Ferien-Akademie

Sarntal / Südtirol
17. bis 29. September 1995

Universität Erlangen-Nürnberg - Technische Universität München

Kurs 2
Randomisierte und parallele Algorithmen

Dozenten: Univ. Prof. Dr. E. W. Mayr[†]
Univ. Prof. Dr. H.-J. Schneider[‡]

Betreuer: Klaus Kühnle[†]

Teilnehmer: Markus Erlacher[†]
Alexander Hall[†]
Arno Klein[‡]
Tobias Knopff[†]
Anja Löser[‡]
Dirk Reichelt[‡]
Markus Reiner[‡]
Andreas Sauter[‡]
Thomas Schickinger[†]
Christopher Schmid[†]
Sascha Uelpenich[‡]
Claudia Ungerer[†]
Josef Weidendorfer[†]
Ralf Werner[‡]

25. Juli 1996

[†] Technische Universität München

[‡] Universität Erlangen-Nürnberg

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Erlanger Vorträge: Parallele Algorithmen

Vortrag 1

Basisalgorithmen auf Listen

Markus Reiner
Universität Erlangen-Nürnberg
`msreiner@cip.informatik.uni-erlangen.de`

Vortrag 2

Basisalgorithmen auf Graphen

Andreas Sauter
Universität Erlangen-Nürnberg

Vortrag 3

Basisalgorithmen für Sortierprobleme

Dirk Reichelt
Universität Erlangen-Nürnberg
`dkreiche@cip.informatik.uni-erlangen.de`

Vortrag 4

PRAM - Modelle

Ralf Werner
Universität Erlangen-Nürnberg
Mathematisches Institut
`werner@mi.uni-erlangen.de`

Zusammenfassung

Jeder Algorithmus für einen Parallelrechner benötigt einen, auf dem er läuft. Im Laufe der Zeit entwickelte man verschiedene Modelle des Parallelrechners - allen gemeinsam ist, daß sie eine Weiterentwicklung der herkömmlichen RAM sind, genannt PRAM (Parallel RAM). Man unterscheidet diese theoretischen Modelle je nach Auflösung von Lese - und Schreibkonflikten. Alle diese Modelle basieren auf folgenden Grundvoraussetzungen:

4.1 PRAM - Modelle

- jede PRAM hat mehrere (i.a. identischen) Prozessoren
- jeder Prozessor besitzt einen eigenen Speicher

- jeder Prozessor hat Zugriff auf einen gemeinsamen Speicher
- alle Prozessoren laufen im gleichen Takt
- jeder Prozessor läuft im Zyklus Lesen - Rechnen - Schreiben

Bis auf gleichzeitiges Lesen und Schreiben im gemeinsamen Speicher entstehen keine Konflikte. Diese werden je nach Modell aufgelöst - wobei natürlich gleichzeitiges Lesen im Gegensatz zu Schreiben keinerlei Inkonsistenz verursacht. Diese Inkonsistenz muß nun vom jeweiligen Algorithmus aufgefangen werden. Man sollte allerdings berücksichtigen, daß auch gleichzeitiges Lesen schon eine starke Eigenschaft ist - z.B. wenn ein Eingabeparameter allen Prozessoren bekannt gemacht werden muß.

Bezeichnung	gleichzeitiges	
	Lesen	Schreiben
EREW (exclusive read, exclusive write)	nein	nein
CREW (concurrent read, exclusive write)	ja	nein
CRCW (concurrent read, concurrent write)	ja	(ja)

Bezeichnung	Auflösung des Schreibkonflikts
WRAM	gleichzeitiges (bitweises) Schreiben ist nur erlaubt, wenn alle Prozessoren das Bit setzen wollen
COMMON CRCW	gleichzeitiges Schreiben ist erlaubt, wenn alle Prozessoren das Gleiche in die Speicherzelle schreiben wollen
ARBITRARY - CRCW	der Algorithmus muß korrekt laufen, egal welcher Prozessor seinen Wert schreibt
PRIORITY - CRCW	die Prozessoren sind nach Prioritäten geordnet und der Prozessor mit der höchsten Priorität schreibt

4.2 Mächtigkeit der PRAM - Modelle

Man überlegt sich nun leicht, daß für die Mächtigkeit der Modelle folgende Reihenfolge gilt:

$$\text{EREW} \leq \text{CREW} \leq \text{CRCW}$$

genauer gilt:

$$\text{WRAM} \leq \text{COMMON} \leq \text{ARBITRARY} \leq \text{PRIORITY}$$

Dies sieht man dadurch, daß jeder Algorithmus, der z.B. auf einer COMMON läuft, ebenfalls (mind. in gleicher Zeit) auf einer PRIORITY läuft - höchstens noch mehr Algorithmen. Durch eine sehr genaue Analyse verschiedener Probleme (Berechnung des logischen Oder, Rangsucheproblem) kann man zeigen, daß in obiger Ungleichungskette jeweils ein '<' gilt. Diese Analysen sind aber sehr involviert und die zugehörigen Beweise ziemlich technisch. Wenn man nun aber weiß, daß zwischen den verschiedenen Modellen ein gewisser 'Abstand' herrscht, fragt man sich, wie groß dieser Abstand ist. Bisher sind folgende (im gewissen Sinne z.T optimale) Transformationsalgorithmen bekannt:

Transformation innerhalb von CRCW: PRIORITY → WRAM

	PRIORITY	WRAM
Prozessoren	p	$p^2 + p$
Speicherzellen	s	$p^2 + p + s$
Laufzeit	f(n)	a·f(n)

Die grundlegende Idee dieser Transformation ist das Anlegen einer p×p - Matrix (hier werden die p^2 Prozessoren benötigt), aus der dann jeder der p Prozessoren ablesen kann , ob und wohin er schreiben darf. DIeser Vorgang wird in jedem Schreibzyklus wiederholt, d.h. er benötigt statt einem Schritt nun (in diesem konkreten Algorithmus) vier Schritte.

Transformation von CRCW nach EREW: PRIORITY → EREW

	PRIORITY	EREW
Prozessoren	p	p
Speicherzellen	s	4p + s
Laufzeit	f(n)	O(f(n)·log(p))

Hierbei ist n die Eingabegröße des Problems und a irgendeine Konstante.

Dieser Algorithmus beruht auf der Idee, die Prozessoren, die in die gleiche Zelle schreiben wollen, nach Priorität zu orden. Dann schreibt nur jeweils der Prozessor mit der höchsten Priorität. Beim Lesen verfährt man analog, nur muß nun jedem Prozessor, der noch nicht gelesen hatte, der Inhalt der zu lesenden Zelle noch mitgeteilt werden. Diese Verbreitung erfolgt nach dem Schneeballprinzip. Der Zeitaufwand des Algorithmus rührt aus der Sortierung und der Verbreitung (jeweils O(log p)) her.

Literaturverzeichnis

[1] S.A. Cook, C. Dwork, R. Reischuk: *Upper and lower bounds for parallel random access machines without simultaneous writes*, SIAM Journ. Comput. 15 (1980), pp. 87 - 97

[2] L. Kučera: *Parallel computation and conflicts in memory access*, Information Processing Letters 14 (1982), pp. 93 - 96

[3] M. Snir: *On parallel searching*, SIAM Journ. Comput. 14 (1985), pp. 688 - 708

[4] U. Vishkin: *Implementation of simultaneous memory address access in models that forbid it*, Journal of Algorithms 4 (1983), pp. 45 - 50

[5] R.M. Karp, V. Ramachandran: *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science, pp. 869 - 941

Vortrag 5

Boolesche Funktionen

Sascha Uelpenich
Universität Erlangen-Nürnberg
`sauelpen@cip.informatik.uni-erlangen.de`

Vortrag 6

Matrixmultiplikation und ihre Anwendungen

Arno Klein
Universität Erlangen-Nürnberg
`aoklein@cip.informatik.uni-erlangen.de`

Zusammenfassung

Die Matrixmultiplikation (MM) stellt eine mathematische Grundoperation, sowohl im theoretischen Bereich, als auch bei numerischen Anwendungen dar. Die durch moderne Parallelrechner erreichten Rechengeschwindigkeiten für MM sind – trotz aller Anstrengungen – teils immer noch sehr mäßig: betrachtet man die Komplexität des Problems so ist klar, daß der Aufwand für MM zumindest mit quadratischem Faktor wächst; auf triviale Weise lassen sich zwei Matrizen mit $O(n^3)$[1] multiplizieren. Die Angabe des Aufwands bezieht sich dabei immer auf die Anzahl der benötigen Prozessoren. Alle angesprochenen Algorithmen zur MM arbeiten mit einer Zeitkomplexität von $O(\mathrm{ld} n)$.

---

[1] n ist dabei die Seitenlänge einer quadratischen Matrix.

Die Suche nach effizienten Algorithmen zur MM, deren Aufwand sich asymptotisch $O(n^2)$ nähert erhält zusätzliches Gewicht, da viele mathematische Probleme MM enthalten und sich ihre Komplexität deswegen durch den Aufwand für MM abschätzen läßt. Zwei dieser Probleme – die Bestimmung der Matrixdeterminante und die BFS-Nummerierung (Single Source - Shortest Path) von Graphen – werden in diesem Vortrag beispielhaft angesprochen und ihre Komplexität mit derjenigen der MM ausgedrückt. Einen umfassenden Einstieg in die Techniken der effizienten MM liefert [4].

6.1 Inhalt der Folien

Matrixmultiplikation und ihre Anwendungen

Grundlagen

- Schranken für Matrixmultiplikation (MM)
- Algorithmus von Strassen (1968)
- Bilineare Algorithmen
- Schreibweise mit Tensoren
- Disjunkte MM
- Vermutung von Strassen
- Theorem von Schönhage

Algorithmen

- Verbesserter Algorithmus von Strassen (1986)
- Algorithmus nach Coppersmith/Winograd (1987)
- Allgemeine Vorgehensweise für schnelle MM

Anwendungen

- Bestimmung der Matrixdeterminante (Chistov, 1985)
- BFS-Nummerierung von Graphen (Gazit/Miller, 1987)

Schranken für Matrixmultiplikation (MM)

• untere Schranke:	$O(N^2)$, da N^2-Eingabewerte
• ‘trivialer’ Weg:	N^3 Multiplikationen + N^3-N^2 Additionen $\Rightarrow$ $O(N^3)$
• Algorithmus von Strassen:	$O(N^{\mathrm{ld}7}) \approx O(N^{2,807})$
• Algorithmus von Coppersmith/Winograd:	$O(N^{2,376})$, z. Z. beste Schranke

Algorithmus von Strassen (1968)

Sei $X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}, Y = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix}$

mit $p_0 = (x_{00} + x_{11})(y_{00} + y_{11})$; $p_1 = (x_{10} + x_{11})y_{00}$; $p_2 = x_{00}(y_{01} + y_{11})$;

$p_3 = (-x_{00} + x_{10})(y_{00} + y_{01})$; $p_4 = (x_{00} + x_{01})y_{11}$; $p_5 = x_{11}(-y_{00} + y_{10})$;

$p_6 = (x_{01} - x_{11})(y_{10} + y_{11})$

$$\Rightarrow \quad XY = \begin{pmatrix} p_0 + p_5 - p_4 + p_6 & p_2 + p_4 \\ p_1 + p_5 & p_0 - p_1 + p_2 + p_3 \end{pmatrix}$$

7 Multiplikationen für MM von 2x2-Matrizen; für größere Matrizen Algorithmus rekursiv anwenden!

$$\Rightarrow \quad M(2^n) = 7^n \ \rightarrow \ \omega = \mathrm{ld}7 \ \text{ für } O(N^{\omega+\varepsilon})$$

Bilineare Algorithmen

$$B_\varphi(X,Y)=\sum_\alpha\sum_\beta g(\alpha,\beta,\varphi)x_\alpha y_\beta=\sum_{q=0}^{M-1} f''(\varphi,q)L_q L'_q\,;\ \varphi=0,\dots,\Gamma-1 \qquad (1)$$

$$\text{mit } L_q=\sum_\alpha f(\alpha,q)x_\alpha\,;\ L'_q=\sum_\beta f'(\beta,q)y_\beta$$

M heißt Rang des Problems: $RK(B_\varphi) = M$.
MM ist ein Spezialfall, der mittels bilinearen Algorithmen beschrieben werden kann.

Bsp.: Multiplikation von komplexen Zahlen

Sei $\bar{x}=(x_0+ix_1),\ \bar{y}=(y_0+iy_1)$

mit $L_0L_0'=x_0y_1;\ L_1L_1'=x_1y_0;\ L_2L_2'=(x_0-x_1)(y_0+y_1)\ \Rightarrow\ RK=3$

$\Rightarrow\ \ \bar{x}\ \bar{y}=(-L_0L_0'+L_1L_1'+L_2L_2'+i(L_0L_0'+L_1L_1'))$

Für die MM von *nxm* mit *mxp* Matrizen (<*m,n,p*>) mit bilinearen Algorithmen gilt folgendes Theorem:

$$\text{Sei } B(X,Y)\ \text{MM} <m,n,p>\ \text{mit } mnp>1\ \Rightarrow\ \omega\le\frac{3\,\mathrm{ld}M}{\mathrm{ld}(mnp)} \qquad (2)$$

Schreibweise mit Tensoren

Sei $\alpha_i=(\alpha_{i1},\dots,\alpha_{ir}),\ \beta_i=(\beta_{i1},\dots,\beta_{is}),\gamma_i=(\gamma_{i1},\dots,\gamma_{it})$

$$\Rightarrow\ B_\varphi(X,Y)=\sum_{j=1}^{s}\sum_{i=1}^{r} b_{ij\varphi}x_i y_j\ ,$$

wobei die $b_{ij\varphi}$ Elemente des Tensors (der Matrizen) $\mathrm{B}=(B_\varphi;\ \varphi<\Gamma-1)=\sum_{q=0}^{M-1}\alpha_{\mathrm{q}}\otimes\beta_q\otimes\gamma_q$ sind.

$\Rightarrow$ für zwei Probleme $<m,n,p>,<m',n',p'>$ gilt:

$<m,n,p>\otimes<m',n',p'>=<mm',nn',pp'>$

$RK(<m,n,p>\otimes<m',n',p'>)\le RK(<m,n,p>)\ RK(<m',n',p'>)$

Disjunkte MM

Für die parallele Ausführung zweier MM *XY* (<*m,n,p*>) und *UV* (<*n,p,m*>) gelten folgenden Aussagen:

$$\sum_j x_{ij}y_{jk}=\sum_j(x_{ij}+u_{jk})(y_{jk}+v_{ki})-\sum_j u_{jk}y_{jk}-\sum_j(x_{ij}+u_{jk})v_{ki}\ ,\quad \forall k,i$$

$$\sum_k u_{jk}v_{ki}=\sum_k(x_{ij}+u_{jk})(y_{jk}+v_{ki})-\sum_k u_{jk}y_{jk}-\sum_k(y_{jk}+v_{ki})x_{ij}\ ,\quad \forall i,j$$

$$\Rightarrow\ M_{m,n,p}=mnp+mn+np+pm$$

Da simultan gelöst wird gilt: $\omega\le\dfrac{3\,\mathrm{ld}(M_{m,n,p}/2)}{\mathrm{ld}(mnp)}$

Vermutung von Strassen

Sei B bilinearer Algorithmus für k disjunkte bilineare Probleme mit Rang M.
Dann gilt: $M=M_1+M_2+...+M_k$

Theorem von Schönhage

Sei B bilinerarer Algorithmus für k disjunkte MM $<m_s,n_s,p_s>$ mit Rang M. Dann kann man einen Algorithmus zur MM von $N\times N$ Matrizen mit $O(N^{3\tau+\varepsilon})$ konstruieren für den gilt:

$$\boxed{\begin{array}{c} \omega \le 3\tau \;\Rightarrow\; \sum_{s=1}^{k}(m_s n_s p_s)^{\tau} = M \;\text{ oder} \\ RK(\bigoplus_{s=1}^{k} < m_s, n_s, p_s >) \le M \;\rightarrow\; \sum_{s=1}^{k}(m_s n_s p_s)^{\frac{\omega}{3}} \le M \end{array}}$$

Beweis:

aus (2): $\omega_s \le 3\tau_s$; $(m_s n_s p_s)^{\tau_s} \le RK(< m_s, n_s, p_s >)$

$$\Rightarrow \sum_{s=1}^{k}(m_s n_s p_s)^{\frac{\omega_s}{3}} \le \sum_{s=1}^{k} RK(< m_s, n_s, p_s >)$$

mit Strassens Vermutung folgt: $\sum_{s=1}^{k} RK(< m_s, n_s, p_s >) = M$

Verbesserter Algorithmus von Strassen (1986)

Beobachtung: Mit einem Algorithmus zur $N\times N$-MM können ca. $\frac{3}{4}N^2$ unabhängige Zahlenpaare multipliziert werden.

$$\text{mit } g=\frac{3}{2}(N+1) \text{ gilt: } \sum_{1\le i,j,k\le N}\left(x_{ij}\lambda^{i^2+2ij}\right)\left(y_{jk}\lambda^{j^2+2j(k-g)}\right)\left(z_{ki}\lambda^{(k-g)^2+2(k-g)i}\right)=\sum_{\substack{i+j+k=g\\1\le i,j,k\le N}} x_{ij}y_{jk}z_{ki}+O(\lambda) \qquad (3)$$

da der Exponent von λ, $i^2+2ij+j^2+2j(k-g)+(k-g)^2+2(k-g)i=(i+j+k-g)^2$,
null ist für $i+j+k=g$.

Außerdem gilt: $\left|\{(\mathrm{i},\mathrm{j},\mathrm{k})/i+j+k=g\}\right| \approx \frac{3}{4}N^2$

Trilineare Basisform:

$$\sum_{i=1}^{q}(x_0+\lambda x_i)(y_0+\lambda y_i)\frac{z_i}{\lambda}+x_0 y_0\left(-\sum_{i=0}^{q}\frac{z_i}{\lambda}\right)=\sum_{i=1}^{q}(x_i^1 y_0^1 z_i+x_0^2 y_i^2 z_i)+O(\lambda)$$

$\Rightarrow$ $(q+1)$ - Multiplikationen

Der Algorithmus stellt <u>keine</u> MM dar, obwohl die Blockstruktur und die Feinstruktur MM sind!
(Die oberen Indizes drücken die Blockstruktur aus und werden Blockindizes genannt.)

Blockstruktur: $\begin{pmatrix} x_i^1 & x_0^2 \end{pmatrix}\begin{pmatrix} y_0^1 \\ y_i^2 \end{pmatrix}$ Block-MM <1,2,1>

Feinstruktur:

$$\sum_{i=1}^{q} x_i y_0 z_i \quad \text{MM} \ < q,1,1 >$$

$q \times 1$-Spaltenvektor · 1×1-Skalar $\rightarrow q \times 1$-Spaltenvektor

$$\sum_{i=1}^{q} x_0 y_i z_i \quad \text{MM} \ < 1,1,q >$$

1×1-Skalar · $1 \times q$-Zeilenvektor $\rightarrow 1 \times q$-Zeilenvektor

Permutiere die Variablen in $x_i y_0 z_i + x_0 y_i z_i$ und bilde das Tensorprodukt:

$$\left.\begin{array}{l} x_i^1 y_0^1 z_i + x_0^2 y_i^2 z_i \\ x_j^1 y_j^2 z_0 + x_j^1 y_0^1 z_j \\ x_0^2 y_k^2 z_k + x_k^1 y_k^2 z_0 \end{array}\right\} \rightarrow \begin{array}{l} \displaystyle\sum_{i,j,k=1}^{q} x_{ij0}^{11} y_{0jk}^{11} z_{i0k} + x_{ijk}^{21} y_{0jk}^{11} z_{i00} + x_{ij0}^{11} y_{00k}^{12} z_{ijk} + x_{ijk}^{21} y_{00k}^{12} z_{ij0} + \\ + x_{0j0}^{12} y_{ijk}^{21} z_{i0k} + x_{0jk}^{22} y_{ijk}^{21} z_{i00} + x_{0j0}^{12} y_{i0k}^{22} z_{ijk} + x_{0jk}^{22} y_{i0k}^{22} z_{ij0} + O(\lambda) \end{array}$$

⇒ Block 2×2-MM; jeder Block ist MM von <*m,n,p*> mit $mnp=q^3$

⇒ N-tes Tensorprodukt bilden ⇒ Block $2^N \times 2^N$-MM mit $M=(q+1)^{3N}$-Multiplikationen

mit (3) erhält man $\frac{3}{4}\left(\left(2^N\right)^2\right)$ disjunkte MM!

⇒ Schönhage anwenden:

$$(q+1)^{3N} = \frac{3}{4} 2^{2N} (q^{3N})^{\tau_N}; \ \omega_N \leq 3\tau_N$$

$$(q+1)^3 = 2^2 q^{3\tau_N}$$

$$3\,\mathrm{ld}(q+1) = 2 + 3\tau \ \mathrm{ld}\ q$$

$$\Rightarrow \ \omega \leq \frac{3\,\mathrm{ld}(q+1) - 2}{\mathrm{ld}\ q}$$

mit q=5 wird der Ausdruck minimal ⇒ $\omega \leq 2{,}4785$

Algorithmus nach Coppersmith/Winograd (1987)

Ähnliche trilineare Form wie Strassen:

$$\sum_{i=1}^{q} \lambda^{-2} (x_0 + \lambda\, x_i)(y_0 + \lambda\, y_i)(z_0 + \lambda\, z_i) -$$

$$-\lambda^{-3} (x_0 + \lambda^2 \sum_{i=1}^{q} x_i)(y_0 + \lambda^2 \sum_{i=1}^{q} y_i)(z_0 + \lambda^2 \sum_{i=1}^{q} z_i) + (\lambda^{-3} - q\lambda^{-2}) x_0 y_0 z_0$$

$$= \sum_{i=1}^{q} (x_0^0 y_i^1 z_i + x_i^1 y_0^0 z_i + x_i^1 y_i^1 z_0) + O(\lambda)$$

Algorithmus benutzt (q+2)-Multiplikationen, ist jedoch ebenfalls kein MM-Algorithmus!
Blockstruktur beschreibt keine MM, die Feinstruktur schon.

- Bestimme das 3*N*-te Tensorprodukt der Gleichung; alle Variablen, die nicht *N* Blockindizes = 0 und 2N Blockindizes = 1 haben eliminieren (alle Terme haben die Form $X_Q{}^I Y_R{}^J Z_S{}^K$, $I,J,K \in \{0,1\}^{3N}$, $Q,R,S \in \{0,\mathrm{i}\}^{3N}$).
- Sei $M = 2\binom{2N}{N} + 1$
- Erzeuge Salem-Spencer-Menge (Folie 12) *B* (mit M) und wähle Zufallszahlen $W = \{0 \le w_j < M \,/\, j = 0,\ldots,3N\}$
- Konstruiere Hash:

$$b_x(I) \equiv_\mu \left(\sum_{j=1}^{3N} I_j w_j\right)$$

$$b_y(J) \equiv_\mu \left(w_0 + \sum_{j=1}^{3N} J_j w_j\right)$$

$$b_z(K) \equiv_\mu \left(w_0 + \sum_{j=1}^{3N} (2 - K_j) w_j\right) / 2$$

- jetzt alle X^I, Y^J, Z^K auf null setzen, für die $b_x(I), b_y(J), b_z(K) \notin \mathrm{B}$

$$\Rightarrow\ b_x(I) + b_y(J) \equiv_M 2\, b_z(K)$$

$$\text{mit } b_x(I) = b_y(J) = b_z(K) \quad (\text{wg. Salem-Spencer Theorem})$$

⇒ Die Blockindizes der Variablen sind in Zahlen gehasht worden, so daß für alle Terme $X^I Y^J Z^K$ der trilinearen Form die Hashwerte den Salem-Spencer-Bedingungen genügen!

- Konstruiere Liste von Tripeln (X^I, Y^J, Z^K) mit $b_x(I) = b_y(J) = b_z(K) = b, \quad \forall b \in \mathrm{B}$.
- Aus allen Listen werden jetzt die Tripel gestrichen, bei denen eine oder mehrere Komponenten bereits in anderen Tripeln enthalten sind, so daß jeder Variablenblock in <u>höchstens einem</u> MM-Blockprodukt enthalten ist

  ⇒ mehrere disjunkte MM!

  und zwar mind. $\mathrm{H} = \frac{1}{4}\binom{3N}{N,N,N} M^{-2}$ pro Liste (H hängt von der Wahl von *W* ab; es muß jedoch mind. eine Wahl geben, für die H erreicht wird).

⇒ $\mathrm{H} \cdot M'$ disjunkte MM der Größe $\langle q^N, q^N, q^N \rangle$, $(M' = |B| > M^{1-\varepsilon})$

mit Schönhage:

$$(q+2)^{3N} \ge \mathrm{H} M' q^{3N\tau_N}; \ \ \omega_N \le 3\tau_N$$

$$\xrightarrow{q=8} \omega \le 2{,}407$$

(zur Berechnung siehe auch Folie 11)

Allgemeine Vorgehensweise für schnelle MM

- rekursiv anwendbare, bi- oder trilineare Form
- Tensorproduktbildung
- Konstruktion mehrerer disjunkter Matrixprodukte

⇒ Theorem von Schönhage anwenden, um Abschätzung zu erhalten

Bestimmung der Tripelanzahl

- Anzahl der Tripel pro Liste:

$$\underbrace{\binom{3N}{N,N,N}}_{\text{Anzahl kompatibler Tripel}} \cdot \underbrace{M^{-2}}_{\substack{\text{Wahrscheinlichkeit}\\ b_X(I)=b=b_Y(J)}}$$

- Anzahl der rausgestrichenen Tripel pro Liste (hier für gemeinsame Z^K):

$$\underbrace{\frac{1}{2}}_{\substack{\text{für gleiche Einträ ge}\\ ((X_I,Y_J,Z_K),(X_{I'},Y_{J'},Z_K))\\ ((X_{I'},Y_{J'},Z_K),(X_I,Y_J,Z_K))}} \cdot \underbrace{\binom{3N}{N,N,N}}_{\text{Anzahl kompatibler Tripel}} \cdot \underbrace{\left(\binom{2N}{N,N}-1\right)}_{\substack{\text{Anzahl kompatibler Blöcke}\\ Y_J \cong Z_K \text{ und } Y_{J'} \cong Z_K\\ \text{aber } Y_{J'} \neq Y_J}} \cdot \underbrace{M^{-3}}_{\substack{\text{Wahrscheinlichkeit}\\ b_Z(K)=b=b_Y(J)\\ \text{und } b_Y(J')=b}}$$

- Für die Anzahl der Tripelpaare mit gemeinsamen X^I oder Y^J gilt die gleiche Beziehung ⇒ Faktor 3
  ⇒ Anzahl der Tripel pro Liste nach dem Streichen:

$$\binom{3N}{N,N,N}M^{-2}-\frac{3}{2}\binom{3N}{N,N,N}\left(\binom{2N}{N,N}-1\right)M^{-3} \geq \frac{1}{4}\binom{3N}{N,N,N}M^{-2} =: \mathrm{H}$$

Theorem von Salem-Spencer

Für ein ε > 0 gibt es ein $M_\varepsilon \cong 2^{c/\varepsilon^2}$, so daß für alle $M > M_\varepsilon$ eine Menge B von $M' > M^{1-\varepsilon}$ unterschiedlichen Ganzzahlen existiert.

$$B = \left\{ b_1, b_2, \ldots, b_{M'} \;/\; 0 < b_1 < b_2 < \ldots < b_{M'} < \frac{M}{2} \right\}$$

Dabei gilt für alle $b \in B$:

Seien $b_i, b_j, b_k \in B \;\Rightarrow\; b_i + b_j = 2b_k$, falls $b_i = b_j = b_k$

Da $\forall b \in B: 0 < b < \frac{M}{2}$ gilt die Aussage auch im Ring Z_m der Integerzahlen Modulo m.

Bestimmung der Matrixdeterminante (Chistov, 1985)

Sei $A=(a_{ij})_{1\leq i,j\leq n}$ quadratische Matrix und $A_\mu=(a_{ij})_{\mu\leq i,j\leq n}$ quadratische Untermatrix von A.

$$A=\begin{pmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \dots & a_{2n} \\ \cdot & & \ddots & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & \ddots & \cdot \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{pmatrix} \Rightarrow A_2=\begin{pmatrix} a_{22} & \dots & \dots & a_{2n} \\ \cdot & \ddots & & \cdot \\ \cdot & & & \cdot \\ \cdot & & \ddots & \cdot \\ a_{n2} & \dots & \dots & a_{nn} \end{pmatrix}$$

Sei $\Delta_\mu(t)=\det(E-tA_\mu)$;

$\mathrm{Char}_A(t)=\sum_{i=0}^{n} c_i t^i = t^n \Delta_1(\mathrm{t}^{-1})$ ist charakteristisches Polynom von A und $\det A = c_0$

mit $(E-tA_\mu)^{-1}=(a_{ij}^{(\mu)})_{\mu\leq i,j\leq n}$ folgt $a_{\mu\mu}^{(\mu)}=\dfrac{\Delta_{\mu+1}(t)}{\Delta_\mu(t)}$, $(\Delta_{n+1}=1)$ und

$$\Rightarrow \frac{1}{\Delta_1(t)}=\prod_{1\leq\mu\leq n} a_{\mu\mu}^{(\mu)}$$

Matrix $(a_{ij}^{(\mu)})$ und damit $a_{\mu\mu}^{(\mu)}$ können durch folgende Gleichung bestimmt werden:

$$(a_{ij}^{(\mu)})_{\mu\leq i,j\leq n}=(E-tA_\mu)^{-1}=E+\sum_{i\geq 1} t^i A_\mu^i$$

Trick: Da das charakteristische Polynom vom Grad höchstens n ist, können die Summenterme auf Grad n gekürzt werden.

$$\Rightarrow (a_{ij}^{(\mu)})_{\mu\leq i,j\leq n} \bmod t^{n+1}=\left(E+\sum_{i=1}^{n} t^i A_\mu^i\right)\bmod t^{n+1}$$

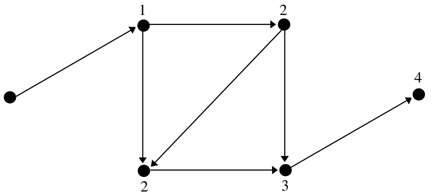
Betrachte $E+\sum_{i=1}^{n} t^i A_\mu^i$: Da jede Potenz von A in $O(\mathrm{ld}^2 n)$ bestimmt werden kann, gilt

für den Zeitbedarf $O(\mathrm{ld}^2 n)$ auf $n\cdot M(n)$ Prozessoren. Alle $a_{\mu\mu}^{(\mu)}$ können parallel berechnt werden!

$\Rightarrow$ Anzahl der Prozessoren: $O(n\cdot M(n))$ pro $E+\sum_{i=1}^{n} t^i A_\mu^i \Rightarrow O(n^2\cdot M(n))$

BFS-Nummerierung von Graphen (Gazit/Miller, 1987)

$G=(V,E)$



Algorithmus in zwei Teilen:

1. Berechne Näherung
2. Berechne Distanz

Berechne Näherung

Sei $B_0[u,v]=\begin{cases}1; & (u,v)\in E \vee u=v\\ 0; & \text{sonst}\end{cases}$ Inzidenzmatrix von G.

Durch $B_{i+1}=B_i^2$ ist eine Folge gegeben, mit der $B_1,\ldots,B_{\lceil \mathrm{ld}\, n\rceil}$ in $O(\mathrm{ld}^2 n)$ mit $M(n)$ Prozessoren berechnet werden kann ($n=|V|$).

$\Rightarrow\ B_i[u,v]=\begin{cases}1; & d(u,v)\le 2^i\\ 0; & \text{sonst}\end{cases}$ mit $d(u,v)$ = Distanz von u und v

Erzeuge $M_i[u,v]=\begin{cases}0; & u=v\\ 2^i; & d(u,v)\le 2^i\\ \infty; & \text{sonst}\end{cases}$ (geht in Zeit $O(1)$)

$\Rightarrow$ Entfernungen sind bis auf einen Faktor 2 bestimmt, d. h. $2^{i-1}+1 \le d(u,v) \le 2^i$. Der Zeitaufwand ist $O(\mathrm{ld}^2 n)$.

Berechne Distanz

Sei $D_u=\begin{cases}0; & u=s\\ \infty; & \text{sonst}\end{cases}$ $\forall 1\le u\le n.$ s ist dabei der Knoten, von dem aus die Entfernungen bestimmt werden sollen. D ist ein in $O(1)$ auf n Prozessoren berechenbarer Zeilenvektor.

Berechne:

```
for $i = \lceil \mathrm{ld}\, n \rceil$ to $0$
    $D=D*M_i$
```

wobei * MM über dem Semiring (N, min, +) bedeutet, d. h.

$$\begin{pmatrix}x_1 & x_2 & x_3\end{pmatrix}\begin{pmatrix}y_{11} & y_{12} & y_{13}\\ y_{21} & y_{22} & y_{23}\\ y_{31} & y_{32} & y_{33}\end{pmatrix}=\begin{pmatrix}\min(x_1+y_{11},x_2+y_{21},x_3+y_{31}) & \mathrm{L} & \min(x_1+y_{13},x_2+y_{23},x_3+y_{33})\end{pmatrix}$$

$\Rightarrow$ **Aufwand**: n^2 Additionen: $O(\mathrm{ld}\, n^2) = O(\mathrm{ld}\, n)$
n min-Operationen: $O(\mathrm{ld}\, n)$

$\Rightarrow$ **Gesamtaufwand BFS**: Zeit: $O(\mathrm{ld}^2 n)$ auf $M(n)$ Prozessoren

6.2 Erläuterungen zu den Folien

6.2.1 Der erste Strassen-Algorithmus (Folie 2)

Eine erste Verbesserung des normalen $O(n^3)$-Algorithmus zur MM wurde 1968 von Strassen entwickelt. Der Algorithmus benutzt für die Multiplikation zweier 2×2-Matrizen 7 Multiplikationen[2]. Gößere Matrizen können durch Substitution mit 2×2-Matrizen und rekursiver Anwendung des Algorithmus behandelt werden. Strassen zeigt, daß diese Verbesserung der MM die Komplexität $O(n^{2{,}807})$ besitzt. Trotz sukzessiver Verbesserungen der Algorithmen ist der Strassen-Algorithmus immer noch das in der Praxis am häufigsten angewandte Verfahren große Matrizen zu multiplizieren.

6.2.2 Bilineare und trilineare Formen (Folie 3 f.)

Sowohl der Strassen-Algorithmus, als auch spätere Verbesserungen der MM bauen auf bilinearen bzw. trilinearen[3] Basisalgorithmen auf. Unter einer bilinearen Form versteht man die Darstellung eines Problems durch Multiplikationen von Linearkombinationen der Eingabegrößen. Die Anzahl der Multiplikationen stellt dabei ein Maß für die Komplexität des Problems dar; verwandte Probleme können durch die gleiche bilineare Form mit unterschiedlichen Koeffizienten beschrieben werden (Schar bilinearer Formen; siehe auch *Disjunkte MM*). Das die MM nur einen Spezialfall der bilinearen Probleme darstellt, wird auf Folie 3 durch die Formulierung der Multiplikation komplexer Zahlen als bilinearen Algorithmus verdeutlicht. Die Formulierung der MM als bilineare Form führt schließlich zu einer Abschätzung der Komplexität eines bilinearenAlgorithmus.

$$\text{Sei } B(X,Y) < m,n,p > \text{ MM mit } mnp > 1 \Rightarrow \omega \leq \frac{3 \,\mathrm{ld} M}{\mathrm{ld}(mnp)}$$

M ist der Rang des Problems (Anzahl der Multiplikationen). Eine gleichartige Darstellung bilinearer bzw. trilinearer Algorithmen läßt sich durch Verwendung mehrdimensionaler Tensoren erreichen. Die Multiplikation zweier bilinearer Probleme (z. B. MM eines $< m,n,p >$-Problems[4], wobei die Matrixelemente Matrizen der Form $m' \times n'$, bzw. $n' \times p'$ sind) kann dann durch das Tensorprodukt erhalten werden.

---

[2] Die Anzahl der Additionen wird bei den folgenden Komplexitätsbetrachtungen stets vernachlässigt.

[3] Trilineare Algorithmen lassen sich stets durch Gleichsetzen der Koeffizienten auf eine bilineare Form reduzieren. Ihre Bedeutung liegt lediglich in einer geschlosseneren mathematischen Darstellung des Problems.

[4] $< m,n,p >$ ist eine Formulierung der MM einer $m \times n$- mit einer $n \times p$-Matrix.

6.2.3 Disjunkte Matrixmultiplikation und Vermutung von Strassen (Folie 4 f.)

Ein Verfahren zur Beschleunigung der MM stellen Algorithmen dar, die mehrere Probleme gleichzeitig lösen. Durch die parallele Ausführung von 'gleichschweren' Problemen folgt mit der *Vermutung von Strassen*, daß der Rang des Gesamtproblems kleiner oder höchstens gleich der Summe der einzelnen Teilränge sein muß; d. h. Teilprobleme können im Verbund effizienter gelöst werden als durch eine isolierte Betrachtung. Da bei einem solchen Algorithmus zur MM keine Zusammenhänge zwischen den Matrixelementen vorausgesetzt werden, spricht man von der *Disjunkten MM*. Verbesserungen des Algorithmus durch Einführung von Parametern erlauben es, den Gesamtrang des Problems weiter zu senken. Dies führt zu den λ-Algorithmen bzw. den *Arbitrary Precision Algorithms* (APA), die im Vortrag jedoch nicht behandelt wurden.

