2...y_m$, beide über einem endlichen Alphabet Σ und $m \leq n$.
Frage: Kommt Y in X als Teilwort vor?

Beispiel 2.9



Der naive Algorithmus („brute force") benötigt $O((n-m+1)m)$ Schritte. Der nachfolgend präsentierte randomisierte Algorithmus von Karp/Rabin folgt dem naiven Ansatz insofern, als daß auch das Muster Y von links nach rechts durch X geschoben wird. Anstelle des direkten Vergleichs von Y mit Teilwörtern von X tritt nun jedoch der Vergleich der jeweiligen „Fingerabdrücke".

Definiere $X(j) := x_j ... x_{j+m-1}$. Zur Vereinfachung sei $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Auf diese Weise lassen sich X, $X(j)$ und Y als natürliche Zahlen interpretieren. Die *Fingerabdrucksfunktion* ist nun

$$F_p : \mathbb{Z} \to \mathbb{Z}_p : F_p(z) := z \pmod p,$$

wobei p eine Primzahl sei.

Der Algorithmus von Karp/Rabin arbeitet wie folgt:

$p :=$ zufällige Primzahl zwischen 2 und $mn^2 \log(mn^2)$;
Match:= *false* ;
$i := 1$;
while Match = *false* **and** $1 \leq i \leq n - m + 1$ **do**
  begin
  if $F_p(X(i)) = F_p(Y)$
    then Match := *true* ;
    else
      begin
      $i := i + 1$;
      berechne $F_p(X(i+1))$ aus $F_p(X(i))$
      end
  end
return Match

Der Schlüssel für die Effizienz des Algorithmus von Karp/Rabin liegt in der einfachen Berechenbarkeit von $F_p(X(i+1))$ aus $F_p(X(i))$:

$$F_p(X(i+1)) = 10 \cdot (F_p(X(i)) - 10^{m-1} \cdot x_i) + x_{i+m} \pmod p$$

$F_p(X(i+1))$ kann also mit konstant vielen arithmetischen Operationen modulo p aus $F_p(X(i))$ berechnet werden, falls $10^{m-1} \pmod p$ schon einmalig vorausberechnet wurde.

Beispiel 2.10

X:	0	2	3	1	4	1	5	2	6	7	3	9	9	2

Y:	3	1	4	1	5

$n = 14$, $m = 5$, $p = 13$:
$F_{13}(Y) = 31415 \pmod{13} = 7$,

$F_{13}(X(1)) = F_{13}(02314) = 0,$
$F_{13}(X(2)) = F_{13}(23141) = 10 \cdot (0 - 10^4 \cdot 0) + 1 \pmod{13} = 1,$
$F_{13}(X(3)) = F_{13}(31415) = 10 \cdot (1 - 10^4 \cdot 2) + 5 \pmod{13} = 7,$ Match!
...
$F_{13}(X(9)) = F_{13}(67399) = 7.$ falscher Match!

Satz 2.11 *Der Karp/Rabin Algorithmus benötigt $O(n+m)$ Schritte und hat eine Fehlerwahrscheinlichkeit von $O(\frac{1}{n})$.*

Beweis: Da $F_p(X(i+1))$ aus $F_p(X(i))$ mit $O(1)$ Operationen berechnet werden kann, folgt die Laufzeitschranke von $O(n+m)$ unmittelbar. Die bedingte Wahrscheinlichkeit $\mathbf{Pr}[F_p(Y) = F_p(X(i)) \mid Y \neq X(i)]$ für einen falschen Match an der Stelle i ergibt sich folgendermaßen: Sei $a := F_p(Y)$ und $b := F_p(X(i))$. Dann gilt $a \pmod{p} \equiv b \pmod{p}$ genau dann, wenn p die Zahl $|a-b|$ teilt. Wir wissen, daß $|a-b| \leq 10^m < 2^{4m}$. Offensichtlich hat eine Zahl kleiner als 2^{4m} höchstens $4m$ verschiedene Primfaktoren. Der Algorithmus wählt für F_p eine zufällige Primzahl p aus dem Bereich zwischen 1 und $mn^2\log(mn^2)$. Nach Mitteilung 2.4 (Primzahlsatz) gibt es davon

$$O\left(\frac{mn^2\log(mn^2)}{\ln(mn^2\log(mn^2))}\right) = O(mn^2).$$

viele. Also folgt für festes i, daß

$$\mathbf{Pr}[F_p(Y) = F_p(X(i)) \mid Y \neq X(i)] = O\left(\frac{4m}{mn^2}\right) = O\left(\frac{1}{n^2}\right).$$

Für die Gesamtfehlerwahrscheinlichkeit ergibt sich somit eine obere Schranke von

$$(n-m+1) \cdot O\left(\frac{1}{n^2}\right) = O\left(\frac{1}{n}\right).$$

□

Der Karp/Rabin-Algorithmus ist vom Typ *Monte-Carlo.* Es gibt zwei Möglichkeiten, daraus einen Las-Vegas-Algorithmus zu konstruieren:

Möglichkeit 1:
Falls im Algorithmus $F_p(X(i)) = F_p(Y)$ festgestellt wird, so überprüfe in $O(n)$ Schritten in direktem Vergleich, ob $X(i) = Y$ gilt. Falls nein, so starte den naiven $O(nm)$-Algorithmus.

Möglichkeit 2:
Analog Möglichkeit 1, nur anstelle des naiven $O(nm)$-Algorithmus starte nun wiederum den Karp/Rabin-Algorithmus mit einer neuen Zufallswahl für p.

Offensichtlich führen beide Varianten zu *Las-Vegas-Algorithmen*, also Algorithmen die keine Fehler machen. Möglichkeit 1 führt zu einem Algorithmus mit erwarteter Laufzeit

$$O\left((n+m)\left(1-\frac{1}{n}\right)\right)+O\left(nm\left(\frac{1}{n}\right)\right)=O(n+m).$$

Die Wahrscheinlichkeit, den $O(nm)$-Algorithmus starten zu müssen, ist hierbei aber relativ hoch. Wir haben also eine hohe *Varianz* der Laufzeit.
Bei Variante 2 hingegen ist die Wahrscheinlichkeit, den Karp/Rabin-Algorithmus t mal neu starten zu müssen durch $O(n^{-t})$ beschränkt. Dies führt zu einer kleineren Varianz der Laufzeit.

Bemerkung 2.12

- Der Karp/Rabin-Algorithmus ist ein Beispiel für das Paradigma *Signaturen (Fingerabdrücke)*.
- Für das Pattern-Matching-Problem gibt es auch *deterministische* $O(n+m)$-Algorithmen („Knuth/Morris/Pratt" und „Boyer/Moore"), die aber deutlich komplizierter sind und im allgemeinen höhere konstante Faktoren haben und mehr Laufzeit/Speicherplatz benötigen.
- Der Karp/Rabin Algorithmus läßt sich auch leicht auf das *zweidimensionale* Pattern-Matching-Problem verallgemeinern.
- Ein weiterer Vorteil des Karp/Rabin-Algorithmus ist, daß er auch eingesetzt werden kann, wenn die Zeichenkette X nur Schritt für Schritt („inkrementell") gegeben wird. Er arbeitet also auch in „Online"- oder „Realzeit"-Umgebungen. Dies ist auch nützlich, wenn X aufgrund seiner Größe nicht in den Hauptspeicher paßt.
- Literatur: R. M. Karp and M. O. Rabin: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, Vol. 31, 249-260, March 1987.

3 Komplexitätsklassen und Grenzen der Randomisierung

In den bisherigen Kapiteln betrachteten wir konkrete algorithmische Probleme und gaben für die vorgestellten Lösungsverfahren mehr oder minder exakt die erwarteten Laufzeiten an. Dabei legten wir implizit die RAM (Random Access Machine) als Maschinenmodell zugrunde. Wir benutzten das *uniforme Kostenmaß*, welches besagt, daß jede Instruktion (Eingabe, Ausgabe, arithmetische Operationen, ...) unter Benutzung von Registern etc. nur einen Rechenschritt koste. In diesem Kapitel erhöhen wir das Abstraktionsniveau — anstelle von einzelnen Problemen betrachten wir Mengen von Problemen (*Komplexitätsklassen*). Um zu allgemeinen und weitreichenderen Aussagen zu kommen, bedarf es gegenüber dem bisherigen Vorgehen zweier Vereinfachungen:

- Statt Laufzeitangaben wie zum Beispiel $O(n^2)$ zu betrachten, beschränken wir uns auf das gröbere Maß Polynomzeit ($n^{O(1)}$).
- Statt RAM's benutzen wir die vom Modellcharakter her deutlich einfacheren (probablistischen) Turingmaschinen.

Die Turingmaschine in ihren Grundzügen wird nachfolgend als bekannt vorausgesetzt (siehe Grundstudium, Informatik III (Theoretische Informatik)). In Anhang C werden einige grundlegende Definitionen und Ergebnisse aus der Komplexitätstheorie (siehe auch die entsprechende Vorlesung im Hauptstudium) bereitgestellt. Kapitel 3 sollte aber prinzipiell auch ohne deren Kenntnis verständlich sein.

Ziel dieses Kapitels ist es, grundlegende randomisierte Komplexitätsklassen vorzustellen, einige ihrer Eigenschaften zu analysieren und miteinander zu vergleichen, und insbesondere einen Blick auf die Einordnung des Konzeptes „Zufall“ zwischen die Konzepte „Determinismus“ und „Nichtdeterminismus“ zu werfen. Dabei ist zu beachten, daß randomisierte Polynomzeit-Algorithmen für RAM's immer in solche für Turingmaschinen, die ebenfalls Polynomzeit benötigen, umgewandelt werden können (und umgekehrt).

3.1 Definitionen

Wie in der Komplexitätstheorie üblich, beschränken wir uns auf *Entscheidungsprobleme*, das heißt, Komplexitätsklassen sind für uns immer Klassen von Sprachen.

Zur Erinnerung:

Definition 3.1

- *P* ist die Menge von Sprachen, die von *deterministischen* Turingmaschinen in Polynomzeit erkannt werden können.
- *NP* ist die Menge von Sprachen, die von *nichtdeterministischen* Turingmaschinen in Polynomzeit erkannt werden können.
- *PSPACE* ist die Menge von Sprachen, die von *deterministischen* Turingmaschinen mit polynomiellem Speicherplatz erkannt werden können.

Zur Vereinfachung (und ohne Einschränkung!) benutzen wir nachfolgend immer folgendes, *normalisiertes* Modell einer (nichtdeterministischen) Turingmaschine ((N)TM):

1. Jeder Schritt der NTM ist nichtdeterministisch,
2. jede Konfiguration hat genau zwei Nachfolgekonfigurationen und
3. für *jede* Eingabe der Länge n hat jeder Berechnungspfad der (N)TM die *gleiche* Länge $t(n)$.

Nachfolgend betrachten wir häufig den mit einer Berechnung einer normalisierten NTM korrespondierenden *Berechnungsbaum* der Tiefe $t(n)$ mit $2^{t(n)}$ Blättern. Dies ist ein vollständiger Binärbaum. Die Knoten sind die Konfigurationen der NTM. Mithilfe solcher Art normalisierter NTM's lassen sich nun sehr leicht *probabilistische Turingmaschinen* lediglich durch Änderung des *Akzeptanzmechanismus* definieren: Die nichtdeterministische Verzweigung wird nun als Münzwurf interpretiert, wobei je nach Ergebnis des Münzwurfs die Berechnung mit einer der beiden Nachfolgekonfigurationen fortgesetzt wird. In Abhängigkeit der Anteile der akzeptierenden und ablehnenden Blätter des Berechnungsbaumes lassen sich so verschiedene probabilistische Komplexitätsklassen definieren.

Definition 3.2 *RP* ist die Klasse von Sprachen, die von normalisierten nichtdeterministischen Turingmaschinen M in Polynomzeit erkannt werden können, für die folgendes gilt:

1. Für $x \in L$ sind mindestens die Hälfte der Blätter des Berechnungsbaumes von M akzeptierend.
2. Für $x \notin L$ sind alle Blätter im Berechnungsbaum von M ablehnend.

Beispiel: Die Sprache *2SAT* aller erfüllbaren aussagenlogischen Formeln in 2KNF ist in RP enthalten (vergleiche Abschnitt 1.3.4).

„RP-Algorithmen“ sind offensichtlich *Monte-Carlo-Algorithmen* mit einseitigem Fehler. Direkt aus der Definition von RP ergibt sich folgender Satz:

Satz 3.3 $P \subseteq RP \subseteq NP$.

Beweis: Sei $L \in P$ und sei M eine ohne Einschränkung normalisierte Turingmaschine, die das bezeugt. Gemäß der Definition von P und der Definition einer zu einer deterministischen Turingmaschine gehörigen normalisierten Turingmaschine, sind für $x \in L$ alle Blätter akzeptierend und für $x \notin L$ alle Blätter ablehnend. Damit sind die Kriterien aus Definition 3.2 erfüllt und es folgt $P \subseteq RP$.
Sei nun $L \in RP$ und sei M eine normalisierte nichtdeterministische Turingmaschine, die das bezeugt. Falls $x \in L$, so sind mindestens die Hälfte der Blätter von M akzeptierend (siehe Definition 3.2), insbesondere gibt es mindestens eine akzeptierende Berechnung. Falls $x \notin L$, so gibt es gemäß Definition 3.2 keine akzeptierende Berechnung. Somit akzeptiert M also L auch im üblichen nichtdeterministischen Sinne, sprich $L \in NP$. □

Sei C eine Klasse von Sprachen über dem Alphabet Σ (ohne Einschränkung sei im Folgenden $\Sigma = \{0,1\}$). Dann definieren wir *Co-C* als die Klasse $\{L \mid (\Sigma^* - L) \in C\}$, das *Komplement* von C. Leicht zu sehen ist, daß sich *Co-RP* wie folgt definieren läßt:

Definition 3.4 *Co-RP* ist die Klasse von Sprachen L, die von normalisierten nichtdeterministischen Turingmaschinen M in Polynomzeit erkannt werden können, für die folgendes gilt:

1. Für $x \in L$ sind alle Blätter im Berechnungsbaum von M akzeptierend.
2. Für $x \notin L$ sind mindestens die Hälfte der Blätter im Berechnungsbaum von M ablehnend.

Beispiel: Das Matrixmultiplikationsproblem (vergleiche Abschnitt 1.3.1), aufgefaßt als Entscheidungsproblem, ist in *Co-RP*. Die Menge aller Primzahlen (kodiert über {0,1}) ist nach Kapitel 2 auch in *Co-RP*. Umgekehrt ist dementsprechend die Menge aller zusammengesetzten Zahlen in RP.

„*Co-RP*-Algorithmen“ sind *Monte-Carlo-Algorithmen* mit einseitigem Fehler.

Die Definition von *RP* verlangt, daß Eingaben in der Sprache mit Wahrscheinlichkeit größer gleich $\frac{1}{2}$ akzeptiert werden, wohingegen Eingaben, die nicht zur Sprache gehören, immer abgelehnt werden müssen. Die Konstante $\frac{1}{2}$ könnte hierbei durch jede andere Konstante ϵ mit $0 < \epsilon < 1$ ersetzt werden, ohne etwas an der Definition von *RP* zu ändern. (Warum?)