6.2.4 Theorem von Schönhage (Folie 5)

Der zentrale Satz zur Aufwandsabschätzung bei bilinearen und trilinearen Algorithmen von Schönhage wird in allen folgenden Verbesserungen der MM verwendet, um eine untere Schranke der MM anzugeben. Das durch diese Ungleichung gewonnene ω zur Komplexitätsabschätzung des Algorithmus stellt keinen absoluten Wert dar, sondern muß im Sinne von $O(n^{\omega+\epsilon})$ mit $\epsilon > 0$ als eine asymptotische Schranke verstanden werden. Mit diesem Theorem reduziert sich die Abschätzung der Komplexität auf die Bestimmung des Gesamtrangs M des disjunkten Algorithmus, sowie der Angabe der benötigten Multiplikationen k.

6.2.5 Verbesserter Algorithmus von Strassen (Folie 6 ff.)

Der 1986 von Strassen (siehe [7]) vorgestellte effiziente Algorithmus zur MM multipliziert zwei $N \times N$-Matrizen mit einem Aufwand von $O(N^{2{,}476})$. Allerdings ist sowohl dieser Algorithmus, als auch der Algorithmus von Coppersmith/Winograd (siehe [1, 2]) lediglich von theoretischem Interesse zur Angabe der denkbaren unteren Schranke zur MM. Das Problem beider Verfahren ist es, keine allgemeine Formel wie der erste Strassen- Algorithmus aufzuzeigen. Vielmehr geben die Algorithmen ein Verfahren zur Berechnung eines MM-Algorithmus für ein spezifisches Problem an die Hand. Diese Berechnung ist leider so aufwendig, daß deswegen beide Verfahren keinerlei praktische Anwendung erfahren.

Die Vorgehensweise zur Konstruktion effizienter MM-Algorithmen ist stets die gleiche:

1. Suche eines Basisalgorithmus in trilinearer Form; dieser Algorithmus muß nicht unbedingt ein MM-Algorithmus sein. Vielmehr ist es die große Entdeckung von Strassen, daß Basisalgorithmen verwendet werden können, bei

denen die Anzahl von Multiplikationen nicht die Anzahl der x- (oder y-, oder z-) Variablen überschreiten muß. Das impliziert jedoch, daß der Basisalgorithmus kein Algorithmus zur MM sein kann. Mit dieser Erleichterung der Konstruktion des Basisalgorithmus ist es nun auch denkbar, Algorithmen zur MM in $O(n^2)$ zu finden[5].

2. Geeignete Tensorproduktbildung des Basisalgorithmus und evt. Potenzierung des Tensorproduktes.
3. Lokalisieren der disjunkten Matrixprodukte.
4. Anwendung des *Theorems von Schönhage* zur Abschätzung der Komplexität.

Der Strassen-Algorithmus beruht auf der Beobachtung, daß bei einem Algorithmus zur $N \times N$-MM ungefähr (d. h. im Sinne des λ-Verfahrens) $\frac{3}{4}N^2$ disjunkte Matrixprodukte zu lokalisieren sind[6]. Dies wird nach der Tensorproduktbildung und der Potenzierung zur Schätzung der benötigten Multiplikationen benutzt; der Rang des Gesamtproblems ergibt sich aus der Anzahl der Multiplikationen im Basisalgorithmus (entsprechend oft potenziert). Das Schönhage-Theorem liefert nun aus diesen beiden Informationen den Komplexitätsparameter $\omega = 2,476$.

6.2.6 Algorithmus von Coppersmith und Winograd (Folie 8 ff.)

Dieser Algorithmus wurde 1987 veröffentlicht (siehe [1, 2]) und hält bis heute den niedrigsten Komplexitätsparameter für MM $\omega = 2,376$. Erreicht wird dieser Wert durch eine ähnliche Vorgehensweise wie beim Strassen- Algorithmus. Die trilineare Basisform stellt wiederum keinen Algorithmus zur MM dar. Erst durch wiederholte Bildung des Tensorprodukts erhält man einen Algorithmus für die MM von Problemen der Größe $< q^N, q^N, q^N >$.

Die Lokalisierung der disjunkten Matrixprodukte erreichen Coppersmith und Winograd durch ein komliziertes Auswahlverfahren basierend auf einem kombinatorischen Theorem von Salem und Spencer (siehe Folie 12, [6]). Um dieses Theorem anwenden zu können werden die durch den Basisalgorithmus gegebenen Blockindizes der Tensorproduktterme in Elemente einer Salem-Spencer-Menge gehasht. Aus diesen Werten können nun diejenigen Terme identifiziert werden, die einer bestimmten – durch das Salem-Spencer-Theorem vorgegebenen – Relation genügen.

---

[5] Coppersmith und Winograd haben 1982 (siehe [3]) gezeigt, daß für jeden Basisalgorithmus zur MM mehrerer Matrizen, der auch ein MM-Algorithmus ist, strikt mehr Multiplikationen nötig sind als x-Variablen. Damit war aber auch klar, daß ein MM-Algorithmus mit Komplexität $O(n^2)$ nicht möglich ist.

[6] In dieser Formulierung ist das Verfahren nicht einmal konstruktiv.

Von den übriggebliebenen Elementen werden durch den Vorgang des „Prunings“ schließlich diejenigen selektiert, die zur Berechnung des Matrixprodukts absolut notwendig sind. Die Konstruktion schließt mit einer kombinatorischen Abschätzung der im Mittel übriggebliebenen Elemente. Die Anwendung des Schönhage-Theorems liefert schließlich den Komplexitätsparameter $\omega = 2,407$. Weitere Verfeinerungen, vor allem bei der Abbildung der Blockindizes in die Salem-Spencer-Menge, führen zu dem oben angegebenen Wert für ω.

6.2.7 Bestimmung der Matrixdeterminante (Folie 13 f.)

Das hier vorgestellte Verfahren zur Berechnung der Matrixdeterminante beruht auf einer Arbeit von Chistov (siehe [8]) von 1985. Der Algorithmus reduziert die Bestimmung der Matrixdeterminante auf die parallele Ausführung sukzessiver Quadrierungen einer bestimmten Matrix. Ziel ist die Bestimmung von $\Delta_1(t)$; dazu muß für alle Untermatrizen $A_\mu = (a_{ij})_{\mu \leq i,j \leq n}$ das linke, obere Elemente von $(E - tA_\mu)^{-1}$ bestimmt werden. Dabei nutzt der Algorithmus die Eigenschaft, daß die Berechnung der Inversen über die übliche Formel im Ring $Z_{t^{n+1}}$ durchgeführt werden kann, so daß keine unendlichen Summenterme entstehen (siehe Folie 14).

In die Komplexitätsbetrachtung geht der Aufwand für MM als Variable ein. Eine Verbesserung der Algorithmen zur MM führt demnach auch zu einer effizienteren Bestimmung der Matrixdeterminante.

6.2.8 BFS-Numerierung von Graphen (Folie 15 f.)

Der Trick des von Gazit und Miller (siehe [5]) 1988 vorgeschlagenen Algorithmus liegt in der Verlagerung der durch den bekannten Algorithmus benötigten MM vom Semiring $(N, \min, +)$ in den Ring $(Z, *, +)$. Hier können dann die effizienten Algorithmen zur MM verwendet werden, wohingegen die MM im Semiring (z. Z.) nicht besser als $O(n^3)$ erledigt werden kann.

Der erste Schritt des Algorithmus besteht aus einer Abschätzung der Entfernung bis auf einen Faktor 2. Die fortgesetzte Quadrierung der Adjazenzmatrix kann mit einem schnellen MM-Algorithmus mit Aufwand $O(\mathrm{ld}^2 n)$ in der Zeit (logarithmischer Aufwand für eine Multiplikation und das ganze insgesamt $\mathrm{ld} n$-mal) auf $M(n)$-Prozessoren durchgeführt werden. Im zweiten Schritt wird dann durch $\mathrm{ld} n$-malige Vektor-Matrix-Multiplikation im Semiring die tatsächliche Entfernung bestimmt.

Literaturverzeichnis

[1] D. Coppersmith and S. Winograd: *Matrix Multiplication via Arithmetic Progressions*, Journal on Symbolic Computation 9: 251-280 (1990).

[2] D. Coppersmith and S. Winograd: *Matrix Multiplication via Arithmetic Progressions*, ACM Symposium on Theory of Computing: 1-6 (1987).

[3] D. Coppersmith and S. Winograd: *On the Asymptotic Complexity of Matrix Multiplication*, SIAM Journal on Computing 11: 472-492 (1982).

[4] V. Pan: *How to Multiply Matrices Faster*, Springer Lecture Notes in Computer Sience 179 (1984).

[5] H. Gazit and G. Miller: *An Improved Parallel Algorithm that Computes the BFS Numbering of a Directed Graph*, Information Processing Letters 28: 61-65 (1988).

[6] R. Salem and D. Spencer: *On sets of integers which contain no three terms in arithmetical progression*, Proc. Nat. Acad. Sci. 28: 561-563 (1942).

[7] V. Strassen: *The Asymptotic Spectrum of Tensors and the Exponent of Matrix Multiplication*, Proc. 27th Ann. IEEE Symposium on Foundations of Computer Science: 49-54 (1986).

[8] A. Chistov: *Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic*, Springer Lecture Notes in Computer Sience 199: 63-79 (1985).

Vortrag 7

Syntaxanalyse

Anja Löser
Universität Erlangen-Nürnberg
aaloeser@cip.informatik.uni-erlangen.de

Teil II

Münchner Vorträge: Randomisierte Algorithmen

Vortrag 8

Randomisierte Algorithmen - Grundlagen

Markus Erlacher
Technische Universität München
erlachem@informatik.tu-muenchen.de

Zusammenfassung

Mit diesem Paper erfahren Sie eine Einführung in das Thema randomisierte Algorithmen. Anhand von Beispielsalgorithmen sollen die wesentlichsten Eigenschaften und der Nutzen von randomisierten Algorithmen dargelegt werden. Außerdem werden noch Komplexitätsklassen für randomisierten Algorithmen und Random Sources diskutiert.

8.1 Motivation

Ein randomisierter Algorithmus ist ein Algorithmus, der zusätzlich zu seinen Eingabedaten noch eine Reihe von Zufallsbits erhält. Diese Bits können nun vom Algorithmus verwendet werden, um Zufallsentscheidungen zu treffen. So ergeben sich bei verschiedenen Durchläufen eines randomisierten Algorithmus mit fester Eingabe durchaus unterschiedliche Ergebnisse. Desweiteren kann man nun auch bei der Ausführungszeit eines solchen Algorithmus von einer Zufallsvariablen sprechen.

Einzelne Beispiele randomisierter Algorithmen können bis zu den frühesten Tagen der Computerwissenschaft zurückverfolgt werden, aber die wirkliche Bedeutung dieses Konzepts wurde erst vor ungefähr 20 Jahren erkannt.

Jetzt ist die Randomisierung in einem großen Bereich von Anwendungen zu einem sehr wichtigen Werkzeug für die Entwicklung von Algorithmen avanciert. Denn für randomisierte Algorithmen gibt es prinzipiell zwei bedeutende Vorteile: Ihre Ausführungszeit und/oder ihre Platzanforderungen sind oftmals kleiner, als das bei entsprechenden deterministischen Algorithmen der Fall ist. Der zweite Vorteil liegt darin, daß sie einfach zu verstehen und zu implementieren sind. So genügt oft die Einführung der Randomisierung, um einen einfachen und naiven deterministischen Algorithmus mit schlechtem Worst-Case-Verhalten in einen randomisierten Agorithmus, der mit hoher Wahrscheinlichkeit jede mögliche Eingabe gut verarbeitet, überzuführen.

8.2 Randomisierte Algorithmen

In diesem Kapitel werden vier randomisierte Algorithmen vorgestellt. Zum einen ein Algorithmus, mit dessen Hilfe man testen kann, ob in einem bipartiten Graphen ein perfektes Matching existiert. Der zweite Algorithmus testet, ob eine aussagenlogische Formel lösbar ist oder nicht. Die letzten beiden Algorithmen beschäftigen sich mit der Zerlegbarkeit von ganzen Zahlen.

8.2.1 Symbolische Determinanten

Wie schon erwähnt, soll in diesem Kapitel ein Algorithmus bestimmt werden, mit dessen Hilfe wir testen können, ob ein bipartiter Graph ein perfektes Matching enthält.

Gegeben sei also ein bipartiter Graph $G = (U, V, E)$ mit $U = \{u_1, u_2, \ldots, u_n\}$, $V = \{v_1, v_2, \ldots, v_n\}$ und $E \subseteq U \times B$. Ein Matching M ist eine Teilmenge von E, wobei aber gelten muß: $(u, v) \in M$ und $(u', v') \in M \Leftrightarrow u \neq u'$ und $v \neq v'$. Für ein perfektes Matching gilt aber noch zusätzlich: $|M| = \frac{|V+U|}{2}$. Mit anderen Worten, gesucht ist eine Permutation π von $\{1, \ldots, n\}$, so daß gilt: $(u_i, v_{\pi(i)}) \in E \forall u_i \in U$.

Dieses Problem läßt sich nun relativ einfach mit Matrizen und Determinanten lösen.

Wir repräsentieren nun die Kantenmenge E durch eine $n \times n$ Matrix A. Die Matrixelemente sind einfache Variablen, falls $(u_i, v_j) \in E$, und Null sonst. Betrachten wir nun die Determinante von A. Sie ist folgendermaßen definiert:

$$detA = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A_{i,\pi(i)}$$

Wobei π über alle n Elemente permutiert. $\sigma(\pi)$ ergibt 1, falls sich π in eine gerade Anzahl von Translationenen zerlegen läßt, ansonsten -1. Damit ist klar, daß nur Terme ungleich Null zum Auffinden des Matchings beitragen. Außerdem folgt damit auch: $detA \neq 0 \Leftrightarrow G$ hat ein Matching.

So eine Matrix mit Variablen als Elemente bzw mit allgemeinen Polynomen über verschiedene Variablen heißt „Symbolische Matrix“. Wenn wir nun aber die Determinante einer solchen Matrix, zB. mit dem Gaußschen Eliminationsverfahren berechenen, dann werden die Elemente der Matrizen in den Zwischenschritten immer komplexer. Selbst bei einer Anfangsmatrix, deren Elemente nur aus einfachen Variablen bestehen.

Aber für unser Vorhaben ist es überhaupt nicht notwendig, den exakten Wert für die Determinante zu erhalten. Es genügt nur zu wissen, ob das Ergebnis Null oder nicht Null ist. Zu diesem Zweck substituieren wir die Variablen von A mit ganzen Zahlen. Wir erhalten somit wieder eine numerische Matrix. Damit können wir die Determinante von A zum Beispiel mit dem Gaußschen Eliminationsverfahren in polynomieller Zeit berechnen. Nur gibt es noch ein kleines Problem aus dem Weg zu räumen. Ergibt die Determinante der numerischen Matrix einen Wert ungleich Null, so ist der Wert der symbolischen ebenso ungleich Null. Hat die Determinante der symbolischen Matrix den Wert Null, so auch die Determinanate der numerischen Matrix. Ist jedoch die Determinante der numerischen Matrix gleich Null, so kann das auch daran liegen, daß falsche Werte substituiert wurden.

Lemma 8.2.1.1 *Sei $\pi(x_1, \ldots, x_m)$ ein Polynom ungleich Null mit m Variablen, jede mit Grad höchstens d. Sei $M > 0$ eine ganze Zahl. Dann gibt es höchstens $m\,d\,M^{m-1}$ m-Tupel $(x_1, \ldots, x_m) \in \{0, 1, \ldots, M-1\}^m$, so daß gilt: $\pi(x_1, \ldots, x_m) = 0$.*

Beweis: Beweis durch Induktion über die Anzahl der Variablen m.
$m = 1 \Rightarrow$ Anzahl der Tupeln $(x_1) \in \{0, 1, \ldots, M-1\}$ für die gilt: $\pi(x_1) = 0$ ist höchstens d.
Lemma gelte nun für $m-1$ Variablen.
$m-1 \to m$: Man schreibe $\pi(x_1, \ldots, x_m)$ so um, daß ein $\pi(x_m)$ mit $\pi(x_1, \ldots, x_{m-1})$ als Koeffizienten entsteht. Angenommen, das Polynom wird für eine beliebige ganze Zahl zu Null ausgewertet. Dann gibt es zwei Fälle: Entweder es ist ein Koeffizient von x_m aus π mit höchstem Grad, oder nicht.

1. Der Koeffizient ist ein Polynom über $x_1, \ldots, x_{m-1}$. Laut Induktionsvoraussetzung gilt: Es gibt $(m-1)\,d\,M^{m-2}$ m-Tupel, so daß gilt: $\pi(x_1, \ldots, x_{m-1}) = 0$. Für die meisten M-Werte von x_m wird das Polynom null, d.h. $(m-1)\,d\,M^{m-1}$ Werte insgesamt.

2. Dieser Fall definiert ein Polynom mit Grad $\leq d$ in x_m, das höchstens d Wurzeln für jede Kombination der Werte $x_1, \ldots, x_{m-1}$ oder $d\,M^{m-1}$ neue

> Wurzeln von π haben kann. Der Beweis folgt schließlich aus der Addition der beiden Abschätzungen.

Mit dem Lemma, kann man jetzt ganz einfach einen Entscheidungsalgorithmus schreiben. $A(x_1, \ldots, x_m)$ bezeichne die Matrix A mit m Variablen. Der Grad der Variablen sei höchstens 1.

Algorithmus:

```
Wähle zufällig m ganze Zahlen $i_1, \ldots, i_m$ zwischen 0 und $M = 2m$;
Berechne $det(A(i_1, \ldots, i_m))$ mit Gaußscher Elimination;
if($det(A(i_1, \ldots, i_m)) \neq 0$) cout << "G hat ein perfektes Matching";
else cout << "G hat wahrscheinlich kein perfektes Matching";
```

Ein solcher Algorithmus heißt auch Monte-Carlo-Algorithmus. Findet er ein perfektes Matching - im allgemeinen Fall eine Positivantwort, so ist das Ergebnis fest und eindeutig bestimmt. Aber antwortet der Algorithmus mit einer Negativantwort - hier: wahrscheinlich kein perfektes Matching, dann besteht die Möglichkeit einer Falschaussage. Aber die Wahrscheinlichket einer Falschaussage beträgt höchstens $\frac{1}{2}$, nach Definition des Monte-Carlo-Algorithmus.

Wenn man nun in unserem Fall M viel größer als md wählt, dann verringert sich die Wahrscheinlichkeit der Falschantworten. Es gibt jedoch eine bessere Methode die Wahrscheinlichkeit der Falschantworten zu senken: Das Durchführen von vielen unabhängigen Experimenten. Die Wahrscheinlichkeit, keine Falschaussage zu erhalten steigt mit jedem Experiment. Bei k Experimenten schließlich auf eine Wahrscheinlichkeit von $1 - (\frac{1}{2})^k$.

8.2.2 Random Walks

Betrachten wir den folgenden randomisierten Algorithmus für SAT:

```
Zuweisung T beliebiger Wahrheitswerte an die Literale einer aussagenlogischen
  Formel;
for (int $i = 1$;$i <= r$;$i++$) {
  if (es gibt keine unerfüllte Klausel) {
    cout << "Formel ist erfüllbar";
    halt;}
  wähle irgendeine unerfüllte Klausel aus;
  wähle zufällig irgendein beliebiges Literal aus und invertiere es;
  bringe T auf neuen Stand;}
cout << "Formel ist wahrscheinlich unerfüllbar";
```

Hierzu sei noch angemerkt, daß die zufällige Auswahl eines Literals gleichwahrscheinlich sei. Dieser Algorithmus wird auch Random Walk Algorithmus genannt.

Ist ein aussagenlogischer Ausdruck unerfüllbar, dann muß der Algorithmus trotzdem „korrekt“ sein. Doch auch hier kann die Formel erfüllbar sein. Wie im ersten Beispiel gilt auch hier: Durch maximal exponentiell viele Wiederholungen wird ein erfüllbarer Ausdruck auch als erfüllbar erkannt.

Eine wichtige Frage lautet: Wie stehen die Chancen, daß ein erfüllbarer Ausdruck als erfüllbar erkannt werden kann, wenn r ein Polynom über die Anzahl der booleschen Variablen ist? Kann diese naive Annahme bzgl. r überhaupt gegen das Erfüllbarkeitsproblem bestehen? In der Tat nicht: Es gibt einfache erfüllbare Beispiele für 3SAT, für die der Random Walk Algorithmus aber schlecht arbeitet. Für 2SAT läßt sich der Algorithmus aber gebrauchen.

Satz 8.2.2.1 *Der Random Walk Algorithmus mit* $r = 2n^2$ *werde auf ein beliebiges erfüllbares Beispiel für 2SAT mit n Variablen angewendet. Dann beträgt die Wahrscheinlichkeit, eine erfüllbare Zuweisung von Wahrheitswerten zu finden mindestens* $\frac{1}{2}$.

Lemma 8.2.2.1 *Sei x eine Zufallsvariable für nichtnegative ganze Zahlen, dann gilt für ein beliebiges* $k > 0$: $prob[x \geq k\mathcal{E}(x)] \leq \frac{1}{k}$. *(*$\mathcal{E}(x)$ *bezeichnet den erwarteten Wert von x.)*

Beweis (Lemma):
$\mathcal{E}(x) = \sum_i iprob[x = i] = \sum_{i<k\mathcal{E}(x)} iprob[x = i] + \sum_{i \geq k\mathcal{E}(x)} iprob[x = i] \geq \sum_{i \geq k\mathcal{E}(x)} iprob[x = i] \geq k\mathcal{E}(x) prob[x \geq k\mathcal{E}(x)]$
$\Rightarrow prob[x \geq k\mathcal{E}(x)] \leq \frac{1}{k}$

Beweis (Satz):
Sei T’ die Zuweisung von Wahrheitswerten, für das das gegebene 2SAT Beispiel erfüllbar ist. t(i) bezeichne die Anzahl der Wiederholungen des Invertierungsschrittes, bis T erfüllbar ist. Dabei sollen sich die Wahrheitswerte in T von denen in T’ um genau i Werte unterscheiden.

$t(0) = 0$, denn wenn $T = T'$ ist, dann müssen keine Wahrheitswerte invertiert werden. Wir müssen t auch nich invertieren, falls wir eine andere erfüllbare Zuweisung von Wahrheitswerten gefunden haben. Im anderen Fall müssen wir mindestens einmal invertieren. Wenn wir also invertieren, wählen wir von den Literalen dasjenige aus, das unter gegenwärtigem T nicht erfüllbar ist. Jetzt ist dann auch mindestens eines der beiden Literale unter T’ wahr. Deshalb bewegen wir uns beim Invertieren eines zufällig ausgewählten Literals mit einer Wahrscheinlichkeit von $\frac{1}{2}$ näher zu T’. Damit können wir für die i im Intervall $]0, n[$ folgende Ungleichung formulieren: $t(i) \leq \frac{1}{2}(t(i+1) + t(i-1)) + 1$ (*). Die Eins müssen wir wegen der gerade durchgeführten Invertierung hinzuaddieren. Desweiteren können wir noch eine weitere Ungleichung anführen: $t(n) \leq t(n-1) + 1$, da wir ja im Fall $i = n$ i nur erniedrigen können.

Betrachten wir eine andere Situation, in der (*) als Gleichung gilt. Dies schließt aus, daß wir auf eine andere Wahrheitswertzuweisung stoßen oder daß

beide Literale einer Klausel in T sich von derselben in T' unterscheiden. Außerdem bedeutet (*) als Gleichung auch, daß wir uns nach jedem Invertierungsschritt T' annähern. Wir definieren also eine neue Funkrtion x :
$x(0) = 0,$
$x(n) = x(n-1) + 1,$
$x(i) = \frac{1}{2}(x(i-1) + x(i+1)) + 1.$
Als weitere Eigenschaft folgt: $x(i) > t(i)$ für alle i.

Die x(i)'s sind aber leicht zu berechnen. Es ergibt sich dafür: $x(i) = 2in - i^2$. Startet man den Algorithmus mit dem Worst-Case-Fall (i=n), so erhalten wir das erwartete Ergebnis: $x(n) = n^2$.

Damit kann die Anzahl der Wiederholungen für die Invertierung nach oben folgendermaßen abgeschätzt werden: $t(i) \leq x(i) \leq x(n) \leq n^2$.

Mit dem obigen Lemma nun, bei dem wir k=2 und $\mathcal{E}(x) = n^2$ setzen, folgt: daß der Random Walk Algorithmus mit $r = 2n^2$ ein polynomieller Monte Carlo Algorithmus für 2SAT ist.

8.2.3 Der Fermatsche Test

Sei N eine Primzahl, dann sei $M = \{1, \ldots, N-1\}$. $\forall a \in M$: $a^{(N-1)} = 1(\text{mod } N)$. Wir stellen deshalb folgende Hypothese auf:

Ist N keine Primzahl, dann gilt für mindestens die Hälfte der Reste $a \neq 0$:

$$a^{(N-1)} \neq 1(\text{mod } N)$$

Mit dieser Hypothese läßt sich sofort ein polynomieller Monte Carlo Algorithmus erstellen:

```
M=$\{1, \ldots, N-1\}$;
wähle zufällig ein $a \in M$;
if ($a^{(N-1)} \neq 1(\text{mod } N)$) cout << "N ist zerlegbar";
else cout << "N ist wahrscheinlich eine Primzahl";
```

Falls die Hypothese wahr ist, dann ist die Wahrscheinlichkeit einer falschen Negativantwort tatsächlich kleiner als $\frac{1}{2}$. Leider ist die Hypothese nicht haltbar. Zum Beispiel durchlaufen alle Reste (modulo 561) den Fermatschen Test ($a^{(N-1)} = 1(\text{mod } N)$), und dennoch gilt: $561 = 3 \cdot 11 \cdot 17$. Ein Grund für dieses Verhalten ist, daß bei dieser Primzahlenzerlegung für alle Teiler von N (=561) gilt: $p - 1 | N - 1$. Eine Zahl mit dieser seltenen Eigenschaft heißt Carmichael Zahl. Diese Carmichael Zahlen scheinen unsere Idee wie eine Seifenblase platzen zu lassen. Aber es gibt Möglichkeiten dieses Problem zu umgehen. Eine wird im nächsten Kapitel vorgestellt.

8.2.4 Square Roots Modulo a Prime

Es sei p eine Primzahl, K={1,...,p-1}. Ein $a \in K$ ist ein quadratischer Rest, wenn es eine perfekte Wurzel in K gibt. In anderen Worten, wenn ein $z \in K$ existiert, so daß gilt: $z^2 = a(\text{mod } p)$. Um nun zu testen, ob ein beliebiges a — im obigen Sinne — ein quadratischer Rest ist, wurde $a^{\frac{p-1}{2}} = 1(\text{mod } p)$ als Indikator eingeführt. Ist die Gleichung erfüllt, so handelt es sich bei a um einen quadratischen Rest. Im anderen Fall, d.h. $a^{\frac{p-1}{2}} \neq 1(\text{mod } p)$, wird die Gleichung so definiert: $a^{\frac{p-1}{2}} = -1(\text{mod } p)$. Diese Indikatorgleichung wird nun abkürzend mit (a/p), dem Legendre Symbol bezeichnet.

Wenn man das Ganze jetzt für alle ganzen Zahlen erweitert, erhält man ein Symbol, das nur noch wenig mit Wurzeln zu tun hat. Aber dieses Symbol ist sehr nützlich und wird auch Jacobi Symbol genannt. Für das Symbol schreibt man kurz (M/N). Seien M und N keine Primzahlen, und N lasse sich als Produkt $\prod_{i=1}^{n}(M/q_i)$ darstellen, wobei q_i ungerade Primzahlen seien. Die Jacobi Symbole lassen sich auf eine ähnliche Art und Weise rekursiv bestimmen wie der größte gemeinsame Teiler. Der Abbruchfall der Rekursion, der das Ergebnis bestimmt, lautet hier $(2/N) = (-1)^{\frac{N^2-1}{8}}$. Die rekursiven Aufrufe bringen wir mit Hilfe folgender Rechenregeln (M/N) in die Form des Abbruchfalls:

- $(M1 * M2/N) = (M1/N)(M2/N)$
- $(M + N/N) = (M/N)$
- $(N/M)(M/N) = (-1)^{\frac{M-1}{2}\frac{N-1}{2}}$ (M und N ungerade)

Ist nun N zerlegbar, dann genügen mindestens die Hälfte der Elemente M aus der Menge $\{x|x \in \{1,\ldots,N-1\} \wedge ggT(x,N) = 1\}$ dem Ausdruck $(M/N) \neq M^{\frac{N-1}{2}} mod\ N$. Und dies wiederum läßt sich sofort in einen Mont Carlo Algorithmus umwandeln:

```
wähle M aus [2,N-1];
if (ggT(M,N) > 1) {
   cout << ”N ist zerlegbar”;
   halt;}
if (($(M/N) \neq M^{\frac{N-1}{2}}$mod $N$) cout << ”N ist zerlegbar”;
else cout << ”N ist wahrscheinlich eine Primzahl”;
```

8.3 Komplexitätsklassen

Randomisierte Algorithmen bilden eine neue, eigene Gruppe von Algorithmen, die sich von der Gruppe der herkömmlichen, deterministischen Algorithmen gut abgrenzen läßt. So wurden für diese neue Gruppe auch neue Komplexitätsklassen definiert. Diese Komplexitätsklassen lassen sich zwischen P und NP einordnen.

Insgesamt gibt es vier neue Komplexitätsklassen - nämlich RP, ZPP, PP, BPP, die im Folgenden eingeführt werden.

In den anschließenden Unterkapiteln sei N eine polynomiell zeitbeschränkte Turingmaschine. N sei exakt, d.h. alle Berechnungen, die auf eine Eingabe x folgen, enden nach der gleichen Anzahl von Berechnungsschritten. Diese Anzahl sei durch ein Polynom über $|x|$ bestimmt. Außerdem soll es bei jedem Berechnungsschritt für die Turingmascheine genau zwei nichtdeterministische Wahlmöglichkeiten geben.

8.3.1 Die Komplexitätsklasse RP

L sei eine Sprache. Eine polynomielle Monte Carlo Turingmaschine für L ist eine Turingmaschine, wie in 4. beschrieben, nur um zwei Eigenschaften erweitert:

1. Die Anzahl der Schritte in jeder Berechnung, die auf einer Eingabe der Länge n folgt, ist durch ein Polynom über n (kurz p(n)) bestimmt.

2. Für jede Eingabe x muß gelten:

   - Wenn $x \in L$, dann hält mindestens die Hälfte der $2^{p(|x|)}$ Berechnungen von N bezüglich x mit „yes“.
   - Wenn $x \notin L$, dann halten alle Berechnungen mit „no“.

Die Klasse aller Sprachen von polynomiellen Monte Carlo Turingmaschinen wird mit RP (für „randomized polynomial time“) bezeichnet.

Mit dieser Definition erhalten wir gleichzeitig eine informelle Beschreibung des Monte Carlo Algorithmus. Die nichtdeterministischen Schritte in der obigen Definition könnte man zum Beispiel durch das „Werfen einer Münze“ ersetzen. Falsche Positivantworten sind ausgeschlossen, da ja alle $x \notin L$ zurückgewiesen werden; und die Wahrscheinlichkeit für eine falsche Negativantwort ist höchstens $\frac{1}{2}$. Die Wahrscheinlichkeit beträgt deshalb $\frac{1}{2}$, da wenn jeder Berechnungsschritt der Turingmaschine eine Auswahl zwischen den beiden Alternativen, mit der Wahrscheinlichkeit von $\frac{1}{2}$ für jede Alternative, trifft, alle Berechnungen gleichwahrscheinliche Ereignisse mit der jeweiligen Wahrscheinlichkeit von $2^{-p(|X|)}$ sind.

Die Mächtigkeit von RP wird nicht angegriffen, wenn die Wahrscheinlichkeit für die Akzeptanz nicht $\frac{1}{2}$ ist, sondern irgendwo zwischen 0 und 1 liegt. Wäre die Wahrscheinlichkeit für eine falsche Negativantwort eines randomisierten Algorithmus höchstens $1 - \varepsilon$, für ein beliebiges $\varepsilon < \frac{1}{2}$, dann könnten wir durch k-maliges Wiederholen des Algorithmus die gewünschte Wahrscheinlichkeit auf einen Wert von höchstens $\frac{1}{2}$ transformieren.

Es bleibt nur noch anzufügen, daß polynomielle deterministische Algorithmen eigentlich nur Spezialfälle des Monte Carlo Algorithmus sind. Bei den polynomiellen deterministischen Algorithmen wird das „Werfen der Münze“ ignoriert und das Ergebnis uneingeschränkt akzeptiert.

8.3.2 Die Komplexitätsklasse ZPP

Betrachten wir Algorithmen, die unter dem Komplement von RP abgeschlossen sind. Diese Klasse wird auch mit coRP abgekürzt. Für Algorithmen dieser Klasse gilt, daß sie nie falsche Negativantworten liefern und höchstens die Hälfte aller Positivantworten falsch ist.

Damit erscheint die Klasse $RP \cap coRP$ als sehr interessant. Ein Problem in dieser Klasse ist dann aus zwei Monte Carlo Algorithmen zusammengesetzt. Einer gibt keine falschen Positivantworten und der andere keine falschen Negativantworten. Wenn wir mit beiden Algorithmen viele unabhängige Experimente durchführen, dann erhalten wir früher oder später definitive Antworten - „yes“ oder „no“. Läßt man beide Algorithmen k-mal unabhängig voneinander laufen, so fällt die Wahrscheinlichkeit, keine definitive Antwort zu erhalten, auf 2^{-k}. Der Unterschied zu einem einfachen Monte Carlo Algorithmus ist also, daß wir am Ende mit Sicherheit eine korrekte Antwort haben. Solche Algorithmen heißen auch Las Vegas Algorithmen.

Die Klasse aller Sprachen von Las Vegas Algorithmen wird ZPP genannt. ZPP steht abkürzend für „polynomial randomized algorithms with zero probability of error“.

8.3.3 Die Komplexitätsklasse PP

Eine Sprache L ist in der Klasse PP, wenn es eine nichtdeterministische polynomiell beschränkte Turingmaschine N, wie in 4. beschrieben, gibt, so daß für jede Eingabe x gilt: $x \in L \Leftrightarrow$ mehr als die Hälfte der Berechnungen von N, die auf x folgen, im akzeptierenden Zustand enden.

Satz 8.3.3.1 $NP \subseteq PP$

Beweis: Sei $L \in NP$ und von einer nichtdeterministischen Turingmaschine N bestimmt. Die Turingmaschine N' wird L mit Mehrheit bestimmen. N und N' seien bis auf zwei Unterschiede identisch. N' besitzt einen neuen Anfangszustand und eine nichtdeterministische Auswahl von seinem Anfangszustand aus. Eine Möglichkeit ist, N' fährt mit den gleichen Berechnungen fort, wie N mit der Eingabe x. Die andere führt genauso viele Berechnungen aus, aber endet immer in einem akzeptierenden Zustand. Jetzt wende man den String x auf N und N' an. Wenn N $p(|x|)$ Schritte rechnet und $2^{p(|x|)}$ Berechnungen anstellt, dann wird N' $2^{p(|x|)+1}$ Berechnungen durchführen. Wobei mindestens die Hälfte der Berechnungen von N' mit „yes“ halten werden. Somit wird die Mehrheit der Berechnungen von N' akzeptiert $\Leftrightarrow$ es mindestens eine Berechnung von N gibt, in der x akzeptiert wird. Und das ist genau dann der Fall, wenn $x \in L$. Deshalb akzeptiert N' L mit Mehrheit und $L \in PP$.

8.3.4 Die Komplexitätsklasse BPP

Die bisher eingeführten Komplexitätsklassen RP, ZPP, PP wurden durch die nichtdeterministische Auswahl motiviert. Wenn wir nun die drei Klassen miteinander vergleichen, dann stellen wir fest, daß es zwischen den ersten beiden und der letzten einen nicht zu verachtenden Unterschied gibt. RP und ZPP sind realistische Vorschläge für praktische Algorithmen. Da $NP \subseteq PP$ gilt, ist PP für die Praxis weniger brauchbar. Ein weiterer Grund für die geringere praktische Einsetzbarkeit von PP liegt darin, daß man PP wegen ihrer Mehrheitskonvention algorithmisch nicht so gut nützen kann. Die Festlegung auf eine Mehrheit, ab der sich ein Algorithmus für „yes“ oder „no“ entscheidet ist bei PP einfach zu schwach.

Diese Überlegung führt zu einer neuen Komplexitätsklasse: Die Komplexitätsklasse BPP (bounded probability of error) enthält alle Sprachen L, für die es eine nichtdeterministische polynomiell begrenzte Turingmaschine mit folgender Eigenschaft gibt: Für jeden String x gilt: Wenn $x \in L$, dann werden mindestens $\frac{3}{4}$ der Berechnungen von N akzeptiert; und wenn $x \notin L$, dann werden mindestens $\frac{3}{4}$ der Berechnungen zurückgewiesen.

Bei BPP haben wir jetzt eine „klare“ Mehrheit für ein Akzeptieren bzw Nichtakzeptieren. BPP läßt sich genau zwischen RP und PP einordnen, anders ausgedrückt: $RP \subseteq BPP \subseteq PP$. Denn läßt man einen Algorithmus, der in RP liegt, zweimal laufen, so ist die Wahrscheinlichkeit für eine falsche Negativantwort schon kleiner als $\frac{1}{4}$. Und BPP ist in PP, denn wenn sich eine Turingmaschine mit „klarer“ Mehrheit für „yes“ bzw „no“ entschieden hat, dann würde sich eine Turingmaschine mit dem Entscheidungskriterium „einfache“ Mehrheit genauso entscheiden.

8.4 Zufallsgeneratoren - Random Sources

Randomisierte Algorithmen sind oft einfach und effizient, aber gibt es sie wirklich? Um nun randomisierte Algorithmen realisieren zu können, müssen diesen Algorithmen Folgen von Zufallsbits zur Verfügung gestellt werden. Aber woher bekommt man diese Zufallsbits, können Computer auf Quellen solcher Zufallsbits zurückgreifen?

Man könnte sich jetzt überlegen eine physikalische Quelle zu konstruieren und zu nutzen, wie zum Beispiel einen Teilchenzähler. Doch da fallen schon einige gravierende Nachteile auf: Die gelieferten Werte sind meist abhängig von den vorausgegangenen und je nach dem, in welcher Umgebung sich die physikalische Quelle befindet, kann sich auch ihr Verhalten ändern.

Einen andere Möglichkeit, eine die in der Praxis eine wichtigere Rolle spielt, ist die, den randomisierten Algorithmen Zufallsgeneratoren zur Verfügung zu stellen. Solchen Zufallsgeneratoren ist eine kurze Bitfolge, „seed“ genannt, gegeben.Die

Seed wird als zufällig vorausgesetzt. Von dieser Seed aus berechnet man anschließend lange Bitfolgen mit mathematischen Eigenschaften, die man zufälligen Bitfolgen unterstellt.

Wie man deutlich sieht, ist man gezwungen, bei den Random Sources Einschränkungen in Kauf zu nehmen. In dem folgenden Kapitel wird erklärt, was man sich genau unter einer perfekten Quelle von Zufallsbits vorzustellen hat. Anschließend wird noch auf eine abgeschwächte Form von Random Sources eingegangen.

8.4.1 Perfect Random Source

Liefert eine Quelle Folgen von Bits mit folgender Eigenschaft: Für alle $n > 0$ und für alle $(y_1, \ldots, y_n) \in \{0,1\}^n$ gilt $prob[x_i = y_i; i = 1, \ldots, n] = 2^{-n}$. Dabei seien die x_i's die Ergebnisse von unabhängigen Experimenten, bei denen jedes x_i mit einer Wahrscheinlichkeit von $\frac{1}{2}$ den Wert Eins annimmt. Dann nennt man diese Quelle Perfect Random Source.

Außerdem ergeben sich zwei wichtige Eigenschaften für Perfect Random Sources:

1. Unabhängigkeit, d.h. die Wahrscheinlichkeit, daß $x_i = 1$ ist, darf nicht von einem früheren oder späteren Bit abhängen.
2. Fairness, d.h. die Wahrscheinlichkeit, daß $x_i = 1$ ist, beträgt $\frac{1}{2}$.

Mit Hilfe einer solchen Quelle ist es möglich, Algorithmen für eine Sprache in RP, ZPP, oder BPP zu implementieren. Wobei jetzt die entsprechenden Turingmaschinen zusätzlich zur problemspezifischen Eingabe x noch eine Folge von Zufallsbits, die eine Perfect Random Source liefert, als Eingabe bekommen. Bei jedem Rechnungsschritt der Turingmaschinen bestimmt dann zum Beispiel das letzte Bit der Bitfolge, welche nichtdeterministische Auswahl getroffen wird. Bevor schließlich zum nächsten Berechnungsschritt übergegangen wird, streicht die Turingmaschine das letzte Bit aus der Folge.

8.4.2 Slightly Random Source

Da aber Perfect Random Sources schwer bzw überhaupt nicht realisierbar sind, wurde das Konzept der Zufälligkeit etwas abgeschwächt.

Seien δ eine Zahl mit der Eigenschaft $0 < \delta \leq \frac{1}{2}$ und p eine Funktion, die einem beliebig langen String aus Nullen und Einsen $\{0,1\}^*$ einen Wert aus dem Intervall $[\delta, 1-\delta]$ zuweist. p sei eine komplexe Funktion und vollständig unbekannt. Eine δ-Random Source S_p ist eine Zufallsvariable, die als Wert eine unendlich lange Bitfolge enthält. Die Wahrscheinlichkeit, daß die ersten n Bits mit den Werten $y_1, \ldots, y_n$ übereinstimmen, beträgt

$$\prod_{i=1}^{n} (y_i p(y_1 \ldots y_{i-1}) + (1-y_i)(1-p(y_1 \ldots y_{i-1})))$$

Damit ist nach dieser Formel die Wahrscheinlichkeit, daß an der i-ten Stelle eine Eins steht, durch $p(y_1 \dots y_{i-1})$ gegeben. Der Wahrscheinlichkeitswert liegt zwischen δ und $1 - \delta$ und hängt von allen vorherigen Ergebnissen der Quelle ab. Wählt man $\delta = \frac{1}{2}$, dann hat man eine Perfect Random Source. Eine δ-Quelle mit $\delta < \frac{1}{2}$ wird Slightly Random Source genannt.

Eine Slightly Random Source ist durch die inneren Abhängigkeiten ein realistischeres Modell als die Perfect Random Source. Doch leider scheinen sie damit für randomisierte Algorithmen wieder unbrauchbar zu sein. Sie sind aber trotzdem sehr nützlich, nur können sie nicht direkt verwendet werden. Doch kann man mit ihnen, zwar mit einem gewissen Effizienzverlust jeden randomisierten Algorithmus simulieren.

8.4.3 Slightly Random Source - Komplexitätsklassen

Bei der Verwendung von Slightly Random Sources sind damit auch die neu eingeführten Komplexitätsklassen nicht mehr richtig. Das bedeutet sie müssen den neuen Umständen angepaßt werden. Exemplarisch soll dies mit den Komplexitätsklassen RP und BPP geschehen.

Sei N eine exakte, polynomiell beschränkte, nichtdeterministische Turingmaschine mit zwei Wahlmöglichkeiten in jedem Rechnungsschritt. Diese Wahlmöglichkeiten werden mit 0-choice und 1-choice bezeichnet. Bei einer Berechnung, die auf eine Eingabe x folgt, baut N einen vollständigen Binärbaum der Tiefe $n = p(|x|)$ auf. Jede der beiden Kanten, die von einem der internen Knoten ausgehen, entspricht einer Wahl.

Es sei δ eine Zahl zwischen 0und$\frac{1}{2}$. Eine δ-Zuweisung F an N(x) ist eine Zuweisung von δ-Werten an die Kanten des Binärbaums aus dem Intervall $[\delta, 1 - \delta]$ derart, daß die Summe der beiden Kanten, die von einem Knoten ausgehen, 1 beträgt.

Für jedes Blatt l des Binärbaums definiert $prob[l] = \prod_{a \in P[l]} F(a)$ die Wahrscheinlichkeit, daß dieses Blatt am Ende aller Berechnungen ausgewählt wurde. P[l] steht abkürzend für den Pfad, der zum Blatt l führt. $prob[M(x) = \text{„}yes\text{“}|F]$ schließlich wird durch die Summe von prob[l] aller „yes“-Blätter bestimmt.

Mit diesen Voraussetzungen sind wir in der Lage, die Komplexitätsklassen BPP und RP neu zu definieren:

Eine Sprache L ist in δ-RP, wenn es eine wie oben beschriebene Turingmaschine N gibt, für die gilt:
Wenn $x \in L$, dann gilt $prob[M(x) = \text{„}yes\text{“}|F] \geq \frac{1}{2}$; und wenn $x \notin L$, dann gilt $prob[M(x) = \text{„}no\text{“}|F] = 1$ für alle δ-Zuweisungen F.

Eine Sprache L ist in δ-BPP, wenn es eine wie oben beschriebene Turingmaschine N gibt, für die gilt:
Wenn $x \in L$, dann gilt $prob[M(x) = \text{„}yes\text{“}|F] \geq \frac{3}{4}$; und wenn $x \notin L$, dann gilt $prob[M(x) = \text{„}no\text{“}|F] \geq \frac{3}{4}$ für alle δ-Zuweisungen F.

Sonderfälle:

- $\delta = 0$: 0-RP=0-BPP=P
- $\delta = \frac{1}{2}$: $\frac{1}{2}$-RP=RP und $\frac{1}{2}$-BPP=BPP

Der Sonderfall b) läßt sich aber erweitern zu:
Für alle $\delta > 0$: δ-RP=RP bzw. δ-BPP=BPP.
Damit folgt, daß wir mit Slightly Random Sources durchaus randomisierte Algorithmen laufen lassen können.

Literaturverzeichnis

[Papa] C.H. Papadimitriou. Computational Complexity 241-267

[Karp] R.M. Karp. Discrete Applied Mathematics 34 (1991) 165-201

Vortrag 9

Primzahltest

Christopher Schmid
Technische Universität München
schmidc@informatik.tu-muenchen.de

9.1 Allgemeines zu Primzahlen

Für viele Numerische Verfahren sind große Primzahlen von elementarem Interesse (bspw. für Kryptographische Verfahren wie RSA). Der bekannte Mathematiker Gauss sagte einmal:

> *Das Problem der Unterscheidung von zusammengesetzten und Primzahlen , sowie das Auflösen ersterer in ihre Primfaktoren ist eines der wichtigsten und nützlichsten in der Arithmetik. (..) Nichtsdestotrotz müssen wir gestehen, daß alle Methoden, die wir bisher entdeckt haben, nur auf Spezialfälle beschränkt oder zu arbeitsaufwendig und komplex sind.*

Glücklicherweise wurden bis heute verbesserte Verfahren entwickelt, die eine Aussage über die Primheit von Zahlen machen. Einige von diesen sollen in dieser Ausarbeitung vorgestellt werden.

9.2 Wichtige Grundbegriffe

Zunächst sollen an dieser Stelle noch einmal kurz grundlegende Begriffe und Defininitionen erläutert werden:

1. **Satz 9.2.1** (LAGRANGE) *Ist G eine endliche Gruppe und H eine Untergruppe, so gilt* $\#H \mid \#G$.

2. **Satz 9.2.2** (FERMAT) *Ist p eine Primzahl und a eine natürliche Zahle mit* $a \not\equiv 0 \ mod\, p$*, so gilt*
$$b^{n-1} \equiv 1 \ mod\, n$$

3. **Folgerung:** Ist p eine Primzahl und a eine natürliche Zahl, so gilt
$$a^p \equiv a \bmod p$$

4. **Satz 9.2.3** *Chinesischer Restesatz Sind* $m_1, \ldots, m_r$ *paarweise teilerfremde positive ganze Zahlen und* $M = m_1 \cdot \ldots \cdot m_r$*, so ist die Abbildung*
$$\varphi \ : \ \mathbf{Z}_M \to \mathbf{Z}_{m_1} \times \ldots \times \mathbf{Z}_{m_r} \ : \ x \mapsto (x \ mod\, m_1, \ldots, x \ mod\, m_r)$$
eine Bijektion.

5. **Defininition 9.2.1** *Einheitengruppe modulo n Die Menge der invertierbaren Elemente in* $\mathbf{Z}_n$ *bildet eine Gruppe, die sog.* Einheitenguppe E_n*:*
$$E_n = \{e \in \mathbf{Z}_n \,;\, ggT(e,n) = 1\}$$
also $\#U_n = \phi(n)$ *(*EULER*-Zahl).*
$$\phi(n) = \prod_{\substack{i \\ \alpha_i > 0}} (p_i^{\alpha_i} - p_i^{\alpha_{i-1}}) = n \cdot \prod_{p|n, p \ prim} \left(1 - \frac{1}{p}\right)$$

6. **Defininition 9.2.2** *Jacobi-Symbol Das Jacobi Symbol* $(\frac{p}{q})$ *ist definiert als entweder 0,1, oder -1 für alle natürlichen Zahlen* $p \geq 0$ *und alle ungeraden natürlichen Zahlen* $q > 1$ *durch die Regeln*
   - $(\frac{p}{q}) = p^{(q-1)/2} \ mod \ q$*, falls q prim,*
   - $(\frac{p}{q}) = (\frac{p}{q_1})..(\frac{p}{q_t})$*, falls q ein Produkt der Primzahlen* $q_1..q_t$ *ist.*

   Es kann sehr effizient berechnet werden durch die Reziprokregel:

   $(\frac{0}{q}) = 0; (\frac{1}{q}) = 1; (\frac{p}{q}) = (\frac{p \ mod \ q}{q}); (\frac{2}{q}) = (-1)^{(q^2-1)/8}; (\frac{p_1 p_2}{q}) = (\frac{p_1}{q})(\frac{p_2}{q}); (\frac{p}{q}) = (-1)^{(p-1)(q-1)/4}(\frac{q}{p})$ *wenn p und q ungerade ;*

7. Zur Schreibweise: Im Folgenden bezeichnen wir den ggT zweier Zahlen a, b mit (a, b).

9.3 Allgemeines zu Pseudo-Primtests

9.3.1 Idee

Man möchte wissen, ob eine gegebene natürliche Zahl n prim ist. Finde ich eine Zahl b, 1 $\leq$b$\leq$ n die mir beweist, daß n zusammengesetzt ist (z.B. $b|n$), so nenne ich dieses b einen *Zeugen für die Zusammengesetztheit von n* und ich weiss, daß n nicht prim ist.

Finde ich nach mehreren Versuchen kein b, welches mir die Zusammengesetztheit von n beweist, und weiß ich überdies, daß im Intervall $[1..n-1]$ sich eine bestimmte Anzahl von Zeugen b befindet, kann ich mit bestimmter Wahrscheinlichkeit sagen, daß n keine Primzahl ist. Dies ist zwar nur eine Annahme, aber durch genügend viele Tests mit unterschiedlichen b kann ich meine Sicherheit beliebig erhöhen.

9.3.2 Defininition

Ein Pseudo-Primtest besteht aus

- einer *Zeugenmenge* $W \subseteq \{\,(b,n)\,;\,1 \leq b \leq n\,\}$
- einem Verfahren t für die Entscheidung, ob ein $(a,b) \in W$

Folgenden Eigenschaften müssen gelten:

- $(b,n) \in W \Rightarrow n$ ist zusammengesetzt
- Laufzeit für $t(b,n)$ ist polynomial in $\log n$

Einen solchen Test nennen wir *brauchbar*, falls gilt:

- $\exists c > 0$ so daß für zusammengesetzte n gilt

$$\#\underbrace{\{\,b\,;\,1 \leq b \leq n, (b,n) \in W\,\}}_{W_n} \geq c \cdot n$$

  d.h. die Zahl der Zeugen $\# W_n$ für die Zusammengesetztheit von n ist direkt proportional zu n.

9.3.3 Anwendung

Iteriert man brauchbare Pseudo-Primtests erhält man effiziente *probabilistische Primzahltests*. Man geht folgendermaßen vor:

- gegeben n, iteriere Test mit input (b,n) k-mal, wobei b mit Gleichverteilung aus $[1 \ldots n]$ gewählt wird

- wird mindestens einmal $(b, n) \in W$ festgestellt:

  n ist mit Sicherheit zusammengesetzt!

- falls k-mal $(b, n) \notin W$ festgestellt wird:

  n ist (wahrscheinlich) Primzahl. Das ist falsch mit Wahrscheinlichkeit $\leq (1-c)^k$!

Mit diesen Tests können wir sowohl sehr effizient Zahlen auf Primheit überprüfen, oder große Primzahlen finden. Dies erreicht man bspw. dadurch, daß man von einer sehr großen Zahl (meist eine hohe 2er-Potenz) schrittweise mehr und mehr abzieht oder aufaddiert und die erhaltenen Zahlen auf Primheit überprüft.

9.3.4 Beispiele für Pseudo-Primzahltests

1. Teilbarkeits-Test:

   $$D = \{\,(b, n)\,;\, 1 < b < n \,\wedge\, b \,|\, n\,\}$$

   ist wegen $\#D_n = d(n) - 2$ unbrauchbar, da $\frac{d(n)-2}{n}$ für zusammengesetzte n nicht nach unten (gegen 0 hin) beschränkt ist

   Bsp.: $D_{24} = \{(2,24), (3,24), (4,24), (6,24), (8,24), (12,24)\}; \#D_{24} = 6$

2. *GGT*-Test (Euklid)

   $$E = \{\,(b, n)\,;\, 1 < b < n \,\wedge\, \mathrm{ggT}(b, n) \neq 1\,\}$$

   ist besser, aber ebenfalls unbrauchbar:

   $$\#E_n = n - \phi(n)$$

   Bsp.: $E_{24} = \{(2,24), (3,24), (4,24), (6,24), (8,24), (9,24), (10,24), (12,24), (14,24), (15,24), (16,24), (18,24), (20,24), (21,24), (22,24)\}; \#E_{24} = 15$

3. Fermat-Test

   $$F = \{\,(b, n)\,;\, 1 < b < n \,\wedge\, (\mathrm{ggT}(b, n) \neq 1 \;\vee\; b^{n-1} \neq 1 \bmod n)\,\}$$

   ist eigentlich praktikabel. Leider gibt es jedoch zusammengesetzte Zahlen, die sich hierin wie Primzahlen verhalten. Man nennt diese nach Ihrem Entdecker Carmichael-Zahlen. Für diese Zahlen gilt also:

   $$b^{n-1} \equiv 1 \bmod n \text{ für alle } 1 \leq b < n \text{ mit } \mathrm{ggT}(b, n) = 1$$

   z.B. $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 * 13 * 17$, $n = 1729 = 7 \cdot 13 \cdot 19$... Seit 1992 (!!) weiß man, daß es unendlich-viele Carmichael-Zahlen gibt (Granville, Pomerance et al.)

9.4 Der Primzahltest nach Miller und Rabin

9.4.1 Prinzip

Zunächst die Definition der Bedingung $W_n(b)$, die ein Zeuge für die Zusammengesetzheit von n erfüllen muß:

Defininition 9.4.1 *Sei n eine natürliche Zahl. Die Eigenschaft $W_n(b)$ einer natürlichen Zahl b sei folgendermaßen definiert:*

1. $1 \leq b < n$;
2. (a) $b^{(n-1)} \not\equiv 1 \bmod n$, *oder*
   (b) $\exists i : 2^i | (n-1)$ *und* $1 < gcd(b^{(n-1)/2^i} - 1, n) < n$.

Dieses b nennt man also einen Zeugen für die Zusammengestztheit von n.

Trivialerweise gilt auch für Primzahlen n, daß diese Bedingung für kein b erfüllt werden kann, denn $b^{(n-1)} \equiv 1 \bmod n$ gilt immer für Primzahlen n nach dem Satz von Fermat (Bedingung 2.a). Außerdem haben Primzahlen mit keiner kleineren Zahl einen ggT ungleich 1 (Bedingung 2.b).

Die Zeugenmenge sieht ganz einfach wie folgt aus:

$$MR_1 = \{(b, n); W_n(b)\}$$

Ist die Zahl n zusammengesetzt, so kann man mit einiger Sicherheit einen Zeugen b finden, der $W_n(b)$ erfüllt:

Satz 9.4.1 *Ist $4 < n$ zusammengesetzt, dann gilt*

$$\frac{3(n-1)}{4} \leq \#\{b | W_n(b)\}$$

.

Beweis:
Einige Deklarationen:

- Bezeichne $e(m)$ als größtes i mit $2^i | m$
- Für eine Sequenz $m =< m_1, ..., m_k >$ ist $res(b, m) =< s_1, ..., s_k >$, wobei $s_i = res(b, m_i), 1 \leq i \leq k$

Wir benötigen zunächst zwei Lemmata:

Lemma 9.4.1 *Gelte* $m_i|n$ *für* $1 \leq i \leq k$, *und* $(m_i, m_j) = 1$, $1 \leq i < j \leq k$ *und sei* $m =< m_1, ..., m_k >$. *Für jedes* $s =< s_1, ..., s_k >, s_1 \in E_{m_1}, ..., s_k \in E_{m_k}$, *ist die Anzahl der Elemente in*

$$\{b \in E_n | res(b, m) = s\}$$

stets diesselbe.

Beweis:

- Sei t Produkt der Primteiler von n, außer den m_i
- Sei $t \neq 1$ ($t = 1$ geringfügig veränderter Beweis)
- Nach CRT $\exists b \in \mathbf{N} : res(b, < m, t >) =< s_1, ..., s_k, 1 >$
- Da $m_i|n, 1 \leq i \leq k$ und $t|n$, gelte $1 \leq b < n$
- Es gilt $b \in E_n$. Falls nicht
  - gäbe es Primzahlen p mit $p|n$ und $p|b$
  - aus $p|n \Rightarrow p|m_i$ oder $p|t$
  - da $s_i = b - q_i m_i$, folgt $p|s_i$, was $(m_i, s_i) = 1$ widerspricht
  - $p|t$ führt zum Widerspruch $p|1$
- Definiere die Operation $res(, m)|E_n$ als f. Aus obigem Abschnitt ergibt sich, daß $f : E_n \to E_{m_1} \times ... \times E_{m_k} = G$ ein Homomorphismus von E_n nach G ist. Daher hat $f^{-1}(< s_1, ..., s_k >)$ immer diesselbe Anzahl von Elementen für alle $< s_1, ..., s_k > \in G$.

Lemma 9.4.2 *Seien* $p_1 \neq p_2$ *Primzahlen* , $q_1 = p_1^{k_1}, q_2 = p_2^{k_2}$. *Gelte* $q_1 q_2 | n$. *Sei ferner* $t_i = (\phi(q_i), n - 1), m_i = \phi(q_i)/t_i, i = 1, 2$. *Nun gilt: Höchstens* $\phi(n)/m_1 m_2$ *der Zahlen* $b \in E_n$ *erfüllen* $W_n(b)$ *nicht. Ist* t_1 *gerade, dann erfüllen höchstens* $\phi(n)/2m_1 m_2$ *der Zahlen* $b \in E_n$ $W_n(b)$ *nicht.*

Schritt 1:
Beweis:

- Sei a_i eine Primitive Einheitswurzel $\bmod q_i, i = 1, 2$, also $a_i^t \equiv 1 \bmod n$ gdw. $\phi(q_i)|t$
- Sei $b \in E_n$, damit also $(b, q_i) = 1$ und $b \equiv a_i^{r_i} \bmod q_i$ für zwei r_1, r_2

- Da $q_1 q_2 | n$

$$b^{(n-1)} (\equiv a_i^{r_i(n-1)}) \equiv 1 \bmod n \Rightarrow \tag{2}$$

$$\phi(q_i) | r_i(n-1) \Rightarrow$$

$$m_i | r_i(n-1) \Rightarrow m_i | r_i$$

- Daher ist, wenn $res(b, < q_1, q_2 > = < s_1, s_2 >$ (2) höchstens für $1/m_1 m_2$ aller Paare $< s_1, s_2 >$ erfüllt.
- Diese Paare kommen nach Lemma 4.1 alle gleichoft als Rest $\bmod b$ vor $\Rightarrow$ (2) ist für höchstens $\phi(n)/m_1 m_2$ der Zahlen $b \in E_n$ erfüllt. Ist aber (2) nicht erfüllt, so ist $W_n(b)$ wahr.

Schritt 2:

- Wir zeigen, daß wenn t_1 gerade und (2) erfüllt ist für b, für mindestens die Hälfte der Fälle 2.(b) aus Definition 4.1. erfüllt, also b immer noch ein Zeuge ist.

  Fall 1:

- Sei jetzt t_1 gerade und $e(t_1) = e(t_2)$
- Wähle i so, daß $(n-1)/2^i = d(i) \in \mathbf{N}$ und $t_j \not| d(i), t_j/2 | d(i), j = 1, 2$.
- $b \in E_n$ erfüllt (2) mit den obigen Bedingungen für $a_1, a_2, r_1, r_2, h_1, h_2$.
- Ist nun h_1 ungerade und h_2 gerade, so gilt

$$\phi(q_1) \not| h_1 m_1 d(i) \wedge \phi(q_2) \not| h_2 m_2 d(i)$$

  Also

$$b^{d(i)} \not\equiv 1 \bmod q_1 \tag{3}$$

$$b^{d(i)} \equiv 1 \bmod q_2 \tag{4}$$

  Für so ein $b \in E_n$ ist 2.(b) erfüllt.

- Dasselbe gilt, wenn h_1 gerade und h_2 ungerade ist. Sind h_1 und h_2 vorzeichengleich ist $(b^{(n-1)/2} - 1, n)$ gleich 1 oder n.
- Also erfüllen höchstens $\phi(n)/2 m_1 m_2$ der $b \in E_n$ $W_n(b)$ nicht.

  Fall 2:

- Sei t_1 gerade , $e(t_1) > e(t_2)$
- Wähle i und j so ,daß $t_2|d(i), t_1 \not| d(i)$ und $t_1/2^j|d(i)$, wobei j minimal ist (also $j \leq 1$)
- $b \in \mathbf{N}$ erfülle weiterhin (2)
- Es gilt $\phi(q_2)|h_2m_2d(i) \Rightarrow$ (4) ist wahr
- Aber $b^{d(i)} \equiv 1 \bmod q_1$ gilt nur dann, wenn $\phi(q_1)|h_1h_2d(i)$, d.h. wenn $2^j|h_1$.
- Außer dem Fall, daß $2^j|h_1$ gilt, ist (3) wahr $\Rightarrow W_n(b)$ ist wahr für höchstens $\phi(n)/2^jm_1m_2b \in E_n$. Aber es ist ja $j \leq 1$.

Beweis für Satz 4.1.

- Sei $4 < n$ zusammengesetzt. Ist $1 \leq b < n$ und $(b, n) \neq 1$ ist $W_n(b)$ wahr (nach 2.(b)). Es genügt also zu zeigen, daß mindestens $3/4\phi(n)$ der $b \in E_nW_n(b)$ erfüllen.

  Fall 1: $n = p^k, 1 < k, p$ Prim
- $\phi(n) = p^{(k-1)}(p-1)$ und $p \not|(n-1)$, also $p^{k-1} \leq \phi(p^k)/(\phi(p^k), n-1) = m$
- Nach Lemma 4.2 (für eine einzelne Primzahl betrachtet) erfüllen höchstens $\phi(n)/m$ der $b \in E_n$ nicht $W_n(b)$.
- Aus $n > 9 \Rightarrow p \geq 5$ oder $2 < k$, daher $4 \leq m$. Ist $n = 9$ gibt es nur zwei Nichtzeugen.

  Fall2: n besitzt mindestens zwei Primteiler p_1, p_2
- Sei $q_i = p_i^{k_i}, i = 1, 2$ maximal mit $q_1|n, q_2|n$
- Gelte $\phi(q_1) \not|(n-1)$
- Es gilt $2 \leq \phi(q_1)/(\phi(q_1), n-1) = m$.

  Fall (a): $\phi(q_2) \not|(n-1)$
- $\Rightarrow 2 \leq m_2 \Rightarrow$ (1) folgt nach Lemma 4.2.

  Fall (b):
- Ist $p_2 \neq 2$ folgt, daß $t_2 = \phi(q_2)$ gerade ist und nach Lemma 4.2. höchstens $\phi(n)/2m_1 \leq \phi(n)/4$ der $b \in E_nW_nb)$ nicht erfüllen.
- Ist $p_2 = 2$ gilt $\phi(n) \leq (n-1)/2$ und damit erfüllen höchstens $\phi(n)/m_1 \leq (n-1)/4$ der $1 \leq b < nW_n(b)$.

  Fall 3: $n = p_1^{k_1}p_2^{k_2}...p_r^{k_r}, 2 \leq r, \phi(p_i^{k_i}|(n-1), 1 \leq i \leq r$

- Aus $p_i^{k_i-1}(p_i-1)|(n-1) \Rightarrow k_i = 1, 1 \leq i \leq r$.

- Eine einfache Rechnung zeigt $r \geq 3$.

- Seien p_1, p_2, p_3 verschiedene Primteiler von n mit $(p_i-1)|(n-1), 1 \leq i \leq 3$.

- Annahme: $e(p_1-1) = e(p_2-1) = e(p_3-1)$

- Sei i so gewählt, daß für $d(i) = (n-1)/2^i$ gilt, daß $(p_j-1) \not| d(i), (p_j-1)/2|d(i), 1 \leq j \leq 3$.

- Ist $b = a_i^{r_i} \mod p_i$ (a_i Primitive Einheitswurzel $\mod p_i$), $1 \leq i \leq 3$, ergibt sich nach dem Beweis von Lemma 4.2. $1 < (b^{d(i)}-1, n) < n$, es sei denn alle r_i sind entweder gerade oder ungerade.

- Ist, z.B., r_1 gerade und r_2 ungerade, dann gilt $p_1|(b^{d(i)-1})$ und $p_2 \not|(b^{d(i)-1})$. Daher ist $W_n(b)$ nicht erfüllt für höchstens $2/8\phi(n)$ der $b \in E_n$.

- Die Fälle $e(p_3-1) = e(p_2-1) < e(p_3-1)$ und $e(p_3-1) < e(p_2-1) \leq e(p_3-1)$ funktionieren in ähnlicher Weise.

9.4.2 Algorithmus

Sei eine natürliche Zahl n gegeben, mit $n-1 = 2^l m$, m ist ungerade.

1. Wähle eine Zahl k, $1 \leq k < n$ und zufällig $1 \leq b_1, ..., b_k < n$.

2. Berechne für $b = b_i$ $b^m \mod n$.

3. Berechne die Reste mod n von $b^{2m}, ..., b^{2^l m}$.

4. Wenn der Rest von $b^{n-1} \bmod n$ nicht gleich 1 ist hält $W_n(b)$. Ist dieser Rest gleich 1, bestimme $(res(b^{2^i m}, n) - 1, n)$ für alle $1 \leq i \leq l$.

5. Ist einer dieser ggT weder 1 noch n (n besitzt Teiler!) ist $W_n(b)$ erfüllt.

6. Sollte für ein $1 \leq i \leq k$ $W_n(b_i)$ erfüllt sein ist n sicher zusammengesetzt. Sind alle $W_n(b_i)$ nicht erfüllt, entscheidet der Test , daß n wahrscheinlich eine Primzahl ist.

9.4.3 Beispiele

1. $n = 25, n-1 = 3 \cdot 8, b = 7$:

$$\begin{aligned} 7^3 &\equiv 18 \bmod 25 \\ 7^6 &\equiv 24 \bmod 25 \\ 7^{12} &\equiv 1 \bmod 25 \\ 7^{24} &\equiv 1 \bmod 25 \\ \Rightarrow ggT - Bestimmung & \\ (18-1, 25) &= 1 \\ (24-1, 25) &= 1 \\ (1-1, 25) &= 25 \end{aligned}$$

   Test bringt keine Information! $b = 7$ ist kein MR_1-Zeuge.

2. $n = 25, n-1 = 3 \cdot 8, b = 2$:

$$\begin{aligned} 2^3 &\equiv 8 \bmod 25 \\ 2^6 &\equiv 14 \bmod 25 \\ 2^{12} &\equiv 21 \bmod 25 \\ 2^{24} &\equiv 16 \bmod 25 \end{aligned}$$

   Test zeigt, daß 25 nicht prim ist! $b = 2$ ist MR_1-Zeuge.

3. $n = 2047 = 23 \cdot 89, n-1 = 2 \cdot 1023, b = 2$:

$$\begin{aligned} 2^{1023} &\equiv 1 \bmod 2047 \\ 2^{2046} &\equiv 1 \bmod 2047 \\ (1-1, n) &= n \end{aligned}$$

   Test bringt keine Information! $b = 2$ ist kein MR_1-Zeuge.

4. $n = 2047 = 23 \cdot 89, n-1 = 2 \cdot 1023, b = 3$:

$$\begin{aligned} 3^{1023} &\equiv 1565 \bmod 2047 \\ 3^{2046} &\equiv 1013 \bmod 2047 \end{aligned}$$

   Test zeigt, daß 2047 nicht prim ist! $b = 3$ ist MR_1-Zeuge.

5. $n = 341 = 11 \cdot 31, n-1 = 4 \cdot 85, b = 2$

$$\begin{aligned} 2^{85} &\equiv 32 \bmod 341 \\ 2^{170} &\equiv 1 \bmod 341 \\ 2^{340} &\equiv 1 \bmod 341 \\ (32-1, 341) &= 31 \end{aligned}$$

Test zeigt, daß 341 nicht prim ist! $b = 2$ ist MR_1-Zeuge.

6. $n = 561 = 3 \cdot 11 \cdot 17, n - 1 = 16 \cdot 35, b = 2$

$$\begin{aligned} 2^{35} &\equiv 263 \bmod 561 \\ 2^{70} &\equiv 166 \bmod 561 \\ 2^{140} &\equiv 67 \bmod 561 \\ 2^{280} &\equiv 1 \bmod 561 \\ 2^{560} &\equiv 1 \bmod 561 \\ (263 - 1, 561) &= 1 \\ (166 - 1, 561) &= 11 \end{aligned}$$

Test zeigt, daß 561 nicht prim ist! $b = 2$ ist MR_1-Zeuge.
In der Tat ist 561 die kleinste CARMICHAEL-Zahl.

7. $n = 2243, n - 1 = 2 \cdot 1121, b = 2$

$$\begin{aligned} 2^{1121} &\equiv 2242 \bmod 2243 \\ 2^{2242} &\equiv 1 \bmod 2243 \\ (2242 - 1, 2243) &= 1 \end{aligned}$$

$b = 2$ ist kein MR_1-Zeuge für die Zusammengesetztheit von 2243. Solche Zeugen darf und kann es nicht geben, denn 2243 ist Primzahl!

9.4.4 Komplexität des Algorithmus

Die Exponentiation mod n (Schritt 2) kann mit etwa $1.5 \log_2 m$ Multiplikationen erreicht werden. Da jedes Mal wenn bei dieser Exponetiation eine Zahl d gröser als n erreicht wird, der Rest modulo n gebildet wird, finden nur Multiplikationen von Zahlen kleiner als n sowie Divisionen von Zahlen kleiner als n^2 statt. Für die restlichen Exponentiationen (Schritt 3) benötigt man, da mit jedem Schritt nur modulo n quadriert wird, l Multiplikationen von Zahlen kleiner als n und l Divisionen von Zahlen kleiner als n^2 . Wir benötigen also $l + 1.5 \log_2 m$ Multiplikationen, sowie $l + 1.5 \log_2 m$ Divisionen.

Da aber aus $n - 1 = 2^l m$ folgt, daß

$$\log_2 n \approx l + \log_2 m$$

und weiterhin gilt, daß

$$l + 1.5 \log_2 m < 1.5(l + \log_2 m) = 1.5 \log_2 n$$

können wir die Komplexität des Algorithmus auch in Abhängigkeit von n angeben: Wir brauchen also höchstens $1.5 \log_2 n$ Multiplikationen, sowie $1.5 \log_2 n$ Divisionen, zusammen also $3 \log_2 n$ Schritte.

Die ggT-Bildung (Schritt 4) kann, da eine einzelne in höchstens $\log_2 n$ Subtraktionen und Divisionen durch 2 erreicht werden kann, mit $l \cdot \log_2 n$ Schritten geschehen.

Insgesamt benötigen wir also

$$3 \log_2 n + l \cdot \log_2 n$$

Operationen.

Satz 9.4.2 *Der obige Algorithmus benötigt für* $n-1 = 2^l m$, m *ungerade, höchstens* $k \cdot (3 \cdot \log_2 n + l \cdot \log_2 n)$ *Schritte. Ist* n *eine Primzahl ist das Ergebnis immer korrekt. Ist* n *zusammengesetzt, kann der Test* n *irrtümlich als Primzahl deklarieren, aber die Wahrscheinlichkeit hierfür ist kleiner als* $(\frac{1}{2})^{2k}$.

Beweis: Ein Fehler tritt nur dann auf, wenn alle ausgewählten $b_i, 1 \leq i \leq k$ in einem Durchlauf des Algorithmus nicht Zeugen für die Zusammengesetzheit von n sind. Die Wahrscheinlichkeit, einen solchen auszuwählen ist kleiner als $\frac{1}{4}$, also muß die Wahrscheinlichkeit k dieser Zahlen auszuwählen kleiner als $(\frac{1}{4})^k = (\frac{1}{2})^{2k}$ sein.