Auch wenn wir durch polynomiell viele Wiederholungen die Fehlerwahrscheinlichkeit exponentiell klein machen können, so können wir doch nie endgültig sicher sein, daß die Ausgabe eines *RP*-Algorithmus korrekt ist. Die Sache ändert sich, wenn wir für eine Sprache L sowohl $L \in RP$ als auch $L \in Co\text{-}RP$ wissen. Dann entscheidet folgender Algorithmus *immer* korrekt, ob $x \in L$ gilt:

repeat
  a :=„Ergebnis, das der *RP*-Algorithmus angewandt auf x liefert";
  b :=„Ergebnis, das der *Co-RP*-Algorithmus angewandt auf x liefert";
until $a = b$;
return a

Da immer einer der beiden benutzten Algorithmen *keinen* Fehler macht, ist die Ausgabe immer korrekt. Somit ist lediglich die Laufzeit eine Zufallsvariable. Diese Betrachtungen legen folgende Definition nahe:

Definition 3.5 $ZPP := RP \cap co - RP$.

ZPP („Zero-error Probabilistic Polynomial time") ist somit die Klasse der Sprachen, für die *Las-Vegas-Algorithmen* mit erwarteter polynomieller Laufzeit existieren.

Beispiel: Randomisiertes Quicksort (Abschnitt 1.3.2) und der Algorithmus zur Spielbaumauswertung (Abschnitt 1.3.3) sind „*ZPP*-Algorithmen". Beide haben sogar garantiert polynomielle Laufzeiten.

Aufgrund eines Ergebnisses von Adleman und Huang (Recognizing primes in random polynomial time, *19th Annual ACM Symposium on Theory of Computing* 462-469, 1987) ist bekannt, daß die Menge aller Primzahlen in *RP* liegt. Zusammen mit dem vorher erwähnten Enthaltensein in *Co-RP* folgt somit, daß diese Sprache sogar in *ZPP* ist. Es ist bis heute jedoch offen, ob sie auch in P liegt.

Die Klassen *RP* und *Co-RP* erlauben nur einseitige Fehler. Es kann hilfreich sein, zweiseitige Fehler zuzulassen:

Definition 3.6 *BPP* (Bounded error Probability, Polynomial time) ist die Klasse von Sprachen L, die von normalisierten nichtdeterministischen Turingmaschinen M in Polynomzeit erkannt werden können, für die folgendes gilt:

1. Für $x \in L$ sind mindestens $\frac{3}{4}$ der Blätter des Berechnungsbaumes von M akzeptierend.
2. Für $x \notin L$ sind mindestens $\frac{3}{4}$ der Blätter des Berechnungsbaumes von M ablehnend.

Wir sprechen bei Definition 3.6 von einer *Fehlerwahrscheinlichkeit* von $\frac{1}{4}$. In Definition 3.6 könnte die Konstante $\frac{3}{4}$ auch durch jede beliebige andere *Konstante* $\epsilon > \frac{1}{2}$ ersetzt werden. Das Interesse an *BPP* rührt von folgendem Satz. Er zeigt, daß wie bei *RP* auch bei *BPP* die Fehlerwahrscheinlichkeit exponentiell klein gemacht werden kann. Der Beweis hierfür ist jedoch nicht mehr ganz trivial.

Satz 3.7 *Sei $L \in BPP$. Dann wird L für ein beliebiges Polynom $p : \mathbb{N}_0 \to \mathbb{N}$ bei Eingabelänge n auch von einem BPP-Algorithmus mit Fehlerwahrscheinlichkeit $2^{-p(n)}$ anstelle von $\frac{1}{4}$ erkannt.*

Beweis: Idee: Führe viele unabhängige Berechnungen der Turingmaschine M für L durch und fälle die Entscheidung für Akzeptanz oder Ablehnung einfach per *Mehrheitsentscheid*. Wir zeigen, daß dies hinreichend zuverlässig ist.
Sei x eine Eingabe mit $|x| = n$. Definiere $\epsilon_x \leq \frac{1}{4}$ als die Wahrscheinlichkeit, mit der M bei Eingabe x die falsche Antwort liefert. Weiter sei $\delta_x := 1 - \epsilon_x$. Leicht ergibt sich, daß $\epsilon_x \cdot \delta_x \leq \frac{1}{4} \cdot \frac{3}{4} = \frac{3}{16}$. Wir führen die Berechnung von M m-mal hintereinander aus, wobei $m := 2 \cdot q + 1$ eine ungerade Zahl sei, für die $q \in \mathbb{N}$ dann später in Abhängigkeit von $p(n)$ festgelegt wird. Wichtig ist, daß q durch ein Polynom beschränkt werden kann. Wir akzeptieren, falls bei m-maliger Ausführung die Turingmaschine M *mehr* als q-mal akzeptiert.
Unsere Antwort ist genau dann falsch, wenn M höchstens q-mal die richtige Antwort liefert. Die Wahrscheinlichkeit hierfür ist

$$\begin{aligned}
\sum_{j=0}^{q} \binom{m}{j} \delta_x^j \epsilon_x^{m-j} &\leq \sum_{j=0}^{q} \binom{m}{j} \delta_x^{\frac{m}{2}} \epsilon_x^{\frac{m}{2}} \\
&\leq \left(\frac{3}{16}\right)^{m/2} \cdot \sum_{j=0}^{q} \binom{m}{j} \\
&\leq \left(\frac{3}{16}\right)^{m/2} \cdot 2^m
\end{aligned}$$

$$= \left(4 \cdot \frac{3}{16}\right)^{\frac{m}{2}} \leq \left(\frac{3}{4}\right)^{q}.$$

Sei $c := 1/(2 - \log 3)$ und setze $q := \lceil c \cdot p(n) \rceil$. Daraus folgt die Behauptung. $\square$

Satz 3.7 ist der Grund, warum *BPP* als die Komplexitätsklasse angesehen wird, deren Probleme noch mit vertretbarem Aufwand gelöst werden können: Die Fehlerwahrscheinlichkeit kann unter Wahrung einer polynomiellen Zeitschranke für praktische Zwecke verschwindend klein gemacht werden.

Als mächtigste probabilistische Komplexitätsklasse definieren wir zum Abschluß *PP*:

Definition 3.8 *PP* ist die Klasse von Sprachen L, die von normalisierten nichtdeterministischen Turingmaschinen M in Polynomzeit erkannt werden können, für die gilt:

1. Für $x \in L$ ist mehr als die Hälfte der Blätter des Berechnungsbaumes von M akzeptierend.
2. Für $x \notin L$ ist mindestens die Hälfte der Blätter des Berechnungsbaumes von M ablehnend.

Obwohl der Unterschied in der Definition von *PP* im Vergleich zu *BPP* marginal erscheint, ist zwischen *BPP* und *PP* ein „komplexitätstheoretisch großer Sprung", wie der nächste Abschnitt zeigt. Im Gegensatz zu in *RP*, *Co-RP*, *ZPP* oder *BPP* liegenden Problemen kann man bei Problemen in *PP* nicht mehr davon ausgehen, daß sie in der Praxis noch gut lösbar sind.

3.2 Beziehungen zwischen den Klassen

Ziel dieses Abschnittes ist die Entwicklung eines Bildes (in Form eines Inklusionendiagramms) der „randomisierten Komplexitätswelt" zwischen P und *PSPACE*. Direkt aus den Definitionen folgt $BPP \subseteq PP$. Nach Satz 3.3 wissen wir schon, daß $P \subseteq RP \subseteq NP$. Analog zu $P \subseteq RP$ folgt auch $P \subseteq Co\text{-}RP$, was unmittelbar $P \subseteq ZPP$ impliziert. Auch leicht zu sehen ist:

Satz 3.9 *$RP \subseteq BPP$ und $Co\text{-}RP \subseteq BPP$.*

Beweis: Für $x \in L$ erlaubt RP die Fehlerwahrscheinlichkeit $\frac{1}{2}$, für $x \notin L$ die Fehlerwahrscheinlichkeit 0. Durch zweimalige Hintereinanderausführung des RP-Algorithmus erreichen wir die Fehlerwahrscheinlichkeiten $\frac{1}{4}$ (für $x \in L$) und 0 (für $x \notin L$), die für die Definition von BPP hinreichend klein sind. Analog für Co-RP. □

Folgender Satz zeigt nun die schon angedeutete „Bösartigkeit" von PP, das heißt, PP enthält NP.

Satz 3.10 *$NP \subseteq PP \subseteq PSPACE$.*

Beweis: „$NP \subseteq PP$": Sei $L \in NP$ und M eine normalisierte nichtdeterministische Turingmaschine, die L in Polynomzeit erkennt. Modifiziere M wie folgt: Am Ende einer Berechnung wird „eine Münze geworfen". Die modifizierte Maschine M' akzeptiert genau dann, wenn die Berechnung von M akzeptiert *oder* die am Ende geworfene Münze „Kopf" zeigt. Sei x ein Eingabewort. Nun gilt: Falls $x \notin L$, so ist genau die Hälfte der Blätter im Berechnungsbaum von M' akzeptierend, denn in diesem Fall kommt es nur auf den Münzwurf an. Falls $x \in L$, so akzeptieren mehr als die Hälfte der Blätter! L ist also genau die von M' „mit Mehrheit" akzeptierte Sprache, sprich $L \in PP$.

„$PP \subseteq PSPACE$": Sei $L \in PP$ und M eine normalisierte nichtdeterministische Turingmaschine, die L in Polynomzeit mit Mehrheit akzeptiert. Wir simulieren einfach nacheinander *alle* möglichen Berechnungen von M und zählen dabei die Anzahl der akzeptierenden Berechnungen und akzeptieren, falls der Bruchteil der akzeptierenden Berechnungen $\frac{1}{2}$ übersteigt. „Offensichtlich" genügt dazu ein Speicherplatz polynomiell in der Eingabegröße. □

Wäre RP *komplementabgeschlossen*, das heißt, gölte $RP = Co\text{-}RP$, so wäre $RP = ZPP$. Dies ist offen. Für BPP hingegen ist leicht zu sehen, daß $BPP = Co - BPP$. Dies liegt im wesentlichen an der Symmetrie in der Definition von BPP. Es genügt die akzeptierenden und die ablehnenden Endzustände miteinander zu vertauschen. Aufgrund der leichten Asymmetrie in der Definition von PP ist das nicht so leicht für PP zu zeigen:

Satz 3.11 *PP ist komplementabgeschlossen, das heißt $PP = Co - PP$.*

Beweis: Sei $L \in PP$ und sei M eine normalisierte nichtdeterministische Turingmaschine, die L in Polynomzeit erkennt. Es genügt zu zeigen, daß M so modifizierbar ist, daß für keine Eingabe x *genau* die Hälfte der Blätter im Berechnungsbaum akzeptierend ist. Läßt sich dies garantieren, so können wir einfach die akzeptierenden und die ablehnenden Endzustände miteinander vertauschen und haben somit eine Maschine für $\bar{L}$ (das Komplement von L) konstruiert. Im folgenden verbleibt es, die besagte Modifikation zu beschreiben. Die Idee ist, jede akzeptierende Endkonfiguration von M „mit geringer Wahrscheinlichkeit verwerfend" zu machen. Wir benutzen dazu folgende normalisierte nichtdeterministische Turingmaschine M':

Simuliere M; {das dauere t Schritte}
Wirf t-mal eine Münze;
if M lehnte ab
  then lehne unabhängig von dem Münzwurf immer ab
  else if t-mal Kopf
    then lehne ab
    else akzeptiere.

Schematisch läßt sich die Arbeitsweise des obigen Algorithmus anhand des Bildes für den zugehörigen Berechnungsbaum wie in Abbildung 2 veranschaulichen:

Die grundlegende Idee ist also, ausgehend vom Berechnungsbaum der Maschine M den Berechnungsbaum der Maschine M' derart zu konstruieren, daß ein akzeptierendes Blatt bei M durch einen Baum mit $2^t - 1$ akzeptierenden Blättern und einem ablehnenden Blatt ersetzt wird; desweiteren wird ein ablehnendes Blatt durch einen Baum mit ausschließlich ablehnenden Blättern (das sind 2^t viele) ersetzt.

Seien jetzt a beziehungsweise a' die Anzahlen der akzeptierenden Blätter im Berechnungsbaum von M beziehungsweise M' auf der Eingabe x. Per Konstruktion gilt $a' = (2^t - 1)a$: OBdA sei $t \geq 2$. Dann gilt:

$$\begin{array}{rrcl}
a \leq 2^{t-1} \text{ (das heißt } M \text{ verwirft)} & \Rightarrow a' & < & 2^{2t-1} \text{ (das heißt } M' \text{ verwirft)} \\
 & \text{und} & & \\
a > 2^{t-1} \text{ (das heißt } M \text{ akzeptiert)} & \Rightarrow a & \geq & 2^{t-1} + 1 \\
 & \Rightarrow a' & \geq & (2^t - 1)(2^{t-1} + 1) \\
 & & = & 2^{2t-1} + 2^t - 2^{t-1} - 1 \\
 & & = & 2^{2t-1} + 2^{t-1} - 1 \\
 & & > & 2^{2t-1} \text{ (das heißt } M' \text{ akzeptiert)}
\end{array}$$

M' akzeptiert also auch L mit Mehrheit, ohne daß jemals für eine Eingabe x genau die Hälfte der Blätter des Berechnungsbaumes von M' auf Eingabe x akzeptierend ist. Daraus folgt nun die Behauptung, da in M' dafür lediglich die akzeptierenden und ablehnenden Endzustände miteinander vertauscht werden müssen. □

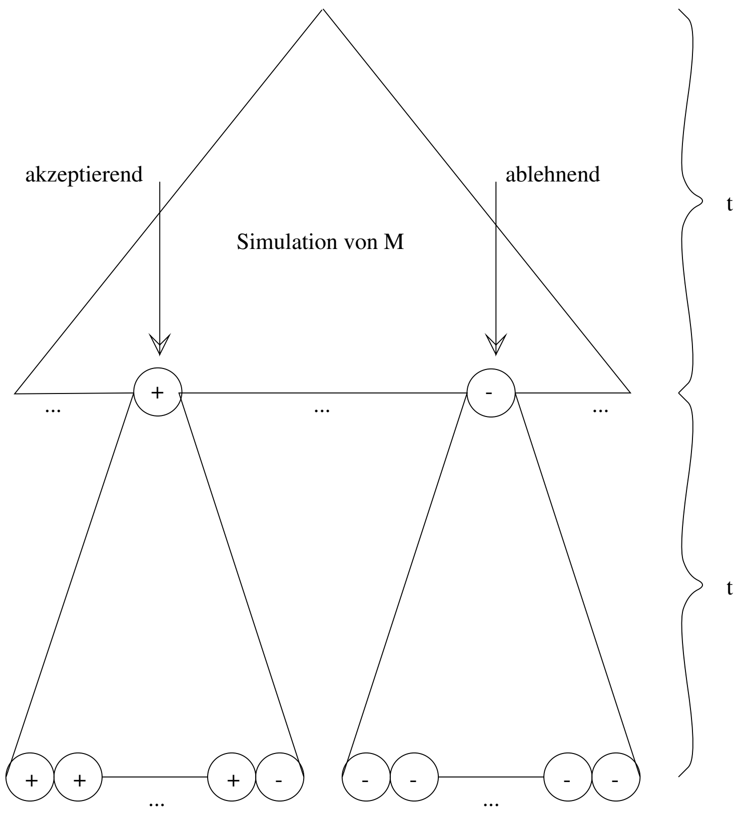


Abbildung 2: Berechnungsbaum der aus M konstruierten Turingmaschine M'.