9.4.5 Brauchbarkeit des Miller-Rabin-Tests

Satz 9.4.3 *Der* Miller-Rabin-*Tests ist ein brauchbarer, effizienter Pseudo-Primtest.*

Beweis:

- Der Miller-Rabin-Test ist ein Pseudo-Primtest, denn er besitzt eine Zeugenmenge und ein Entscheidungsverfahren.
- Der Test ist effizient. Dies ergibt sich aus obiger Komplexitätsüberlegung.
- Der Test ist auch brauchbar. Dies ergibt sich direkt aus dem Beweis von Satz 4.1.

9.4.6 Experimentelle Ergebnisse

Die Experimente mit diesem Algorithmus fanden auf einem mittelgroßen Computer statt. Als erstes wurden Mersenne-Zahlen getestet, also Zahlen der Form $n = 2^p - 1$, p prim. Für diese Zahlen mit $p \leq 500$ benötigte der Rechner etwa 10 Minuten ohne einen einzigen Fehler.

Neben weiteren Test wurden auch große Primzahlen gesucht, startend von einer hohen Zweierpotenz, von der man immer größere ungerade Zahlen abzog. Die Zahlen

$$2^{300} - 153, \quad 2^{400} - 593,$$

wurden als größte Primzahlen identifiziert.

9.5 Variante von D.Knuth et.al.

9.5.1 Prinzip

Wir betrachteten gerade eben die Variante des Tests, wie sie M.O.RABIN vorgeschlagen hat. D.KNUTH formulierte seinen Test in leicht abgewandelter Weise. Wir werden später sehen, daß sich seine Testdefinition leicht aus der ersteren ableiten läßt.

Zunächst die Defininition des Pseudo-Primtests nach MILLER:

Defininition 9.5.1 *Sei n eine natürliche Zahl mit* $n-1=2^s t$*, wobei s und t natürliche Zahlen und t ungerade ist. Wir sagen, n besteht den* MILLER-*Test für die Basis b, wenn entweder* $b^t \equiv 1 \ mod \ n$ *oder* $b^{2^j t} \equiv -1 \ mod \ n$ *für irgendein j mit* $0 \leq j \leq s-1$.

Besteht eine Zahl den MILLER-Test, so ist sie also kein Zeuge für die Zusammengesetztheit von n. Wir zeigen jetzt, daß, falls n prim ist, n den MILLER-Test besteht für alle Basen b mit $n \not| b$:

Satz 9.5.1 *Ist n eine Primzahl und b eine natürliche Zahl, mit* $n \not| b$*, dann besteht n den* MILLER-*Test.*

Beweis: Ist n eine Primzahl gilt:

- $b^{n-1} \equiv 1 \bmod n$
- $\mathbf{Z}_n$ ist ein *Körper.*

Die obige Folge endet also mit einer 1. Da in einem Körper die 1 nur zwei Quadratwurzeln hat, nämlich sich selbst und die -1, gilt:

- entweder ist bereits $b^t \equiv 1 \text{ mod } n$ und alle Glieder der obigen Folge sind dann ebenfalls $\equiv 1 \text{ mod } n$;
- oder es gibt eine Stelle $i \in [0..s-1]$ mit $b^{2^i t} \equiv -1 \text{ mod } n$

Die Zeugenmenge sieht also wie folgt aus:

$$\begin{aligned} MR_2 = \{ \quad & (b,n); 1 < b < n \wedge \\ & (n-1 = 2^q m, m \text{ ungerade }) \wedge \\ & b^m \not\equiv 1 \text{ mod } n \wedge \\ & \forall i \in [0..q-1] : b^{m2^i} \not\equiv -1 \text{ mod } n\} \end{aligned}$$

Für zusammengesetzte Zahlen gilt jetzt folgendes:

Satz 9.5.2 *Ist n eine natürliche Zahl, n ungerade und zusammengesetzt, dann besteht n den* MILLER-*Test für höchstens $\frac{n-1}{4}$ Basen b mit $1 \leq b \leq n-1$.*

Der Beweis hierfür folgt in 5.2. Wie weiter oben erwähnt, können wir obigen Pseudo-Primtest durch Iteration in einen effizienten *probabilistischen Primzahltest* überführen:

Defininition 9.5.2 (Probabilistischer Primzahltest von Rabin) *Sei n eine natürliche Zahl. Wähle k verschiedene natürliche Zahlen $b_1, b_2, ..., b_k$ mit $1 < b_i < n, 1 \leq i \leq k$ und führe den* MILLER-*Test auf n mit eben jenen Basen aus. Ist n zusammengesetzt, ist die Wahrscheinlichkeit, daß n alle Tests besteht, kleiner als $(\frac{1}{4})^k$.*

Ich kann also , indem ich mit genügend vielen Basen b den MILLER-Test ausführe, mit beliegig hoher Wahrscheinlichkeit sicher sein, daß die Zahl eine Primzahl ist.

9.5.2 Überführung der Methode von RABIN in die von KNUTH

Wir verwenden wieder folgende Notation:

Sei $n-1 = 2^l \cdot m$, wobei m ungerade ist. Sei ferner $0 \leq x < n$, $x_0 \equiv x^m \bmod n$, $x_i \equiv x_{i-1}^2 \bmod n$, $1 \leq i \leq l$, $x_l \equiv x^{n-1} \bmod n$.

Satz 9.5.3 *Ist $4 < n$ zusammengesetzt und ungerade, so gilt für mindestens $\frac{3}{4}(n-1)$ der $1 \leq x < n$ entweder $x_l \neq 1$ oder für ein $1 \leq i < l$ gilt $x_i = 1$ und $x_{i-1} \neq n-1$.*

Beweis: Satz 5.3. besagt, daß, wenn n zusammangesetzt ist, für mindestens $\frac{3}{4}(n-1)$ der $1 \leq x < n$ $W_n(x)$ gilt. Angenommen $W_n(x)$ ist erfüllt. Ist jetzt $x_l \neq 1$, ist alles klar. Sei also $x_l = 1$. Außerdem ist für mindestens ein $0 \leq j < l$: $1 < (x_j - 1, n) < n$.

Wir wählen ein $i-1, j \leq i-1 < l$, wobei $i-1$ der letzte Index ist, für den $x_{i-1} \neq 0$, also ist $x_i = 0$. Aus obiger ggT-Beziehung folgt, daß $x_j - 1$ einen gemeinsamen Teiler p mit n besitzt, oder anders ausgedrückt

$$x_j \equiv 1 \bmod p$$

.

Aus

$$x_{i-1} \equiv x_j^{2^{i-j-1}} \bmod n$$

folgt auch

$$x_{i-1} \equiv 1 \bmod p$$

.

Wäre jetzt $x_{i-1} = n - 1$ gilt auch $x_{i-1} - 1 = n - 2$, was durch p teilbar ist. Es würde also gelten, daß

$$p|n \wedge p|(n-2) \Rightarrow p = 2$$

Dies ist ein Widerspruch zur Tatsache, daß n ungerade ist! Also muß gelten

$$x_{i-1} \neq n - 1$$

9.5.3 Algorithmus (nach D.Knuth)

Sei $n - 1 = 2^k q$, q ungerade.

1. **Wähle Basis:** Für ein gegebenes n wähle zufällig ein x mit $1 < x < n$.
2. **Exponentiation:** Setze $j \leftarrow 0$ und $y \leftarrow x^q \bmod n$.
3. **Fertig?** (Jetzt ist $y = x^{2^j q} \bmod n$). Wenn $j = 0$ und $y = 1$, oder wenn $y = n - 1$, terminiere mit dem Ergebnis "n ist wahrscheinlich prim". Ist $j > 0$ und $y = 1$ gehe nach 5.
4. **Erhöhe j:** Erhöhe j um 1. Ist $j < k$, setze $y \leftarrow y^2 \bmod n$ und gehe zu 3 zurück.
5. **Keine Primzahl:** Terminiere Algorithmus mit dem Ergebnis n ist sicher nicht prim.

Führen wir also obigen Algorithmus mehrmals nacheinander aus, wobei wir x jedesmal zufällig auswählen und liefert der Algorithmus einmal das Ergebnis, daß n nicht prim ist, so wissen wir sicher, daß n zusammengesetzt ist. Terminiert er 25 mal hintereinander mit "n ist wahrscheinlich prim", ist die Wahrscheinlichkeit eines Irrtums (n ist doch keine Primzahl), kleiner als $(\frac{1}{4})^2 5$. Selbst wenn wir eine Billion Primzahlen mit einem solchen Test verifiziert haben, ist die Wahrscheinlichkeit eines Fehlers immer noch kleiner als $\frac{1}{1000000}$.

9.5.4 Beispiele

1. $n = 25, n - 1 = 3 \cdot 8, b = 7$:

$$\begin{aligned} 7^3 &\equiv 18 \bmod 25 \\ 7^6 &\equiv 24 \bmod 25 \\ 7^{12} &\equiv 1 \bmod 25 \\ 7^{24} &\equiv 1 \bmod 25 \end{aligned}$$

Test bringt keine Information! $b = 7$ ist kein MR_2-Zeuge.

2. $n = 25, n - 1 = 3 \cdot 8, b = 2$:

$$\begin{aligned} 2^3 &\equiv 8 \bmod 25 \\ 2^6 &\equiv 14 \bmod 25 \\ 2^{12} &\equiv 21 \bmod 25 \\ 2^{24} &\equiv 16 \bmod 25 \end{aligned}$$

   Test zeigt, daß 25 nicht prim ist! $b = 2$ ist MR_2-Zeuge.

3. $n = 2047 = 23 \cdot 89, n - 1 = 2 \cdot 1023, b = 2$:

$$\begin{aligned} 2^{1023} &\equiv 1 \bmod 2047 \\ 2^{2046} &\equiv 1 \bmod 2047 \end{aligned}$$

   Test bringt keine Information! $b = 2$ ist kein MR_2-Zeuge.

4. $n = 2047 = 23 \cdot 89, n - 1 = 2 \cdot 1023, b = 3$:

$$\begin{aligned} 3^{1023} &\equiv 1565 \bmod 2047 \\ 3^{2046} &\equiv 1013 \bmod 2047 \end{aligned}$$

   Test zeigt, daß 2047 nicht prim ist! $b = 3$ ist MR_2-Zeuge.

5. $n = 341 = 11 \cdot 31, n - 1 = 4 \cdot 85, b = 2$

$$\begin{aligned} 2^{85} &\equiv 32 \bmod 341 \\ 2^{170} &\equiv 1 \bmod 341 \\ 2^{340} &\equiv 1 \bmod 341 \end{aligned}$$

   Test zeigt, daß 341 nicht prim ist! $b = 2$ ist MR_2-Zeuge.

6. $n = 561 = 3 \cdot 11 \cdot 17, n - 1 = 16 \cdot 35, b = 2$

$$\begin{aligned} 2^{35} &\equiv 263 \bmod 561 \\ 2^{70} &\equiv 166 \bmod 561 \\ 2^{140} &\equiv 67 \bmod 561 \\ 2^{280} &\equiv 1 \bmod 561 \\ 2^{560} &\equiv 1 \bmod 561 \end{aligned}$$

   Test zeigt, daß 561 nicht prim ist! $b = 2$ ist MR_2-Zeuge. (Carmichael-Zahl).

7. $n = 2243, n - 1 = 2 \cdot 1121, b = 2$

$$\begin{aligned} 2^{1121} &\equiv 2242 \bmod 2243 \\ 2^{2242} &\equiv 1 \bmod 2243 \end{aligned}$$

   $b = 2$ ist kein MR_2-Zeuge für die Zusammengesetztheit von 2243.

(Beispiele aus [7]).

9.5.5 Brauchbarkeit des Miller-Rabin-Tests nach D.Knuth

Satz 9.5.4 *Der* Miller-Rabin-*Tests ist ein brauchbarer, effizienter Pseudo-Primtest.*

Beweis:

- Aus $(b,n) \in MR_2 \Rightarrow n$ ist zusammengesetzt. Dies wurde bereits vorher gezeigt.
- Der Test ist effizient, da jedes b^m mod n mit $\mathcal{O}(\log_2 m)$ Operationen (Multiplikationen von $\log n$-bit-Zahlen) berechnet werden kann (schnelle Exponentiation); die Anzahl solcher mod-Operationen ist $\mathcal{O}(\log_2 n)$.
- Nach Satz 4.2 wächst die Anzahl der der Zeugen für die Zusammengesetztheit von n proportional mit n, und zwar um den Faktor $\frac{3}{4}$.

9.6 Der schnelle Monte-Carlo-Primzahltest von Solovay† und Strassen

9.6.1 Algorithmus

Sein wieder n eine ungerade narürliche Zahl.

- Wähle zufällig eine Zahl a, mit $1 \leq a \leq n-1$.
- Sind a und n relativ prim zueinander, so berechne $\epsilon \equiv a^{(n-1)/2} \bmod n$, wobei $-1 \leq \epsilon < n-2$.
- Berechne das Jacobi-Symbol $\delta = (\frac{a}{n})$ mit der Reziprokregel.
-
    - Ist nun $\epsilon = \delta$, so terminiere mit dem Ergebnis "n ist Primzahl".
    - Ist $\epsilon \neq \delta$, so terminiere mit dem Ergebnis "n ist zusammengesetzt".

Wieder gilt offensichtlich daß unser Test immer korrekt verläuft, falls n eine Primzahl ist. Weiter unten werden wir zeigen, daß die Wahrscheinlichkeit eines Testirrtums für zusammengesetzte n wieder begrenzt ist, nämlich beim Monte-Carlo-Primzahltest durch $\frac{1}{2}$.

9.6.2 Kosten des Algorithmus

Wir benötigen folgenden Aufwand für die einzelnen Schritte:

- ggT von (a, n) mit Euklid'schem Algorithmus:

  $1.5 log_2 n$ Divisionen mit Rest von zwei Zahlen $\leq n^2$ [4].

- Berechnung von ϵ:

  $1.25 log_2 n$ Multiplikationen, gefolgt von einer Reduktion $\bmod n$, mit $2.5 log_2 n$ Divisionen mit Rest von zwei Zahlen $\leq n^2$.

- Berechnung von δ mit der Kehrwertregel des Jacobi-Symbols, in etwa so aufwendig wie ggT-Berechnung:

  $1.5 log_2 n$ Divisionen mit Rest von zwei Zahlen $\leq n^2$ [4].

Insgesamt kommt der Algorithmus mit $6 log_2 n$ Operationen aus.

9.6.3 Fehlerwahrscheinlichkeit

Satz 9.6.1 *Die Wahrscheinlichkeit, daß der MC-Test eine zusammengesetzte Zahl n als Primzahl bezeichnet ist $\leq \frac{1}{2}$.*

Beweis: Sei n zusammengesetzt.

- Die Menge $G_n = \{a + (n) | a \in \mathbf{Z}, (a, n) = 1, a^{(n-1)/2} \equiv (\frac{a}{n}) \bmod n\}$ ist offensichtlich eine Untergruppe der E_n.

- Es genügt also zu zeigen, daß $G_n \neq E_n$ ist, da aus dem Satz von LAGRANGE folgt: $\#G_n \leq \#E_n \leq (n-1)/2 \Rightarrow$ höchstens die Hälfte der Zahlen zwischen 1 und n führen zu einem falschen Ergebnis

- Beweis durch Widerspruch: Annahme:
  (1) $$a^{(n-1)/2} \equiv (\tfrac{a}{n}) \bmod n$$
  für alle $a \in \mathbf{Z}$ mit $(a, n) = 1$.

- Wäre $n = p^e$ (p Primzahl), folgt aus (1)

  - $$a^{(p-1)/2} \equiv 1 \bmod p^e \Rightarrow a^{(p-1)} \equiv 1 \bmod p^e$$

    für alle a, mit $p \not| a$ (Teilerfremd zu n)

  - Da E_{p^e} zyklisch ist und die Ordnung $p^{e-1}(p-1)$ besitzt, gilt

    $$p^{e-1}(p-1) | p^e - 1$$

    $\Rightarrow e \leq 1$, was im Widerspruch zur Annahme steht

  $\Rightarrow n$ besitzt nichttriviale Zerlegung $n = r \cdot s$, mit $(r, s) = 1$

- Aus (1) folgt
  (2) $$a^{(n-1)/2} y \equiv \pm 1 \bmod n$$
  (0 nur für $a = 0$), für alle $a \in \mathbf{Z}$ mit $(a, n) = 1$.

- Wäre $a^{(n-1)/2} \equiv -1 \bmod n$:
  - Da $(r,s) = 1$ können wir nach dem CRT ein b finden, mit
    $$b \equiv 1 \bmod r, \quad b \equiv a \bmod s$$
  - $\Rightarrow$ b ist teilerfremd zu r, s, damit auch zu n $\Rightarrow$
  - $$b^{(n-1)/2} \equiv 1 \bmod r, \quad b^{(n-1)/2} \equiv -1 \bmod s$$

    was im Widerspruch zu (2) steht

  $\Rightarrow$

  $$a^{(n-1)/2} \equiv 1 \bmod n \tag{3}$$

  für alle $a \in \mathbf{Z}$ mit $(a,n) = 1$.

- Aus (3) folgt, $(\frac{a}{n}) = 1$, für alle $a \in \mathbf{Z}$ mit $(a,n) = 1$, was unmöglich ist, gdw. n quadratfrei.

- Sei n nicht quadratfrei, z.B. $n = p^e q$, q ungerade Primzahl, $e > 1$ und $(p,q) = 1$.
  - Aus (1) folgt $a^{(n-1)} \equiv 1 \bmod n$, für alle $a \in \mathbf{Z}$ mit $(a,n) = 1$.
  - Aus CRT folgt $a^{(n-1)} \equiv 1 \bmod p^e$, für alle $a \in \mathbf{Z}$ mit $(a,n) = 1$.
  - E_{p^e} zyklisch, mit Ordnung $p^{e-1}(p-1) \Rightarrow$ mit $p^{e-1}(p-1)|(n-1)$
  - Da $e > 1 \Rightarrow p|(n-1)$,$p|n$ ergibt sich ein Widerspruch $\Rightarrow$ n quadratfrei

  $\Rightarrow$ Widerspruch zu (3) $\Rightarrow$ Satz ist korrekt.

Literaturverzeichnis

[1] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. Journal of Number Theory 12, 128-138(1980).

[2] R.Solovay und V.Strassen. A fast Monte-Carlo test for primality. Siam J.Comput Vol.6, No.1, März 1977.

[3] R.Solovay und V.Strassen. Erratum: A fast Monte-Carlo test for primality. Siam J.Comput Vol.7, No.1, Februar 1978

[4] D.E.Knuth. The Art of computer programming, Vol.2, Second Edition. Addison-Wesley Co., 1981.

[5] K.H.Rosen. Elementary number theory and its applications. Second Edition. Addison-Wesley Co., 1988.

[6] D.Bressoud. Factorization and primality testing. Addison-Wesley Co., 1989.

[7] Strehl. Script bzgl. Vorlesung “Einführung in die Theoretische Informatik III”. 1994

Vortrag 10

Routing mit randomisierten Algorithmen

Alexander Hall
Technische Universität München
hall@informatik.tu-muenchen.de

10.1 Allgemeines zu Parallelrechnern

In der Literatur sind verschiedene Modelle von Parallelrechnern geläufig. Im Zusammenhang mit effizienten synchronen Multiprozessor Algorithmen wird oft ein eher idealistisches Modell hergenommen. N Prozessoren können dabei fast beliebig parallel auf einen gemeinsamen Speicher zugreifen. Durch Zugriffsbeschränkungen treten während der Laufzeit keine Speicherkonflikte auf. Dieses Modell ist technisch so nicht umsetzbar.

Ein realistischer Ansatz ist, daß N Prozessoren über Kanten verbunden sind, wobei maximal d Kanten aus jedem Prozessor hinaus- bzw. hineingeführt sind. Dementsprechend spricht man von einem Eingangs- bzw. Ausgangsgrad d. Jeder Prozessor ist quasi wie ein kleiner Computer, der eigenen Speicher besitzt. Im folgenden werden zwei „realistische Modelle“ vorgestellt.

10.2 Problemstellung

In einem Netzwerk von N Prozessoren sollen Nachrichten verschickt werden. Bei jedem Prozessor befinden sich zum Startzeitpunkt mehrere Pakete, die aus Daten und einer Zieladresse (Zielprozessor) bestehen. Dabei bekommt jedes Paket ein Ticket, das die Routing-Informationen enthält. Das Ticket bestimmt die Route, die das Paket durch das Netzwerk nimmt.

Pro Kante (zwischen zwei Prozessoren) und Zeiteinheit kann jeweils nur ein Paket übertragen werden.

Nach dem Durchlauf eines solchen Systems sollen sich alle Pakete an ihrem Ziel befinden. Gesucht wird ein Algorithmus, mit dem ein solcher Durchlauf möglichst schnell beendet ist, ideal wären $\mathcal{O}(\log N)$ Zeiteinheiten. Dazu dürfen sich zum Startzeitpunkt natürlich nicht beliebig viele Pakete bei jedem Prozessor befinden. Sind h Pakete bei einem Prozessor, der den Ausgangsgrad d hat, so dauert es $\frac{h}{d}$ Zeiteinheiten, bis die Pakete den Prozessor überhaupt verlassen haben. D. h. bei keinem Prozessor dürfen anfangs mehr als $d \cdot \log N$ Pakete sein.

Durch randomisierte Algorithmen kann man hier im Vergleich zu anderen Ansätzen einen erheblichen Geschwindigkeitsgewinn erzielen.

10.3 Parallele Kommunikations Struktur

10.3.1 Definition

Die Parallele Kommunikations Struktur (PKS), die zur Beschreibung der im folgenden vorgestellten Netzwerkarten dient, besteht im wesentlichen aus einem gerichteten Graphen $G = (V, E)$. $V = \{0, 1, \ldots, N-1\}$ ist die Menge der N Knoten, E ist die Menge der Kanten. Alle Knoten haben den Ein- und Ausgangsgrad d $\Rightarrow |E| = dN$.

Zu jedem Zeitpunkt im Durchlauf sind genau T Pakete im System. Zum Startzeitpunkt enthalten die Tickets nur die Zieladresse des zugehörigen Pakets.

Pro Knoten gibt es d Warteschlangen, für jede Kante eine. Die einzige Voraussetzung, die die Schlangen erfüllen müssen, ist, daß sie ein „Kopfelement“ haben müssen, falls sie nicht leer sind. Dieses wird dann im nächsten Zeitintervall an der korrespondierenden Kante übertragen.

10.3.2 Eigenschaften einer PKS

- nicht adaptiv: Veränderungen im Ticket eines Pakets hängen nicht vom Vorhandensein, Inhalt oder Ticket eines anderen Pakets ab.

- symmetrisch: Der Verkehr ist an allen Kanten gleich. Der Verkehr (oder Traff) von einer Kante ist die erwartete Anzahl Pakete, die bei einem Durchlauf diese Kante passieren.

- nicht wiederholend: Zwei Pakete nehmen die Routen $e_1 e_2 \cdots e_r$ und $\overline{e}_1 \overline{e}_2 \cdots \overline{e}_s$ (e_n und $\overline{e}_n$ sind Kanten). Falls $e_i = \overline{e}_j$ und $e_k = \overline{e}_m$ $(k > i)$ gilt, ist $k - i = m - j$ und $e_p = \overline{e}_{p+j-i}$ für $i \leq p \leq k$. D.h. falls zwei Routen sich zwei verschiedene Kanten teilen, so teilen sie sich auch alle Kanten die dazwischen liegen.

10.4 Obere Grenze für m Treffer

Im folgenden sollen Schranken für die Wahrscheinlichkeit, daß ein Algorithmus mehr als eine gewisse Zeit benötigt, angegeben werden. Dazu muß man die Wahrscheinlichkeit von m Teffern bei N unabhängigen Poisson Experimenten nach oben abschätzen können.

10.4.1 Hoeffdings Theorem

Zur Erinnerung: Ein Poisson Experiment ist ein Zufallsexperiment, das nur zwei mögliche Ereignisse hat, die im Allgemeinen als Treffer bzw. Nicht-Treffer bezeichnet werden.

Eine obere Grenze für die Wahrscheinlichkeit von m Treffer bei N unabhängigen Poisson Experimenten, deren Trefferwahrscheinlichkeiten $p_1, \ldots, p_N$ sind, ist gegeben durch die Binomialverteilung:

$$\mathrm{B}(m, N, p)$$

Dabei ist $\sum p_i = Np$; es muß gelten: $m \geq Np + 1$

10.4.2 Chernoff

Falls $m \geq Np$ und $m \neq N$, so gilt nach Chernoff:

$$\mathrm{B}(m, N, p) \leq \left(\frac{Np}{m}\right)^m \cdot \left(\frac{N - Np}{N - m}\right)^{N-m}$$

mit $(1 + \frac{1}{x})^x < \mathrm{e}$ für $x > 0$ folgt:

$$\left(\frac{N - Np}{N - m}\right)^{N-m} = \left(1 + \frac{m - Np}{N - m}\right)^{\frac{N-m}{m-Np} \cdot (m-Np)} \leq \mathrm{e}^{m-Np}$$

$$\Rightarrow \underline{\mathrm{B}(m, N, p) \leq \left(\frac{Np}{m}\right)^m \cdot \mathrm{e}^{m-Np}}$$

10.5 n Dimensionaler Boolescher Würfel / Hypercube

10.5.1 Graph

Beim n Dimensionalen Booleschen Würfel handelt es sich um einen gerichteten Graphen $G = (V, E)$, mit $N = 2^n$ Knoten ($V = \{0, 1, \ldots, N - 1\}$). Die Kanten sind immer zwischen zwei Knoten, deren Binärrepräsentation sich nur in einer Stelle unterscheiden. D. h. $E = \{(x, x//i) \mid x \in V \;,\; i \in \{1, \ldots, n\}\}$, dabei ist $x//i$ die Zahl, deren Binärrepräsentation bis auf die i. Stelle gleich der von x ist.

Die Binärrepresentation einer Zahl a wird folgenderweise notiert: $a^n a^{n-1} \cdots a^1$, dabei ist a^1 die erste Stelle.

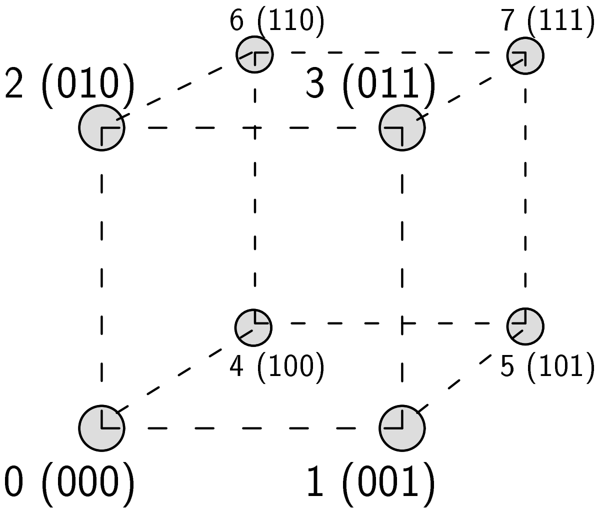


Abbildung 10.1: Beispiel für $n = 3$

In Abbildung 10.1 ist als Beispiel das Bild eines dreidimensionalen Hypercubes zu sehen. Die eingezeichneten Verbindungen sind jeweils Kanten in zwei Richtungen.

10.5.2 Routing Algorithmus

Der Routing Algorithmus ist in zwei Phasen aufgeteilt:

- Phase A: Jedes Paket an einen zufällig gewählten Knoten schicken.
- Phase B: Die zufällig verteilten Pakete an ihre tatsächlichen Ziele schicken.

Die Randomisierung in Phase A garantiert, daß ein Durchlauf, unabhängig von der Verteilung der T Pakete zum Startzeitpunkt, immer mit der gleichen Wahrscheinlichkeit in einer bestimmten Zeit terminiert.

```
begin
  for i:=1 step 1 until Tx[i+1]=0 or i=n do
    ``Sende X vom aktuellen Knoten y
    zum Knoten y//Tx[i]''
end.
```

Abbildung 10.2: Routing Algorithmus bei gegebenem Ticket

Das Ticket `Tx` eines Pakets ist ein n-stelliger Vektor. In Abbildung 10.2 steht der Algorithmus zum Verschicken eines Pakets anhand dieses Tickets. Dieser wird in beiden Phasen nach der Erstellung des Tickets durchlaufen.

Erstellung des Tickets in Phase A

Der in Abbildung 10.3 gezeigte Algorithmus belegt die i. Stelle des Tickets zufällig entweder mit i oder mit 0. Dazu wird die Funktion `Pick a` $\in C$ verwendet, die zufällig ein Element aus der Menge C auswählt (bei gleichmäßiger Wahrscheinlichkeitsverteilung).

Anschließend wird mit `Pack Tx` noch erreicht, daß die r Elemente von `Tx`, die ungleich Null sind, in der gleichen Reihenfolge an die Stellen `Tx[1],...,Tx[r]` geschrieben werden. Die restlichen Stellen werden mit Null belegt.

```
begin
  for i:=1 step 1 until n do
  begin
    Tx[i]:=i;
    Pick a ∈ {0,1};
    if a=0 then Tx[i]:=0;
  end;
  Pack Tx;
end.
```

Abbildung 10.3: Erstellung des Tickets in Phase A

Die Dimensionen werden also beim Routen anhand eines solchen Tickets in aufsteigender Reihenfolge abgelaufen. D. h. wenn z. B. schon die Position des Zielknotens in der zweiten Dimension (bzw. das zweite Bit des Zielknotens) feststeht, so steht auch schon die Position in der ersten Dimension fest (bzw. das erste Bit).

Erstellung des Tickets in Phase B

Das Paket befindet sich am Knoten u und muß zum Knoten v. Es werden die Bits von u und v verglichen, alle die nicht übereinstimmen werden in `Tx` eingetragen (Abbildung 10.4).

Die Dimensionen werden wieder in aufsteigender Reihenfolge abgelaufen.

```
begin
  for i:=1 step 1 until n do
  begin
    if $u^i = v^i$ then Tx[i]:=0 else Tx[i]:=i;
  end;
  Pack Tx;
end.
```

Abbildung 10.4: Erstellung des Tickets in Phase B

10.5.3 Eigenschaften des Booleschen n-Würfel

Zur Ermittlung der benötigten Zeiteinheiten des Algorithmus wird im folgenden ein allgemeines Theorem bewiesen, das voraussetzt, daß eine PKS die in Abschnitt 10.3.2 vorgestellten Eigenschaften besitzt.

Deswegen wird nun nachgewiesen, daß der Hypercube-Routing Algorithmus diese Eigenschaften hat.

nicht adaptiv

In Phase A hängt die Route nur von zufälligen Entscheidungen ab, in Phase B nur vom aktuellen Standpunkt relativ zum Ziel.

symmetrisch

Voraussetzung: Zum Startzeitpunkt müssen an jedem Knoten h Pakete sein. Unabhängig davon müssen h Pakete dasselbe Ziel haben.

Phase A: Sei (u, v) eine beliebige Kante mit $u^i \neq v^i$. Das Paket p startet am Knoten x, falls es durch u geht, gilt:

$$x^j = u^j \qquad (i \leq j \leq n)$$

Das Paket p passiert u mit einer Wahrscheinlichkeit $\frac{1}{2}^{i-1}$. Da es bei den Dimensionen $1, \ldots, i-1$ jeweils mit einer Wahrscheinlichkeit von $\frac{1}{2}$ in die richtige Richtung geht. Ist p am Knoten u, so geht es mit einer Wahrscheinlichkeit von $\frac{1}{2}$ entlang (u, v) und es gibt 2^{i-1} mögliche Startknoten x.

⇒ Der Erwartungswert für die Anzahl der Pakete, die die Kante (u, v) während eines Durchlaufs passieren, ist:

$$h^{(1)} \cdot 2^{i-1(2)} \cdot \left(\frac{1}{2}\right)^{i-1} {}^{(3)} \cdot \frac{1}{2}^{(4)} = \underline{\frac{1}{2}h}$$

Das Ergebnis ist unabhängig von der Kante (u, v) und gilt also $\forall$ Kanten. Der Verkehr ist somit an allen Kanten gleich.

Phase B: Analog.

nicht wiederholend

Da in beiden Phasen die Dimensionen in aufsteigender Reihenfolge abgelaufen werden, gilt:

Falls sich zwei Routen eine Kante teilen und dann z. B. in der i. Dimension auseinander gehen, so unterscheiden sich alle folgenden Knoten der beiden Routen auf jeden Fall in der i. Dimension.

⇒ Zwei Routen, die sich zwei verschiedene Kanten teilen, müssen sich also auch alle dazwischenliegenden Kanten teilen. Keine der Routen kann dazwischen „abbiegen“ und einen anderen Weg nehmen.

10.6 Allgemeines Theorem

10.6.1 Voraussetzungen

Es wird, wie bereits erwähnt, eine PKS betrachtet, die nicht adaptiv, nicht wiederholend und symmetrisch ist. Sie hat N Knoten und ist vom Grad d, d. h. sie hat dN Kanten. Die Menge der Kanten wird mit $E = \{e_1, \ldots, e_{dN}\}$ bezeichnet. Zum Startzeitpunkt sollen sich h Pakete an jedem Knoten befinden (= volle h-Relation). Insgesamt befinden sich also $T = hN$ Pakete im System.

s bezeichne die maximale Länge einer Route und l den Erwartungswert der Länge einer Route.

10.6.2 Wartezeit eines Pakets

Es wird eine feste Route $R = e_1 e_2 \ldots e_r$ betrachtet. P_x ist die Wahrscheinlichkeit, daß ein Paket x mindestens eine Kante in R berührt. P_{x_i} ist die Wahrscheinlichkeit, daß x die Kante e_i berührt.

Dann gilt:

---

[1] Anzahl der Pakete, die anfangs an jedem Knoten waren

[2] Mögliche Startknoten

[3] Wahrscheinlichkeit, daß p zu u gelangt

[4] Wahrscheinlichkeit, daß p die Kante u entlang geht

$$P_x \leq \sum_{i=1}^{r} P_{x_i} \tag{10.1}$$

Der Verkehr entlang einer Kante e_i wird definiert als der Erwartungswert der Anzahl der Pakete x, die die Kante berühren: $\mathrm{Traff}(e_i) = \sum\limits_x P_{x_i}$ Da die PKS symmetrisch ist, ist der Verkehr an allen Kanten gleich:

$$\mathrm{Traff}(e_i) = \mathrm{Traff}(e_j)$$

$$\Rightarrow \sum_x P_{x_i} = \sum_x P_{x_j} \qquad \forall\, i, j \quad (1 \leq i, j \leq Nd) \tag{10.2}$$

Wenn man (10.1) über alle x aufsummiert und (10.2) für $j = 1$ einsetzt, ergibt sich:

$$\underline{\sum_x P_x} \leq \sum_x \sum_{i=1}^{r} P_{x_i} = \sum_{i=1}^{r} \sum_x P_{x_i} = \underline{r \cdot \sum_x P_{x_1}} \tag{10.3}$$

(10.4) und (10.5) geben jeweils Gleichungen an, für den Gesamtverkehr an allen Kanten. In (10.4) wird der Erwartungswert der Länge einer Route mit der Anzahl der Pakete multipliziert. In (10.5) wird der Verkehr an allen Kanten aufsummiert.

$$Tl = hNl \tag{10.4}$$

$$\sum_{i=1}^{Nd} \mathrm{Traff}(e_i) = Nd \cdot \mathrm{Traff}(e_1) = Nd \sum_x P_{x_1} \tag{10.5}$$

(10.4) und (10.5) gleichsetzen:

$$\sum_x P_{x_1} = \frac{hl}{d}$$

in (10.3) einsetzen:

$$\sum_x P_x \leq r \cdot \sum_x P_{x_1} = \frac{rhl}{d}$$

Die Wahrscheinlichkeiten P_x sind wegen der Nicht-Adaptivität von einander unabhängig (da kein Ticket vom anderen abhängt).

Es handelt sich also um T unabhängige Poisson Experimente, deren Trefferwahrscheinlichkeiten sich zu höchstens $\frac{rhl}{d}$ aufaddieren.

Man kann eine obere Grenze für die Wahrscheinlichkeit von mindestens m erfolgreichen Experimenten angeben (siehe Abschnitt 10.4):

$\left(\mathrm{B}(m, T, p) \quad ; \quad Tp = \frac{rhl}{d}\right)$

$$\mathrm{B}\left(m, T, \frac{rhl}{dT}\right) = \mathrm{B}\left(m, hN, \frac{rl}{dN}\right)$$

obere Grenze für $\mathrm{B}\left(m, hN, \frac{rl}{dN}\right)$, falls $m > hN \cdot \frac{rl}{dN} + 1 = \frac{hrl}{d} + 1$:

$$\mathrm{B}\left(m, T, \frac{rl}{dN}\right) \leq \left(\frac{rhl}{dm}\right)^m \cdot \mathrm{e}^m = \left(\frac{\mathrm{e}rhl}{dm}\right)^m \tag{10.6}$$

Für ein bestimmtes Paket y mit Route R gibt (10.6) die obere Grenze an, daß mindestens m andere Pakete den Pfad von y berühren ⇒ (10.6) ist auch eine obere Grenze für die Wahrscheinlichkeit, daß y mindestens m Zeiteinheiten in Schlangen warten muß, bei jeder „Berührung" mit einem anderen Paket könnte y ja verzögert werden. Insgesamt sind in dem System T Pakete vorhanden ⇒ die obere Grenze für die Wahrscheinlichkeit, daß mindestens eines der Pakete mindestens m Zeiteinheiten warten muß, ist T mal so groß wie (10.6), vorausgesetzt man setzt für die Pfadlänge r die maximale Länge einer Route s ein:

$$P_\Delta = \left(\frac{\mathrm{e}shl}{md}\right)^m \cdot T$$

Auch bei einer partiellen h-Relation (zum Startzeitpunkt sind an jedem Knoten zwischen 0 und h Pakete) ist P_Δ eine obere Grenze, es sind ja im Vergleich zur vollen h-Relation weniger Pakete im System.

10.7 Allgemeines Theorem auf den Hypercube übertragen

Die maximale Länge einer Route beim vorgestellten Hypercube Algorithmus ist der Grad $d = n$ des Würfels (⇒ $s = n$). Der Erwartungswert der Länge eines Pfades l ist $\frac{n}{2}$. Dies gilt für beide Phasen.

Der Hypercube hat $N = 2^n$ Knoten ⇒ bei einer vollen h-Relation sind $T = hN = h2^n$ Pakete im System.

P_Δ ist eine obere Grenze für die Wahrscheinlichkeit, daß eine Phase nicht in der Zeit $t = s + m$ (s ist die Zeit, die ein „unverzögertes" Paket maximal unterwegs sein kann) beendet ist. Mit $m = Khn$, wobei $K = const.$, ergibt sich:

$$P_\Delta = \left(\frac{\mathrm{e}nh\frac{n}{2}}{Khnn}\right)^{Khn} \cdot h2^n = \left(\frac{\mathrm{e}}{2K}\right)^{Khn} \cdot h2^n$$

für $K > 6$:

$$\underline{P_\Delta \leq} \left(\frac{\mathrm{e}}{12}\right)^{Khn} h2^n \leq \left(\frac{1}{4}\right)^{Khn} h2^n = h\left(\frac{2^{\frac{1}{Kh}}}{4}\right)^{Khn} \leq h\left(\frac{1}{2^n}\right)^{Kh} = \underline{hN^{-Kh}}$$

⇒ die Beendigung innerhalb der Zeit

$$t = s + m = n + Khn = (1+Kh)n = (1+Kh)\log_2(N) = \underline{\mathcal{O}(\log N)}$$

ist bei großem K sehr wahrscheinlich.

10.8 2 Dimensionales Gitter

Bei hohen Taktfrequenzen kann es durchaus verlangt sein, daß die Leitungen zwischen allen verbundenen Prozessoren gleich lang sind, da nur so gewährleistet werden kann, daß Pakete an allen Kanten gleich schnell übertragen werden können. Beim n Dimensionalen Booleschen Würfel ist dies bei größerer Anzahl Prozessoren schwierig, bzw. unmöglich. Deswegen wird noch ein einfacheres Modell vorgestellt: Das n Dimensionale Gitter, zur Erklärung wird der Spezialfall $n = 2$ betrachtet.

10.8.1 Graph

Der Graph des 2 Dimensionalen Gitters besteht aus der Knotenmenge $V = \{[i,j] \mid 0 \leq i < n \ , \ 0 \leq j < n\}$ und der Kantenmenge $E = \{([i,j],[l,m]) \mid (i = l \ \wedge \ j = m+1) \ \vee \ (j = m \ \wedge \ i = l+1)\}$. Alle Kanten gehen dabei in beide Richtungen. Es gibt $N = |V| = n^2$ Knoten. Abbildung 10.5 zeigt ein solches Gitter.

10.8.2 Routing Algorithmus

Der Routing Algorithmus ist in drei Phasen aufgeteilt. Ähnlich wie beim Hypercube dient die erste Phase zur Randomisierung:

- Phase A: Jedes Paket in vertikaler Richtung zufällig verschicken.
- Phase Z: Die Pakete in die richtige Spalte schicken (horizontale Korrektur).
- Phase B: Die Pakete in die richtige Zeile schicken (vertikale Korrektur).

In Phase A wird ein Paket am Knoten $[i,j]$ entlang der Spalte i zum Knoten $[i,k]$ geschickt, wobei k mit `Pick k` $\in$ `{0,...,n-1}` zufällig gewählt wird. Hat das Paket das Ziel $[l,m]$, so wird es in Phase Z vom Knoten $[i,k]$ entlang der k. Zeile zum Knoten $[l,k]$ geschickt. In Phase B wird das Paket entlang der Spalte l zu $[l,m]$ geschickt. In Abbildung 10.5 wird dies veranschaulicht.

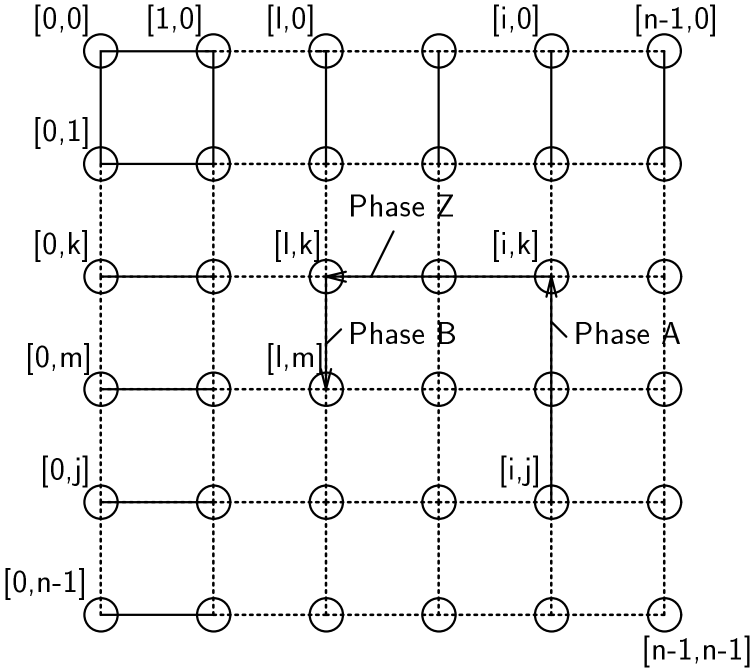


Abbildung 10.5: 2 Dimensionales Gitter

10.8.3 Effizienz des Routing Algorithmus

Voraussetzungen

Zum Startzeitpunkt befinde sich an jedem Knoten ein Paket. Die Warteschlangen sollen Paketen die sich bereits bewegt haben den Vorrang geben ⇒ ein Paket wird nicht mehr angehalten, sobald es sich bewegt (†).

Phase A

Da sich an jedem Knoten zum Startzeitpunkt nur ein Paket befindet, ergibt sich bei der Phase A ein kontinuierlicher Fluß der Pakete, keines wird durch ein anderes blockiert.

Phase Z

Sei x ein Paket, daß sich am Anfang von Phase Z am Knoten $[k-1, s]$ befindet und einem Knoten rechts davon als Ziel hat. x wartet wegen † höchstens soviele Zeiteinheiten, wie es Pakete an den Knoten $\left\{ \left[\bar{i}, s\right] \mid \bar{i} < k \right\}$ gibt. Dies gilt entsprechend auch für Pakete, die nach links verschickt werden sollen ⇒ da sich beide Laufrichtungen nicht gegenseitig beeinflußen, werden nur die Pakete die nach rechts müssen, betrachtet.

Vor Phase A befinden sich n Pakete in Spalte i $(0 \leq i < n)$. Nach der Phase A ist jedes Paket der i. Spalte mit der Wahrscheinlichkeit $\frac{1}{n}$ am Knoten $[i, s]$ $\Rightarrow$ die Wahrscheinlichkeit, daß am Anfang von Phase Z mindestens $k + g$ Pakete an den Knoten $\left\{ \left[\bar{i}, s\right] \mid \bar{i} < k \right\}$ sind, ist:

$$\mathrm{B}\left(k + g, n \cdot k, \frac{1}{n}\right) \tag{10.7}$$

Ein Paket am Knoten $[k - 1, s]$ hat noch höchstens $n - k$ Schritte zu gehen $\Rightarrow$ (10.7) gibt eine obere Grenze für die Wahrscheinlichkeit an, daß ein Paket in Phase Z mehr als $(n - k) + (k + g) = n + g$ Zeiteinheiten benötigt.

Phase B

Am Anfang von Phase B ist jedes Paket zufällig in seiner Zielspalte plaziert (in jeder Spalte sind n Pakete). Ein Paket in der Spalte l befindet sich also mit der Wahrscheinlichkeit $\frac{k}{n}$ über der k. Zeile $\Rightarrow$ die Wahrscheinlichkeit, daß an den Knoten $\left\{ \left[l, \bar{i}\right] \mid \bar{i} < k \right\}$ mindestens $k + g$ Pakete sind, ist:

$$\mathrm{B}\left(k + g, n, \frac{k}{n}\right) \tag{10.8}$$

Analog zu Phase Z gibt (10.8) eine obere Schranke für die Wahrscheinlichkeit an, daß ein Paket in Phase B mehr als $(n - k) + (k + g) = n + g$ Zeiteinheiten benötigt.

Phase Z und B

Nach Chernoff (siehe Abschnitt 10.4) lassen sich sowohl $\mathrm{B}\left(k + g, n \cdot k, \frac{1}{n}\right)$ als auch
$\mathrm{B}\left(k + g, n, \frac{k}{n}\right)$ mit

$$\left(\frac{k}{k+g}\right)^{k+g} \cdot \mathrm{e}^g$$

nach oben abschätzen.

Fallunterscheidung:

1. Fall: $g > 2k$

$$\underbrace{\left(\frac{k}{k+g}\right)}_{<1}^{k+g} \mathrm{e}^g \leq \left(\frac{k}{k+g}\right)^g \mathrm{e}^g \leq \left(\frac{k}{3k}\right)^g \mathrm{e}^g = \underline{\left(\frac{\mathrm{e}}{3}\right)^g}$$

2. Fall: $2k \geq g \geq k$

$$\left(\frac{k}{k+g}\right)^{k+g} \mathrm{e}^{g} \leq^{5} \left(\frac{k}{k+g}\right)^{\frac{g}{2}+g} \mathrm{e}^{g} \leq^{6} \left(\frac{k}{2k}\right)^{\frac{3}{2}g} \mathrm{e}^{g} = \underline{\frac{\mathrm{e}}{2\sqrt{2}}}$$

3. Fall: $k > g \geq 0$

$$\text{nach Chernoff}: \mathrm{B}\left(k+g, n\cdot k, \frac{1}{n}\right) \text{ und } \mathrm{B}\left(k+g, n, \frac{k}{n}\right) \leq \underline{\mathrm{e}^{-\frac{g^2}{2k}}}$$

Sei $g = Kn^{\frac{3}{4}}$ ($K = const.$). $\forall\ k \in \{0, \ldots, n-1\}$ ist $\overline{C}^{K\sqrt{n}}$ ($0 < \overline{C} < 1$, $\overline{C} = const.$) eine obere Schranke für die Wahrscheinlichkeit bei allen drei Fällen. Insgesamt sind immer n^2 Pakete im System $\Rightarrow$ die Wahrscheinlichkeit, daß mindestens ein Paket $n + g = n + Kn^{\frac{3}{4}}$ Zeiteinheiten in Phase Z bzw. B benötigt, ist höchstens $n^2\overline{C}^{K\sqrt{n}}$.

Insgesamt

$\exists\ C = const.$ mit $0 < C < 1$, so daß gilt:
Die Wahrscheinlichkeit, daß mindestens ein Paket nicht in der Zeit

$$\underbrace{n}_{\text{Phase A}} + \underbrace{n + Kn^{\frac{3}{4}}}_{\text{Phase Z}} + \underbrace{n + Kn^{\frac{3}{4}}}_{\text{Phase B}} = \underline{3n + 2Kn^{\frac{3}{4}}}$$

seinen Zielknoten erreicht hat, ist kleiner als: $C^{K\sqrt{n}}$

Bei genügend großem K terminiert der Algorithmus mit hoher Wahrscheinlichkeit nach $\mathcal{O}(n)$ Zeiteinheiten.

---

[5] $k = \frac{g}{2}$ (minimal)

[6] $g \geq k$

Literaturverzeichnis

[1] L. G. Valiant, G. J. Brebner: *Universal schemes for parallel communication*, STOC Milwaukee (1981).

[2] I. N. Bronstein, K. A. Semendjajew: *Taschenbuch der Mathematik*, B. G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig (1991).

Vortrag 11

Byzantinische Generäle

Claudia Ungerer
Technische Universität München
ungerer@informatik.tu-muenchen.de

Zusammenfassung

Kann in einem unzuverlässigen verteilten System Übereinkunft erreicht werden?
Dieses Problem wurde von L. Lamport, R. Shostak und M. Pease 1980 als das *Problem der Byzantinischen Generäle* bezeichnet. Im Folgenden wird im Rahmen der Ferienakademie 1995 dieses Problem aus der Sicht randomisierter Algorithmen beleuchtet.
Zunächst wird das Problem allgemein vorgestellt und die Motivation zu dessen Lösung begründet. Bevor ein randomisierter Algorithmus in zwei Varianten vorgestellt wird, werden zwei Methoden erläutert, die dieser Algorithmus benötigt:

- digitale Unterschrift
- Teilen eines Geheimnisses

Abschließend wird gezeigt, daß das Problem nicht zu lösen ist, wenn ein deterministischer Ablauf verwendet wird.

11.1 Einleitung

11.1.1 Problemstellung

Der Name *Byzantinische Generäle* rührt von folgender Vorstellung:
Auf Bergen rund um ein Tal stehen Generäle der byzantinischen Armee mit ihren Divisionen und wollen das im Tal liegende feindliche Dorf angreifen und einnehmen. Dies ist aber nur dann möglich, wenn alle Generäle zum gleichen Zeitpunkt angreifen, d.h. die Generäle müssen sich auf einen gemeinsamen Angriffszeitpunkt einigen. Zum Nachrichtenaustausch stehen ihnen aber lediglich Boten zur Verfügung. Die Schwierigkeit besteht darin, daß sich unter den Generälen Verräter befinden, die mit dem Feind paktieren und versuchen, eine Einigung der anderen (loyalen) Generäle zu verhindern, indem sie verfälschte Nachrichten verschicken. Das Problem ist nun, ein Entscheidungsverfahren zu finden, das garantiert, daß die loyalen Generäle trotz Verrätern in den eigenen Reihen stets zu einer gemeinsamen Entscheidung finden.

11.1.2 Motivation

Übertragen auf die Rechnerwelt stellt sich obiges Problem wie folgt dar:
Zuverlässige Rechensysteme müssen mit defekten Komponenten, die widersprüchliche Information an andere Teile des Systems weitergeben, umgehen können. Das Problem, eine Einigung zwischen verteilten Prozessoren zu erreichen, ist daher eines der fundamentalsten Probleme in der verteilten Programmierung, wie z.B.:

- verteilte Datenentwicklung
- verteilte Datenbanksysteme
- komplexe Steuerungssysteme in der Luft- und Raumfahrt

Im Falle der verteilten Datenbanksysteme müssen sich z.B. alle an einer Transaktion beteiligten Datenbankmanager darauf einigen, ob sie die Ergebnisse dieser Transaktion in die Datenbank übernehmen oder verwerfen. Alle Datenbankmanager müssen die gleiche Entscheidung treffen, um die Konsistenz der Datenbank zu bewahren (transaction commit problem). In einem absolut zuverlässigen System ist eine Übereinkunft freilich einfach. In realen Systemen aber können Fehler unterschiedlichster Art auftreten, wie z.B. Zusammenbruch von Prozessoren, Nachrichten gehen verloren, werden zerstört oder verdoppelt oder Prozessoren versuchen absichtlich zu stören. Trotz dieser Fehler soll eine Einigung erreicht werden. Je zahlreicher und schwerwiegender die Fehler, desto schwieriger natürlich die Einigung.

11.2 Randomisierte Byzantinische Generäle [1]

11.2.1 Einleitung

Der hier vorgestellte Algorithmus zur Lösung des Problems ist randomisiert im Sinne von M. O. Rabin, d.h. es werden Zufallszahlen verwendet, aber keine Annahmen über das zufällige Verhalten der fehlerhaften Prozessoren gemacht. Die Lösung kann sowohl im synchronen wie im asynchronen Fall angewandt werden. Sie beruht auf Authentisierung der Nachrichten mittels *digitaler Unterschrift*. Das Besondere an diesem Algorithmus ist die Verwendung zufällig gewählter Geheimnisse, die von den Prozessoren nach einer von Shamir eingeführten Methode geteilt werden. Für die Lösung werden die möglichen Fehlerarten in einem System auf den Zusammenbruch von Prozessoren beschränkt.
Bevor auf diese Besonderheiten genauer eingegangen wird, werden im folgenden die grundlegenden Konzepte behandelt.

11.2.2 Grundlegende Konzepte

Das System besteht aus n Prozessoren G_i (die Generäle), wobei $1 \leq i \leq n$. Jeder Prozessor G_i kann mit jedem anderen Prozessor G_j direkt Nachrichten $M \in \mathcal{M}$ austauschen (mit $\mathcal{M}$ Menge aller Nachrichten und $1 \leq j \leq n, i \neq j$). Durch diesen Nachrichtenaustausch sollen sich die Prozessoren auf eine gemeinsame Nachricht einigen. Jeder Prozessor G_i hat eine *Anfangsnachricht* M_i und führt ein *Übereinkunftsprotokoll* P_i aus. Solange sich ein Prozessor G_i an sein Protokoll P_i hält, wird er als *fehlerfrei* bezeichnet. Rückt ein Prozessor einmal von seinem Protokoll ab, so wird er *fehlerhaft* und von da an immer als fehlerhaft bezeichnet, selbst für den Fall, daß er später P_i wieder gehorchen sollte. Von den n Prozessoren dürfen höchstens t fehlerhaft sein ($t < \frac{n}{10}$ wird sich später als günstig erweisen). Jeder Prozessor G_i besitzt eine lokale Variable *message(i)*, in der er seine aktuelle Nachrichtenversion speichert.

Definition 1: Die Prozessoren in einer Menge $\{G_i : i \in P\}$ (mit $P \subseteq \{1 \ldots n\}$) erreichen eine *Einigung* über einen Nachrichtenwert, wenn jeder Prozessor aus dieser Menge anhält und nach Ausführung ihrer Protokolle $message(i) = message(j)$ für alle $i, j \in P$ gilt.

Definition 2: Die fehlerfreien Prozessoren G_i erreichen eine *byzantinische Einigung*, wenn

1. alle fehlerfreien Prozessoren G_i eine Einigung erreichen
2. Wenn alle fehlerfreien Prozessoren G_i die gleiche Anfangsnachricht $M_i = M$ haben, dann einigen sich die fehlerfreien Prozessoren auf die Nachricht M.

Man beachte, daß bei byzantinischer Einigung nichts über die fehlerhaften Prozessoren ausgesagt wird. Wenn alle fehlerfreien Prozessoren die gleiche Anfangsnachricht haben, dann wird das System als fehlerfrei bezeichnet, sonst als fehlerhaft. Es sollte erwähnt werden, daß in einem System mit bis zu t fehlerhaften Prozessoren, diese nicht von vornherein festgelegt sind. Es wird vielmehr der worst-case-Fall angenommen, daß zu jeder Zeit während des Ablaufs, Prozessoren fehlerhaft werden können, solange die Gesamtanzahl an fehlerhaften Prozessoren die obere Grenze t nicht überschreitet.

Die Prozessoren erreichen Einigung über Nachrichtenaustausch. Dabei wird ein Prozessor G_i von k anderen Prozessoren Nachrichten anfordern. Da bis zu t Prozessoren fehlerhaft sein können und somit u.U. niemals antworten, sollte G_i auf nicht mehr als $k - t$ Antworten warten. Daher

Definition 3: Eine *Runde* in den Berechnungen eines fehlerfreien Prozessors G_i ist das Zeitintervall zwischen G_i's Informationsanforderung an k andere Prozessoren, dem Empfang von mindestens $k - t$ Antworten und der Verarbeitung der Information.

Jeder Prozessor G_i hat eine lokale Phasenuhr $p(i)$, die am Ende jeder Runde um Eins erhöht wird.

Diese Ausarbeitung stellt einen randomisierten Algorithmus in zwei Varianten vor. In der ersten Variante wird bei einer festen Anzahl an Runden mit geringer Fehlerwahrscheinlichkeit byzantinische Einigung erreicht. In der zweiten Variante wird nach einer zu erwartenden Anzahl von Runden byzantinische Einigung erreicht, und zwar ohne Fehler. Für den asynchronen Fall werden folgende Annahmen getroffen:

1. Wann immer ein fehlerfreier Prozessor G_i eine Nachricht an einen anderen fehlerfreien Prozessor G_j schickt, wird diese Nachricht G_j letztendlich erreichen.
2. Ein fehlerfreier Prozessor G_i arbeitet sein Protokoll P_i in endlicher Zeit ab.

Diese Eigenschaften könnten auch einfach in die Definition eines fehlerfreien Prozessors aufgenommen werden.

11.2.3 Authentisierung mittels digitaler Unterschrift

Die hier vorgestellten Lösungen benutzen zur Authentisierung der Nachrichten die sog. digitale Unterschrift. Somit kann jeder Prozessor G_i sicher sein, daß die Nachricht, die er von einem anderen Prozessor erhält, auch wirklich von diesem stammt. Das Prinzip der digitalen Unterschrift lautet:

Ein Sender G_i „unterschreibt“ die Nachricht, die er verschicken will, dadurch, daß er die Nachricht mit seinem geheimen Schlüssel σ_i verschlüsselt. Der Empfänger kann dann diese Nachricht mit G_i's öffentlichem Schlüssel κ_i wieder entschlüsseln.

Die Tatsache, daß es sehr schwierig ist, von dem öffentlichen Schlüssel κ_i auf den geheimen Schlüssel σ_i zu kommen, garantiert, daß kein anderer außer G_i eine mit σ_i verschlüsselte Nachricht erstellen kann. Fortan soll $\sigma_i(M)$ für die von G_i „unterschriebene" Nachricht M stehen.
Damit stets die neueste Nachricht erkannt werden kann, wird zusätzlich zur aktuellen Nachrichtenversion der Wert der aktuellen Runde mitgesendet (time stamping). Ein Prozessor G_i erzeugt also eine verschlüsselte Nachricht $N = \sigma_i(message(i), p(i))$. Die Originalnachricht kann jeder Prozessor mit Hilfe des öffentlichen Schlüssels von G_i berechnen: $\kappa_i(N) = message(i), p(i)$. Die öffentlichen Schlüssel werden in einem öffentlichen Verzeichnis gehalten und von einem fehlerfreien Prozessor, dem sog. „Dealer", den G_i zu Beginn zugeteilt. Der zur Verschlüsselung verwendete RSA-Algorithmus kann in [3] nachgelesen werden.

11.2.4 Teilen eines Geheimnisses

Das besondere Merkmal des später vorgestellten Algorithmus ist, wie bereits erwähnt, die Verwendung von A. Shamir's Methode zum Teilen eines Geheimnisses.
Auch hier wird ein fehlerfreier Prozessor D, der „Dealer", benötigt. Für jede Runde r bestimmt D zufällig ein Geheimnis $S^{(r)} = 0$ oder 1. Shamir gibt nun einen Algorithmus vor, mit dessen Hilfe man dieses Geheimnis in n Teile $s_i^{(r)}$ zerlegen kann, so daß mindestens k Teile benötigt werden, um das Geheimnis zu rekonstruieren. Mit weniger als k Teilen kann $S^{(r)}$ jedoch nicht gefunden werden. Der Dealer wählt $k = t + 1$. Da t die obere Grenze für die Anzahl fehlerhafter Prozessoren ist, können keine t „Verräter" das Geheimnis erraten. Der Dealer zerlegt $S^{(r)}$ in n Teilgeheimnisse $s_i^{(r)}$, unterzeichnet sie mit seiner digitalen Unterschrift und sendet sie an die Prozessoren G_i. Somit kann jeder Prozessor sicher sein, daß die Geheimnisse auch wirklich von D stammen und nicht verfälscht wurden. Will ein Prozessor G_i in der r-ten Runde das Geheimnis $S^{(r)}$ wissen, so fordert er von $2t$ anderen Prozessoren ihre Teilgeheimnisse an und wartet auf t Antworten (t Prozessoren können fehlerhaft sein). Mit seinem eigenen Teilgeheimnis hat er nun $t + 1 = k$ Werte, aus denen er $S^{(r)}$ rekonstruieren kann. Zur selben Zeit berechnen auch alle anderen Prozessoren das Geheimnis. Am Ende dieser Runde teilen zumindest alle fehlerfreien Prozessoren das Geheimnis $S^{(r)}$.
Es folgt nun, kurz erläutert, Shamir's Algorithmus zur Berechnung der Teilgeheimnisse aus einem Geheimnis S, so daß mindestens k Teile zur Rekonstruktion nötig sind. Solch ein Schema wird auch (k, n) *threshold scheme* genannt [4].

1. Zunächst wird ein zufälliges Polynom folgender Gestalt gewählt:
   $q(x) = a_0 + a_1x + a_2x^2 + ... + a_{k-1}x^{k-1}$, wobei $q(0) = a_0 = S$.

2. Dann die Werte s_i berechnet:
   $s_1 = q(1), s_2 = q(2), ..., s_i = q(i), ..., s_n = q(n)$

Mit beliebigen k Werten dieser s_i können mit Interpolation die Koeffizienten von $q(x)$ ermittelt und somit $S = a_0$ berechnet werden (Mit k Werten kann ein Polynom von Grad $(k-1)$ gelöst werden).
Mit dieser Methode können auch Hierarchien unterstützt werden, indem die Prozessoren ihrer Wichtigkeit nach mehrere Teilgeheimnisse erhalten.
Ein (k, n) threshold scheme mit $n = 2k - 1$ ist sehr robust, da das Geheimnis selbst dann noch rekonstruiert werden kann, wenn $\lfloor n/2 \rfloor = k - 1$ der n Teile zerstört sind. Die Verräter können jedoch selbst mit $k-1$ der k übrigbleibenden Teile das Geheimnis nicht erhalten.

11.2.5 Der Algorithmus

Erste Variante

Es wird nun der Algorithmus vorgestellt, wo die Prozessoren mit geringer Fehlerwahrscheinlichkeit byzantinische Einigung erreichen: Nach R Runden soll mit einer Wahrscheinlichkeit $1 - 2^{-R}$ eine Einigung erreicht werden. Die Grundstruktur des Übereinkunftsprotokolls (Byzantine Agreement Protokol) ist wie folgt:

```
PROCEDURE BAP; {for $G_i$}
BEGIN
    $message(i) := M_i$; {$G_i$'s Anfangsnachricht}
    FOR $r := 1$ TO $R$ DO
    BEGIN
        Polling($r$);
        Lottery($r$);
        Decision;
    END;
END;
```

Mit der Prozedur Polling fordert ein Prozessor G_i von allen anderen Prozessoren deren aktuelle Nachrichtenversion an.

```
PROCEDURE Polling($r$);
BEGIN
    sende $\sigma_i(message(i), r)$ an alle Prozessoren;
    fordere die r-te Nachrichtenversion aller Prozessoren an;
    sammle die eintreffenden $\sigma_j(message(j), r)$ bis $n - t$ Nachrichten erhalten;
    $temp(i)$ :=die Nachricht, die am häufigsten empfangen wurde;
    $count(i) := |\{j : (message(j), r) = (temp(i), r)\}|$;
END;
```

Streng genommen wird die r-te Nachrichtenversion nicht nur in der r-ten Iteration der Prozedur Polling gesammelt, sondern es existiert ein Sammel-Prozeß L_i, der alle ankommenden Nachrichtenversionen speichert und zur Verfügung stellt. Nun zum Herzstück des randomisierten Algorithmus:
Nach Ablauf der Prozedur Polling steht in $temp(i)$ die am meisten empfangene Nachricht, also nach Meinung von G_i der Kandidat für die gemeinsame Nachricht; wie oft G_i diese Nachricht empfangen hat steht in $count(i)$. Nun muß entschieden werden, ob die Nachricht $temp(i)$ angenommen wird oder nicht. Dazu kann man entweder $n - 2t \leq count(i)$ oder $\frac{n}{2} \leq count(i)$ vergleichen. Die Entscheidung, welcher Vergleich betrachtet wird, wird abhängig von dem zufällig gewählten Geheimnis $S^{(r)} = 0$ oder 1 getroffen. Dieses Geheimnis ermitteln die Prozessoren G_i in der Prozedur Lottery.

```
PROCEDURE Lottery($r$);
BEGIN
    fordere alle Teilgeheimnisse $s_j^{(r)}$ an;
    sende $s_i^{(r)}$ an alle Prozessoren ;
    warte, bis t verschiedene Werte von $s_j^{(r)}$ eingetroffen sind;
    berechne $S^{(r)}$ aus $s_i^{(r)}$ und den eingetroffenen $s_j^{(r)}$;
END;
```

Man beachte, daß das Geheimnis $S^{(r)}$ keinem fehlerhaften Prozessor bekannt ist, bevor mindestens ein fehlerfreier Prozessor Polling beendet hat. Nun kann die Prozedur Decision aufgerufen werden, in der letztendlich über die Annahme von $temp(i)$ entschieden wird.

```
PROCEDURE Decision;
BEGIN
    IF ($S^{(r)} = 0$ AND $\frac{n}{2} \leq count(i)$) OR ($S^{(r)} = 1$ AND $n - 2t \leq count(i)$)
        THEN $message(i) := temp(i)$
    ELSE $message(i)$ :='system faulty';
END;
```

Theorem 1:
Sei $t < \frac{n}{10}$. Falls jeder fehlerfreie Prozessor die Anfangsnachricht M hat (fehlerfreies System), dann haben am Ende von BAP alle fehlerfreien Prozessoren $message(i) = M$.
Falls nicht alle fehlerfreien Prozessoren die Anfangsnachricht M haben (fehlerhaftes System), dann haben alle fehlerfreien Prozessoren am Ende von BAP mit einer Wahrscheinlichkeit von $1 - 2^{-R}$ den gleichen Wert in $message(i)$.

Beweis: (partielle Korrektheit)
Es sei vorrausgesetzt, daß jeder fehlerfreie Prozessor terminiert.
Fallunterscheidung:

1. Alle fehlerfreien Prozessoren haben bei Eintritt in die r-te Iteration des FOR-Statements von BAP $message(i) = M$:

   Dann wird diese Iteration mit $message(i) := temp(i) = M$ beendet, da nach Beendigung von Polling alle fehlerfreien Prozessoren $temp(i) = M$ und $count(i) \geq n - 2t$ haben werden. Somit ist in der Prozedur Decision sowohl die Ungleichung $(n - 2t) \leq count(i)$ als auch die Ungleichung $\frac{n}{2} \leq count(i)$ erfüllt, also unabhängig vom zufälligen Wert $S^{(r)}$ ($count(i) \geq n - 2t > n - \frac{2n}{10} = n - \frac{n}{5} = \frac{4}{5}n > \frac{n}{2}$). Haben alle fehlerfreien Prozessoren obigen Stand erreicht, wird sich der gemeinsame Nachrichtenwert in $message(i)$ nicht mehr ändern und die Einigung ist erreicht.
   Insbesondere gilt, wenn das System fehlerfrei ist: alle fehlerfreien Prozessoren G_i einigen sich auf ihre Anfangsnachricht.

2. Mindestens $n - 4t$ fehlerfreie Prozessoren haben zu Beginn der r-ten Runde von BAP $message(i) = M$ (d.h. es gibt höchstens $4t$ andere Nachrichtenwerte):

   In der r-ten Runde von BAP bekommt ein fehlerfreier Prozessor mindestens l-mal die Nachricht M, wobei $l := n - 4t - t = n - 5t$. Also gilt:
   $count(i) \geq n - 5t > n - 5\frac{n}{10} = \frac{n}{2}$. Wenn nun $S^{(r)} = 0$, was mit einer Wahrscheinlichkeit von $\frac{1}{2}$ der Fall ist, dann haben alle fehlerfreien Prozessoren am Ende der r-ten Runde von BAP $message(i) := temp(i) = M$.

3. Weniger als $n - 4t$ fehlerfreie Prozessoren haben zu Beginn der r-ten Runde von BAP $message(i) = M$:

   Dann gilt $count(i) < n - 4t < n - 2t$. Wenn nun $S^{(r)} = 1$, dann haben am Ende der r-ten Iteration alle fehlerfreien Prozessoren $message(i)$:='system faulty', was auch einer Einigung auf eine gemeinsame Nachricht entspricht. Dies geschieht ebenfalls mit einer Wahrscheinlichtkeit von $\frac{1}{2}$.

Ist einmal eine Einigung erreicht, so bleibt sie bestehen. Also ist die Wahrscheinlichkeit, keine Einigung in R Runden zu erreichen, gleich 2^{-R}.

Zweite Variante

Im Folgenden wird der Algorithmus mit erwarteter Einigung nach einer konstanten Anzahl an Runden vorgestellt. Sei weiterhin $t < \frac{n}{10}$. Der entscheidende Punkt ist nun:
Falls ein fehlerfreier Prozessor in der r-ten Runde mehr als $n - 2t$ gleiche Nachrichten zählt und $S^{(r)} = 0$ ist, dann empfangen alle anderen fehlerfreien Prozessoren mindestens $n - t - 2t - t = n - 4t$-mal diese Nachricht. Jetzt ist aber $count(j) \geq n - 4t > n - \frac{4n}{10} > n - \frac{5n}{10} = \frac{n}{2}$. Also wird jeder fehlerfreie Prozessor am Ende dieser Runde $message(j) := temp(j)$ setzen. Gilt also obige Situation für einen fehlerfreien Prozessor, werden alle fehlerfreien Prozessoren den gleichen Wert in $message(i)$ halten. Um diese Tatsache auszunutzen wird nun die Prozedur Decision leicht verändert. Decisionproof übernimmt nun auch die Erhöhung des Rundenindex.

```
PROCEDURE Decisionproof;
BEGIN
    IF ($S^{(r(i))} = 0$ AND $\frac{n}{2} \leq count(i)$) OR
        ($S^{(r(i))} = 1$ AND $n - 2t \leq count(i)$)
        THEN $message(i) := temp(i)$
    ELSE $message(i)$ :='system faulty';

    IF ($S^{(r(i))} = 0$ AND $n - 2t \leq count(i)$)
        THEN sende $\sigma_i$('Einigung über Wert V erreicht$'$) an alle;
    $r(i) := r(i) + 1$
END;
```

Da nun keine feste Rundenanzahl mehr vorgegeben ist, wird in die Prozedur BAP die boolsche Variable $inround(i)$ als Haltesignal eingeführt.

```
PROCEDURE BAP2; {for $G_i$}
BEGIN
    $inround(i) := TRUE$;
    $r(i) := 1$;
    $message(i) := M_i$;
    REPEAT BEGIN
        Polling($r$);
        Lottery($r$);
        Decisionproof;
    UNTIL $inround(i) = FALSE$;
END;
```

Da die Prozedur BAP2 nicht von selbst abbricht, läuft gleichzeitig die Prozedur Closefinish, die das Halten von BAP2 garantiert.

```
PROCEDURE Closefinish; {for $G_i$}
BEGIN
    REPEAT
        IF neue Nachricht $\sigma_j$('Einigung über Wert $V$ erreicht') erhalten
        THEN sende $\sigma_j$('Einigung über Wert $V$ erreicht') zu allen;
    UNTIL $t+1$ Nachrichten mit demselben $V$ gezählt wurden;
    $message(i) := V$;
    $inround(i) :=$ FALSE;
END;
```

Beachte, daß in der Prozedur Closefinish auf $(t+1)$ gleiche Nachrichten V gewartet wird, damit die fehlerhaften Prozessoren nicht eine Einigung vortäuschen können. D.h. mindestens ein fehlerfreier Prozessor G_i hat σ_i('Einigung über Wert V erreicht') gesendet. Somit endet das Programm korrekt.

Theorem 2:
Innerhalb von erwartungsgemäß vier Runden wird jeder fehlerfreie Prozessor ('Einigung über Wert V erreicht') für ein beliebiges V senden.

Beweis:
Decisionproof ist eine Erweiterung von Decision. Somit ist wegen der Überlegungen in dem letzten Kapitel eine Einigung der Prozessoren in den ersten zwei Runden zu erwarten. Nach weiteren zwei Runden kann damit gerechnet werden, daß $s^{(r)} = 0$ wird. Nun senden alle fehlerfreien Prozessoren die Nachricht ('Einigung über Wert V erreicht').