Schließlich sei noch erwähnt, daß folgende Variante des Erfüllbarkeitsproblems aussagenlogischer Formeln in PP enthalten ist (Warum?):

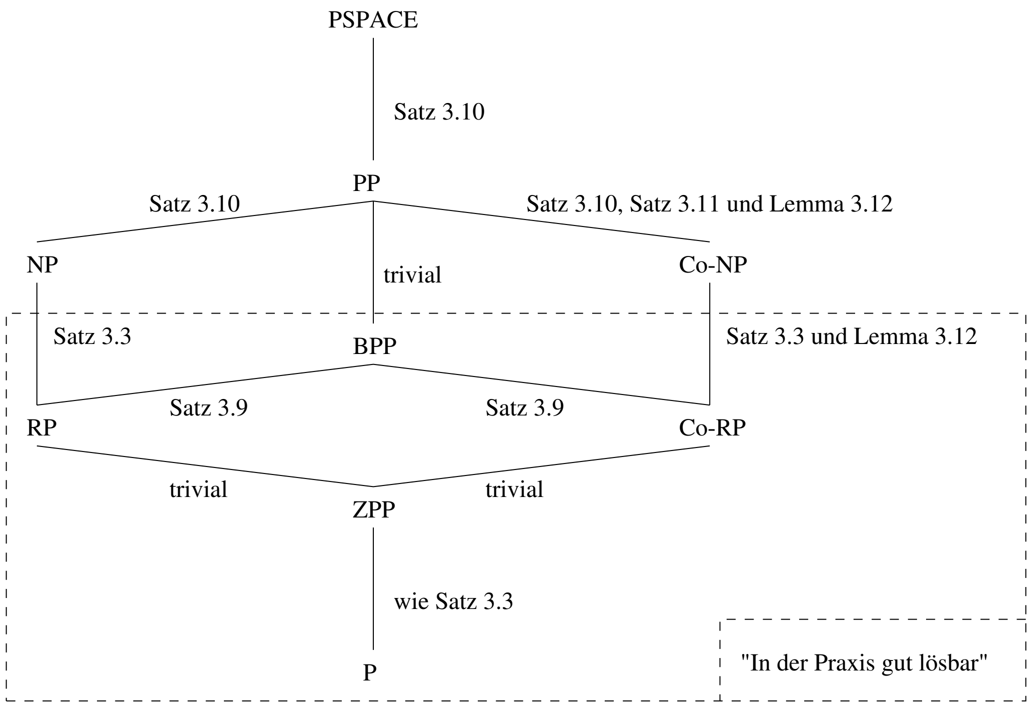
#SAT:
Gegeben: Eine aussagenlogische Formel F und eine Zahl $k \in \mathbb{N}_0$.
Frage: Hat F mehr als k erfüllende Belegungen?

Es läßt sich sogar zeigen, daß #SAT zu den „harten“ Problemen in PP gehört, das heißt *PP-vollständig* ist. Hingegen ist offen, ob die Klassen ZPP, RP, $Co - RP$ oder BPP vollständige Probleme besitzen.
Zur Erzeugung unseres abschließenden Inklusionendiagramms ist noch folgendes Lemma sehr nützlich:

Lemma 3.12 *Sind A und B Komplexitätsklassen, so gilt* $A \subseteq B \Rightarrow Co - A \subseteq Co - B$.

Beweis: $L \in Co - A \Rightarrow \bar{L} \in A \Rightarrow \bar{L} \in B \Rightarrow L \in Co - B$. □

Die randomisierte Komplexitätswelt zwischen *P* und *PSPACE* sieht zum gegenwärtigen Zeitpunkt wie folgt aus:



Bei allen Inklusionsrelationen ist unbekannt, ob sie echt sind. Andererseits sind auch viele Klassenbeziehungen offen. So ist zum Beispiel unbekannt, ob $BPP \subseteq NP$ (oder umgekehrt). Es gibt jedoch Indizien, die sogar auf $BPP = P$ hindeuten, wie der nächste Abschnitt zeigt.

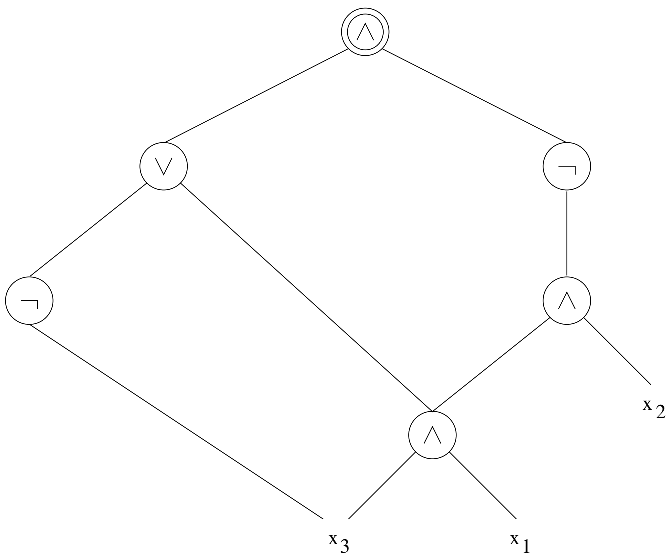
Als Fazit dieses Abschnittes läßt sich insbesondere festhalten, daß wir nicht erwarten können, daß *NP*-vollständige Probleme *effizient* durch randomisierte Verfahren gelöst werden können.

3.3 Schaltnetze

Boolesche Schaltnetze sind ein noch elementareres Berechnungsmodell als Turingmaschinen. Sie eignen sich besonders zum Studium *unterer Schranken*, aber auch zur Modellierung von *parallelem Rechnen*. Sie zeigen uns aber auch gewisse Grenzen randomisierter Algorithmen.

Definition 3.13 Ein *(Boolesches) Schaltnetz* ist ein azyklischer, gerichteter Graph, dessen Knoten mit den logischen Operatoren $\vee$, $\wedge$ und $\neg$ beschriftet sind und entsprechenden Eingangsgrad haben. Zudem gibt es Knoten ohne Vorgänger (Eingangsgrad 0), die entweder mit Konstanten (0 oder 1) oder mit Variablen $(x_1, x_2, \ldots, x_n)$ beschriftet sind. Schließlich gibt es einen ausgezeichneten Ausgangsknoten, der bei Belegung der (Eingabe-)Variablen einen bestimmten Ausgabewert liefert. Somit berechnet ein Boolesches Schaltnetz in natürlicher Weise eine *Boolesche Funktion*.

Beispiel 3.14 Ein Boolesches Schaltnetz der Größe neun, drei Eingangsvariablen und ein Ausgabeknoten (doppelt eingekreist).



Gemäß Definition 3.13 lassen sich Schaltnetze auch als *Sprachakzeptoren* verstehen: Wörter über $\{0,1\}^n$ repräsentieren die Belegung der Eingangsvariablen und werden bei Ausgabeknotenwert 1 akzeptiert und bei 0 abgelehnt. Damit Schaltnetze auch als Sprachakzeptoren für Sprachen in $\{0,1\}^*$ dienen, bedarf es folgender Erweiterung:

Definition 3.15 Die *Größe* eines Booleschen Schaltnetzes ist die Anzahl seiner Knoten (*Gatter*). Eine *Schaltnetzfamilie* ist eine unendliche Sequenz $\mathcal{C} = (C_0, C_1, C_2, \ldots)$ von Booleschen Schaltnetzen, wobei C_n genau n Eingabevariablen habe. Wir sagen, daß eine Sprache $L \subseteq \{0,1\}^*$ *Schaltnetze polynomieller Größe* besitzt, falls es eine Familie $\mathcal{C}$ von Schaltnetzen gibt, so daß gilt:

1. Jedes C_n hat Größe höchstens $p(n)$ für ein festes Polynom p.
2. $\forall x \in \{0,1\}^* : x \in L \iff C_{|x|}$ liefert auf Eingabe $x = x_1 x_2 x_3 \ldots x_{|x|}$ den Wert 1.

Es besteht ein enger Zusammenhang zwischen polynomzeitbeschränkten Turingmaschinen und Schaltnetzen polynomieller Größe.

Satz 3.16 *Alle Sprachen in P haben Schaltnetze polynomieller Größe.*

Beweisidee: Simuliere die $t(n)$ Konfigurationsübergänge einer Turingmaschine mit Laufzeit $t(n)$ mithilfe von $t(n)$ Schichten im Schaltnetz, wobei die Nachfolgekonfiguration einer Konfiguration mit sehr einfacher Boolescher Logik aus dieser berechnet werden kann. Jede Schicht repräsentiere also quasi eine Konfiguration der zu simulierenden Turingmaschine. □

Die Umkehrung von Satz 3.16 gilt nicht. Denn:

Mitteilung 3.17 *Es gibt unentscheidbare Sprachen, die Schaltnetze polynomieller Größe haben.*

Der Grund für die überraschende Mächtigkeit von Schaltnetzen polynomieller Größe liegt darin, daß wir keinerlei Forderungen an die „Konstruierbarkeit" der Schaltnetze stellten. Deshalb:

Definition 3.18 Wir nennen eine Schaltnetzfamilie $\mathcal{C} = (C_0, C_1, C_2, \ldots)$ *uniform*[2], falls es eine deterministische Turingmaschine gibt, die in Polynomzeit bei Eingabe 1^n (das heißt die Zahl n in Unärkodierung) das Schaltnetz C_n ausgibt.

Für uniforme Schaltnetzfamilien gilt auch die Umkehrung von Satz 3.16.

Satz 3.19 *Eine Sprache hat uniforme Schaltnetze polynomieller Größe genau dann, wenn sie in P liegt.*

Beweisidee: Die im Beweis zu Satz 3.16 skizzierte Konstruktion ist uniform. Für die Umkehrrichtung konstruiere zunächst in Polynomzeit das Schaltnetz auf dem Arbeitsband und werte es dann ebenfalls in Polynomzeit aus. □

Es wird vermutet, daß *NP*-vollständige Probleme weder uniforme noch nicht-uniforme Schaltnetze polynomieller Größe besitzen. Probleme in *BPP* jedoch haben nicht-uniforme Schaltnetze polynomieller Größe, wie das folgende Hauptergebnis dieses Abschnittes zeigen wird. Das heißt insbesondere, daß sich randomisierte Polynomzeitalgorithmen durch deterministische Schaltnetze polynomieller Größe *derandomisieren* lassen.

[2] Üblicherweise wird für die „Uniformitätsmaschine" weniger „Berechnungskraft" angenommen, wie zum Beispiel eine logarithmisch platzbeschränkte Turingmaschine oder noch schwächer. Hier ist dies unerheblich.

Satz 3.20 *Alle Probleme in BPP haben Schaltnetze polynomieller Größe.*

Beweis: Wir geben zu einer beliebigen Sprache $L \in BPP$ ein Schaltnetz polynomieller Größe an, das L erkennt. Dabei setzen wir ohne Einschränkung $L \subseteq \{0,1\}^*$ voraus. Nach Satz 3.7 gibt es einen randomisierten Polynomzeit-Algorithmus A, der L mit Fehlerwahrscheinlichkeit kleiner als 2^{-n} erkennt, wobei n die Länge des Eingabewortes x ist. Definiere die Zufallsvariable C_x wie folgt:

$$C_x = \begin{cases} 1, & \text{falls } A \text{ bei Eingabe } x \text{ einen Fehler macht,} \\ 0, & \text{sonst.} \end{cases}$$

Der Wert von C_x kann von einem Lauf von A zum nächsten verschieden sein. Wir schätzen nun den Erwartungswert für die Anzahl der Eingaben der Länge n, auf welchen A einen Fehler macht, auf zwei verschiedene Weisen ab. Zunächst ist dieser:

$$\begin{aligned} E[\sum_{x\in\{0,1\}^n} C_x] &= \sum_{x\in\{0,1\}^n} E[C_x] \\ &= \sum_{x\in\{0,1\}^n} \mathbf{Pr}[C_x = 1] \\ &< \sum_{x\in\{0,1\}^n} 2^{-n} \\ &= 1. \end{aligned}$$

Sei $r(n)$ die (polynomielle!) Anzahl von Zufallsbits, die A auf Eingaben der Länge n benutzt (ohne EInschränkung benutze A für alle Eingaben der Länge n die gleiche Anzahl an Zufallsbits). Weiter sei $C_x(y)$ der Wert, den C_x annimmt, wenn A die Zufallsfolge $y \in \{0,1\}^{r(n)}$ benutzt und sei $F(y)$ die Anzahl der Eingaben x, für die $C_x(y)$ den Wert 1 annimmt. Dann gilt für obigen Erwartungswert auch:

$$\begin{aligned} \sum_{x\in\{0,1\}^n} \mathbf{Pr}[C_x = 1] &= \sum_{x\in\{0,1\}^n} \frac{|\{y \mid C_x(y) = 1\}|}{2^{r(n)}} \\ &= 2^{-r(n)} \cdot \sum_{x\in\{0,1\}^n} \sum_{y\in\{0,1\}^{r(n)}} C_x(y) \\ &= 2^{-r(n)} \cdot \sum_{y\in\{0,1\}^{r(n)}} \sum_{x\in\{0,1\}^n} C_x(y) \\ &= 2^{-r(n)} \cdot \sum_{y\in\{0,1\}^{r(n)}} F(y). \end{aligned}$$

Da nach vorheriger Abschätzung der Erwartungswert kleiner als 1 ist, folgt, daß es mindestens ein $y \in \{0,1\}^{r(n)}$ mit $F(y) = 0$ geben muß. Benutze im randomisierten Algorithmus A nun immer dies y anstelle einer Folge von Zufallsbits. Da $F(y) = 0$ heißt das, daß dieser nunmehr deterministische Algorithmus auf keiner Eingabe der Länge n einen Fehler macht, seine Ausgabe ist also immer korrekt. Wie in Satz 3.16 läßt sich schließlich zu diesem deterministischen Polynomzeitalgorithmus ein Schaltnetz polynomieller Größe angeben, das ihn simuliert. □

Der Beweis von Satz 3.20 ist hochgradig nicht-konstruktiv, es handelt sich lediglich um einen *Existenzbeweis*. Wir wissen nicht, wie das besagte y effizient zu finden ist, und es kann für verschiedene Eingabelängen n verschieden sein. Der Beweis ist ein Beispiel für das Paradigma „probabilistische Methode." Könnte eine *uniforme* Schaltnetzfamilie konstruiert werden, die BPP simuliert, so würde daraus $P = BPP$ folgen.