11.3 Ein deterministischer Versuch [2]

Im letzten Kapitel wurden zwei randomisierte Lösungen für das Problem der byzantinischen Generäle vorgestellt. In diesem Kapitel soll nun gezeigt werden, daß es keine Möglichkeit gibt, dieses Problem deterministisch zu lösen. Dabei soll gelten:
Das System arbeitet asynchron, d.h. es werden keine Annahmen über die Prozessorgeschwindigkeit oder Zeitverzögerungen im Nachrichtenaustausch gemacht. Es gibt außerdem keine Möglichkeit, den Ausfall eines Prozessors zu erkennen. Es gibt n Prozessoren mit $n \geq 2$. Jeder Prozessor hat ein Eingabe- und ein Ausgaberegister, die die Werte $\{b, 0, 1\}$ annehmen können, und einen unendlichen inneren Speicher. Die aktuellen Werte des Eingaberegisters und des Ausgaberegisters, der Programmzähler und der innere Speicher bestimmen den *inneren Zustand* eines Prozessors. Ein Prozessor ist im *Anfangszustand*, wenn Ausgaberegister $= b$ und im *Entscheidungszustand*, wenn Ausgaberegister $= 0$ oder 1. Ein Prozessor G_i geht deterministisch nach einer Transitionsfunktion vor. Diese kann den Wert im Ausgaberegister nicht mehr ändern, sobald ein Entscheidungszustand erreicht ist. Eine Nachricht ist ein Tupel (i, M), wobei i den Zielprozessor und $M \in \mathcal{M}$ den Wert der Nachricht angibt. Den Prozessoren steht ein Nachrichtensystem mit Nachrichtenspeicher zur Verfügung, das zwei Operationen unterstützt:

send(i,M): plaziert die Nachricht (i, M) im Speicher.

receive(i): löscht ein beliebiges (i, M) aus dem Speicher und sendet M zu Prozessor i, wobei auch endlich oft der Nullmarker $\emptyset$ gesendet werden kann.

Eine *Konfiguration* des Systems besteht aus den inneren Zuständen aller Prozessoren und dem Inhalt des Nachrichtenspeichers. In der *Anfangskonfiguration* befinden sich alle Prozessoren im Anfangszustand und der Nachrichtenspeicher ist leer. Eine Zustandsänderung erfolgt in zwei Phasen:
Sei C_1 eine Konfiguration. Zunächst erfolgt ein $receive(i)$, um aus dem Nachrichtenspeicher ein $M \in \mathcal{M} \cup \{\emptyset\}$ zu bekommen. Dann wird der neue, innere Zustand von G_i berechnet und endlich viele Nachrichten an andere Prozessoren gesendet $\rightarrow$ neue Konfiguration C_2.
Damit ist eine Zustandsänderung eindeutig durch das *Ereignis* $e = (i, M)$ bestimmt (Erhalt von M durch G_i). Man kann somit schreiben: $C_2 = e(C_1)$. Mehrere Zustandsänderungen werden zu einer Sequenz σ zusammengefaßt. Zwei Ereignisse $e_1 = (i_1, M_1)$ und $e_2 = (i_2, M_2)$ sind *disjunkt*, wenn gilt $i_1 \neq i_2$. Zwei Sequenzen sind *disjunkt*, wenn alle Ereignisse der ersten Sequenz paarweise disjunkt zu den Ereignissen der zweiten Sequenz sind. Das folgende Lemma verdeutlicht die Kommutativität von Sequenzen.

Lemma 1:
Seien σ_1 bzw. σ_2 zwei disjunkte Sequenzen, die von einer Konfiguration C zu den Konfigurationen C_1 bzw. C_2 führen. Dann kann σ_1 auf C_2 und σ_2 auf C_1 angewandt werden und beide Wege führen zur gleichen Konfiguration C_3.

Beweis:
Folgt unmittelbar aus der Definition des deterministischen Systems (vgl. Abbildung 1).

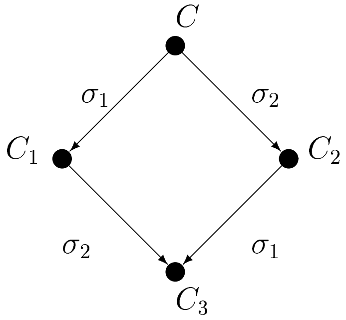


Abbildung 1

Um zu beweisen, daß es kein Ablaufprotokoll gibt, das alle fehlerfreien Prozessoren zu einer Einigung führt, wird ein Widerspruchsbeweis in zwei Teilen geführt. Zuerst wird gezeigt, daß es eine Anfangskonfiguration gibt, in der die Entscheidung noch nicht vorherbestimmt ist. Im zweiten Teil wird ein Ablauf konstruiert, wobei alle Nachrichten des Nachrichtensystems empfangen wurden, aber trotzdem keine Einigung erzielt wird. Dazu noch einige Begriffe:
Eine Konfiguration heißt bivalent, falls noch beide Entscheidungswerte 0 und 1 erreichbar sind. Eine Konfiguration ist univalent, falls nur mehr ein Wert erreicht werden kann. Im Speziellen werden diese Konfigurationen als 0-valent, falls der Entscheidungswert 0 ist und als 1-valent, falls der Entscheidungswert 1 ist, bezeichnet.

Lemma 2:
Es gibt eine bivalente Anfangskonfigurationen.

Beweis:
Annahme des Gegenteils, d.h. es gibt nur Anfangskonfigurationen, die entweder 1-valent oder 0-valent sind, wo der Entscheidungswert also feststeht. Alle Anfangskonfigurationen werden nun in eine Liste geschrieben, so daß sich zwei Nachbarkonfigurationen genau im Wert des Eingaberegisters eines Prozessors G_i unterscheiden. In dieser Liste muß es nun zwei Nachbarkonfigurationen C_0 und C_1 mit Entscheidungswert 0 bzw. 1 geben. Diese beiden Konfigurationen unterscheiden sich lediglich in dem Eingaberegisterwert eines Prozessors G_i. Wenn

nun dieser Prozessor in beiden Läufen ausfällt, dann ist es nicht mehr möglich, daß sich die beiden Entscheidungswerte unterscheiden, da ja nun beide Anfangskonfigurationen 'gleich' sind. Dies ist ein Widerspruch zur Annahme, daß es nur Anfangskonfigurationen gibt, bei denen der Wert der Einigung schon vorherbestimmt ist.

Lemma 3:
Sei C eine bivalente Konfiguration und $e = (p, M)$ ein Ereignis, das auf C anwendbar ist. Sei außerdem $\mathcal{C}$ die Menge der Konfigurationen, die von C aus erreichbar sind, ohne e anzuwenden und sei $\mathcal{D} = e(\mathcal{C}) = \{e(E) | E \in \mathcal{C}$ *und e kann auf E angewandt werden*$\}$. Dann enthält die Menge $\mathcal{D}$ eine bivalente Konfiguration.

Beweis:
Annahme des Gegenteils, d.h. $\mathcal{D}$ beinhaltet keine bivalente Konfiguration, also ist jede Konfiguration $D \in \mathcal{D}$ univalent.
Sei E_i eine i-valente Konfiguration, die von C erreichbar ist, $i = 0$ oder 1. E_i existiert, da C bivalent ist. Dann gilt entweder

- $E_i \in \mathcal{C}$ und sei $F_i = e(E_i) \in \mathcal{D}$, oder
- e wurde beim Erreichen von E_i angewendet, so daß ein $F_i \in \mathcal{D}$ existieren muß, von dem aus E_i erreichbar ist.

Es müssen nun zwei Konfigurationen $C_0, C_1 \in \mathcal{C}$ existieren, die durch die Anwendung eines Ereignisses $e' = (p', M')$ auseinander hervorgehen (o.E.d.A. gelte $C_1 = e'(C_0)$) und zwar derart, daß $D_i = e(C_i)$ i-valent ist, mit $i = 0, 1$).
Wenn nun $p' \neq p$, dann wäre $D_1 = e'(D_0)$ nach Lemma 1. Das ist aber nicht möglich, da aus jeder 0-valenten Konfiguration wieder eine 0-valente Konfiguration entsteht (vgl. Abbildung 2).

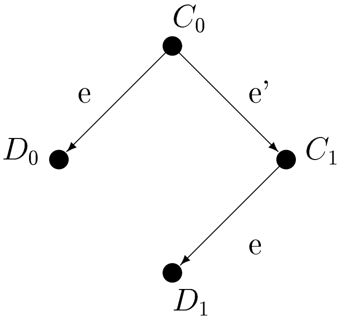


Abbildung 2

Wenn $p' = p$, stelle man sich eine Sequenz σ vor, in der keine Nachricht an den Prozessor p gesendet wird. Diese Sequenz führt angewandt auf C_0 zu einer Entscheidung. Sei $A = \sigma(C_0)$. Wegen Lemma 1 ist σ anwendbar auf D_i und führt zur i-valenten Konfiguration $E_i = \sigma(D_i), i = 0, 1$. Wegen Lemma 1 ist $e(A) = E_0$ und $e(e'(A)) = E_1$ (siehe Abbildung 3). Nun ist aber A bivalent, was aber ein Widerspruch zur Annahme ist, daß A zu einer Einigung führt. ⊓

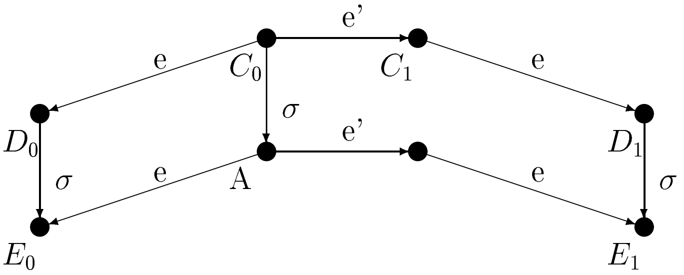


Abbildung 3

Mit Hilfe der Lemmata 2 und 3 kann nun bewiesen werden, daß es keinen Ablauf gibt, der zu einer Einigung führt, sobald ein falscher Prozessor existiert.

Es wird angenommen, daß ein solcher Ablauf existiert. Es muß also einen Schritt geben, der von einer bivalenten Konfiguration zu einer univalenten führt. Es ist aber immer möglich, das System so laufen zu lassen, daß solche Entscheidungsschritte vermieden werden.

Der Lauf wird in Etappen konstruiert, gestartet von einer Anfangskonfiguration. Ein gültiger Lauf (alle Nachrichten werden letztendlich verbraucht) wird wie folgt sichergestellt: es wird eine Warteschlange für die Prozessoren eingeführt, die in einer zufälligen Reihenfolge aufgebaut ist. Die Nachrichten werden in der Folge ihres Eintreffens im Nachrichtensystem gespeichert. Jede Etappe besteht aus einer oder mehreren Zustandsänderungen. Eine Etappe endet, wenn der erste Prozessor in der Warteschlange die früheste Nachricht aus dem Nachrichtenspeicher, der zu Beginn der Etappe nicht leer war, empfängt. Dieser Prozessor wird dann an das Ende der Warteschlange gesetzt. In irgendeiner unendlichen Sequenz von solchen Etappen ändert jeder Prozessor unendlich oft seinen Zustand und empfängt jede Nachricht, die ihm zugesandt wird.

Sei C_0 eine bivalente Anfangskonfiguration, deren Existenz durch Lemma 2 sichergestellt ist. Man stellt nun sicher, daß jede Etappe mit einer bivalenten Konfiguration beginnt. Angenommen C ist eine bivalente Konfiguration und der Prozessor p steht am Anfang der Warteschlange. Sei M die früheste Nachricht für p im Nachrichtenspeicher, falls diese existiert, sonst sei $M = \emptyset$. Sei $e = (p, M)$. Wegen Lemma 3 gibt es eine bivalente Konfiguration C', die von C aus mit Hilfe einer Sequenz, in der e das letzte Ereignis ist, das angewendet wird, erreichbar ist. Diese Sequenz kennzeichnet diese Etappe.

Jede Etappe endet also in einer bivalenten Konfiguration. Der Lauf, der dabei entsteht, ist gültig, da alle Nachrichten gelesen werden, aber trotzdem wird nie eine Einigung erreicht.

11.4 Zusammenfassung

In dieser Ausarbeitung wurde ein randomisierter Algorithmus in zwei Varianten zur Lösung des Problems der Byzantinischen Generäle vorgestellt und gezeigt, daß ein deterministischer Ablauf das Problem nicht löst, wenn auch nur ein Prozessor fehlerhaft ist. Der randomisierte Algorithmus kann sowohl im synchronen wie im asynchronen Fall angewandt werden. In letzterem Fall kann ein Prozessor G_j seine k-te Nachrichtenversion zu G_i schicken, während sich dieser noch in einer früheren Iteration befindet. Die einzelne Rundendauer kann also unterschiedlich sein von Prozessor zu Prozessor. Daher hat jeder Prozessor eine Liste L_i für die einkommenden Nachrichten. Im synchronen Fall haben alle Runden den gleichen Start- und Endpunkt und die Prozessoren lesen immer die gleiche Uhr.

Literaturverzeichnis

[1] M.O. Rabin *Randomized Byzantine Generals*, Proc. of the 24th Ann. IEEE Symp. on Foundations of Computer Science, 1983, pp. 403-409

[2] M.J. Fischer, N. Lynch, M. Paterson *Impossibility of distributed consensus with one faulty process*, J. ACM Vol. 32(2), 1985, pp. 374-382

[3] R. Rivest, A. Shamir, L. Adleman *A method for obtaining digital signatures and public-keys cryptosystems*, CACM Vol. 21, 1978, pp. 120-126

[4] A. Shamir *How to share a secret*, CACM Vol. 22, 1979, pp. 612-613

Vortrag 12

Minimale Spannbäume

Tobias Knopff
Technische Universität München
`knopff@informatik.tu-muenchen.de`

12.1 Motivation

Das Problem einen minimalen Spannbaum eines Graphen mit m Kanten und n Knoten zu finden ist lang bekannt und dementsprechend viele Lösungsansätze gibt es. Einer der ersten Algorithmen ist der von Boruvka mit einer Komplexität von $O(\min\{m \log m, n^2\})$, der auch diesem hier vorzustellenden Algorithmus zugrunde liegt. Weitere Standardalgorithmen sind der von Kruskal ($O(m \log m)$) und der von Prim ($O(m+n \log m)$ mit Fibonacci-Heaps). Der schnellste bekannte deterministische Algorithmus ist der von Gabow, Galil und Spencer mit einer Zeitkomplexität von $O(m \log \beta(m,n))$, wobei $\beta(m,n) = \min\{i | \log^{(i)} n \leq \frac{m}{n}\}$. Allen diesen Algorithmen zugrunde liegt die vergleichsbasierte RAM[1], d.h. bei der Berechnung werden Kantengewichte lediglich verglichen. Erlaubt man ein stärkeres Modell, bei dem die Bitdarstellungen der Kantengewichte manipuliert werden können, so gibt es einen Algorithmus von Fredman und Willard [6], der Zeitkomplexität $O(m)$ hat.

Der hier vorzustellende Algorithmus ([1]) ist randomisiert und benutzt auch nur die vergleichsbasierte RAM als Modell. Wie dann gezeigt wird benötigt er dazu als Erwartungswert nur $O(m)$ Zeit um den minimalen Spannbaum eines Graphen zu finden.

---

[1] RAM : random access machine

12.2 Grundlagen

Um das Problem des minimalen Spannbaums zu erläutern, sollen zuerst ein paar Begriffe eingeführt werden. Gegeben sei ein ungerichteter, zusammenhängender Graph $G = (V, E)$, bestehend aus einer n-elementigen Menge V von Knoten und einer m-elementigen Menge E von Kanten. Als Dichte des Graphen ist das Verhältnis $\frac{m}{n}$ von Anzahl der Kanten zu Anzahl der Knoten zu verstehen. Ein Spannbaum $T = (V, E')$ mit $E' \subseteq E$ ist ein zusammenhängender azyklischer Teilgraph von G, der alle Knoten enthält. Ist weiterhin eine Gewichtsfunktion $w : E \mapsto \mathbb{R}$ gegeben, so lässt sich das Gewicht eines Spannbaums definieren als Summe der Kantengewichte des Baums, also $w(T) := \sum_{e \in E'} w(e)$. Ein Spannbaum T heißt minimal, falls es keinen Baum gibt, dessen Gewicht echt kleiner ist. Für nichtzusammenhängende Graphen sind minimale Spannwälder definiert, die aus den Spannbäumen der Zusammenhangskomponenten bestehen. Im weiteren wird davon ausgegangen, daß alle Kantengewichte unterschiedlich sind und somit der minimale Spannbaum eindeutig ist. Ist dies von vornherein nicht der Fall, so kann dies durch eine zusätzliche Kantennummerierung erreicht werden.

Es gibt zwei wichtige Aussagen über minimale Spannbäume :

Satz 1 *Sei $S, \bar{S}$ ein Schnitt*[2] *und e eine Kante minimalen Gewichts aus $E_{S,\bar{S}}$. Dann gibt es einen minimalen Spannbaum, der e enthält.*

Satz 2 *Sei e eine Kante mit maximalen Gewicht auf einem Zyklus. Dann gibt es einen minimalen Spannbaum, der e nicht enthält.*

Sei F nun ein Wald in G. Dann bezeichne $F(v, w)$ den Pfad zwischen v und w in F, falls es diesen gibt. Weiterhin sei $w_F(v, w)$ das Gewicht der schwersten Kante auf $F(v, w)$ bzw. $w_F(v, w) = \infty$ falls es keinen Pfad zwischen v und w in F gibt. Eine Kante $\{v, w\}$ aus G heiße jetzt F-heavy, falls $w(v, w) > w_F(v, w)$, d.h. sie ist schwerer als alle Kanten, die auf dem Pfad zwischen v und w in F liegen. Anderfalls heie die Kante F-light. Daraus ergibt sich sofort, daß alle Kanten in F F-light sind. Weiterhin können als Folge von Satz 1 F-heavy Kanten nicht Teil des minimalen Spannbaums sein.

12.3 Ein randomisierter Algorithmus zum Finden eines minimalen Spannbaums

Als erstes wird der Algorithmus ausführlich vorgestellt. Danach wird ein Satz bzgl. des zufälligen Auswählens von Teilgraphen hergeleitet, wie es im Algorithmus durchgeführt wird. Im folgenden Abschnitt wird dieser Satz dann verwendet um den Zeitbedarf unter verschiedenen Gesichtspunkten zu bestimmen.

---

[2] Ein Schnitt ist eine Knotenpartition in S und $\bar{S}$ mit $V = S \uplus \bar{S}$. Die Kanten zwischen den Knotenmengen werden bezeichnet mit $E_{S,\bar{S}} := E \cap (S \times \bar{S})$.

12.3.1 Der Algorithmus

Der rekursive Algortithmus löst das Problem einen Minimalen Spannwald zu finden. Dieses ist etwas allgemeiner als das Problem einen minimalen Spannbaum zu finden, da es auch nichtzusammenhängende Graphen umfaßt. Da jedoch der Algorithmus evtl. rekursiv angewandt wird auf zufällig ausgewählte Teilgraphen, die dann i.a. nicht zusammenhängend sind, ist dies eine grundlegende Eigenschaft.

Der Algrithmus hat folgende Grundidee : Solange die Dichte $\frac{m}{n}$ gering ist, werden Kanten gesucht, die im minimalen Spannwald sind. Wird die Anzahl der Kanten zu groß in Relation zu den noch zu findenden Spannwaldkanten (diese sind in Anazhl gleich der Knoten), werden Kanten mit Hilfe eines für einen Teilgraphen mit dann weniger Kanten berechneten Spannwalds ausgeschlossen.

Am Anfang wird davon ausgegange, daß der Graph keine isolierten Knoten enthält. Der Algorithmus im einzelnen :

1.Schritt: Zu jedem Knoten wird die inzidente Kante minimalen Gewichts ermittlet. Die durch die Knoten und minimalen Kanten entstandenen Zusammenhangskomponenten werden nun zu Superknoten verschmolzen wobei alle anderen Kanten, die innerhalb einer solchen Zusammenhangskomponenten verlaufen, eliminiert werden, sowie entstandene isolierte Knoten. Ergeben sich parallele Kanten zwischen Knoten, so werden alle bis auf die mit dem niedrigsten Gewicht zwischen einem Knotenpaar eliminert.

2.Schritt: Falls die Dichte des Graphen größer als 6 ist, wird folgender Zwischenschritt ausgeführt : Ein Teilgraph H wird erstellt indem jede Kante mit Wahrscheinlichkeit $\frac{1}{2}$ zu H hinzugenommen wird. Daraufhin wird der Algorithmus rekursiv auf H angewandt und liefert einen Spannwald F_H für H. Danach werden alle F_H-heavy Kanten im ganzen Graphen G eliminiert.

3.Schritt: Falls G noch Knoten und Kanten hat, wird der Algorithmus rekursiv auf G angewendet. Der minimale Spannwald ergibt sich dann aus dem berechneten Sapnnwald aus dem rekursiven Aufruf zusammen mit den verschmolzenen Zusammenhangskomponenten aus Schritt 1.

Für das Finden der F_H-heavy Kanten in Schritt 2 wird ein Algorithmus zur Verifikation von minimalen Sapnnbäumen leicht verändert, wobei das Verifizieren eines minimalen Spannbaums in linearer Zeitkomplexität möglich ist (siehe Ergänzungen). Der Algorithmus prüft für jede nicht-Baumkante, ob diese Satz 2 aus Abschnitt 1 erfüllt. Ist dies der Fall, ist die Kante F_H-heavy und kann gelöscht werden.

Die Korrektheit des Algorithmus läßt sich induktiv beweisen :
Alle Kanten, die in Schritt 1 ausgewählt werden sind nach Satz 1 im minimalen Spannbaum. Alle Kanten, die in Schritt 1 und 2 gelöscht werden sind F_H-heavy

und nach oben gesagtem nicht im minimalen Spannbaum. Unter der Induktionsannahme, daß die rekursiv berechneten minimalen Spannwälder aus Schritt 2 und 3 korrekt sind, ist auch der gesamte Algorithmus korrekt.

12.3.2 Sampling Lemma

Durch das zufällige Auswählen eines Teilgraphen in Schritt 2 ergibt sich folgendes Problem :
Um möglichst viele F_H-heavy Kanten in Schritt 2 eliminieren zu können, muß H aus einer "günstigen" Wahl von Kanten bestehen. Ist dies nicht der Fall, sind zu viele Kanten in G F_H-light und können folglich nicht eliminiert werden. Da die Auswahl von H aber zufällig erfolgt, ist die Güte von H bzgl. der Kantenelimination nicht gewiss. Im folgenden soll daher die Wahrscheinlichkeit gezeigt werden, mit der F_H-light Kanten einen bestimmte Anzahl nicht überschreiten, sowie der Erwartungswert für die Anzahl von F_H-light Kanten.

Für den Beweis wird eine Erweiterung von Kruskals Algorithmus verwendet. Dabei wird mit einem leeren Wald F_H begonnen und die Kanten in aufsteigender Reihenfolge nach ihren Gewicht sortiert bearbeitet. Dabei werden für jede Kante e folgende beide Schritte ausgeführt :

- Bestimme, ob die Kante e getrennte Zusammenhangskomponenten in F_H verbindet
- Eine Münze wird geworfen, die mit Wahrscheinlichkeit p mit der "Kopf"-Seite nach oben liegen bleibt.

Falls der Münzwurf "Kopf" ergibt, wird die Kante e in den Teilgraphen H aufgenommen. Wird "Kopf" geworfen und die Kante verbindet Zusammenhangskomponenten in F_H, dann wird diese zusätzlich in F_H eingesetzt. Nach Konstruktion liefert diese Vorgehensweise für H den gleichen Spannwald F_H wie Kruskals Algorithmus und daher ist F_H der minimale Spannwald von H.

Betrachtet man die Konstruktion von F_H mit Hinblick auf die Konstruktion eines minimalen Spannwalds F_G für G ergibt sich folgendes : Kannten, deren Endpunkte in der selben Zusammenhangskomponente liegen, sind nicht im minimalen Spannwald von H und folglich auch nicht im minimalen Spannwald von G. Diese entsprechen den Kanten, die im Algorithmus in Schritt 2 ausgeschlossen werden, entweder durch Konstruktion des minimalen Spannwalds für den Teilgraphen H ("Kopf" bei Mnzwurf) oder bei der Elimination der F_H-heavy Kanten ("Zahl" bei Mnzwurf). Da diese Kanten also sowieso wegfallen ist der Ausgang des Münzwurfs bzgl. G irrelevant und wird im weiteren nicht mitgezählt. Auch die Kanten seien im weiteren ausgeschlossen. Übrig bleiben also nur die Kanten und ihre entsprechenden Münzwürfe, die Zusammenhangskomponenten verbinden würden, falls sie in H aufgenommen würden. Diese Kanten entsprechen den

F_H-light Kanten, die im Schritt 2 übrigbleiben. Schlecht im Hinblick auf F_G sind die Kanten, bei denen der Münzwurf "Zahl" ergibt und diese nicht in H aufgenommen werden. Diese sind sozusagen die "besseren Kandidaten" für F_H bzgl. F_G als die Kanten die die Auswahl für H letztendlich ausmachen, da für jede solche Kante eine andere Kante mit höheren Gewicht gefunden werden muss, die die Zusammenhangskomponenten verbindet. Eine weitere Kante muss es aber nicht mehr geben oder auch sie muss nicht in H wegen des Münzwurfs aufgenommen. Die anderen Kanten, deren Münzwurf "Kopf" ergab, sind F_H-Baumkanten und daher in der Anzahl höchstens $n - 1$[3]. Damit die Anzahl der F_H-light Kanten nun einen gegebenen Wert k überschreitet, müssen also nach k Münzwürfen ungenügend viele Kanten ausgewählt sein, die einen minimalen Spannwald von H ergeben. Die Anzahl i der schon gefundenen F_H-Kanten kann dabei jeder Wert bis $n - 1$ sein. Dies entspricht der Wahrscheinlichkeit, daß nach k Münzwürfen höchstens $n - 1$ "Kopf"-Ereignisse stattgefunden haben. Die Wahrscheinlichkeit dafür ist höchstens gegeben durch

$$\sum_{i=0}^{n-1} \binom{k}{i} p^i (1-p)^{k-i}$$

Damit ergibt sich folgendes Lemma :

Lemma 1 *Sei G ein Graph mit n Knoten und H ein Teilgraph von G, der durch zufälliges Auswählen mit Wahrscheinlichkeit p von Kanten aus G entsteht. Weiterhin sei F_H der minimale Spannwald von H. Dann übersteigt die Anzahl der F_H-light Kanten in G ein gegebenes k mit einer Wahrscheinlichkeit von höchstens*

$$\sum_{i=0}^{n-1} \binom{k}{i} p^i (1-p)^{k-i}$$

Korollar 1 *Es ergibt sich als Erwartugswert für die Anzahl der F_H-light Kanten $\frac{n}{p}$.*

In diesem Fall $p = \frac{1}{2}$ ist der Erwartungswert für die Anzahl der F_H-light Kanten $2n$.

12.3.3 Zeitkomplexität des Algorithmus

Es soll nun ermessen werden, wieviel Zeit der Algorithmus im Erwartungfall und im schlechtesten Fall braucht, um das Problem zu lösen. Anschliessend wird noch gezeigt, daß der Algorithmus mit hoher Wahrscheinlichkeit in Zeit linear abhängig von der Anzahl der Kanten läuft.

---

[3] Ein Wald mit n Knoten hat höchstens $n - 1$ Kanten.

Zuerst wird die Zeit, die der Algorithmus in Schritt 1 bis 3 braucht, ohne rekursive Aufrufe betrachtet : Für Schritt 1 wird mittels einfacher Graphenalgorithmen lediglich $O(m)$ Zeit benötigt. Auch Schritt 2 braucht nur $O(m)$ Zeit, falls ein Algorithmus zur Verifikation von minimalen Spannbäumen verwendet wird, der in linearer Zeit läuft. Damit ist die ganze benötigte Zeit pro Inkarnation nur $O(m)$, sei cm mit geeigneter Konstatnte c im weieren eine obere Schranke dafür.

Erwartungswert des Zeitbedarfs bei einem worst-case Graphen

Sei $T(m)$ der Erwartungswert der Zeit, die der Algorithmus braucht für einen worst-case[4] Graphen mit m Kanten. Per Induktion soll nun gezeigt werden, daß $T(m) \leq 7cm$ gilt, wobei c die Konstante aus dem letzten Abschnitt ist:
Der Induktionanfang mit einer Kante gilt trivialerweise.
Induktionschritt : sei n' die Anzahl der Knoten und m' die Anzahl der Kanten im Graphen nach dem 1. Schritt. Es werden mindestens $\frac{n}{2}$ Zusammenhangskomponenten zu Superknoten geschrumpft, da sich schlechtestenfalls für jeweils zwei benachbarte Knoten dieselbe Kante als minimale ergibt. Dies hat zur Folge, daß die Anzahl der Knoten mindestens halbiert wird, d.h. $n' \leq \frac{n}{2}$, und da die Anzahl der Kanten um mindestens $\frac{n}{2}$ abnimmt, also $m' \leq m - \frac{n}{2}$. Resultierend aus den zwei Ungleichungen ergibt sich $m' \leq m - n'$. Nun muß unterschieden werden, ob der Zwischenschritt in Schritt 2 ausgeführt wird, also ob die Dichte $\frac{m'}{n'}$ die Grenze 6 überschreitet :

1.Fall – $m' \leq 6n'$ **:** Wird diese Bedingung in obige Ungleichung eingesetzt, so ergibt sich $m' \leq m - \frac{m'}{6}$ und damit $m' \leq \frac{6}{7}m$. Das heißt, daß in der nächsten Phase die Anzahl der Kanten um mindestens $\frac{1}{7}$ geringer ist. Mit der Induktionshypothese, daß $T(\frac{6}{7}m) \leq 7c\frac{6}{7}n$ gilt, ergibt sich $T(m) \leq cm + T(\frac{6}{7}m) \leq cm + 7c\frac{6}{7}m = 7cm$.

2.Fall – $m' > 6n'$ **:** In diesem Fall gibt es zwei rekursive Aufrufe : einmal für den Teilgraphen H und einmal für den Graphen, der nach der Kantenelimination übrig bleibt, d.h. der Graph der F_H-light Kanten. Es sei X die Zufallsvariable für die Anzahl der Kanten in H. Nach Schritt 1 sind nur noch höchstens $m - 1$ Kanten in G. Da jede Kante mit Wahrscheinlichkeit $\frac{1}{2}$ in den Teilgraphen H hinzugenommen wird, ergibt sich ein Erwartungswert für X von höchstens $\frac{1}{2}(m-1)$. Da die Anzahl der Kanten in G nach Schritt 1 höchstens $m - 1$ ist, kann X auch nicht größer als $m - 1$ sein und somit gilt die Induktionsannahme $T(X) \leq 7cX$. Weiterhin ergibt sich als Erwartungswert für $T(X)$ höchstens der Erwartungswert von $7cX$, welcher $7c\frac{m}{2}$ beträgt.

---

[4] Worst-Case bezieht sich hierbei auf Schritt 1, d.h. nur jeweils zwei Knoten mit einer gemeinsamen Kante (ggf. einmal ein Knotentripel mit zwei Kanten bei Graphen mit ungerader Knotenzahl) ergeben eine Zusammenhangskomponente, die dann zu einem Superknoten geschrumpft wird.

Für den Restgraphen, der aus den F_H-light Kanten besteht, sei die Zufallsvariable Y die Anzahl der Kanten. Wegen Korollar 1 beträgt der Erwartungswert von Y $2n'$, d.h. höchstens doppelt soviel Kanten wie Knoten sind F_H-light im Graphen in Schritt 2. Da gilt $n' \leq \frac{m'}{6} < \frac{m}{6}$, folgt daraus, daß der Erwartungswert von Y höchstens $\frac{m}{3}$ ist. Genau wie bei X kann auch Y höchstens $m-1$ sein und damit gilt die Induktionsannahme $T(Y) \leq 7cY$. Als Erwartungswert für $T(Y)$ ergibt sich $T(Y) \leq 7cY \leq 7c\frac{m}{3}$.

Der Erwartungswert der Zeit, die der Algorithmus braucht ergibt sich in diesem Fall, zusammen mit den zwei rekursiven Aufrufen : $T(m) \leq cm + T(X) + T(Y) \leq cm + 7c\frac{m}{2} + 7c\frac{m}{3} \leq 7cm$.

Damit ist gezeigt, daß auch für schlechte Graphen der Erwartungswert der Laufzeit linear von der Anzahl der Kanten abhängt.

Worst-case Zeitbedarf

Betrachtet wird jetzt der Graph nach Schritt 1 mit m' Kanten und n' Knoten. Wie schon oben gezeigt gelten die Ungleichungen $n' \leq \frac{n}{2}$ und $m' < m$. Falls die Dichte $\frac{m'}{n'}$ die Grenze 6 nicht überschreitet, wird der Algorithmus auf den Restgraphen sofort angewendet. Ansonsten erfolgt vorher noch ein rekursiver Aufruf auf einen zufällig ausgewählten Teilgraphen.

Im folgenden sei nun die Dichte mehr als 6. Wie im letzten Abschnitt sei X die Anzahl der Kanten im Teilgraphen H und Y die Anzahl der F_H-light Kanten nach Schritt 2. Der Teilgraph mit X Kanten und der Teilgraph mit Y Kanten ergeben zusammen höchstens den gesamten Graphen, der nach Schritt 1 übrigbleibt. Die einzigen Kanten, die in beiden Teilgraphen enthalten sind, sind die des minimalen Spannbaums von H. Somit ergibt sich[5] $X+Y \leq m'+n'-1 < m$. Mittels dieser Ungleichung kann die Anzahl der rekursiven Aufrufe nach oben abgeschätzt werden. Sei $I(m)$ die worst-case Anzahl von rekursiven Aufrufen. Dann gilt :

$$\begin{aligned} I(1) &= 1 \\ I(m) &\leq 1 + \max\{I(x) + I(y) : x + y < m\} \quad \text{falls} \quad m > 1 \end{aligned}$$

Durch Induktion zeigt sich $I(m) \leq m$.

Augrund des eben gezeigten, läßt sich der worst-case Zeitbedarf $T(n,m)$ für einen Graphen mit n Knoten und m Kanten abschätzen. Es ergeben sich folgende Rekursionsgleichungen :

$$\begin{aligned} T(1,m) &= O(m) \\ T(n,m) &\leq O(m) + \max\{T(\lfloor \tfrac{n}{2} \rfloor, x) + T(\lfloor \tfrac{n}{2} \rfloor, y) : x + y < m\} \quad \text{falls} \quad n > 1 \end{aligned}$$

---

[5] mit Hilfe von $|A| + |B| = |A \cup B| + |A \cap B|$ und der schon gezeigten Formel $m' \leq m - n' \leadsto m' + n' \leq m$.

Per Induktion läßt sich beweisen, daß $T(n,m) = O(m \log n) = O(m \log m)$ gilt.

Zeitbedarf mit hoher Wahrscheinlichkeit

Der Algorithmus soll nun mit wahrscheinlichkeitsthoeretischen Hilfsmittel analysiert werden. Wie schon gezeigt, gibt es nur einen rekursiven Aufruf mit einem Graphen mit höchstens $\frac{6}{7}m$ Kanten, falls die Dichte des Graphen nach Schritt 1 die Grenze 6 nicht erreicht. Anderfalls bei $\frac{m'}{n'} > 6$ erfolgen zwei rekursiven Aufrufe : einmal mit dem zufällig ausgewählten Teilgraphen H mit X Kanten und zum anderen mit dem Graphen bestehend aus den F_H-light Kanten, deren Anzahl Y ist. Es gilt nun zu zeigen, daß $X + Y \leq (1-\epsilon)m$ mit hoher Wahrscheinlichkeit gilt für eine geeignete kleine positive Konstante ϵ. Mittels Rekursionsformeln kann dann eine Zeitkomplexität von $O(m)$ hergeleitet werden.

Die Anzahl X der Kanten im Teilgraphen H ist binomialverteilt mit einem Mittelwert von $\frac{m'}{2} \leq \frac{m}{2}$. Eine Abschätzung der Binomialverteilung ergibt, daß die Wahrscheinlichkeit, daß X den Wert $\frac{m}{2}(1+\delta_1)$ überschreitet, exponentiell klein ist, nämlich $\exp(-\Omega(m))$ für jede Konstante $\delta_1 > 0$. Sei δ_1 nun $\frac{1}{10}$, dann ergibt sich, daß X größer als $\frac{11}{20}m$ mit Wahrscheinlichkeit $\exp(-\Omega(m))$ ist.

Als nächstes soll nun die Anzahl Y der F_H-light Kanten im Restgraphen abgeschätzt werden. Aus Lemma 1 mit $p = \frac{1}{2}$ ergibt sich eine obere Schranke von $\sum_{i=0}^{n'} \binom{k}{i} \left(\frac{1}{2}\right)^k$. Dies entspricht der Wahrscheinlichkeit, daß höchstens n' mal "Kopf" geworfen wird bei k Münzwürfen mit einer Laplace-Münze. Der Erwartungswert ist $\frac{k}{2}$. Sei k nun $\frac{m}{3}(1+\delta_2)$ mit positiver Konstanten δ_2. Weiterhin ist $n' \leq \frac{m'}{6} \leq \frac{m}{6}$ und somit ergibt sich $n' \leq \frac{k}{2(1+\delta_2)}$. Wiederum zeigt die Abschätzung der Binomialveerteilung, daß die Wahrscheinlichkeit, mit der höchstens n' "Kopf"-Ereignisse während des k-maligen Werfens einer Münze auftreten, $\exp(-\Omega(m))$ ist. Sei nun $\delta_2 = \frac{1}{20}$, dann ergibt sich, daß Y größer als $\frac{7}{20}m$ ist mit Wahrscheinlihkeit $\exp(-\Omega(m))$.