Starke Indizien dafür, daß $P = BPP$ gelten könnte, lieferten jüngst N. Nisan und A. Wigderson Hardness vs Randomness, *Journal of Computer and System Sciences*, Vol. 49, 149–167, 1994) und R. Impagliazzo und A. Wigderson („*P =BPP if E requires exponential circuits: Derandomizing the XOR Lemma*", In Proceedings of the 29th ACM Symposium on Theory of Computing, May 1997). Sie zeigten, daß wenn irgendein Problem in $E = DTIME(2^{O(n)})$ eine Schaltnetzgröße von $2^{\Omega(n)}$ erfordert, daß dann $P = BPP$ gilt. Erstere Vermutung wird von vielen Leuten vertreten. Andererseits würde $P = BPP$ auch aus $P = NP$ folgen (siehe M. Sipser: A complexity-theoretic approach to randomness, *In Proceedings of the 15th ACM Symposium on Theory of Computing*, 330-335, 1983).

Wie bereits erwähnt wird vermutet, daß nicht alle Probleme in NP Schaltnetze polynomieller Größe haben, was $NP \neq BPP$ implizieren würde. Ein erster Schritt in Richtung auf einen Beweis dieser Vermutung wurde von dem Russen A. Razborov getan: Bezeichne ein Boolesches Schaltnetz, das keine Negationen enthalten darf, als *monotones Schaltnetz*. Razborov zeigte, daß ein monotones Schaltnetz zur Berechnung der $\text{CLIQUE}_{n,k}$-Funktion (Gegeben: Graph G mit n Knoten; Frage: Besitzt G eine Knotenmenge der Größe k, so daß zwischen allen Knoten dieser Menge eine Kante in G zu jedem anderen Knoten der Menge existiert?) exponentielle Größe benötigt. Genauer: Es gibt eine Konstante c, so daß für n groß genug jedes monotone Schaltnetz zur Berechnung von $\text{CLIQUE}_{n,k}$ mit $k = \sqrt[4]{n}$ mindestens eine Größe von $2^{c \cdot \sqrt[8]{n}}$ benötigt.

In natürlicher Weise lassen sich auch *randomisierte Schaltnetze* einführen. Neben den üblichen Eingabevariablen gibt es hier zusätzlich Eingabeknoten, die Zufallsbits liefern, ansonsten sind sie wie die üblichen (deterministischen) Schaltnetze definiert. In Analogie zu den Definitionen von ZPP, RP, *Co-RP* und *BPP* lassen sich so auch verschiedene randomisierte Schaltnetzfamilien definieren. Satz 3.20 kann dann dahingehend umgedeutet werden, daß randomisierte Schaltnetze polynomieller Größe nicht mächtiger sind als deterministische Schaltnetze polynomieller Größe (im nichtuniformen Fall).

Zum Abschluß dieses Kapitels noch ein Wort zu *nicht-uniformen Algorithmen*: Im Beweis von Satz 3.20 wurde ein deterministischer Algorithmus, der eine feste Bitfolge $y \in \{0,1\}^{r(n)}$ verwendet, angegeben. Dieses y nennt man in der Literatur auch „Advice". Man sagt, daß ein Algorithmus A eine Sprache L mit Advice a entscheidet, falls A neben dem Eingabewort $x \in \{0,1\}^n$ auf einem zusätzlichen Read-Only-Eingabeband eine Bitfolge $a(n) \in \{0,1\}^{n^{O(1)}}$ bekommt und mit deren Hilfe „$x \in L$?" entscheidet. In anderen Worten heißt das, daß eine *einzige* Advice-Bitfolge $a(n)$ es dem Algorithmus A ermöglicht, „$x \in L$?" für *alle* $x \in \{0,1\}^n$ zu entscheiden. Solche Algorithmen heißen nicht-uniform. *Uniforme Algorithmen* hingegen benutzen keinen Advice.

Auf diese Weise läßt sich dann zum Beispiel die Komplexitätsklasse $P_{/poly}$ definieren als die Klasse von Problemen, die deterministische Polynomzeit-Algorithmen besitzen, die (polynomiell großen) Advice benutzen. Mit dieser Terminologie ergibt sich unmittelbar aus Satz 3.20:

Korollar 3.21 $BPP \subseteq P_{/poly}$.

4 Parallele Algorithmen

Das vielleicht mit wichtigste Anwendungsfeld für randomisierte Verfahren ist das der parallelen Algorithmen.

„Beschleunigung durch Parallelisierung“ ist heutzutage ein zentrales Forschungsthema der Informatik. Parallele Algorithmen tragen dazu bei, indem sie eine effiziente Zusammenarbeit mehrerer, meist eng gekoppelter Prozessoren an einem Problem gewährleisten.

Es gibt eine Vielzahl von Parallelrechnermodellen, auf die wir hier nicht im Einzelnen eingehen wollen. Wichtiger für uns in diesem Kapitel sind exemplarische Anwendungen von Randomisierung im parallelen Bereich. In Abschnitt 4.1 über perfekte Matchings lernen wir mit dem *Isolationslemma* eine Technik kennen, die es uns ermöglicht die Zusammenarbeit vieler Prozessoren dadurch zu organisieren, daß sie an einer mittels Randomisierung eindeutig bestimmten Lösung arbeiten. In Abschnitt 4.2 (Routing in Parallelrechnern) dient Randomisierung der gleichmäßigen Verteilung von Kommunikationslast über die Verbindungskanäle eines Parallelrechners. Hierbei lernen wir einen randomisierten Algorithmus kennen, der jedem deterministischen Algorithmus *beweisbar* überlegen ist. Für seine Analyse nutzen wir die *Chernoff-Schranken*, die Abschätzungen über die Summen unabhängiger 0-1-Zufallsvariablen liefern.

4.1 Perfektes Matching[3]

Das Finden eines „sehr schnellen“ parallelen Algorithmus zur Bestimmung perfekter Matchings in Graphen ist ein Problem, das bislang nur randomisiert zufriedenstellend gelöst werden konnte.

4.1.1 Problem und Berechnungsmodell

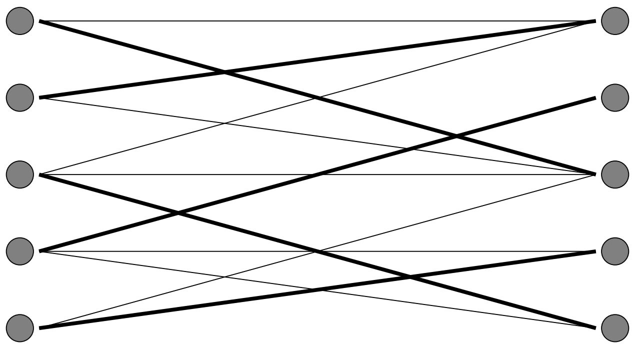
Gegeben: Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E.
Frage: Besitzt G ein perfektes Matching, das heißt eine Kantenmenge $M \subseteq E$ so, daß keine zwei Kanten aus M einen gemeinsamen Knoten besitzen und jeder Knoten aus V zu einer Kante in M gehört?

Natürlich ist man in der Regel nicht nur an der obigen Entscheidungsvariante des Problems interessiert, sondern auch an der konkreten Konstruktion eines

[3]Der Inhalt dieses Abschnittes wurde in einer englischsprachigen Gastvorlesung von Eric Allender (Rutgers University, New Brunswick) präsentiert. Diese Ausarbeitung stützt sich auf seinen Tafelanschrieb.

perfekten Matchings.
Nachfolgend beschränken wir uns auf einen Spezialfall des Problems, nämlich der Suche nach perfekten Matchings für *bipartite* Graphen. Ein bipartiter Graph ist ein Graph G, bei welchem sich die Knotenmenge V zerlegen läßt gemäß $V = V_1 \cup V_2$ mit $V_1 \cap V_2 = \emptyset$, so daß *jede* Kante in E zwischen einem Knoten aus V_1 und einem aus V_2 verläuft.

Beispiel 4.1 Ein bipartiter Graph mit eingezeichnetem perfekten Matching (fett durchgezogene Kanten).



Offensichtlich kann für bipartite Graphen ein perfektes Matching nur existieren, falls $|V_1| = |V_2|$ gilt.[4] Es verbleibt nun noch, den eingangs benutzten Begriff des „sehr schnellen parallelen Algorithmus“genauer zu spezifizieren.

Bekanntlich gelten die in deterministischer (oder randomisierter...) Polynomzeit berechenbaren Probleme als die *sequentiell effizient lösbaren* Probleme, was zum Beispiel zu Komplexitätsklassen wie P oder RP führte. Man geht dabei also immer von *einem* Prozessor aus, der mithilfe eines sequentiellen Algorithmus ein Problem schnell lösen kann. Immer mehr drängt sich heute die Frage auf, wie Probleme unter dem Einsatz von mehreren, parallel arbeitenden Prozessoren schneller gelöst werden können. In der Theorie geht es dabei insbesondere auch um die Grenzen der Parallelisierbarkeit, sprich der Frage nach maximal erzielbarer Beschleunigung unter Einsatz einer „vernünftigen“ Anzahl von Prozessoren. Diese Ansätze werden in der Komplexitätsklasse NC aufgefangen, welche die Klasse von Problemen umfaßt, die mit einer polynomiellen Prozessorenzahl ($n^{O(1)}$) in polylogarithmischer Laufzeit ($(\log n)^{O(1)}$) gelöst werden können. Dabei spielt die Wahl

[4] Das Problem des perfekten Matching für bipartite Graphen ist auch als *Heiratsproblem* bekannt. Dabei wird V_1 als Menge von Damen und V_2 als Menge von Herren interpretiert und eine Kante drückt aus, daß das entsprechende Paar befreundet ist, also eine Heirat nicht ausgeschlossen ist. Ein perfektes Matching bedeutet dann, daß alle Damen und Herren einen Ehepartner finden, global betrachtet also perfekte Zufriedenheit hergestellt wird...

des „Parallelrechnermodells“ keine so große Rolle wie man vermuten könnte. Die Klasse NC ist definierbar mit einer Vielzahl von parallelen Modellen — wir beschränken uns hier auf das vielleicht einfachste: Schaltnetze. Wie bereits in Abschnitt 3.3 erwähnt, lassen sich (Boolesche) Schaltnetze als sehr elementares paralleles Berechnungsmodell verstehen. Unter der Größe eines Schaltnetzes verstehen wir die Anzahl seiner Gatter (vergleiche Definition 3.15). Unter seiner *Tiefe* verstehen wir den längsten Pfad von einem Eingabeknoten des Schaltnetzes bis zu einem Ausgabeknoten.

Definition 4.2

- NC^k ist die Menge von Funktionen, die von uniformen Schaltnetzfamilien der Größe $n^{O(1)}$ und der Tiefe $O((\log n)^k)$ berechnet werden können.
- $NC := \bigcup_{k\in\mathbb{N}} NC^k$.

Definition 4.2 führt NC^k gleich als „Funktionenklasse“ (und nicht nur als „Sprachklasse“) ein, da wir an der konkreten Berechnung eines perfekten Matching (und nicht nur an seiner Existenz) interessiert sind. Klar ist, daß alle Funktionen in NC sequentiell auch in Polynomzeit berechnet werden können. (Warum?) Wie auch schon in Abschnitt 3.3 erwähnt, lassen sich in natürlicher Weise randomisierte Schaltnetze einführen. Der einzige Unterschied zu deterministischen Schaltnetzen ist die zusätzliche Bereitstellung von Zufallsbits an den Eingängen des Schaltnetzes.

Definition 4.3

- RNC^k ist die Menge von Funktionen, die von uniformen randomisierten Schaltnetzfamilien der Größe $n^{O(1)}$ und der Tiefe $(\log n)^k$ derart berechnet werden können, daß das Schaltnetz bei beliebiger Eingabe mindestens mit Wahrscheinlichkeit $\frac{1}{2}$ das korrekte Ergebnis liefert.
- $RNC := \bigcup_{k\in\mathbb{N}} RNC^k$.

Das Ergebnis dieses Abschnittes wird ein RNC^2-Algorithmus für bipartites perfektes Matching sein. Es ist unbekannt, ob es hierfür auch einen NC-Algorithmus gibt!

4.1.2 Einige Grundlagen und Existenz eines perfekten Matching

Wie sich herausstellen wird, ist die Berechnung eines perfekten Matchings eng verknüpft mit der Berechnung der Determinanten spezieller Matrizen.

Sei M eine $n \times n$-Matrix. Dann ist die *Determinante* von M definiert als:

$$\det = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)},$$

wobei S_n die symmetrische Gruppe der Permutationen der Größe n bezeichnet und $\operatorname{sgn}(\pi) \in$ {-1,1} das Vorzeichen der Permutation π ist.[5]

Beispiel 4.4 $\pi = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\}$, $M = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$.

Dann ist $\prod_{i=1}^{3} M_{i,\pi(i)} = M_{1,2} \cdot M_{2,1} \cdot M_{3,3} = 4 \cdot 2 \cdot 9 = 72$.

Unter direkter Benutzung obiger Formel zur Berechnung von $\det(M)$ müßten $n!$ Terme ausgewertet werden. Unter Ausnutzung anderer Eigenschaften der Determinante läßt sich jedoch ein deterministischer Algorithmus mit polynomieller Laufzeit zur Berechnung der Determinante einer ganzzahligen $n \times n$-Matrix angeben. Uns interessiert hier besonders die parallele Komplexität der Determinantenberechnung. Dazu ist bekannt:

Mitteilung 4.5 *Die Berechnung der Determinante ganzzahliger $n \times n$-Matrizen ist in NC^2 ausführbar.*

Eng verwandt zur Determinante ist die *Permanente*:

$$\operatorname{perm}(M) = \sum_{\pi \in S_n} \prod_{i=1}^{n} M_{i,\pi(i)}.$$

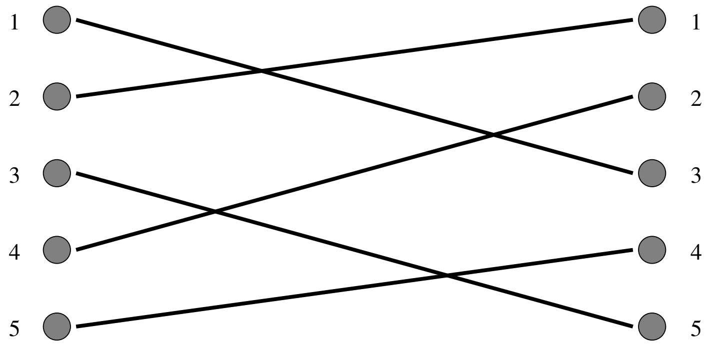
Im Vergleich zur Determinante fällt also lediglich der Faktor $\operatorname{sgn}(\pi)$ weg. Leider macht das die Permanentenberechnung schon zu einem extrem schwierigen Problem: Es ist kein Polynomzeitalgorithmus hierfür bekannt (das Problem ist NP-hart) und somit kann insbesondere auch nicht mit der Existenz eines NC- oder RNC-Algorithmus für dieses Problem gerechnet werden.

[5] Zur Erinnerung: $\operatorname{sgn}(\pi) = (-1)^t$, wobei t die Anzahl der Transpositionen ist, die benötigt werden, um die Identität in π umzuwandeln.

Im Rest dieses Unterabschnittes machen wir nun noch einige wichtige Beobachtungen, die als Grundlage für den im nächsten Unterabschnitt präsentierten RNC^2-Algorithmus zur Konstruktion eines perfekten Matching dienen.

Ein perfektes Matching für einen bipartiten Graphen $G = (V_1 \cup V_2, E)$ kann als Permutation gedeutet werden: Interpretiere die Knotenmenge V_1 als Urbild- und die Menge V_2 als Bildmenge.

Beispiel 4.6 Wir betrachten das Beispiel 4.1 und numerieren jeweils die Knoten von 1 bis 5 durch:



Das gezeichnete perfekte Matching entspricht der Permutation
$\{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 5, 4 \mapsto 2, 5 \mapsto 4\}$.

Ein bipartiter Graph mit $2n$ Knoten läßt sich in natürlicher Weise durch eine $n \times n$- Matrix mit 0-1-Einträgen repräsentieren. Die Zeilen repräsentieren beispielsweise die Knoten in V_1 und die Spalten die in V_2; falls eine Kante zwischen zwei Knoten in V_1 und V_2 existiert, so hat die Matrix an der entsprechenden Stelle eine 1, sonst eine 0. Demgemäß werden wir nachfolgend der Einfachheit halber von einem perfekten Matching einer Matrix sprechen.

Lemma 4.7 *Eine 0-1-Matrix M hat ein perfektes Matching genau dann, wenn* $\mathrm{perm}(M) \neq 0$.

Beweis: Der Beweis ergibt sich direkt aus der Definition der Permanente und der Beobachtung, daß ein perfektes Matching eine Permutation ist. $\square$

Genauere Betrachtung liefert sogar:

Lemma 4.8 *Sei M eine $n \times n$-Matrix. Dann ist* $\mathrm{perm}(M)$ *die Anzahl der perfekten Matchings von M.*

Leider lassen sich die Permanenten nicht effizient berechnen, so daß folgender „Umweg“ beschritten werden muß, um effizient die Existenz eines perfekten Matchings eines bipartiten Graphen zu entscheiden.

Ein zentraler Punkt für das Nachfolgende ist der Übergang von 0-1-Matrizen zu Matrizen, deren Einträge (symbolische) Variablen sind.

Beispiel 4.9

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \rightsquigarrow M' = \begin{pmatrix} x_{11} & 0 & x_{13} \\ x_{21} & 0 & 0 \\ 0 & x_{32} & 0 \end{pmatrix}$$

Lemma 4.10 *Sei M' die Matrix, die aus einer 0-1-Matrix M hervorgeht, indem eine 1 an der Stelle M_{ij} durch die Variable x_{ij} ersetzt wird und 0 sonst. Dann gilt:*

$$\mathrm{perm}(M) = 0 \iff \det(M') \equiv 0$$

Beweis: Die Richtung von links nach rechts ist trivial. Angenommen es sei $\mathrm{perm}(M) \neq 0$, dann existiert eine Permutation π mit $\prod_{i=1}^{n} M_{i,\pi(i)} \neq 0$. Setze nun alle Variablen in M' gleich 0, außer den Variablen $\{x_{i,\pi(i)} \mid 1 \leq i \leq n\}$, die auf 1 gesetzt werden. Unter dieser Variablenbelegung hat das multivariate Polynom $\det(M')$ den Wert 1, also folgt $\det(M') \not\equiv 0$. Dies liefert die Richtung von rechts nach links. □

Lemma 4.10 zusammen mit Lemma 4.7 haben die Frage der Existenz eines perfekten Matchings auf die Berechnung von $\det(M')$ reduziert. Es muß also nur festgestellt werden, ob das multivariate Polynom $\det(M')$ in den Variablen $x_{11}, x_{12}, \ldots, x_{nn}$ identisch gleich dem Nullpolynom ist. Am Beginn der Vorlesung (Beispiel 1.1) wurde schon angedeutet, daß dies leicht mit Hilfe eines randomisierten Algorithmus überprüfbar ist. Dieser ist auch leicht parallelisierbar. Da wir jedoch nicht nur an Existenzaussagen, sondern an einer expliziten Berechnung eines perfekten Matching interessiert sind, müssen wir noch etwas mehr Aufwand treiben. Leider hilft dabei Lemma 4.10 wenig.

4.1.3 Das Isolationslemma und die Konstruktion eines perfekten Matching

Das zentrale Problem, das bei der Konstruktion eines perfekten Matchings im Gegensatz zu der bloßen Frage nach seiner Existenz überwunden werden muß, ist das mögliche Vorhandensein sehr vieler verschiedener perfekter

Matchings. Dadurch entsteht die Schwierigkeit, dass sich die parallelen Prozessoren quasi darüber einigen müssen, welches der vielen möglichen perfekten Matchings sie gemeinsam konstruieren wollen. Dabei hilft das Isolationslemma. Mit seiner Hilfe können wir mit Wahrscheinlichkeit größer als $\frac{3}{4}$ ein *eindeutig* bestimmtes perfektes Matching aus der Menge aller perfekten Matchings „herausisolieren“, an dessen Konstruktion dann alle Prozessoren gemeinsam arbeiten können.

Wir geben das Isolationslemma nachfolgend in einer für unsere Anwendung geeigneten speziellen Form. Es ist jedoch auf beliebige Mengensysteme verallgemeinerbar und damit auch für andere Anwendungen sehr nützlich.

Definition 4.11 Sei S die Menge aller perfekten Matchings einer $n \times n$-Matrix M. Ordne jeder Kante e in M per Zufall ein ganzzahliges *Gewicht* $w(e) \in \{1, 2, \ldots, n^3\}$ zu. Für ein $Y \in S$ ist dann das Gewicht $w(Y)$ definiert als $\sum_{e \in Y} w(e)$.

Das Isolationslemma zeigt nun, daß die in Definition 4.11 beschriebene zufällige Gewichtszuordnung zu perfekten Matchings mit hoher Wahrscheinlichkeit ein Y mit minimalem Gewicht aus der Menge S isoliert:

Lemma 4.12 *(Isolationslemma[6]) Das in Definition 4.11 vorgestellte Verfahren liefert mit Wahrscheinlichkeit größer als* $\frac{3}{4}$ genau ein *Element aus S mit minimalem Gewicht.*

Beweis: Wir untersuchen die Wahrscheinlichkeit, daß es mindestens zwei perfekte Matchings gibt. Dies ist genau dann der Fall, wenn es eine Kante e in M gibt so, daß e in einem perfekten Matching ist, nicht aber in dem anderen. Definieren wir abkürzend das Prädikat $\mathrm{BAD}(e)$, das genau diesen Sachverhalt ausdrücke. Gesucht ist somit die Wahrscheinlichkeit

$$\begin{aligned} \mathbf{Pr}[\exists e : \mathrm{BAD}(e) = true] &\leq \sum_{e \in M} \mathbf{Pr}[\mathrm{BAD}(e) = true] \\ &= \sum_{e \in M} \sum_{w'} \mathbf{Pr}[\mathrm{BAD}(e) = true \mid w'] \cdot \mathbf{Pr}[w'] \qquad (*) \end{aligned}$$

Dabei sei w' eine Gewichtsfunktion, die allen Kanten in M *außer* e ein Gewicht aus $\{1, 2, \ldots, n^3\}$ zuordne und obige Summe wird über alle solche w' genommen. $\mathbf{Pr}[w']$ drückt dann das Auftreten genau dieser Gewichtsfunktion bei unserem in Definition 4.11 beschriebenen Zufallsexperiment aus und

[6] Das womöglich erstaunliche am Isolationslemma ist, daß keine Aussage über die Größe von S nötig ist!

$\mathbf{Pr}[\text{BAD}(e) = true \mid w']$ ist die bedingte Wahrscheinlichkeit, daß BAD(e) den Wert true hat, unter der Bedingung, daß w' gewählt wurde. Nun gilt

$$\mathbf{Pr}[\text{BAD}(e) = true \mid w'] \leq \frac{1}{n^3}. \qquad (**)$$

Dies ist wie folgt zu sehen. Zunächst ist klar, daß wenn es kein Gewicht für e gibt, so daß BAD(e) zu $true$ wird, die bedingte Wahrscheinlichkeit gleich 0 ist und die Behauptung gilt. Falls es aber ein Gewicht r gibt, mit der Eigenschaft

$$w(e) = r \Rightarrow \text{BAD}(e) = true,$$

so muß folgendes gelten. Würden wir $w(e)$ größer als r setzen, dann wäre e in *keinem* perfekten Matching minimalen Gewichts; würden wir $w(e)$ kleiner als r setzen, so wäre e in *jedem* perfekten Matching minimalen Gewichts. In beiden Fällen ergibt sich somit ein Widerspruch zu BAD(e)=$true$, wir können e also nur das Gewicht r geben. Da es n^3 gleich wahrscheinliche Gewichte für e gibt, geschieht dies mit Wahrscheinlichkeit $1/n^3$. Mit Einsetzung von $(**)$ in $(*)$ erhalten wir nun:

$$\begin{aligned} \mathbf{Pr}[\exists e : \text{BAD}(e) = true] &\leq \sum_{e \in M} \sum_{w'} \frac{1}{n^3} \cdot Prob[w'] \\ &\leq \sum_{e \in M} \frac{1}{n^3} \\ &\leq \frac{1}{n}. \end{aligned}$$

Setzen wir noch $n \geq 4$ voraus, so folgt die Aussage. □

Das Isolationslemma ist nun der Hauptbaustein bei der Entwicklung des gesuchten Algorithmus für perfektes Matching.

Sei M die gegebene 0-1 $n \times n$-Matrix, für die wir ein perfektes Matching suchen. Wir bilden eine Matrix M', indem wir jede Kante e in M (das sind die Stellen in M, wo eine 1 steht) ersetzen durch $2^{w(e)}$, wobei $w(e)$ das in Definition 4.11 beschriebene Gewicht ist. Dann gilt:

$$\begin{aligned} \det(M') &= \sum_{\pi \in S_n} \text{sgn}(\pi) \prod_{i=1}^{n} M'_{i,\pi(i)} \\ &= \sum_{Y \in S} \text{sgn}(Y) \prod_{i=1}^{n} 2^{w(e)} \\ &= \sum_{Y \in S} \text{sgn}(Y) \cdot 2^{\sum_{e \in Y} w(e)} \\ &= \sum_{Y \in S} \text{sgn}(Y) \cdot 2^{w(Y)}. \end{aligned}$$

Die entscheidende Beobachtung ist nun, daß trotz der Tatsache $\mathrm{sgn}(Y) \in \{-1, +1\}$ für ein *eindeutig bestimmtes*, perfektes Matching *minimalen Gewichts* der Term $2^{w(Y)}$ nicht durch andere Terme dieser Form „ausgelöscht" werden kann.

Sei M_e diejenige Matrix, die aus M' entsteht, indem der Wert $2^{w(e)}$ für die Kante e ersetzt wird durch 0. Daraus ergibt sich nun folgender paralleler Algorithmus:

Berechne $\det(M')$;
for all e in M **in parallel do**
  Berechne $\det(M_e)$;
  if Übereinstimmung der Binärdarstellungen von $\det(M_e)$ und $\det(M')$
    an der niederwertigsten Bitposition, bei der mindestens in einer der
    beiden der Wert 1 auftritt
    then e ist nicht im perfekten Matching
    else e ist im perfekten Matching

Satz 4.13 *Gegeben sei ein bipartiter Graph mit mindestens einem perfekten Matching. Dann läßt sich ein solches von einem RNC^2-Algorithmus mit Wahrscheinlichkeit größer als $\frac{3}{4}$ konstruieren.*

Beweisskizze: Nach Mitteilung 4.5 lassen sich Determinanten mit NC^2 - Schaltnetzen berechnen. Die randomisierte Gewichtszuordnung, wie sie für die Anwendung des Isolationslemmas nötig ist, läßt sich leicht mittels eines RNC^1 - Schaltnetzes erledigen. Zusammen ergibt das, daß obiger Algorithmus mit einem RNC^2 - Schaltnetz implementierbar ist. $\square$

Bemerkung 4.14

- Das obige Ergebnis für bipartite Graphen ist auch auf allgemeine Graphen übertragbar.
- Das Verfahren ist ein Beispiel für das Paradigma *Isolierung und Aufbrechen von Symmetrien.*
- Es ist offen, ob es für das Problem auch einen (deterministischen!) NC-Algorithmus gibt.
- Literatur: K. Mulmuley, U. V. Vazirani and V.V. Vazirani: Matching is as easy as matrix inversion. *Combinatorica*, Vol. 7, 105-113, 1987.

4.2 Routing in Parallelrechnern

Das Routing-Problem, das heißt das Verschicken von Nachrichtenpaketen zwischen den einzelnen Prozessoren eines Parallelrechners, ist von grundlegender Bedeutung beim parallelen Rechnen.

4.2.1 Grundlagen

Wir betrachten *Parallelrechner* als Netzwerke von parallel arbeitenden Prozessoren. Diese lassen sich mithilfe von Graphen modellieren, bei welchen die Knoten Prozessoren und die Kanten Verbindungskanäle zwischen Prozessoren entsprechen. Dabei nehmen wir an, daß die Prozessoren in *synchroner* Weise arbeiten, insbesondere also die Kommunikation zwischen miteinander verbundenen Prozessoren in synchronen Schritten abläuft. Jeder Verbindungskanal kann in einem Zeitschritt eine „elementare Nachricht“ in jeweils jede der beiden Richtungen transportieren. Diese Elementarnachrichten nennen wir *Pakete*. Während eines Berechnungsschrittes kann jeder Prozessor höchstens ein Paket an jeden seiner Nachbarn schicken. In der Regel besitzen die Prozessoren eindeutig bestimmte Nummern zu ihrer Identifikation.

Nachfolgend studieren wir das *Permutations-Routing-Problem*:
Gegeben: Ein Parallelrechner, bei dem jeder Prozessor ein Paket erhält, das für einen anderen Prozessor bestimmt ist. Dabei soll jeder Prozessor Start und Ziel von jeweils genau einem Paket sein.
Aufgabe: Führe den Datentransport durch, so daß jedes Paket möglichst schnell von seinem Start- zu seinem Zielprozessor gelangt.

Um von seinem Start zu seinem Ziel zu gelangen, durchläuft jedes Paket eine gewisse *Route* durch das Netzwerk, ausgehend vom Start- und endend im Zielprozessor. Dabei kann es vorkommen, daß ein Paket unterwegs warten muß, da mehrere Pakete gleichzeitig über einen Verbindungskanal gehen wollen. Dazu nehmen wir an, daß jeder Netzwerkknoten eine *Warteschlange* besitzt, in der solche Pakete zwischengespeichert werden können. Wie sich zeigen wird, ist es für unsere Zwecke im wesentlichen egal, welche „Warteschlangendisziplin“ gewählt wird, das heißt in welcher Reihenfolge die Pakete gespeichert und weitergeschickt werden. Der Einfachkeit halber nehmen wir an, die Warteschlange sei gemäß FIFO („first in, first out“) organisiert.