Im folgenden wird eine Phase des Algorithmus als *Fehlschlag* bezeichnet, falls der Zwischenschritt in Schritt 2 ausgeführt wird, jedoch X größer als $\frac{11}{20}m$ oder Y größer als $\frac{7}{20}m$ ist. Mit dem oben gezeigten ist die Wahrscheinlichkeit, daß es zu einem *Fehlschlag* kommt $\exp(\Omega(m))$. Sei r eine bisher nicht bestimmter Parameter. Eine Phase heiße nun *reichlich*, falls die Anzahl der Kanten im Graphen r überschreitet. Unter der Annahme, daß keine *reichliche* Phase zu einem *Fehlschlag* wird, wird nun die Laufzeit des Algorithmus bestimmt.

Sei im weiteren $A(m)$ die benötigte Laufzeit des Algorithmus für einen Graphen mit m Kanten unter der Annahme, daß keine *reichliche* Phase zu einem *Fehlschlag* wird. Außerdem sind c_0 und c_1 geeignete positive Konstante. Dann ergibt sich mit der worst-case Zeitschranke von $O(m \log m)$ und der Definition von *Fehlschlag* folgende Rekursionsgleichungen :

$$\begin{aligned} A(m) &\le c_0 m \log m \qquad \text{falls} \quad m \le r \\ A(m) &\le c_1 m + \max\{A(\tfrac{11}{20}m) + A(\tfrac{7}{20}m), A(\tfrac{6}{7}m)\} \qquad \text{falls} \quad m > r \end{aligned}$$

Daraus läßt sich herleiten :

Lemma 2 *Für ein geeignetes* $\epsilon < 1$ *ergibt sich* $A(m) \le 10c_1 m + c_0 m^{1-\epsilon} r^{\epsilon} \log r$.

Der Beweis erfolgt mittels Induktion über m.
Falls $m \le r$ ist, dann gilt $c_0 m^{1-\epsilon} r^{\epsilon} \log r \ge c_0 m \log m$ und somit ist der Induktionsanfang bewiesen. Für den Induktionschritt sei nun $m > r$. Dann ergibt sich mit der Induktionshypothese :

$$A(\tfrac{11}{20}m) \le \tfrac{11}{2} c_1 m + c_0 (\tfrac{11}{20}m)^{1-\epsilon} r^{\epsilon} \log r$$

Wird ϵ nun so gewählt, daß $(\frac{11}{20})^{1-\epsilon} \le \frac{3}{5}$ ist, dann ergibt sich :

$$A(\tfrac{11}{20}m) \le \tfrac{11}{2} c_1 m + \tfrac{3}{5} c_0 m^{1-\epsilon} r^{\epsilon} \log r$$

Ebenso, falls ϵ die Ungleichung $(\frac{7}{20})^{1-\epsilon} \le \frac{2}{5}$ erfllt, ergibt sich :

$$A(\tfrac{7}{20}m) \le \tfrac{7}{2} c_1 m + \tfrac{2}{5} c_0 m^{1-\epsilon} r^{\epsilon} \log r$$

Aus beiden folgt dann :

$$c_1 m + A(\tfrac{11}{20}m) + A(\tfrac{7}{20}m) \le 10 c_1 m + c_0 m^{1-\epsilon} r^{\epsilon} \log r$$

Zusätzlich folgt aus der Induktionshypothese :

$$c_1 m + A(\tfrac{6}{7}m) \le 10 c_1 m + c_0 m^{1-\epsilon} r^{\epsilon} \log r$$

Damit ist gezeigt, daä die Ungleichung des Lemmas für alle Fälle gilt.

Zu zeigen bleibt noch, daß die Annahme, daß keine *reichliche* Phase zum *Fehlschlag* wird, mit hoher Wahrscheinlichkeit zutrifft. Wird der Algorithmus auf einen Graphen mit m Kanten angewendet, so wird $r = \frac{m}{\log^{1/\epsilon} m}$ gewählt. Solange keine *reichliche* Phase zum *Fehlschlag* wird, ergibt sich mit Lemma 2 eine Laufzeit des Algorithmus von $O(m)$. Die Wahrscheinlichkeit, daß eine *reichliche* Phase zum *Fehlschlag* wird ist $\exp(-\Omega(r))$. Es gibt höchstens m rekursive Anwendungen und damit ist die Wahrscheinlichkeit, daß es auch nur zu einen *Fehlschlag* kommt $m \exp(-\Omega(r)) = \exp(-\Omega(r))$. Damit läuft der Algorithmus nicht in $O(m)$ Zeit mit Wahrscheinlichkeit $\exp(-\Omega(m/\log^{O(1)} m))$.

12.4 Ergänzungen

Ein wesentlicher Teil des Algorithmus ist das Finden der F_H-heavy Kanten in Schritt 2 in linearer Zeit. Da diese nicht mittels eines randomisierten Algorithmus erfolgt wurde im bisherigen Teil nicht weiter darauf eingegangen. Kompletterweise soll nun hier kurz erläutert werden, wie ein Spannbaum in linearer Zeit als minimal verifiziert werden kann. Allerdings sind die verwendeten Algorithmen teilweise zeimlich komplex, so daß hier nur die groben Grundzüge der Algorithmen vorgestellt werden können. Für ausführliche Beschreibung sei auf die Literaturhinweise verwiesen.

Der Verifikationsalgorithmus baut auf weiteren Algorithmen auf, zum einen auf einem zum Finden von niedrigsten gemeinsamen Vorfahren in $O(m)$ Zeit und zum anderen auf einen weiteren Verifikationsalgorithmus, der aber $O(m\,\alpha(m,n))$[6] Zeit benötigt. Genauere Beschreibungen der Algorithmen sind [4] für den Algorithmus zum Finden niedrigster gemeinsamer Vorfahren, [2] für den Algorithmus zur Verifikation in $O(m\,\alpha(m,n))$ Zeit und [3] für den Verifikationsalgorithmus in $O(m)$ Zeit.

12.4.1 Verifikation eines minimalen Spannbaums in $O(m)$ Zeit

Der Algorithmus benutzt mehrere Techniken, um einen Baum als minimalen Spannbaum zu verifizieren. Die Grundidee ist, den Graphen in kleinere Teilgraphen zu unterteilen, in denen dann die Spannbäume effizient verifiziert werden können, wie noch gezeigt werden soll. Da jedoch die zu verifizierenden nicht-Baumkanten im allgemeinen nicht schon innerhalb eines dieser Teilgraphen verlaufen, werden sie gegebenenfalls durch mehrere andere Kanten, mit gleichem Gewicht, ersetzt.

Als erstes wird eine beliebige Wurzel r für den Baum ausgewählt um dann für die Endpunkte v und w jeder nicht-Baumkante der Form $\{v,w\}$ deren niedrigsten gemeinsamen Vorfahren $LCA(v,w)$ zu ermitteln. Falls die beiden Knoten nicht direkt verwandt sind (d.h. $v \neq LCA(v,w) \neq w$), dann wird die Kante $\{v,w\}$ durch die Kanten $\{v, LCA(v,w)\}$ und $\{LCA(v,w), w\}$ ersetzt, wobei die beiden neuen Kanten das Gewicht der alten Kante erhalten. Dadurch verdoppelt sich höchstens die Anzahl der nicht-Baumkanten, dafür sind diese nur Rückwärtskanten und der minimale Spannbaum hat sich nicht verändert[7]. Dieser Schritt kann in $O(m)$ Zeit durchgeführt werden, falls ein Algorithmus verwendet wird, der alle

---

[6] Die Funktion α ist die Inverse zu einer Variante der Ackermannfunktion : sei $A(i,0)=0$, $A(0,j)=2^j$ für $j \geq 1$, $A(i,1)=A(i-1,2)$ für $i \geq 1$ und $A(i,j)=A(i-1,A(i,j-1))$ für $i \geq 1, j \geq 1$. Dann ist $\alpha(m,n)=\min\{i \geq 1 | A(i,\lfloor\frac{2m}{n}\rfloor) > \log_2 n\}$.

[7] Dies folgt aus Satz 2 : falls eine nicht-Baumkante $\{v,w\}$ nicht in den MST gehört, so ist sie maximal auf dem Zyklus, der durch die Kanten $\{v,w\}$ und $T(v,w)$ beschrieben wird. Damit ist sie auch maximal auf jedem Teil dieses Zyklus.

benötigten LCAs in $O(m)$ Zeit finden kann, z.B. [4].

Als nächstes erfolgt die Aufteilung des Spannbaums in die Teilbäume. Die Anzahl der Knoten in den Teilbäumen soll höchstens g sein, wobei g eine Größe logarithmisch zu n ist, für den genauen Wert sei jedoch auf [3] verwiesen. Um die Aufteilung nun durchzuführen werden die Knoten in Postorder (d.h. die Kinder eines Knotens vor dem Knoten) bearbeitet und jedem Knoten ein Wert s zugewiesen wie folgt : zuerst wird die Summe der s-Werte der Kinder des Knotens plus 1 berechnet. Dieser Wert wird das s-Label vom aktuellen Knoten, falls er höchstens g ist, ansonsten wird das s-Label 1 gesetzt und zusätzlich der Knoten als Teilbaumwurzel markiert. Abschließend wird noch die Wurzel r als Teilbaumwurzel markiert. Nach Konstruktion ergibt sich folgende Eigenschaft : nach Löschen der inzidenten Kanten der Teilbaumwurzeln zu den darunterliegenden Bäumen entstehen nur Bäume, die höchstens g Knoten haben.

Weiterhin wird nun ein sog. Macrotree konstruiert, dessen Knotenmenge die Teilbaumwurzeln sind und die Teilbaumwurzeln w ist im Macrotree der Vater von Teilbaumwurzel v, falls w der niedrigste Vorfahr von v im Baum ist, der eine Teilbaumwurzel ist (der Macrotree stellt also im Wesentlichen die Topologie der Teilbäume zueinander dar). Das Gewicht der Kanten im Makrotree wird erst später berechnet. Dieser Baum kann durch eine DFS konstruiert werden, indem ein Stack der Teilbaumwurzeln zusätzlich benutzt wird. Damit wird auch ein weiteres Knotenlabel p' definiert, wobei $p'(v)$ die Wurzel des Teilbaums ist, in dem v liegt ($p'(r)$ sei undefiniert).

Die nicht-Baumkanten werden jetzt ggf. nochmals vervielfältigt. Sei $\{u, v\}$ eine nicht-Baumkante (wobei v Nachkomme von u ist). Sei jetzt $r_u = p'(u)$, falls u keine Teilbaumwurzel ist, ansonsten sei $r_u = u$. Genauso sei $r_v = p'(v)$, falls v keine Teilbaumwurzel ist, ansonsten $r_v = v$. Falls nun $r_u = r_v$, so verläuft die Kante bereits im gleichen Teilbaum und wird nicht ersetzt. Andernfalls sei r diejenige Teilbaumwurzel, die Vorfahr von v im Baum ist und ein Kind von r_u im Macrotree. Die Berechnung von r erfolgt wiederum mittles DFS mit Teilbaumwurzel-Stack, wobei r der Knoten vor r_u auf dem Stack ist, wenn der aktuelle Knoten r_v ist. Nun wird die Kante $\{u, v\}$ erstzt durch die Kanten $\{u, r\}$, $\{r, r_v\}$ und $\{r_v, v\}$, wobei Kanten entfallen, die zu Schleifen werden. Wiederum erhalten die neuen Kanten das Gewicht der alten. Die Anzahl der nicht-Baumkanten wird dadurch höchstens verdreifacht, jedoch bleibt der minimale Spannbaum unverändert.

Als nächstes wird nun noch ein Knotenlabel $high$ berechnet, welches dem Maximum der Kantengewichte auf dem Pfad von der Teilbaumwurzel zu einem Knoten innerhalb dieses Teilbaums entspricht, also für einen Knoten v ist $high(v) := w_T(v, p'(v))$. Dies erfolgt wiederum mittles DFS in allen Teilbäumen wobei hier zusätzlich ein Stack mit Heap der Kantengewichte Verwendung findet.

Ein Teil der nicht-Baumkanten kann nun verifiziert werden. Zum einen die Kanten der Form $\{r_v, v\}$, d.h. nicht-Baumkanten, deren ein Endknoten eine Teilbaumwurzel ist und der andere Endknoten ein Knoten in diesem Teilbaum ist. Es muß hierzu lediglich geprüft werden, ob $w(r_v, v) \geq high(v)$ gilt für Satz 2.

Die anderen nicht-Baumkanten, die verifiziert werden können sind die der Form $\{r, r_v\}$, also Kanten deren Endknoten Teilbaumwurzeln sind. Diese Kanten werden zum Macrotree hinzugenommen. Die Kanten im Macrotree erhalten nun die Gewichte $high$, d.h. für eine Kante $\{x, y\}$ im Macrotree mit x Vater von y, ist $w(x, y) = high(y)$. Der Makrotree kann nun mit dem Algorithmus zur Verifikation von minimalen Spannbäumen von Tarjan ([2], siehe unten) berechnet werden, wobei der Zeitbedarf nur $O(m)$ ist, da der Makrotree logarithmisch kleiner ist als der ursprüngliche Baum, sodaß sich der Wert der α-Funktion sozusagen wegkürzt.

Die Kanten, die jetzt noch üebriggeblieben sind, habe die Form $\{u, r\}$, d.h. die Kanten zwischen zwei Knoten, die beide in genau einem Teilbaum liegen und beide nicht Wurzel dieses Teilbaums sind[8]. Nun wird aus dem Spannbaum eine Menge von Teilbäumen indem für jede Teilbaumwurzel die Kanten zu ihren Kinder gelöscht werden. Wie oben schon gezeigt, entstehen so Teilbäume, die höchstens g Knoten enthalten. Zu den Teilbäumen werden jetzt noch die zu verifizierenden Kanten der Form $\{u, r\}$ genommen, die dann eine Menge von Teilgraphen ergeben. Alle Teilgraphen zusammen beinhalten alle n Knoten und $O(m)$ Kanten.

Um nun die Teilgraphen mit ihren Spannbäumen zu verifizieren, wird erst eine Tabelle erstellt, die Entscheidungsbäume für Graphen mit höchstens g Kanten enthält, und mit dieser Tabelle werden dann die Teilgraphen abgearbeitet. Um die Tabelle erstellen zu können ist eine Codierung der Graphen mit höchstens g Knoten notwendig. Dazu werden die Knoten bei 1 beginnend durchnummeriert. Kanten werden kodiert als Konkatenation ihrer Endknotennummern. Die Darstellung eines Graphen ist dann die Konkatenation aller Kantenkodierungen, wobei die Spannbaumkanten den vorderen Teil der Kodierung ausmachen. Dabei wurde g so gewählt, daß es höchstens $\sqrt{n}$ Kodierungen gibt von Graphen mit höchstens g Knoten. Nun wird für alle Graphen mit höchstens g Knoten und ausgezeichnetem Spannbaum ein Entscheidungsbaum erstellt, mit dem bestimmt werden kann, ob der Spannbaum minimal ist. Nach [5] kann ein Entscheidungsbaum bei geeigneten g in Zeit $O(\sqrt{m})$ berechnet werden und alle Entscheidungsbäume in Zeit $O(m)$, wobei die Tiefe eines Entscheidungsbaumes linear abhängig ist von der Anzahl der Kanten im Graphen. Diese Entscheidungsbäume werden jetzt in einer Tabelle mit den Graphkodierungen als Indizes gespeichert. Danach werden die Teilgraphen, die aus dem ursprünglichen Graphen entstanden sind mittels dieser Tabelle verifiziert, indem der entsprechende Entscheidungsbaum durchlaufen wird. Die Zeit die dafür gebraucht wird ist auch $O(m)$ und die Verifikation des Spannbaums ist danach abgeschlossen.

---

[8] Um es nochmal zu verdeutlichen : r ist zwar Teilbaumwurzel, aber von dem Teilbaum, der an r am Teilbaum, in dem die Kante verläuft, darunter dranhängt.

12.4.2 Verifikation eines minimalen Spannbaums in $O(m\,\alpha(m,n))$ Zeit

Um einen Spannbaum T als minimalen Spannbaum zu verifizieren reicht es für alle nicht-Baumkanten Satz 2 zu überprf̈en. Dabei wird folgendermasen vorgegangen : Zuerst wird ein beliebiger Knoten als Wurzel ausgewählt damit LCAs definiert sind. Danach wird jede nicht-Baumkante der Form $\{v, w\}$ an ihrem $LCA(v, w)$ vermerkt. Danach werden alle Knoten des Baums durchlaufen und für jeden vermerkte nicht-Baumkante $\{v, w\}$ die Pfade $T(v, LCA(v, w))$ und $T(w, LCA(v, e))$ durchlaufen und dabei auf ihren Kanten das maximale Kantengewicht ermittelt. Für einen minimalen Spannbaum sollte der gefundene Wert höchstens so groß sein wie das Gewicht der nicht-Baumkante.

Um nun die Suche nach dem Kantengewichtsmaximum auf den Baumpfaden möglichst efiizient zu finden, wird eine zusätzliche Datenstruktur verwendet - ein balanzierter Baum mit Pfadkompression bzw. vielmehr ein Wald dieser Bäume. Der balanzierte Baum mit Pfadkompression stellt dabei die Pfade des ursprünglichen Baums in anderer Weise dar, speziell ausgerichtet auf die auszuführende Operation, in diesem Falle, der Maximumsbestimmung der Kantengewichte. Folgende Beobachtung ist dabei ausschlaggebend : wenn für einen Pfad bereits das Maximum bestimmt ist, so kann dieses Ergebnis genutzt werden, um für einen anderen Pfad, der den abgearbeiteten Pfad enthält (auch nur zum Teil), das Maximum zu bestimmen, ohne den gemeinsamen Teil nochmals durchlaufen zu müssen. Dabei wird zu Anfang jeder Knoten mit einem leeren Baum assoziiert. Dann werden die Knoten in Postorder (d.h. die Kinder eines Knoten vor diesem) abgearbeitet. Für jeden Knoten wird folgende Prozedur ausgeführt : Ist an dem Knoten eine nicht-Baumkante vermerkt, so wird das Maximum der beiden Pfade der Endknoten der Kante zu diesem Knoten bestimmt. Für jeden der beiden abzuarbeitenden Pfaden wird folgendes gemacht : falls der Endknoten der Kante kein Kind des aktuelen Knoten ist, wird der Pfad rekursiv abgearbeitet. Dazu wird für den Vaterknoten des Endknotens der Kante das Maximum auf dem Pfad zum aktuellen Knoten bestimmt, dann das Gewicht der verbindenden Kante zur Maximumsbestimmung mitgerechnet und der Endknoten wird zum Kind des aktuellen Knotens gemacht, wobei die verbindende Kante das ermittelte Maximum als Gewicht erhält. Durch die rekursive Anwendung werden so alle Knoten auf dem Pfad zu Kindern des aktuellen Knotens und so in gewisser Weise "hochgezogen" und der Pfad komprimiert, ohne daß dabei Information verlorengeht. Damit ergibt sich ein deutlich besseres Laufzeitverhalten als wenn Pfade immer erneut in voller Länge abgearbeitet werden müssten.

Nach der Bearbeitung aller vermerkten Kanten des Knotens, wird dieser mit dem Baum seines Vaterknotens durch eine Kante verbunden. Um nun wiederum besonders günstig dieses Verschmelzen der beiden Bäume auszunutzen, wird der kleinere Baum an den größeren drangehängt - daher balanzierter Baum. Da jedoch die binäre Operation *max*, die aus zwei Werten das Maximum ergibt, keine

Inverse hat, kann dies in diesem Fall nur mit viel Aufwand erreicht werden. Die Idee dabei ist allerdings dieselbe, und daher soll die konkrete Realisierung hier nicht beschrieben werden und es wird auf die Literatur [2] verwiesen. So wachsen schließlich alle zuerst leeren Bäume zu einem einzigen Baum zusammen, der den ursprünglichen Baum repräsentiert. Dadurch, daß bei der Komprimierung immer solange nach oben gegangen wird, bis die Wurzel des aktuellen Baums erreicht wird, ist es notwendig, daß der Baum bottom-up, also in Postorder abgearbeitet wird.

Literaturverzeichnis

[1] P.N.Klein, R.E.Tarjan : *A randomized liner-time algorithm for Finding Minimum Spanning Trees.* Proc. 26th Ann. ACM Symp. on Theory of Computing, 1994 , pp. 9 - 15.

[2] R.E.Tarjan : *Applications of path compression on balanced trees.* J. Assoc. Comput. Mach., 26 (1979), pp.690 -715.

[3] B.Dixon, M.Rauch, R.E.Tarjan : *Verification and sensitivity analysis of minimum spanning trees in linear time* , SIAM J. Comput., 21 (1992), pp. 1184-1192.

[4] B.Schieber, U. Vishkin : *On finding lowest common ancestors : simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253-1262.

[5] J. Komlos : *Linear verification for spanning trees*, Combinatorice, 5 (1985), pp. 57-65.

[6] M. Fredman, D.E. Willard : *Transdichotomous algorithms for minimum spanning trees and shortest paths*, Proc. 31st Annual IEEE Symp. on Foundations of Computer Science, IEEE IEEE Computer Society Press, Los Alamitos, CA, 1990, pp.719-725.

12.5 Anhang : Veranschaulichung einer Phase des Algorithmus anhand eines Beispielgraphen

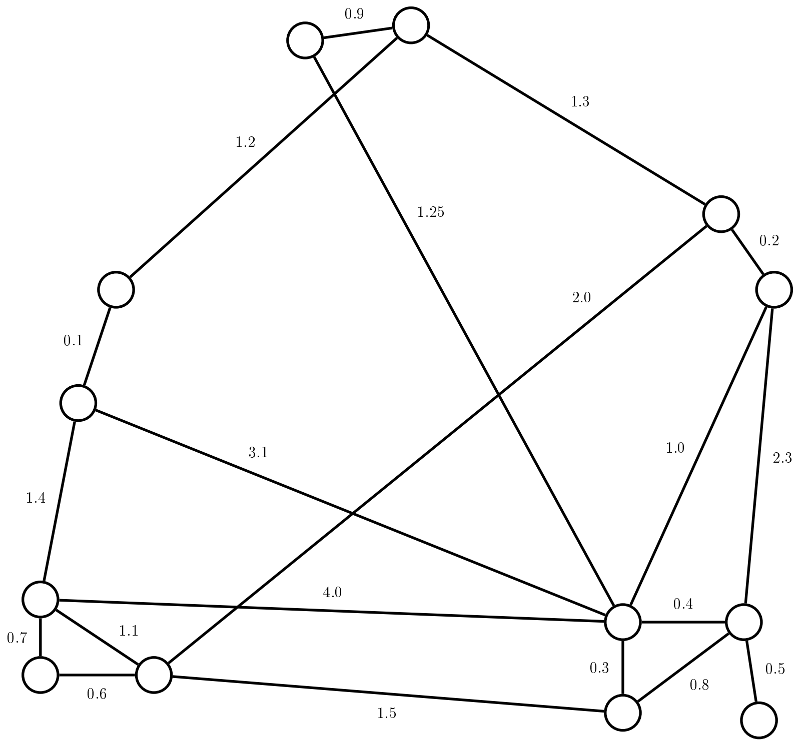


Abbildung 12.1: Ausgangsgraph

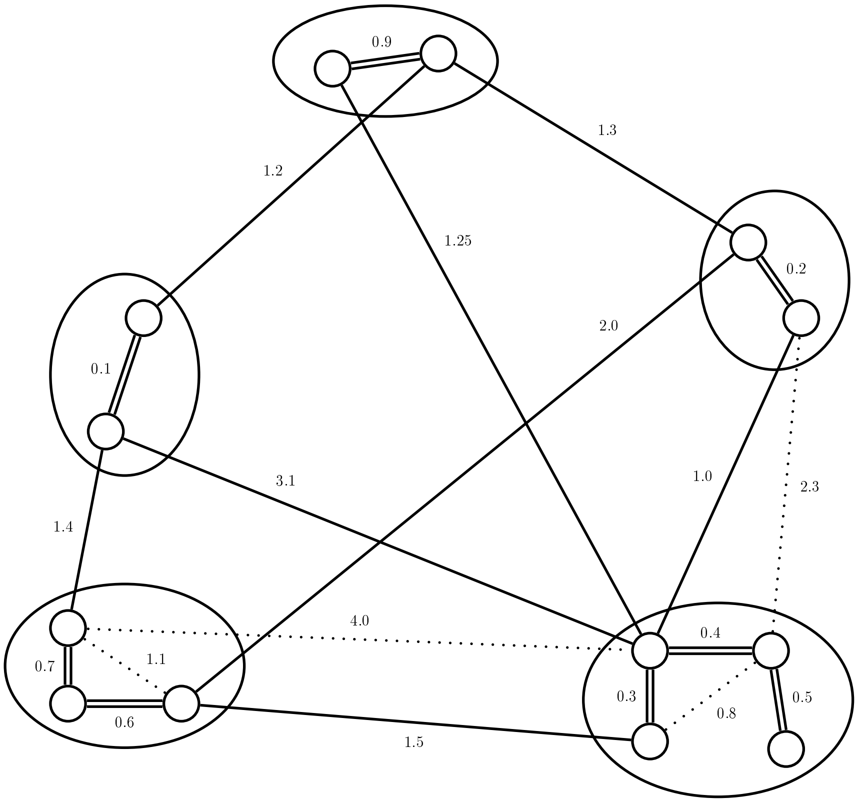


Abbildung 12.2: Finden der inzidenten Kante mit minimalen Gewicht für jeden Knoten und Elimination von Kanten innerhalb Zusammenhangskomponenten (Schritt 1).

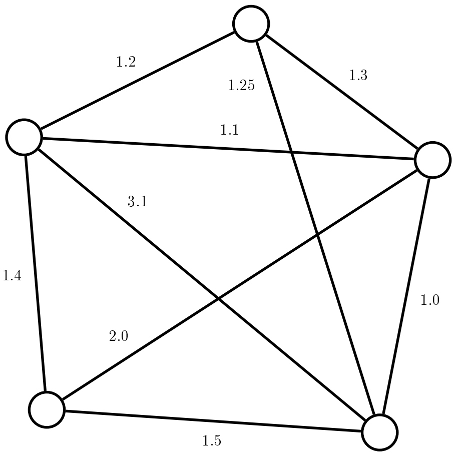


Abbildung 12.3: Verschmelzen der Zusammenhangskomponenten zu Superknoten verkleinert den Graphen (Schritt 1)

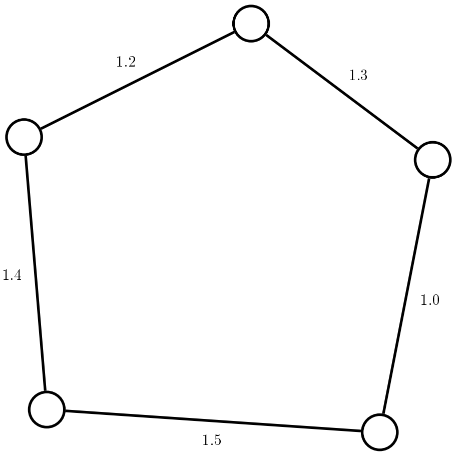


Abbildung 12.4: Erstellen eines Teilgraphen H durch zufälliges Auswählen von Kanten.

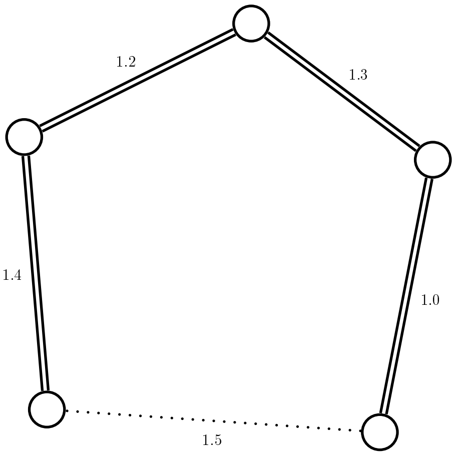


Abbildung 12.5: Ergebnis Spannwald F_H des Algorithmus auf Teilgraph H

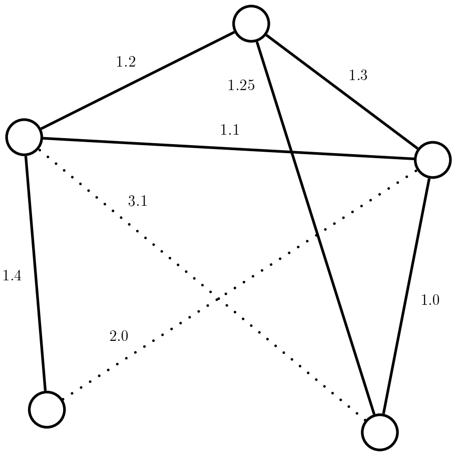


Abbildung 12.6: Löschen der F_H-heavy Kanten im ganzen Graphen.

Vortrag 13

Parallele und randomisierte Algorithmen zum Problem der maximal unabhängigen Menge

Thomas Schickinger
Technische Universität München
schickin@informatik.tu-muenchen.de

Zusammenfassung

In diesem Artikel werden vier parallele Algorithmen zum Maximal Independent Set (MIS) Problem vorgestellt. Drei dieser Algorithmen benötigen Zufallsbits, während der vierte deterministisch arbeitet. Besonders bemerkenswert ist, daß dieser Algorithmus systematisch aus den randomisierten Algorithmen abgeleitet wird. Das hierbei zur Anwendung kommende Verfahren kann allgemein auf randomisierte Algorithmen mit einer bestimmten Struktur angewendet werden und steht deshalb im Zentrum dieses Artikels.

Dieser Artikel faßt den Inhalt eines Vortrages zusammen, im Rahmen von Kurs 2 „Parallele und randomisierte Algorithmen“ auf der Ferienakademie ’95 gehalten wurde, und beruht auf einer Arbeit vom M. Luby [Lu].

13.1 Problemstellung

Beim Maximal Independet Set (MIS) Problem geht man von einem ungerichteten Graph (V, E) aus und sucht eine Menge von Knoten $V' \subset V$ mit den Eigenschaften

- *unabhängig:* Es gibt keine benachbarten Knoten in V', d. h. $\neg(\exists i, j \in V' : (i, j) \in E)$
- *maximal:* V' kann um keinen Knoten aus $V \backslash V'$ erweitert werden, ohne die Unabhängigkeit zu verlieren, d. h. $\forall i \in V \backslash V' \; \exists j \in V' : (i, j) \in E$

Offensichtlich besitzt dieses Problem im Normalfall keine eindeutige Lösung. Wenn also im Folgenden von „der Lösung" die Rede sein wird, so ist damit immer nur die jeweils vom Algorithmus produzierte Lösung gemeint. Abbildung 13.1 zeigt zwei Beispiele dazu.

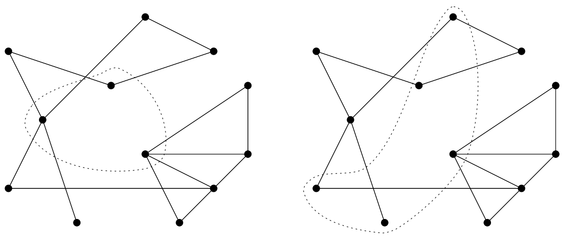


Abbildung 13.1: Zwei Beispiele für unabhängige Mengen

Für das MIS-Problem läßt sich problemlos ein sequentieller Algorithmus angeben, der $O(n \log n)$ Zeit benötigt. Hierbei wird davon ausgegangen, daß die Kanten in einer Datenstruktur mit logarithmischer Zugriffszeit abgespeichert werden.

```
$I \leftarrow \emptyset$
for $i = 1, \ldots, n$ do
    if $\nexists j \in I : (i, j) \in E$ then
        $I \leftarrow I \cup \{i\}$
end
```

Dieser Algorithmus ist inhärent sequentiell, da in jedem Durchlauf der **for**-Schleife die Ergebnisse aller vorherigen Durchläufe benutzt werden. Deshalb war lange zweifelhaft, ob überhaupt ein effektiver paralleler Algorithmus existiert.

Alg.	*CPUs*	*Zeit (auf EREW)*	*Zufallsbits*
A	$O(m)$	$EO((\log n)^2)$	$EO(n(\log n)^2)$
B	$O(m)$	$EO((\log n)^2)$	$EO(n \log n)$ (?)
C	$O(m)$	$EO((\log n)^2)$	$EO((\log n)^2)$
D	$O(n^2 \cdot m)$	$O((\log n)^2)$	keine

Tabelle 13.1: Leistungsdaten der vorgestellten Algorithmen ($n = |V|, m = |E|$)

13.2 Inhaltsübersicht

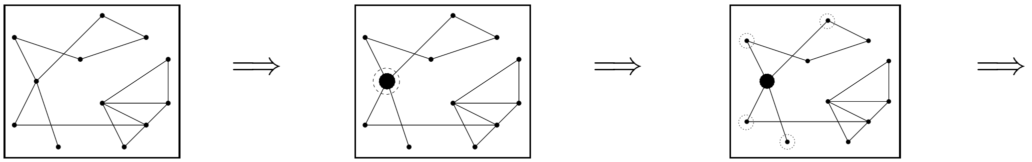
In diesem Artikel werden nun vier Algorithmen zur Lösung MIS-Problems vorgestellt, welche die in Tabelle 13.1 angegebenen Leistungsdaten besitzen.
Die Algorithmen B, C und D sind eng miteinander verwandt, während Algorithmus A nach einem etwas anderen Grundprinzip arbeitet.
Die determistische Variante D wird mittels einer *allgemeinen Technik* zur Transformation von randomisierten in *deterministische* Algorithmen aus Variante B abgeleitet. Dabei entsteht zusätzlich Variante C, die weniger Zufallsbits als B benötigt.

13.3 Schema der Monte Carlo MIS Algorithmen

Alle vorgestellten Algorithmen haben dieselbe Grundstruktur und unterscheiden sich nur in einem einzelnen Punkt. Zunächst soll nun diese Struktur vorgestellt werden.

13.3.1 Informelle Beschreibung und exemplarischer Ablauf

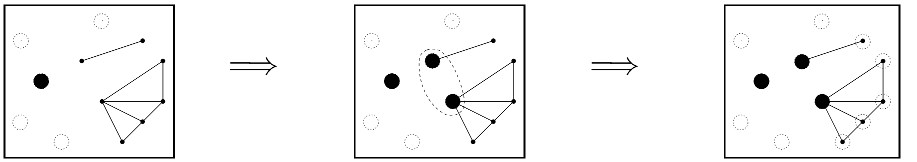
Die Lösung wird nach und nach in einer Menge I angesammelt, d. h. I wird in mehreren Iterationsschritten um neue Elemente erweitert, bis die Maximalität erfüllt ist. I wird anfangs mit der leeren Menge initialisiert. Während der Iterationen muß als Invariante gelten, daß I unabhängig ist.
In jedem Iterationsschritt wird zunächst eine Menge von untereinander unabhängigen Knoten I' gewählt. Diese Knoten werden zu I hinzugefügt und mit ihren Nachbarn aus dem Graphen entfernt. Durch wiederholte Anwendung des Algorithmus wird der Graph solange verkleinert, bis keine Knoten mehr übrig sind. I enthält dann eine maximal unabhängige Menge. Abbildung 13.2 zeigt einen exemplarischen Ablauf.



Ausgangsgraph

I' wählen

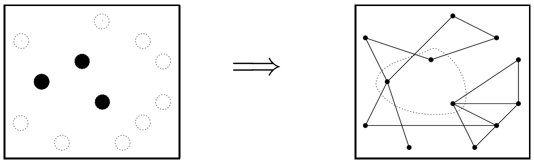
Nachbarn suchen ...



... und entfernen

Wieder: I' wählen

Nachbarn suchen ...



... und entfernen

Endergebnis

Abbildung 13.2: Exemplarischer Ablauf

13.3.2 Formale Beschreibung

Mit folgender Abkürzung für die Nachbarmenge einer Menge von Knoten

$$\operatorname{adj}(W) = \{i \in V' : \exists j \in W : (i,j) \in E'\}$$

kann man das allgemeine Schema der MIS-Algorithmen so schreiben:

```
begin
$I \leftarrow \emptyset$
$G' = (V', E') \leftarrow G = (V, E)$
while $G' \neq \emptyset$ do
    begin
        select $I' \subseteq V'$, so daß $I'$ unabhängig in $G'$
        $I \leftarrow I \cup I'$
        $Y \leftarrow I' \cup \operatorname{adj}(I')$
        $G' = (V', E')$ ist der durch $V' \backslash Y$ induzierte Teilgraph
    end
end
```

Die Korrektheit des Schemas kann man sich mit den folgenden Invarianten für die **while**-Schleife klar machen:

- I ist unabhängig in G.
- $\forall i \in I\ \forall j \in V' : (i,j) \notin E$, d. h. es gibt keinen Knoten in I der mit einem Knoten aus V' benachbart ist.
- $\forall j \in E \backslash E'\ \exists i \in I : (i,j) \in E$, d. h. jeder Knoten in $E \backslash E'$ hat mindestens einen Nachbarn in I.