In der Praxis ist man im allgemeinen an sehr einfachen (Implementierbarkeit!) Routing-Verfahren interessiert. Einfache Verfahren sind *datenunabhängige („oblivious“)* Routing-Algorithmen: Seien die Prozessoren von 1 bis N durchnumeriert. Sei weiter v_i das Paket, das im Prozessor $i \in \{1, \ldots, N\}$ startet, und sei $d(i)$ sein Ziel. Dann hängt bei einem datenunabhängigen Algorithmus für jedes v_i die gewählte Route *ausschließlich*

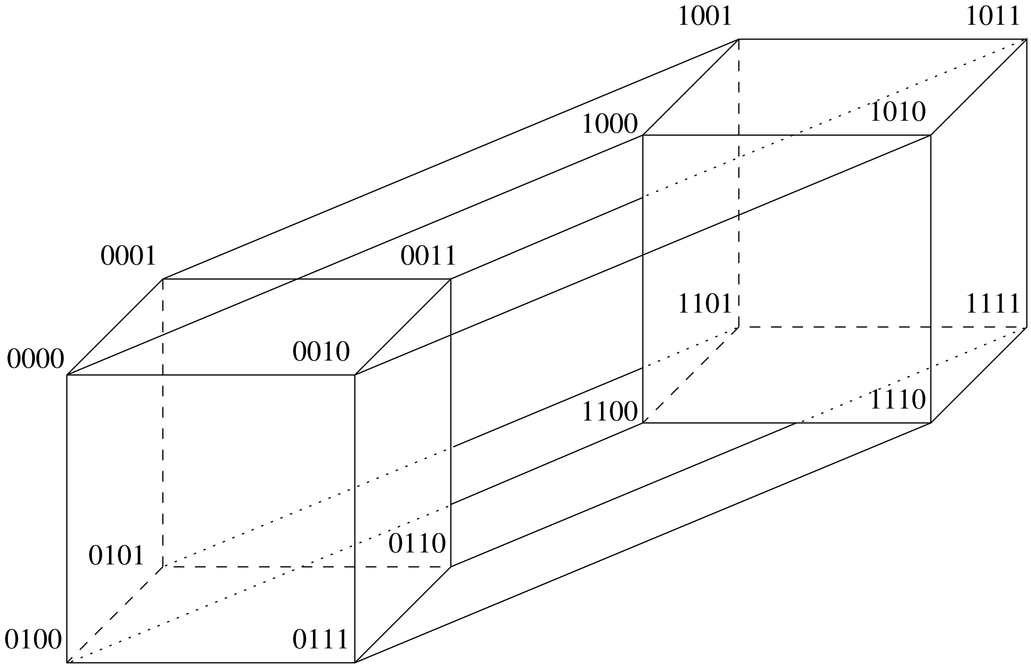


Abbildung 3: Ein vierdimensionaler Hyperwürfel.

von $d(i)$ ab, und nicht zum Beispiel von einem $d(j)$ für $i \neq j$. Das heißt die Kommunikations-Hardware kann für jedes Paket und jeden Knoten im Netzwerk *allein* aufgrund des Starts und des Ziels eines Paketes bestimmen, auf welchem Verbindungskanal das Paket weitergeschickt werden soll.

In diesem Abschnitt befassen wir uns mit einer speziellen Parallelrechner-Topologie, der des *Hyperwürfels (Hypercube)*. Ein (Boolescher) Hyperwürfel besteht aus 2^n Knoten — jeder Knoten wird also mit einem n-Bitstring numeriert. Zwei Knoten $(i_0 \ldots i_{n-1})$ und $(j_0 \ldots j_{n-1}) \in \{0,1\}^n$ sind genau dann miteinander verbunden, wenn sie sich in genau einem Bit unterscheiden.

Beispiel 4.15 Abbildung 3 zeigt einen vierdimensionalen Hyperwürfel.

Auf Borodin und Hopcroft (1985) geht folgende untere Schranke für deterministisches, datenunabhängiges Permutations-Routing zurück:

Mitteilung 4.16 *Für jeden deterministischen, datenunabhängigen Permutations-Routing-Algorithmus auf einem Netzwerk der Größe N, bei welchem jeder Knoten höchstens d Nachbarn hat, gibt es eine Instanz des Permutations-Routing-Problems, so daß der Algorithmus $\Omega(\sqrt{N/d})$ Schritte benötigt.*

Im Spezialfall des Hyperwürfels liefert Mitteilung 4.16 eine untere Schranke von $\Omega(\sqrt{N/n})$ für jeden deterministischen Algorithmus. Der nächste Unterabschnitt präsentiert einen sehr einfachen randomisierten, datenunabhängigen Algorithmus mit erwarteter Laufzeit $O(n) = O(\log N)$.

4.2.2 Randomisiertes, datenunabhängiges Permutations-Routing

Der Algorithmus von Valiant, der in zwei Phasen arbeitet, benutzt in beiden Phasen folgendes, *datenunabhängiges* Verfahren zur Festlegung der Transportrouten: Paket v_i wird jeweils über denjenigen Transportkanal eines Knoten (weiter-)transportiert, so daß die am weitesten links stehende Bitposition, in der sich die augenblickliche Position im Netzwerk und die Zielposition unterscheiden, korrigiert wird.
Beispiel: Die Route von Startknoten (1011) zum Zielknoten (0000) geht erst über den Zwischenknoten (0011) und dann über Zwischenknoten (0001).

Der randomisierte, datenunabhängige Permutations-Routing-Algorithmus arbeitet wie folgt:

Phase 1: Für jedes Paket v_i wähle per Zufall ein Ziel $r(i) \in \{1, \ldots, N\}$. Schicke Paket v_i zum Knoten $r(i)$.
Phase 2: Von dem Zwischenziel $r(i)$ schicke nun jedes Paket v_i zu seinem endgültigen Ziel $d(i)$.

Der Zeitbedarf des Algorithmus setzt sich zusammen aus der Distanz vom Ziel ($\leq n$) und der Anzahl der Zeiteinheiten, in denen ein Paket nicht weitergeschickt werden kann, also in einer Warteschlange verbringen muß. Wir analysieren dies zunächst für Phase 1.

Lemma 4.17 *Gehe die Route von v_i in Phase 1 über die Kantensequenz $\rho_i = (e_1, \ldots, e_k)$ und sei S die Menge von von v_i verschiedenen Paketen, deren Route mindestens eine der Kanten in $\{e_1, \ldots, e_k\}$ passiert. Dann muß v_i höchstens $|S|$ Schritte in einer Warteschlange verbringen.*

Beweis: Wir sagen, ein Paket in S verlasse ρ_i zu einem bestimmten Zeitpunkt, falls es dann zum letzen Mal eine Kante von ρ_i passiert. Wir sagen, ein Paket, das zum Zeitpunkt t bereit ist, e_j zu passieren, habe Verzögerung $t - j$. Die Gesamtverzögerung des Pakets v_i ist dann seine Verzögerung, wenn es e_k passiert. Wir zeigen nachfolgend, daß jeder Zeitpunkt, zu dem sich die Verzögerung von v_i um eins vergrößert, einem anderen Element von S zugeschrieben werden kann.
Behauptung: Falls die Verzögerung von v_i um 1 von l auf den Wert $l + 1$ erhöht wird, so gibt es ein Paket in S, das ρ_i mit Verzögerung l verläßt.

Da ein Paket, das ρ_i verläßt, „offensichtlich" nie wieder eine Kante in ρ_i passiert (dank unserer Routenfestlegung) folgt mit obiger Behauptung und dem zuvor Gesagten das Lemma. Es bleibt also, obige Behauptung zu beweisen.

Zunächst gilt: Falls sich die Verzögerung von v_i von l auf $l+1$ erhöht, so muß es mindestens ein Paket aus S geben, das die selbe Kante wie v_i zum selben Zeitabschnitt benutzen will. Also enthält S mindestens ein Paket mit Verzögerung l. Sei nun t' der letzte Zeitpunkt, zu dem irgendein Paket in S Verzögerung l hat. Das heißt, es gibt ein Paket p, das zum Zeitpunkt t' bereit ist, der Kante $e_{j'}$ zu folgen und es gilt $t' - j' = l$. Wir zeigen, daß es ein Paket in S gibt, das ρ_i zum Zeitpunkt t' verläßt. Da p bereit ist, die Kante $e_{j'}$ zum Zeitpunkt t' zu passieren, muß es ein Paket q (womöglich p selbst) in S geben, das $e_{j'}$ zum Zeitpunkt t' passiert. Würde nun q die Route ρ_i nicht zum Zeitpunkt t' verlassen, so würde es damit im Widerspruch zur Maximalität von t' ein Paket geben, das die Kante $e_{j'+1}$ zum Zeitpunkt $t'+1$ mit Verzögerung l passiert. Diesem q schreiben wir nun die Zunahme der Verzögerung von v_i von l auf $l+1$ zu. Da q die Route ρ_i verläßt, kann ihm nie mehr eine solche Zunahme zugeschrieben werden, und die Behauptung folgt. $\square$

Lemma 4.17 ist der Hauptbaustein, um für den randomisierten Algorithmus eine erwartete Laufzeit von $O(n)$ zu beweisen. Es ergibt, daß wir „nur" noch die Anzahl der Pakete abschätzen müssen, deren jeweilige Route mindestens eine Kante gemeinsam mit e_i hat, um eine Abschätzung für die Gesamtverzögerung zu erhalten. Dafür benötigen wir noch folgende Definition von Zufallsvariablen H_{ij} für Routen ρ_i und ρ_j mit $1 \leq i, j \leq N$:

$$H_{ij} = \begin{cases} 1 & : \text{ falls } \rho_i \text{ und } \rho_j \text{ mindestens eine gemeinsame Kante haben,} \\ 0 & : \text{ sonst.} \end{cases}$$

Offensichtlich ist die Gesamtverzögerung des Pakets v_i durch $\sum_{j=1}^{N} H_{ij}$ beschränkt. Da in der ersten Phase die Ziele $r(i)$ zufällig und für verschiedene i unabhängig voneinander gewählt wurden, sind somit auch die H_{ij} in $\sum_{j=1}^{N} H_{ij}$ voneinander unabhängige 0-1-Zufallsvariablen für $i \neq j$.

An dieser Stelle (und nicht nur an dieser) sind Chernoff-Schranken sehr hilfreich, um $\sum_{j=1}^{N} H_{ij}$ von oben abzuschätzen. Wir geben sie in einer für unsere Zwecke günstigen Form an:

Mitteilung 4.18 *(Chernoff-Schranken)*
Seien $X_1, X_2, \ldots, X_m$ unabhängige 0-1-Zufallsvariablen mit $\mathbf{Pr}[X_i = 1] = p_i$ und $\mathbf{Pr}[X_i = 0] = 1 - p_i$ für $i \in \{1, \ldots, m\}$. Weiter sei $X := \sum_{i=1}^{m} X_i$ und $\mu := E[X] = \sum_{i=1}^{m} p_i$. Dann gilt für $\delta_1 > 0$ und $0 < \delta_2 \leq 1$:

$$\mathbf{Pr}[X > (1+\delta_1)\mu] \quad < \quad (e/(1+\delta_1))^{(1+\delta_1)\mu},$$

$$\mathbf{Pr}[X < (1-\delta_2)\mu] \quad < \quad \exp(-\mu\delta_2^2/2).$$

Mithilfe der Chernoff-Schranken ließe sich also $\sum_{j=1}^{N} H_{ij}$ abschätzen, falls der Erwartungswert dieser Summe bekannt wäre. Daher suchen wir zunächst nach einer Abschätzung für $E[\sum_{j=1}^{N} H_{ij}]$.

Definiere dazu für eine Kante e im Hyperwürfel die Zufallsvariable $T(e)$ als die Anzahl der Routen, die über e gehen. Für eine beliebige Route $\rho_i = (e_1, e_2, \ldots, e_k)$ mit $k \leq n$ gilt dann

$$\sum_{j=1}^{N} H_{ij} \leq \sum_{l=1}^{k} T(e_l)$$

und deshalb

$$E[\sum_{j=1}^{N} H_{ij}] \leq E[\sum_{l=1}^{k} T(e_l)]. \quad (*)$$

Aufgrund von Symmetrieüberlegungen folgt leicht $E[T(e_l)] = E[T(e_m)]$ für zwei beliebige Kanten e_l und e_m im Hyperwürfel. Da die erwartete Anzahl der Kanten, die von einem Paket v_i passiert werden, offenbar $n/2$ ist, folgt per Summation über die Anzahl aller Pakete, daß die erwartete aller in Phase 1 stattfindenden Kantenpassierungen gleich $Nn/2$ ist. Da es im Hyperwürfel insgesamt $Nn/2$ Kanten gibt, folgt mit obigem also $E[T(e)] = 1$ für jede Kante e. Mit $(*)$ folgt somit

$$E[\sum_{j=1}^{N} H_{ij}] \leq k \leq n.$$

Setzen wir dies in die erste Ungleichung von Mitteilung 4.18 ein, so erhalten wir

$$\mathbf{Pr}[\sum_{j=1}^{N} H_{ij} > (1+5)\frac{n}{2}] < \left(\frac{e}{1+5}\right)^{(1+5)n} < 2^{-6n}.$$

Zu beachten ist hier, daß die Chernoff-Schranke *nicht* auf $\sum_{l=1}^{k} T(e_l)$ angewendet werden kann, da die Zufallsvariablen $T(e_l)$ *nicht unabhängig* und außerdem keine 0-1-Zufallsvariablen sind!

Wir haben somit gezeigt, daß die von v_i erlittene Verzögerung höchstens mit einer Wahrscheinlichkeit von 2^{-6n} den Wert $6n$ übersteigt. Da es ingesamt $N = 2^n$ Pakete gibt, folgt somit, daß die Wahrscheinlichkeit, daß mindestens ein Paket in Phase 1 eine Verzögerung größer als $6n$ hat, kleiner als $2^n \cdot 2^{-6n} = 2^{-5n}$ ist. Zur Verzögerung muß jetzt nur noch n für die maximale Routenlänge hinzuaddiert werden.

Dies beendet die Analyse von Phase 1. Glücklicherweise läßt sich Phase 2 als Phase 1 betrachten, welche „rückwärts durchlaufen“ wurde. Deshalb läßt sich für Phase 2 die gleiche Analyse wie für Phase 1 anwenden. Insgesamt erhalten wir:

Satz 4.19 *Mit Wahrscheinlichkeit mindestens* $1 - (1/N)$ *erreicht jedes Paket beim randomisierten, datenunabhängigen Permutations-Routing-Algorithmus sein Ziel in höchstens* $14n$ *Schritten.*

Beweis: Nach der vorangehenden Analyse ist die Wahrscheinlichkeit, daß ein Paket in $14n$ Schritten sein Ziel nicht erreicht höchstens $2 \cdot 2^{-5n} < 2^{-n} = 1/N$ für $n \geq 1$. Dabei setzen wir voraus, daß sich die beiden Phasen nicht überlappen, was leicht mittels einer Uhr erreicht werden kann, die bis $7n$ zählt, bevor jeder Prozessor Phase 2 startet. □

Korollar 4.20 *Die erwartete Laufzeit des randomisierten, datenunabhängigen Permutations-Routing-Algorithmus ist* $15n$.

Beweis: Mit Wahrscheinlichkeit mindestens $(N-1)/N$ benötigt der Algorithmus höchstens $14n$ Schritte. Die Höchstschrittanzahl ist offensichtlich beschränkt durch Nn und wird höchstens mit Wahrscheinlichkeit $1/N$ erforderlich. Somit ist eine obere Schranke für den Erwartungswert gegeben durch

$$14n \cdot \frac{N-1}{N} + Nn \cdot \frac{1}{N} \leq 15n.$$

□

Ein Vergleich von Korollar 4.20 mit Mitteilung 4.16 zeigt nochmals die Güte des randomisierten im Vergleich zu deterministischen Algorithmen für datenunabhängiges Permutations-Routing.

Bemerkung 4.21

- Der vorgestellte Algorithmus fällt unter die Paradigmen *zufällige Umordnung* beziehungsweise *Lastbalancierung*.
- Aufgrund ihrer Schärfe sind Chernoff-Schranken ein häufig eingesetztes Werkzeug bei Entwurf und Analyse (randomisierter) Algorithmen. Einen guten Überblick bietet: T. Hagerup and C. Rüb: A guided tour of Chernoff bounds. *Information Processing Letters*, Vol. 33, 305-308, 1990.

- Mitteilung 4.16 geht zurück auf: A. Borodin and J. E. Hopcroft: Routing, merging and sorting on parallel models of computation. *Journal of Computer and Systems Sciences*, Vol. 30, 130-145, 1985 und C. Kaklamanis, D. Krizanc and T. Tsantilas: Tight bounds for oblivious routing in the hypercube. *In Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, 31-36, 1991.

- Der präsentierte Algorithmus wurde von Valiant vorgestellt und analysiert: L. G. Valiant A scheme for fast parallel communication. *SIAM Journal on computing*, Vol. 11, 350-361, 1982. Eine vereinfachte Analyse findet sich in L. G. Valiant and G.J. Brebner: Universal schemes for parallel communication. *In Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, 263-277, May 1981.

4.2.3 Verminderung der Zufallsbits — ein existentieller Beweis[7]

Um die erwartete Laufzeit von $15n$ zu erreichen, benutzte der Algorithmus des vorigen Unterabschnitts Nn Zufallsbits. Wir untersuchen nun die Existenz eines Algorithmus, der mit deutlich weniger Zufallsbits auskommt.

Wieviele Zufallsbits sind nötig, um einen randomisierten, datenunabhängigen Permutations-Routing-Algorithmus für Hyperwürfel mit Laufzeit $O(n)$ zu gewinnen? Nach Mitteilung 4.16 ist klar, daß dies nicht mit einem deterministischen Algorithmus zu erreichen ist.

Für nachfolgende Zwecke ist es hilfreich, einen randomisierten Algorithmus als eine Wahrscheinlichkeitsverteilung über einer Menge deterministischer Algorithmen zu betrachten. So kann jeder randomisierte, datenunabhängige Algorithmus als eine Menge von Paaren $\{(A_1, p_1), (A_2, p_2), \ldots, (A_r, p_r)\}$ gedacht werden, wobei A_i ein deterministischer Algorithmus ist und p_i die Wahrscheinlichkeit, daß wir A_i in einem Lauf des randomisierten Algorithmus auswählen. Natürlich gilt $\sum_{i=1}^{r} p_i = 1$.

Satz 4.22 *Jeder randomisierte, datenunabhängige Algorithmus für Permutations-Routing auf dem Hyperwürfel mit $N = 2^n$ Knoten, welcher k Zufallsbits benutzt, benötigt eine erwartete Laufzeit von $\Omega(2^{-k}\sqrt{N/n})$.*

Beweis: Wie zuvor ausgeführt, betrachten wir jeden randomisierten, datenunabhängigen Algorithmus als eine Wahrscheinlichkeitsverteilung über einer Menge deterministischer Algorithmen. Da nur k Zufallsbits benutzt

[7] Der Inhalt dieses Unterabschnittes wurde im Rahmen der Vorlesung im Sommersemester 1997 nur noch im Überblick präsentiert.

werden, wird mindestens einer dieser deterministischen Algorithmen mit Mindestwahrscheinlichkeit 2^{-k} gewählt. Dieser sei A_1. Nach Mitteilung 4.16 gibt es eine Instanz des Permutations-Routing-Problems, für die A_1 mindestens $\Omega(\sqrt{N/n})$ Schritte benötigt. Arbeitet der randomisierte Algorithmus auf dieser Instanz, so benötigt er also mit Wahrscheinlichkeit 2^{-k} mindestens $\Omega(\sqrt{N/n})$ Schritte und daher ist die erwartete Laufzeit für den randomisierten Algorithmus mindestens $\Omega(2^{-k}\sqrt{N/n})$. □

Korollar 4.23 *Jeder randomisierte, datenunabhängige Algorithmus für Permutations-Routing auf dem Hyperwürfel mit $N = 2^n$ Knoten muß $\Omega(n)$ Zufallsbits benutzen, um eine erwartete Laufzeit von $O(n)$ zu erreichen.*

Mithilfe der *probabilistischen Methode* können wir zumindest die Existenz eines randomisierten Algorithmus beweisen, der die in Korollar 4.23 angegebenen Schranken erreicht.

Satz 4.24 *Es existiert ein randomisierter, datenunabhängiger Algorithmus für Permutations-Routing auf dem Hyperwürfel mit $N = 2^n$ Knoten, welcher $3n$ Zufallsbits benutzt und eine erwartete Laufzeit von $18n$ hat.*

Beweis: Wir nennen eine Menge $\mathcal{A} = (A_1, A_2, \ldots, A_t)$ von deterministischen, datenunabhängigen Permutations-Routing-Algorithmen auf einem N-Knoten-Hyperwürfel ein *effizientes N-Schema,* falls für jede Instanz des Permutations-Routing-Problems gilt, daß die erwartete Anzahl von Schritten unter Benutzung eines per Zufall aus $\mathcal{A}$ gewählten Algorithmus höchstens $18n$ ist. Wir zeigen nun, daß für jedes $N = 2^n$ ein effizientes N-Schema der Größe $t = N^3$ existiert. Daraus folgt die Behauptung, da nur $\log t$ Zufallsbits für die Auswahl eines dieser Algorithmen benötigt werden.

Der im vorigen Unterabschnitt vorgestellte, randomisierte Permutations-Routing-Algorithmus von Valiant repräsentiert N^N verschiedene deterministische Algorithmen, kann also als effizientes N-Schema der Größe $t = N^N$ interpretiert werden. Andererseits gibt es in einem N-Knoten-Netzwerk $N!$ verschiedene Instanzen des Permutations-Routing-Problems. Sei π_i mit $1 \leq i \leq N!$ eine solche Instanz. Wir nennen einen deterministischen Algorithmus A_j des besagten Schemas *gut* für π_i, wenn er π_i in höchstens $14n$ Schritten erfolgreich durchführt, andernfalls heißt er *schlecht.* Dank Satz 4.19 wissen wir, daß für beliebiges π_i höchstens ein Anteil von $1/N$ der insgesamt N^N Algorithmen schlecht ist.

Wir betrachten folgendes Zufallsexperiment: Wähle N^3 Indizes $i_1, i_2, \ldots, i_{N^3}$ per Zufall aus der Menge $\{1, 2, \ldots, N^N\}$. Wir zeigen, daß die Menge $\mathcal{A} :=$

$(A_{i_1}, A_{i_2}, \ldots A_{i_{N^3}})$ ein effizientes N-Schema mit Wahrscheinlichkeit größer als Null ist (und erhalten daraus unmittelbar einen Beweis für seine Existenz).

Die erwartete Anzahl von schlechten Algorithmen für eine Instanz π_i in dem N-Schema $\mathcal{A}$ ist nach dem zuvor Gesagten höchstens $N^3 \cdot (1/N) = N^2$. Definiere die charakteristische Zufallsvariable X_j gemäß

$$X_j := \begin{cases} 1: & \text{falls } A_{i_j} \text{ schlecht ist,} \\ 0: & \text{sonst.} \end{cases}$$

Somit gilt $E[\sum_{j=1}^{N^3} X_j] \leq N^2$. Anwendung der Chernoff-Schranke (Mitteilung 4.18, Teil 1) liefert (mit $\delta_1 = 3$)

$$\mathbf{Pr}[\sum_{j=1}^{N^3} X_j > 4N^2] < (e/4)^{4N^2} < e^{-N^2},$$

d.h. die Wahrscheinlichkeit, daß mehr als $4N^2$ der Algorithmen in $\mathcal{A}$ schlecht sind, ist kleiner als $\exp(-N^2)$. Sei nun B_i das *schlechte Ereignis*, daß mehr als $4N^2$ Algorithmen in $\mathcal{A}$ schlecht für π_i sind. Dann gilt für $N \geq 4$

$$\begin{aligned} \mathbf{Pr}[\bigcup_{i=1}^{N!} B_i] &\leq \sum_{i=1}^{N!} \mathbf{Pr}[B_i] \\ &\leq N! \cdot e^{-N^2} \\ &< 1, \end{aligned}$$

wobei die letzte Ungleichung aus der *Stirlingschen Formel*

$$N! = \sqrt{2\pi N} \left(\frac{N}{e}\right)^N \left(1 + \frac{1}{12n} + O\left(\frac{1}{N^2}\right)\right)$$

folgt. Wir haben also gezeigt, daß eine Teilmenge von N^3 Algorithmen von $\{A_1, A_2, \ldots, A_{N^N}\}$ derart *existiert*, daß höchstens $4N^2$ der Algorithmen in der Teilmenge schlecht sind für beliebiges π_i. Diese Teilmenge bildet ein effizientes N-Schema: Auf beliebiger Instanz π_i transportiert ein zufällig aus dieser Teilmenge gewählter Algorithmus mit Wahrscheinlichkeit $1 - (4N^2/N^3)$ alle Pakete in Zeit höchstens $14n$ an ihr Ziel. Mit Wahrscheinlichkeit $4N^2/N^3$ benötigt er maximal Nn Schritte. Für den Erwartungswert der benötigten Laufzeit ergibt sich damit

$$14n \cdot \frac{N-4}{N} + Nn \cdot \frac{4}{N} \leq 18n.$$

□

An dieser Stelle ist es wichtig sich zu vergegenwärtigen, daß Satz 4.24 nur eine Existenzaussage liefert, also nicht zur konkreten Konstruktion des besagten Algorithmus taugt. Wir haben in Analogie zu Satz 3.20 und Abschnitt 3.3 wiederum eine *nicht-uniforme Konstruktion.*

Bemerkung 4.25

- Satz 4.24 fällt unter das Paradigma *probabilistische Methoden und Existenzbeweise.*
- Bislang konnte noch kein nur $O(n)$ Zufallsbits benutzender $O(n)$-Zeit-Algorithmus für das datenunabhängige Permutations-Routing-Problem auf 2^n-Hyperwürfeln *explizit* angegeben werden.
- Die beste explizite Konstruktion liefert einen (uniformen) Algorithmus, der $O(n^2)$ Zufallsbits benötigt und $O(n)$ erwartete Laufzeit hat.
- Literatur: D. Peleg and E. Upfal: A time-randomness tradeoff for oblivious routing. *SIAM Journal on Computing*, Vol. 19, 256-266, 1990.

A Etwas Wahrscheinlichkeitstheorie

In der Vorlesung beschäftigen wir uns lediglich mit Wahrscheinlichkeitsräumen, bei welchen es nur endlich viele mögliche Ereignisse gibt.

Definition A.1 Ein (endlicher) *Wahrscheinlichkeitsraum* besteht aus einem endlichen Ereignisraum Ω und einem Wahrscheinlichkeitsmaß $\mathbf{Pr}$.
Ein endlicher *Ereignisraum* Ω ist eine endliche Menge von elementaren Ereignissen. Eine Teilmenge $\mathcal{E} \subseteq \Omega$ nennt man ein *Ereignis*.
Ein *Wahrscheinlichkeitsmaß* $\mathbf{Pr} : \mathcal{P}(\Omega) \rightarrow \mathbf{R}$ ist eine Funktion, die die folgenden Bedingungen erfüllt:

1. $\forall \mathcal{E} \subseteq \Omega, 0 \leq \mathbf{Pr}[\varepsilon] \leq \mathbf{1}$.
2. $\mathbf{Pr}[\Omega] = 1$.
3. $\forall \mathcal{E} \subseteq \Omega, \mathbf{Pr}[\mathcal{E}] = \sum_{\omega \in \mathcal{E}} \mathbf{Pr}[\{\omega\}]$.

Definition A.2 Die *bedingte Wahrscheinlichkeit* von $\mathcal{E}_1$ unter $\mathcal{E}_2$ wird geschrieben als $\mathbf{Pr}[\mathcal{E}_1|\mathcal{E}_2]$ und errechnet sich durch $\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2]/\mathbf{Pr}[\mathcal{E}_2]$, wobei $\mathbf{Pr}[\mathcal{E}_2] > 0$ vorausgesetzt wird. Die bedingte Wahrscheinlichkeit $\mathbf{Pr}[\mathcal{E}_1|\mathcal{E}_2]$ entspricht der Wahrscheinlichkeit, daß ein Zufallsexperiment ein Ergebnis in $\mathcal{E}_1$ hat, wenn wir bereits wissen, daß es eines in $\mathcal{E}_2$ hat.

Definition A.3 Eine Menge von Ereignissen $\{\mathcal{E}_i : i \in I\}$ ist *unabhängig*, wenn für alle Teilmengen $S \subseteq I$ gilt: $\mathbf{Pr}[\bigcap_{i \in S} \mathcal{E}_i] = \prod_{i \in S} \mathbf{Pr}[\mathcal{E}_i]$.
Eine Menge von Ereignissen ist also unabhängig, wenn die Wahrscheinlichkeit, daß ein Experiment ein Ergebnis in der Menge $\mathcal{E}_i$, $i \in I$, hat, nicht davon abhängt, ob es ein Ergebnis in $\bigcup_{j \in S} \mathcal{E}_j$ hat, für ein beliebiges $S \subseteq I - \{i\}$.

Definition A.4 Eine Menge von Ereignissen $\{\mathcal{E}_i : i \in I\}$ ist *k-unabhängig*, wenn für alle Teilmengen $S \subseteq I$ mit $|S| \leq k$ gilt: $\mathbf{Pr}[\bigcap_{i \in S} \mathcal{E}_i] = \prod_{i \in S} \mathbf{Pr}[\mathcal{E}_i]$.
Eine Menge von Ereignissen ist *paarweise unabhängig*, wenn sie 2-unabhängig ist.

Definition A.5 Eine *Zufallsvariable* X ist eine reellwertige Funktion über einem Wahrscheinlichkeitsraum, $X : \Omega \rightarrow \mathbb{R}$.

Definition A.6 Die *Verteilungsfunktion* $F : \mathbb{R} \rightarrow [0, 1]$ für eine Zufallsvariable X ist definiert als $F_X(x) = \mathbf{Pr}[X \leq x]$.

Definition A.7 Für ein Ereignis $\mathcal{E} \subseteq \Omega$ ist die *charakteristische Zufallsvariable* X definiert als:

$$X(\omega) = \begin{cases} 1 : & : \quad \omega \in \mathcal{E}, \\ 0 : & : \quad \omega \notin \mathcal{E}. \end{cases}$$

Eine charakteristische Zufallsvariable X wird benutzt, um das Auftreten oder Nichtauftreten eines Ereignisses $\mathcal{E}$ zu kennzeichnen.

In Analogie zur Unabhängigkeit bei Ereignissen läßt sich die Unabhängigkeit von Zufallsvariablen definieren.

Definition A.8 Eine Menge $\{X_i : i \in I\}$ von Zufallsvariablen ist *unabhängig*, wenn für alle Teilmengen $S \subseteq I$ und für alle $\{x_i \in \mathbb{R} : i \in I\}$ gilt:
$\mathbf{Pr}[\forall i \in S \colon X_i = x_i] = \prod_{i \in S} \mathbf{Pr}[X_i = x_i]$.

Definition A.9 Eine Menge $\{X_i : i \in I\}$ von Zufallsvariablen heißt *k-unabhängig*, wenn für alle Teilmengen $S \subseteq I$, $|S| \leq k$, und für alle $\{x_i \in \mathbb{R} : i \in I\}$ gilt:
$\mathbf{Pr}[\forall i \in S \colon X_i = x_i] = \prod_{i \in S} \mathbf{Pr}[X_i = x_i]$.

Definition A.10 Der *Erwartungswert* einer Zufallsvariable X ist definiert als:
$\mathbf{E}[X] = \sum_{x \in \mathbb{R}} x \cdot \mathbf{Pr}[X = x]$.

Die folgenden zwei Punkte sind wichtige Eigenschaften von Erwartungswerten:

Mitteilung A.11 (Linearität des Erwartungswertes)
Für beliebige Zufallsvariablen X und Y gilt:
$\mathbf{E}[aX + bY] = a\mathbf{E}[X] + b\mathbf{E}[Y]$ *für Konstanten a und b.*

Mitteilung A.12 *Für beliebige, unabhängige Zufallsvariablen X und Y gilt:*
$\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$.

Definition A.13 Die *Varianz* einer Zufallsvariable X ist definiert als:
$\mathbf{var}[X] = \mathbf{E}[(X - \mathbf{E}[X]^2]$.

Mitteilung A.14 *Für beliebige Zufallsvariablen X gilt:*
$\mathbf{var}[X] = \mathbf{E}[X^2] - \mathbf{E}[X]^2$.

Mitteilung A.15 *Für beliebige unabhängige Zufallsvariablen X und Y gilt:*
$\mathbf{var}[X + Y] = \mathbf{var}[X] + \mathbf{var}[Y]$.

Zu beachten ist dabei, daß im Unterschied zu Erwartungswerten Mitteilung A.15 nicht mehr aufrecht erhalten werden kann, wenn die Zufallswerte abhängig voneinander sind.

Mitteilung A.16 *Für eine beliebige Zufallsvariable X und ein beliebiges $c \in \mathbb{R}$ gilt:*
$\mathbf{var}[cX] = c^2\mathbf{var}[X]$.

B Landau-Symbole, Asymptotik

Wachstumsvergleich von Funktionen $f, g : \mathbb{R}^+ \to \mathbb{R}^+$:

- f *wächst echt langsamer* als g (bzw. f geht schneller gegen 0 als g):

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

- f *wächst nicht schneller* als g:

$$f = O(g) :\Longleftrightarrow \limsup_{x\to\infty} \frac{f(x)}{g(x)} < \infty.$$

- $f = \Omega(g) \iff g = O(f)$.
- $f = \omega(g) \iff g = o(f)$.
- $f = \Theta(g) \iff f = O(g) \land g = O(f)$.

Beachte: $f = o(g)$ impliziert $f = O(g)$, und $f = \omega(g)$ impliziert $f = \Omega(g)$.

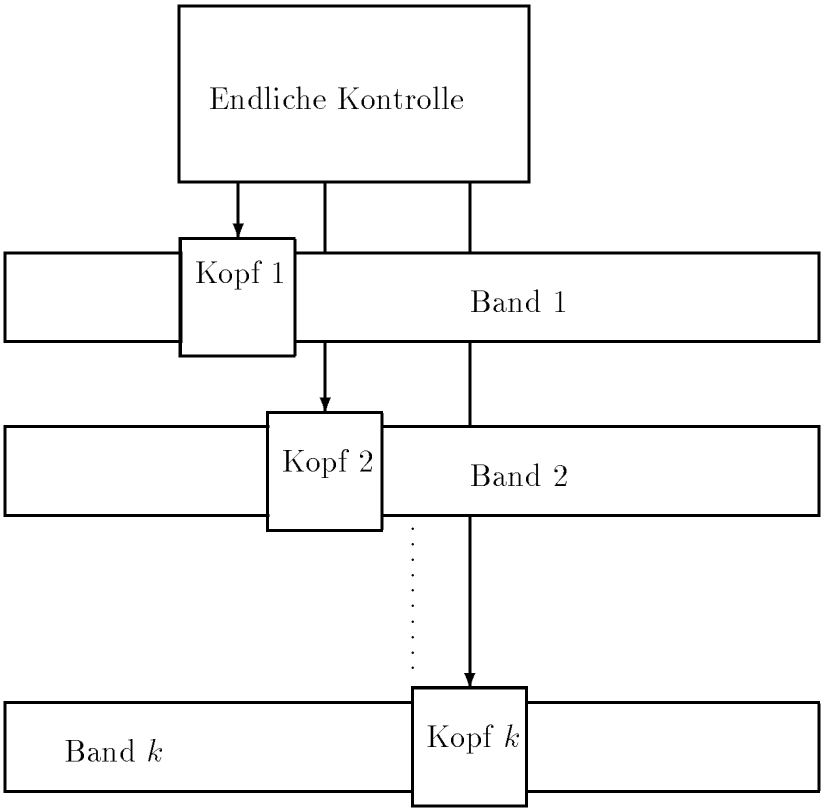
Einige Beispiele:

$$\begin{aligned}
2x &= o(x^2); \\
2x &= O(x); \\
\ln(x) &= o(\sqrt{x}); \\
\sqrt{x} &= o(x); \\
x^{\ln(\ln(x))} &= O(\ln(x)^{\ln(x)}); \\
\sqrt{x} &= o(x); \\
x^{\ln(\ln(x))} &= O(\ln(x)^{\ln(x)}); \\
\frac{1}{x} &= o(\frac{1}{\ln(x)}).
\end{aligned}$$

C Turingmaschinen und etwas Komplexitätstheorie

C.1 Die Turingmaschine

- k-Band-TM



Übergangsrelation:

$$\Delta : (q, a_1, a_2, \ldots, a_k) \mapsto (q', b_1, \ldots, b_k, m_1, \ldots, m_k).$$

Anfangskonfiguration: Das Eingabewort v auf Band 1 und Bänder 2 $\ldots k$ leer. **Ein** Turingmaschinenschritt:

- Lesen auf allen Bändern
- Abhängig vom gelesenen und vom alten Zustand:
  - Neuen Zustand einnehmen
  - Schreiben auf allen Bändern
  - Bewegen der Köpfe

- Eine TM *akzeptiert* ihre Eingabe , wenn sie aus der Anfangskonfiguration in endlich vielen Schritten einen akzeptierenden Endzustand erreicht, sonst *lehnt* sie ihre Eingabe ab.
- Die Anzahl dieser Schritte, also die Länge dieser Berechnung, entspricht der zur Berechnung benötigten *Zeit.*
- Die Anzahl aller je erreichten Speicherzellen entspricht dem zur Berechnung benötigten *Speicherplatz.*

C.2 Platzklassen

Definition C.1 $S : \mathbb{N} \to \mathbb{N}, L \subseteq \Sigma^*$ und $k \in \mathbb{N}$.

- Eine TM heißt *$S(n)$-platzbeschränkt* gdw. für jedes Eingabewort der Länge n die TM höchstens $\lceil S(n) \rceil$ Speicherplätze benutzt, bevor sie hält, wobei das (read-only-)Eingabewort nicht mitgezählt wird.
- L heißt *„von einer S-platzbeschränkten k-Band-Maschine erkennbar"* gdw. es eine k-Band-Maschine M derart gibt, daß L von M akzeptiert wird, also $L = L(M)$, und für jedes $v \in \Sigma^*$ der Länge n die Berechnung von M auf v höchstens $S(n)$ Speicherzellen benötigt.
- $DSPACE_k(S)$ bezeichne die Menge aller von S-platzbeschränkten k-Band-Maschinen erkennbaren Mengen.
  $DSPACE(S)$ sei die Vereinigung über k aller $\mathrm{DSPACE}_k(S)$, also
  $\bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}_k(S)$.

„Natürlich": $\mathrm{DSPACE}_1(S) \subseteq \mathrm{DSPACE}_2(S) \subseteq \ldots \subseteq \mathrm{DSPACE}(S)$.

Mitteilung C.2 (Bandreduktion) *Für beliebiges S gilt*

$$DSPACE_1(S) = DSPACE(S).$$

Mitteilung C.3 (Bandkompression)

$$\forall c > 0 : DSPACE(\lceil c \cdot S(n) \rceil) = DSPACE(S(n))$$

wobei $\lceil x \rceil := \min\{n \in \mathbb{N} |\ x \leq n\}$.

Definition C.4 PSPACE $:= \bigcup_{p\in POLY} \mathrm{DSPACE}(p(n))$, wobei POLY die Menge aller Polynome bezeichnet.

C.3 Zeitklassen

Definition C.5 Eine TM heißt *T(n)-zeitbeschränkt* gdw. für jedes Eingabewort der Länge n die TM höchstens $\max(n+1, \lceil T(n) \rceil)$ Schritte macht, bevor sie hält.

$\mathrm{DTIME}_k(T)$ bezeichne die Klasse aller von T-zeitbeschränkten k-Band-Maschinen erkennbaren Mengen. DTIME(T) sei die Vereinigung aller $\mathrm{DTIME}_k(T)$, also $\mathrm{DTIME}(T) = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}_k(T)$.

Analog zu Mitteilung C.3 gilt:

Satz C.6 (Lineare Beschleunigung) *$T : \mathbb{N} \to \mathbb{N}$ erfülle $\frac{1}{T(n)} = o(\frac{1}{n})$. Dann gilt:*

$$\forall k > 1 \forall c > 0 \; DTIME_k(\lceil c \cdot T \rceil) = DTIME_k(T);$$

und daher auch

$$DTIME(\lceil c \cdot T \rceil) = DTIME(T).$$

Analog zu Mitteilung C.2 nur in schwächerer Form gilt:

Mitteilung C.7 (Bandreduktion) *$T : \mathbb{N} \mapsto \mathbb{N}$.*

a) $DTIME(T) \subseteq DTIME_1(T^2)$;

b) (Hennie & Stearns) $DTIME(T) \subseteq DTIME_2(T \cdot \log(T))$.

Definition C.8 *Wichtige Zeitklassen:*

- $P := \bigcup_{p \in POLY} \mathrm{DTIME}(p(n))$,
- $E := \mathrm{DTIME}(2^{O(n)})$,
- $EXPTIME := \bigcup_{p \in POLY} \mathrm{DTIME}(2^{p(n)})$.

C.4 Zeit und Platz im Zusammenhang

Wie leicht zu sehen ist, gilt

$$\mathrm{DTIME}(T) \subseteq \mathrm{DSPACE}(T).$$

Es läßt sich noch zeigen:

Mitteilung C.9 (Hopcroft, Paul & Valiant)

$$DTIME(T) \subseteq DSPACE\left(\frac{T}{\log(T)}\right).$$

Umgekehrt gilt:

Mitteilung C.10 $S : \mathbb{N} \to \mathbb{N}$.

$$DSPACE(S(n)) \subseteq \bigcup_{c \geq 1} DTIME(c^{S(n)})$$

C.5 Modell: Nichtdeterminismus

Beim Konzept Nichtdeterminismus darf die Maschine „raten“, d.h. die Auswahl zwischen mehreren Folgeschritten treffen.
Akzeptierung $\equiv$ Existenz einer Erfolgsrechnung. So hat man die Möglichkeit des Durchprobierens von Fällen.

Nichtdeterminismus ist kein reales Rechnerkonzept!

- *Maschinenmodell (NTM):* Nichtdeterminismus in der Kontrolle
  - (Zweiweg-) Eingabeband (nur Lesen)
  - (Zweiweg-) Arbeitsbänder (Lesen+Schreiben)
  - Nichtdeterministische, endliche Kontrolle ($\approx$ Programm) u.a. mit akzeptierenden, ablehnenden und neutralen Zuständen.
- Eine *Berechnung* von M auf einem Wort v ist eine Folge von Konfigurationen (= Bandinhalte, Kopfstellung, Zustand) die durch 1-Schritt-Übergänge auseinander hervorgehen. Ist der Zustand der letzten Konfiguration akzeptierend (bzw. ablehnend), so heißt die Berechnung *akzeptierend* (bzw. *ablehnend*) .
- M *akzeptiert* genau ein Wort v, wenn es eine akzeptierende Berechnung auf v gibt (Existenzialkonzept).

C.6 Nichtdeterministische Komplexitätsklassen

Definition C.11
a) Eine NTM M heißt *S-platzbeschränkt* (bzw. *T-zeitbeschränkt*) gdw. für alle Eingaben v und alle Berechnungen von M auf v M höchstens $S(|v|)$ Plätze benutzt (bzw. $T(|v|)$ Schritte macht).
b) NSPACE$_k(S)$ bezeichne die Menge aller von S-platzbeschränkten k-Band NTM'n erkennbaren Sprachen und NSPACE(S) deren Vereinigung über k.
c) Ganz entsprechend werden $NTIME_k(T)$ und $NTIME(T)$ definiert.

Mitteilung C.12 (Bandreduktion)

$$NSPACE_k(S) = NSPACE_1(S) = NSPACE(S).$$

Mitteilung C.13 (Bandkompression)

$$\forall_{c>0} : NSPACE(S) = NSPACE(\lceil c \cdot S \rceil).$$

Wie im deterministischen Fall gilt:

Mitteilung C.14 $NTIME(T) \subseteq NTIME_1(T^2)$.

Mitteilung C.15 (Lineare Beschleunigung)

$$\forall_{c>0} NTIME(T) = NTIME(\lceil c \cdot T \rceil),\ T \geq n,\ bzw.$$

$$NTIME_k(T) = NTIME_k(\lceil c \cdot T \rceil),\ k \geq 2.$$

Einige wichtige nichtdeterministische Klassen:

Definition C.16

- $NLOGSPACE := NSPACE(\log n)$
- $CS := NSPACE(n)$
- $NP := \bigcup_{p \in POLY} NTIME(p(n))$
- $NEXPTIME := \bigcup_{p \in POLY} NTIME(2^{(p(n))})$

Index