Nach Ablauf des Algorithmus ist E' leer. Daraus folgt unmittelbar die Korrektheit.
Alle vorgestellten Algorithmen folgen dem obigen Schema. Die Unterschiede liegen nur im *select*-Schritt, d. h. in der Wahl von I'. Dafür werden in den nächsten Absätzen zwei verschiedene Lösungsmöglichkeiten vorgestellt.
Offensichtlich benötigt der Rumpf der **while**-Schleife ohne den den **select**-Schritt auf einer CRCW-PRAM nur $O(1)$ Zeit, bzw. auf einer EREW-PRAM $O(\log n)$ (Vgl. Vortrag von Ralf Werner zur Simulation der CRCW-PRAM auf einer EREW-PRAM). Um die in der Einleitung angegebenen Laufzeiten von $O((\log n)^2)$ zu beweisen, ist es also erforderlich, eine Implementierung des *select*-Schritts anzugeben, die genauso schnell läuft, wie der Rest der **while**-Schleife, und die dazu führt, das V' nach $O(\log n)$ Schleifendurchläufen auf den leeren Graphen reduziert ist.

13.4 Algorithmus A

13.4.1 Implementierung des *select*-Schritts

O. B. d. A. sei $V' = \{1, \ldots, k\}$, mit $k \leq n$. Der *select*-Schritt wird nun folgendermaßen durchgeführt:

1. Wähle eine zufällige Permutation π von V'
2. $I' \leftarrow \{i \in V' : \pi(i) < min\{\pi(j) : j \in \text{adj}(i)\}\}$

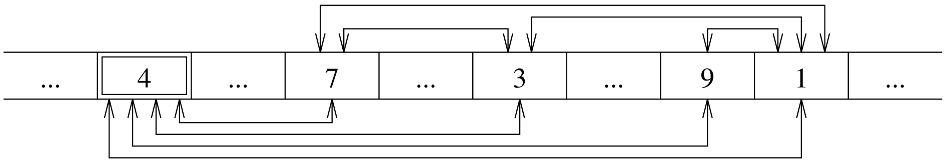


Abbildung 13.3: Beispiel für Permutation der Knoten mit „4“ $\in I'$

Korrektheit: I' ist unabhängig, denn:

Angenommen: $\exists i,j \in I'$ mit $(i,j) \in E'$, wobei o. B. d. A gilt $\pi(j) \leq \pi(i)$, dann folgt:
$j \in \mathrm{adj}(i) \implies \pi(i) < \pi(j)$ (nach Def. von I') $\implies$ Widerspruch

Der gesamte *select*-Schritt kann so notiert werden:

```
ALGVERTEX(i)
begin
    $\pi(i) \leftarrow$ Zufallszahl aus $\{1,\dots,n^4\}$
end

ALGEDGE(i,j)
begin
    if $\pi(i) \geq \pi(j)$ then $I' \leftarrow I' \setminus \{i\}$
    else $I' \leftarrow I' \setminus \{j\}$
end

begin
    In parallel , $\forall i \in V'$, execute ALGVERTEX(i)
    $I' \leftarrow V'$
    In parallel , $\forall (i,j) \in E'$, execute ALGEDGE(i,j)
end
```

Achtung: Diese Implementierung weicht leicht von dem oben angegebenen Vorgehen ab, denn es wird keine echte Permutation von V' erzeugt. Erstens gilt $\pi(i) \in \{1,\dots,n^4\}$ statt $\pi(i) \in \{1,\dots,k\}$, $(k \leq n)$. Dies ist aber unerheblich, da es nur auf die *Anordnung* der $\pi(i)$ ankommt. Zweitens kann der Fall $\pi(i) = \pi(j)$ auftreten. Damit ist aber nur mit einer Wahrscheinlichkeit von höchstens $1-1/2n^2$ zu rechnen. Es gibt nämlich höchstens $\binom{n}{2}$ Paare von Knoten in V'. Für jedes Paar $(i,j) \in E'$ ist die Wahrscheinlichkeit für $\pi(i) = \pi(j)$ gleich $1/n^4$. Also:

$$\Pr[\text{„korrekte Permutation“}] \geq 1 - \binom{n}{2} \cdot \frac{1}{n^4} = 1 - \frac{n(n-1)}{2n^4} \geq 1 - \frac{1}{2n^2}$$

Bei jeder Ausführung des *select*-Schritts werden $O(n)$ Zufallsvariablen mit einem Wertebereich der Mächtigkeit n^4 benötigt. Dies ergibt einen Aufwand von $O(n \log n)$ Zufallsbits pro Durchlauf und $O(n(\log n)^2)$ insgesamt.

13.4.2 Mathematische Analyse

Analog zu Algorithmus B (siehe dort) kann man für Algorithmus A zeigen (Y_k^A sei die Anzahl der Kanten in E' vor dem k-ten Durchlauf der **while**-Schleife):

$$E[Y_k^A - Y_{k-1}^A] \geq \frac{1}{8} \cdot Y_k^A - \frac{1}{16}$$

Der Beweis verläuft ähnlich wie für Algorithmus B allerdings mit folgendem

Lemma A *Bei Algorithmus A gilt* $\forall i \in V'$ *mit* $d(i) \geq 1$*:*

$$\Pr[i \in \mathrm{adj}(I')] \geq \frac{1}{4} \cdot \min\{\mathrm{sum}(i), 1\} \cdot \left(1 - \frac{1}{2n^2}\right)$$

statt Lemma B. Auf eine Durchführung des Beweises wird hier verzichtet.

13.5 Algorithmus B

13.5.1 Implementierung des *select*-Schritts

Die Implementierung des *select*-Schritts von Algorithmus B benützt die Anzahl der Nachbarn eines Knotens, d. h. seinen Grad (engl.: degree):

$$d(i) := |adj(i)|$$

```
begin
    In parallel , $\forall i \in V'$, compute $d(i)$
    $X \leftarrow \emptyset$
    In parallel , $\forall i \in V'$
        Wähle Zufallsvariable $coin(i) \in \{0,1\}$ mit $\Pr[coin(i) = 1] = 1/2d(i)$
        if $coin(i) = 1$ then $X \leftarrow X \cup \{i\}$
    $I' \leftarrow X$
    In parallel , $\forall (i,j) \in E'$ (*)
        if $i \in X$ and $j \in X$ then
            if $d(i) \leq d(j)$ then $I' \leftarrow I' \setminus \{i\}$
            else $I' \leftarrow I' \setminus \{j\}$
end
```

Bei jedem Durchlauf der **while**-Schleife müssen parallel $O(n)$ Zufallsvariablen mit dem Wertebereich $\{0,1\}$ berechnet werden. Dies ergibt einen Aufwand von $O(n)$ pro Durchlauf, falls diese Zufallsvariablen direkt zur Verfügung stehen, und führt zu einem Gesamtaufwand von $O(n \log n)$, wie er auch in [Lu] zu finden ist. Da $coin(i)$ die Werte 0 und 1 nicht gleichwahrscheinlich annimmt, müßten diese allerdings vermutlich durch gleichverteilte Zufallszahlen mit $O(\log n)$ Bits simuliert werden. Dann ist der Aufwand pro Durchlauf wie bei Algorithmus A gleich $O(n \log n)$.

13.5.2 Informelle Begründung der Effizienz von Algorithmus B

Bei der Durchführung des *select*-Schritts sollen pro Durchlauf der **while**-Schleife möglichst viele Knoten entfernt werden, um V' schnell auf den leeren Graphen zu

reduzieren. D. h. wir wollen möglichst *viele Knoten* mit möglichst *vielen Nachbarn* in I' aufnehmen.
Algorithmus B erreicht dies durch folgende Maßnahmen:

- In (*) (siehe obiges Listing) wird der Knoten mit der kleineren Anzahl von Nachbarn entfernt.
- Die Verteilung der Zuvallsvariablen *coin* mit „$\Pr[coin(i) = 1] = 1/2d(i)$“ ist so gewählt, daß nur wenige Knoten mit vielen Nachbarn werden in X aufgenommen werden.

Der Sinn der zweiten Maßnahme ist im Gegensatz zur ersten nicht sofort einsichtig. Wollen wir denn nicht gerade Knoten mit vielen Nachbarn in I' aufnehmen? Man kann sich aber plausibel machen, daß eine Aufnahme vieler solcher Knoten in die Vorauswahl X dieses Ziel nicht erreicht.
Angenommen, X enthielte viele solcher Knoten. Dann wäre die Wahrscheinlichkeit sehr groß, daß etliche dieser Knoten untereinander benachbart wären. Im nächsten Schritt würden somit viele dieser Knoten wieder entfernt. Da diese Knoten viele Nachbarn haben, würden sie auch zur Entfernung vieler anderer Knoten mit weniger Verbindungen führen. Zuviele Knoten mit hohem Grad richten also mehr Schaden an, als sie Nutzen bringen (siehe dazu Abbildung 13.4).
Genaueres wird später eine mathematische Analyse zeigen.

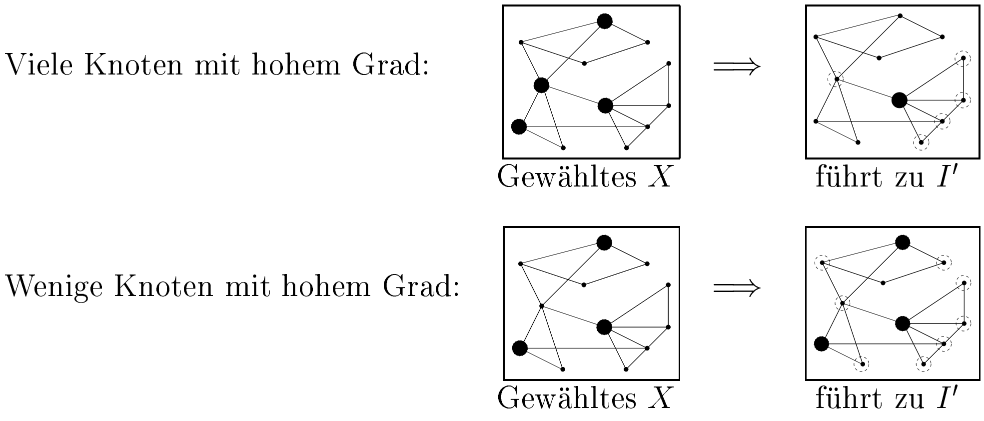


Abbildung 13.4: Erklärung für die Effizienz von Alg. B (2)

13.5.3 Mathematische Analyse von Algorithmus B

Technisches Lemma

Folgende Aussage wird zur Untersuchung der Laufzeit der Algorithmen A und B benötigt:

Technisches Lemma *Seien* $p_1 \geq \ldots \geq p_n \geq 0$ *reelle Variablen. Für* $1 \leq l \leq n$ *und konstantem* $c > 0$ *sei*

$$\alpha_l = \sum_{j=1}^{l} p_j, \quad \beta_l = \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k, \quad \gamma_l = \alpha_l - c \cdot \beta_l$$

Dann gilt:

$$\max\{\gamma_l : 1 \leq l \leq n\} \geq \frac{1}{2} \cdot \min\left\{\alpha_n, \frac{1}{c}\right\}$$

Beweis:

Durch Induktion kann man zeigen:

$$\beta_l \text{ maximal für } p_1 = \ldots = p_n = \frac{\alpha_l}{l}$$

Damit gilt:

$$\beta_l = \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k = \frac{l(l-1)\alpha_l^2}{2l^2} \implies \gamma_l \geq \alpha_l \cdot \left(1 - c \cdot \alpha_l \cdot \frac{(l-1)}{2l}\right)$$

Falls $\alpha_n \leq \frac{1}{c}$, so $\gamma_n \geq \frac{\alpha_n}{2}$. Falls $\alpha_1 \geq \frac{1}{c}$, so $\gamma_1 \geq \frac{1}{c}$.
In allen anderen Fällen gilt:

$$\exists l : 1 < l < n, \quad \text{so daß} \quad \alpha_{l-1} \leq \frac{1}{c} \leq \alpha_l \overset{(*)}{\leq} \frac{1}{c} \cdot \frac{l}{l-1} \implies \gamma_l \geq \frac{1}{2c}$$

Die Ungleichung (*) folgt aus $p_1 \geq \ldots \geq p_n$.

Induktionsbeweis für: β_l maximal für $p_1 = \ldots = p_n = \alpha_l / l$.

Induktionsanfang ($n = 2$):

$$\begin{aligned} & \beta_2 = p_1 \cdot p_2 = (\alpha_2 - p_2) \cdot p_2 \\ \implies \quad & \text{Maximal für } p_2 = \alpha_2/2 \text{ (Scheitel der Parabel!)} \end{aligned}$$

Induktionsschluß ($n \to n+1$):

$$\begin{aligned} \beta_{l+1} &= \beta_l + p_{l+1}\alpha_l \overset{IV}{\leq} \alpha_l^2 \tfrac{l-1}{2l} + p_{l+1}\alpha_l \\ &= (\alpha_{l+1} - p_{l+1})^2 \cdot \tfrac{l-1}{2l} + p_{l+1}(\alpha_{l+1} - p_{l+1}) =: h(p_{l+1}) \\ h'(p_{l+1}) &= (\alpha_{l+1} - p_{l+1})\tfrac{1-l}{l} + \alpha_{l+1} - 2p_{l+1} \overset{!}{=} 0 \\ \iff \quad p_{l+1} &= \tfrac{\alpha_{l+1}}{l+1} \end{aligned}$$

h ist eine nach unten geöffnete Parabel $\implies$ Maximum!

Dieser Beweisschritt kann statt durch Induktion auch mittels der **Methode der Lagrange-Multiplikatoren** erfolgen (s. [Br], [Gr]).

Die Lagrangefunktion lautet:

$$L(p_1,\ldots,p_n,\alpha_l) = \sum_{j=1}^{l}\sum_{k=j+1}^{l} p_j \cdot p_k + \lambda \cdot \left(\alpha_l - \sum_{j=1}^{l} p_j\right)$$

$$\frac{\partial L}{\partial p_i} = \sum_{\substack{j=1\\ j\neq i}}^{l} p_j - \lambda \overset{!}{=} 0 \iff \lambda = \alpha_l - p_i \iff p_i = \lambda + \alpha_l$$

$$\Longrightarrow \alpha_l - l\cdot\lambda - l\cdot\alpha_l = 0 \Longrightarrow \lambda = \frac{\alpha_l}{l} - \alpha_l \Longrightarrow p_i = \frac{\alpha_l}{l}$$

Nachweis von Ungleichung (*):
Ang.: $\exists l : \alpha_{l-1} \leq \frac{1}{c} \wedge \alpha_l > \frac{1}{c}\cdot\frac{l}{l-1}$

$$\begin{aligned} & p_l = \alpha_l - \alpha_{l-1} > \frac{1}{c}\cdot\frac{l}{l-1} - \frac{1}{c} = \frac{1}{c}\cdot\frac{1}{l-1} \\ \Longrightarrow\quad & p_1,\ldots,p_{l-1} > \frac{1}{c}\cdot\frac{1}{l-1} \Longrightarrow \alpha_{l-1} > (l-1)\cdot\frac{1}{c}\cdot\frac{1}{l-1} > \frac{1}{c} \\ \Longrightarrow\quad & \text{Widerspruch} \end{aligned}$$

Untersuchung der Laufzeit

Mit

Theorem 1 *Sei Y_k^B die Anzahl der Kanten in E' vor dem k-ten Durchlauf der* **while**-*Schleife in Algorithmus B.*

$$E[Y_k^B - Y_{k-1}^B] \geq \frac{1}{8}\cdot Y_k^B$$

folgt sofort: Die Anzahl der Durchläufe der **while**-Schleife ist von der Ordnung $EO(\log n)$.

Man kann sogar zeigen (ohne Beweis):

$$\Pr[Y_{k+1}^B \leq \frac{15}{16}Y_k^B] \geq \frac{1}{15} \Longrightarrow \Pr[t_B \geq 700\log n] \leq 2n^{-2000}$$

Die Laufzeit bleibt also mit hoher Ws logarithmisch.

Zum Beweis von Theorem 1 benötigt man

Lemma B *Bei Algorithmus B gilt $\forall i \in V'$ mit $d(i) \geq 1$:*

$$\Pr[i \in \mathrm{adj}(I')] \geq \frac{1}{4}\cdot\min\left\{\frac{1}{2}\mathrm{sum}(i), 1\right\}$$

Dabei sei

$$\mathrm{sum}(i) = \sum_{j \in adj(i)} \frac{1}{d(j)}$$

Zum Beweis von Lemma B dienen folgende **Definitionen**:

- $\forall j \in V'$ sei E_j das Ereignis „$coin(j) = 1$“ mit $p_j = \Pr[E_j] = 1/2d(j)$.
- O. B. d. A. sei $\mathrm{adj}(i) = \{1, \ldots, d(i)\}$ und $p_1 \geq \ldots \geq p_{d(i)}$
- Die Ereignisse E_i' und A_i seien definiert durch

$$E_1' \equiv E_1 \quad ; \quad \forall j : 2 \leq j \leq d(i) : E_j' = \left(\bigcap_{k=1}^{j-1} \neg E_k \right) \cap E_j$$

$$A_j = \bigcap_{\substack{v \in adj(j) \\ d(v) \geq d(j)}} \neg E_v$$

„Alle Knoten, die zur Entfernung von j aus I' führen, sind nicht in X“

Nun zum eigentlichen **Beweis**:

$$\Pr[i \in \mathrm{adj}(I')] \geq \sum_{j=1}^{d(i)} \Pr[E_j'] \cdot \Pr[A_j | E_j'] \geq \sum_{j=1}^{d(i)} \Pr[E_j'] \cdot \frac{1}{2}$$

denn (Unabhängigkeit der E_i wird ausgenützt!):

$$\Pr[A_j | E_j'] \overset{U}{=} \Pr[A_j] \geq 1 - \sum_{\substack{v \in adj(j) \\ d(v) \geq d(j)}} p_v \geq \frac{1}{2}$$

Weitere Abschätzung der Summe ($\forall i, j : E_i' \cap E_j' = \emptyset$):

$$\sum_{j=1}^{d(i)} \Pr[E_j'] = Pr\left[\bigcup_{j=1}^{d(i)} E_j' \right] = Pr\left[\bigcup_{j=1}^{d(i)} E_j \right]$$

Für $k \neq j$ gilt $\Pr[E_j \cap E_k] = p_j \cdot p_k$. Damit folgt für $1 \leq l \leq d(i)$

$$Pr\left[\bigcup_{j=1}^{d(i)} E_j \right] \geq Pr\left[\bigcup_{j=1}^{l} E_j \right] \geq \sum_{j=1}^{l} p_j - \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k$$

Sei nun $\alpha = \sum_{j=1}^{d(i)} p_j$. Dann folgt mit dem technischen Lemma

$$Pr\left[\bigcup_{j=1}^{d(i)} E_j \right] \geq \frac{1}{2} \cdot \min\{\alpha, 1\} \geq \frac{1}{2} \cdot \min\left\{ \frac{1}{2}\mathrm{sum}(i), 1 \right\}$$

und daraus dann die Behauptung.

Damit kann nun der **Beweis von Theorem 1** durchgeführt werden:

Sei $G' = (V', E')$ der Graph vor dem k-ten Durchlauf der **while**-Schleife. In diesem Durchlauf werden alle Knoten mit mindestens einem Endpunkt in $I' \cup \mathrm{adj}(I')$ eliminiert.
Also:

$$\begin{aligned} E[Y_k^B - Y_{k+1}^B] &\geq \frac{1}{2} \cdot \sum_{i \in V'} d(i) \cdot \Pr[i \in I' \cup \mathrm{adj}(I')] \\ &\geq \frac{1}{2} \cdot \sum_{i \in V'} d(i) \cdot \Pr[i \in \mathrm{adj}(I')] \end{aligned}$$

Daraus folgt mit Lemma B:

$$\begin{aligned} E[Y_k^B - Y_{k+1}^B] &\geq \frac{1}{8} \cdot \left(\frac{1}{2} \cdot \sum_{\substack{i \in V' \\ sum(i) \leq 2}} d(i) \cdot sum(i) + \sum_{\substack{i \in V' \\ sum(i) > 2}} d(i) \right) \\ &\geq \frac{1}{8} \cdot \left(\sum_{\substack{i \in V' \\ sum(i) \leq 2}} \sum_{j \in adj(i)} \frac{d(i)}{2 \cdot d(j)} + \sum_{\substack{i \in V' \\ sum(i) > 2}} \sum_{j \in adj(i)} 1 \right) \\ &\geq \frac{1}{8} \cdot \left(\sum_{\substack{(i,j) \in E' \\ sum(i) \leq 2 \\ sum(j) \leq 2}} \frac{1}{2} \cdot \left(\frac{d(i)}{d(j)} + \frac{d(j)}{d(i)} \right) + \right. \\ &\quad \left. + \sum_{\substack{(i,j) \in E' \\ sum(i) \leq 2 \\ sum(j) > 2}} \left(\frac{d(i)}{2 \cdot d(j)} + 1 \right) + \sum_{\substack{(i,j) \in E' \\ sum(i) > 2 \\ sum(j) > 2}} 2 \right) \geq \frac{1}{8} \cdot |E'| = \underline{\underline{\frac{1}{8} \cdot Y_k^B}} \end{aligned}$$

Bem.: $f(x) = x + 1/x$ hat Minimum bei 1, denn $f'(x) = 1 - 1/x^2 \overset{!}{=} 0 \iff x^2 = 1$ (Steigungsverhalten klar)

13.6 Ein allgemeines Verfahren zur Derandomisierung

Am Beispiel von Algorithmus B soll nun ein allgemein verwendbares Verfahren zur Derandomisierung von parallelen randomisierten Algorithmen vorgestellt werden.

13.6.1 Transformierbare Algorithmen

Damit die Derandomisierung durchgeführt werden kann, muß der betreffende Algorithmus folgende Eigenschaften haben:

- Die Randomisierung ist konzentriert in einen *choice step*, der aber auch mehrmals ausgeführt werden darf.
- Im *choice step* werden Werte für *unabhängige* Zufallsvariablen $X_0, \ldots, X_{n-1}$ erzeugt, die im Mittel *gut* sind.
- *Gute* Werte führen in begrenzter Zeit zu einem korrekten Ergebnis.
- Es kann „schnell" festgestellt werden, ob die Werte von $X_0, \ldots, X_{n-1}$ *gut* sind oder nicht.

13.6.2 Modifikation des Algorithmus

Wenn ein Algorithmus die obigen Eigenschaften hat, dann besteht die Aussicht, ihn so modifizieren zu können, daß er auch noch die folgenden Eigenschaften erfüllt:

1. Man findet $r \in N$ und eine Primzahl $q \geq n$, wobei r und q polynomial von n abhängen, mit denen man die Zufallsvariablen $X_0, \ldots, X_{n-1}$ so modifiziert, daß
    - $X_i \in R = \{R_1, \ldots, R_r\}$
    - $\Pr[X_i = R_j] = n_{ij}/q$ mit $n_{ij} \in N_0$ und $\sum_{j=1}^{r} n_{ij} = q$.
2. Auch wenn $X_0, \ldots, X_{n-1}$ nur *paarweise unabhängig* sind, ist die Wahrscheinlichkeit für *gute* Werte noch positiv.

13.6.3 Simulation des *choice steps*

Der deterministische Algorithmus stimmt bis auf den *choice step* mit dem randomisierten Algorithmus überein. Dieser wird folgendermaßen (deterministisch!) simuliert:

1. Erzeuge einen W-Raum mit q^2 Elementen, in dem $X_0, \ldots, X_{n-1}$ *paarweise* unabhängig sind.
2. Durchsuche parallel alle q^2 Belegungen nach *guten* Belegungen und benutze eine davon als Ergebnis des *choice steps*.

Aus Bedingung 2 in 13.6.2 folgt, daß man sicher eine *gute* Belegung in den q^2 Möglichkeiten findet.

13.6.4 Konstruktion des Wahrscheinlichkeitsraums

Vorgaben für $X_0, \ldots, X_{n-1}$ (zur Erinnerung):

- $X_i \in R = \{R_1, \ldots, R_r\}$
- $\Pr[X_i = R_j] = n_{ij}/q$ mit $n_{ij} \in N_0$ und $\sum_{j=1}^{r} n_{ij} = q$.

Für *paarweise* unabhängige $X_0, \ldots, X_{n-1}$, welche diese Vorgaben erfüllen (polynomial beschränkter Wertebereich, Wahrscheinlichkeiten lassen sich als Bruchteile einer Primzahl q angeben, die ebenfalls polynomial beschränkt ist) kann ein Wahrscheinlichkeitsraum (kurz: W-Raum) mit q^2 (gleichwahrscheinlichen) Elementen erzeugt werden (Zum Vergleich: Die Mächtigkeit eines W-Raums von n *vollständig* unabh. Var. beträgt r^n.)

Dazu konstruiert man eine $n \times q$ Matrix A, deren i-te Zeile die Werte für X_i enthält, d. h. genau n_{ij} Einträge sind gleich R_j (siehe Abbildung 13.5).
Diese Matrix benutzt man, um eine Belegung $b^{x,y}$ von $X_0, \ldots, X_{n-1}$ finden. Für $0 \leq x, y \leq q-1$ sei $b^{x,y} = (b_0, \ldots, b_{n-1})$ definiert durch $b_i = A_{i,(x+y\cdot i) \bmod q}$. Jede Belegung $b^{x,y}$ habe die Wahrscheinlichkeit $1/q^2$.

$$\begin{array}{r|ccc|l} & & q \text{ Spalten} & & \\ & \vdots & \vdots & \vdots & \\ n \text{ Zeilen} & \underbrace{R_1 \cdots R_1}_{n_{i1}\text{-mal}} & \cdots & \underbrace{R_n \cdots R_n}_{n_{in}\text{-mal}} & \leftarrow \; i\text{-te Zeile enthält Werte für } X_i \\ & \vdots & \vdots & \vdots & \end{array}$$

Abbildung 13.5: Matrix zur Konstruktion eines paarweise unabh. W-Raums

Die **Korrektheit** dieser Konstruktion sieht man so ein:

- $\Pr[X_i = R_j] = n_{ij}/q$, denn:
  Für festes l gibt es genau q Paare (x, y) mit $(x+y\cdot i) \bmod q = l$. Es existieren n_{ij} Werte von l, so daß $A_{i,j} = R_j$. (Die Restklassen modulo q bilden einen Körper, da q eine Primzahl ist!)
- $\Pr[X_i = R_j \wedge X_{i'} = R_{j'}] = (n_{ij} \cdot n_{i'j'}/q^2)$, denn:
  Für festes l und l' gibt es genau ein Paar (x, y), so daß $(x + y \cdot i) \bmod q = l$ und zugleich $(x + y \cdot i') \bmod q = l'$.

13.6.5 Verallgemeinerung der Konstruktion

Das beschriebene Konstruktionsschema läßt sich leicht auf k-fache Unabhängigkeit erweitern. Hierzu benutzt man dieselbe Matrix A wie bei paarweiser Unabhängigkeit und erzeugt q^k Belegungen von $X_0, \dots, X_{n-1}$ durch

$$b^{x_0,\dots,x_{k-1}} = (b_0, \dots, b_{n-1}) \text{ mit } 0 \leq x_0, \dots, x_{k-1} \leq q-1$$

Dabei sei b_i der Eintrag aus A mit den Koordinaten $\left(i, \left(\sum_{j=0}^{k-1} x_j \cdot i^j\right) \bmod q\right)$. Jede Belegung habe die Wahrscheinlichkeit $1/q^d$.
Wenn diese Konstruktion korrekt ist, muß gelten:

$$P[X_i^{(1)} = R_j^{(1)} \wedge \dots \wedge X_i^{(k)} = R_j^{(k)}] = \frac{n_{ij}^{(1)}}{q} \cdot \ldots \cdot \frac{n_{ij}^{(k)}}{q}$$

Beweisskizze:
Das Gleichungssystem für eine bestimmte Belegung von $(b_0, \dots, b_{n-1})$ (d. h. jedem b_i soll der Wert aus einer bestimmten Spalte von A zugewiesen werden) besitzt genau eine Lösung (Vandermondesche Determinante!). Für jedes b_i stehen n_{ij} mögliche Spaltenindizes zur Verfügung.

13.6.6 Transformation von Algorithmus B

Analyse von Alg. B bei paarweiser Unabhängigkeit

Theorem 2 *Für Algorithmus B gilt bei* paarweise *unabhängigen Zufallsvariablen* $\{coin(i) : i \in V'\}$*:*

$$E[Y_k^B - Y_{k+1}^B] \geq \frac{1}{16} Y_k^B$$

Dies ist eine unmittelbare Konsequenz von

Lemma C *Für Algorithmus B gilt bei* paarweise *unabhängigen Zufallsvariablen* $\{coin(i) : i \in V'\}$*:*

$$\Pr[i \in \text{adj}(I')] \geq \frac{1}{8} \cdot \min\{\text{sum}(i), 1\}$$

Beweis: Es gelten diesselben Definitionen wie für Lemma B. Außerdem sei

$$\alpha_0 = 0 \quad ; \quad \alpha_l = \sum_{j=1}^{l} p_j \text{ für } 1 \leq l \leq d(i)$$

Wie im Beweis von Lemma B gilt:

$$\Pr[i \in \mathrm{adj}(I')] \geq \sum_{j=1}^{d(i)} \Pr[E'_j] \cdot \Pr[A_j | E'_j]$$

Suche untere Schranke für $\Pr[A_j|E'_j] = 1 - \Pr[\neg A_j|E'_j]$:

$$\Pr[\neg A_j|E'_j] \leq \sum_{\substack{v \in adj(j) \\ d(v) \geq d(j)}} \Pr[E_v|E'_j] \leq \sum_{\substack{v \in adj(j) \\ d(v) \geq d(j)}} \frac{p_v}{1-\alpha_{j-1}} \text{ , denn}$$

$$\Pr[E_v|E'_j] = \frac{\Pr[E_v \cap E'_j] \cdot \Pr[E_j]}{\Pr[E_j] \cdot \Pr[E'_j]} = \frac{\Pr[E_v \cap \neg E_1 \cap \ldots \cap \neg E_{j-1}|E_j]}{\Pr[\neg E_1 \cap \ldots \cap \neg E_{j-1}|E_j]}$$

$$\begin{array}{rcl} \text{Zähler} & \leq & \Pr[E_v|E_j] \overset{U}{=} \Pr[E_v] = p_v \\ \text{Nenner} & = & 1 - Pr\left[\bigcup_{l=1}^{j-1} E_l \middle| E_j\right] \geq 1 - \sum_{l=1}^{j-1} \Pr[E_l|E_j] \overset{U}{=} 1 - \alpha_{j-1} \end{array}$$

Daraus folgt (mit $p_v = 1/2d(v) \leq 1/2d(j)$):

$$\Pr[A_j|E'_j] \geq 1 - \underbrace{\sum_{\substack{v \in adj(j) \\ d(v) \geq d(j)}} \frac{p_v}{(1-\alpha_{j-1})}}_{\geq d(j) \text{ Summ.}} \geq 1 - \frac{1}{2(1-\alpha_{j-1})} = \frac{1-2\alpha_{j-1}}{2(1-\alpha_{j-1})}$$

Suche untere Schranke für $\Pr[E'_j]$

$$\begin{array}{rcl} \Pr[E'_j] & = & \Pr[E_j] \cdot \Pr[\neg E_1 \cap \ldots \cap \neg E_{j-1}|E_j] \\ & = & p_j \left(1 - Pr\left[\bigcup_{l=1}^{j-1} E_l \middle| E_j\right]\right) \geq p_j(1-\alpha_{j-1}) \end{array}$$

Für $1 \leq l \leq d(v)$ und $\alpha_l < \frac{1}{2}$ gilt somit

$$\begin{array}{rcl} \Pr[i \in \mathrm{adj}(I')] & \geq & \sum_{j=1}^{l} \frac{1}{2} p_j (1-2\alpha_{j-1}) \\ & = & \frac{1}{2} \cdot \left(\sum_{j=1}^{l} p_j - 2 \cdot \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k \right) \\ & \geq & \frac{1}{4} \cdot \min\{\alpha_{d(i)}, \frac{1}{2}\} \geq \frac{1}{8} \cdot \min\{\mathrm{sum}(i), 1\} \end{array}$$

Die letzte Ungleichung folgt aus dem technischen Lemma.

Durchführung der Derandomisierung

Algorithmus B erfüllt fast alle Bedingungen, die wir für die Derandomisierung gefordert haben:

- *Gute* Belegung der Zufallsvariablen $\{coin(i) : i \in V'\}$: mindestens $\frac{1}{16}$ der Kanten in E' werden entfernt.
- Aus Theorem 2 folgt: Für *paarweise* unabhängige ZV $coin(i)$ ist die Wahrscheinlichkeit für eine *gute* Belegung positiv.
- Nach der Ausführung des Rumpfes der **while**-Schleife steht bereits fest, ob eine Belegung *gut* war.
- Für den Wertebereich der ZV gilt: $coin(i) \in \{0, 1\}$

Aber: Kriterium 1 ist nicht voll erfüllt $\implies$ Suche Primzahl q und definiere Wahrscheinlichkeiten neu.

Dazu nehmen wir folgende kleine Modifikationen am *choice step* vor:

- Suche Primzahl q mit $n \leq q < 2n$ (Existiert nach dem *Bertrandschen Postulat* (siehe [Ni]))
- Neue Wahrscheinlichkeit für „coin(i) = 1“:
  $P[E_j] = p'_j = \lfloor p_i \cdot q \rfloor / q$ mit $p_i = 1/2d(i)$
- Durchführung des *choice steps* mit Fallunterscheidung:
  1. $\exists i \in V' : d(i) \geq n/16 \implies I' \leftarrow \{i\}$
  2. $\forall i \in V' : d(i) \leq n/16 \implies$ Erzeuge Wahrscheinlichkeitsraum mit q^2 Elementen und berechne I' wie beschrieben.

Eine kurze Analyse zeigt, daß diese kleine Veränderung der Wahrscheinlichkeiten die Effizienz des Algorithmus nicht wesentlich verschlechtert:

Theorem 3 *Für Algorithmus C gilt im Fall* $\forall i \in V' : d(i) \leq n/16$*:*

$$E[Y_k^C - Y_{k+1}^C] \geq \frac{1}{18} Y_k^C$$

Dies ist eine unmittelbare Konsequenz von

Lemma D *Für Algorithmus C gilt im Fall* $\forall i \in V' : d(i) \leq n/16$*:*

$$\Pr[i \in \mathrm{adj}(I')] \geq \frac{1}{9} \min\{\mathrm{sum}(i), 1\}$$

Beweis:

$$d(i) < \frac{n}{16} \implies \frac{q}{2d(i)} \geq \frac{n}{2d(i)} > 8 \implies p_i' \geq \frac{8}{q} \implies \frac{1}{q} \leq \frac{p_i'}{8}$$

$$\frac{(\lfloor p_i \cdot q \rfloor + 1)}{q} \geq p_i \implies p_i' + \frac{1}{q} \geq p_i \iff \underline{\frac{8}{9} p_i \leq p_i' \leq p_i}$$

Der Rest des Beweises erfolgt analog zur Analyse von Algorithmus B mit den neuen Schranken für p_i'.

Die Durchführung der Derandomisierung bereitet nun keine Schwierigkeiten mehr und führt zu folgender Implementierung:

```
Wähle $x, y$ mit $0 \leq x, y \leq q - 1$ (Variante C)
bzw.
In parallel $\forall x, y : 0 \leq x, y \leq q - 1$ (Variante D)
begin
    $X \leftarrow \emptyset$
    In parallel $\forall i \in V'$
    begin
        $n(i) \leftarrow \lfloor q/2d(i) \rfloor$
        $l(i) \leftarrow (x + y \cdot i) \bmod q$
        if $l(i) \leq n(i)$ then $X \leftarrow X \cup \{i\}$
    end
end
```

Bei jedem Durchlauf der **while**-Schleife müssen Werte für die ZV $x, y \in \{0, \ldots, q-1\}$ ermittelt werden. Dazu braucht man $(O(\log n))$ Zufallsbits.

Literaturverzeichnis

[Lu] M. Luby: *A simple parallel algorithm for the maximal independent set problem.* SIAM J. Comput. 15(4): 1036-1053, 1986

[Br] I. Bronstein, A. Semendjajew, G. Musiol, H. Mühlig: *Taschenbuch der Mathematik.* Verlag Harry Deutsch: 240-241, 1993

[Gr] M. Greiner: *Wahrscheinlichkeitstheorie und Statistik für Informatiker - Aufgabensammlung.* Institut für Informatik der TU München: 219-222, 1994

[Ni] I. Niven, H. S. Zuckerman: *Einführung in die Zahlentheorie II.* B.-I.-Wissenschaftsverlag: 248-250, 1972

Vortrag 14

Zufallszahlengeneratoren

Josef Weidendorfer
Technische Universität München
weidendo@informatik.tu-muenchen.de

14.1 Einleitung

Nachdem viel über probabilistische Algorithmen gesprochen wurde, handelt dieser Vortrag über Pseudo-Zufallsgeneratoren, die Grundlage einer effektiven Durchführung eben genannter Art von Algorithmen sind.
Welche Eigenschaften sollte ein idealer Zufallsgenerator besitzen? Er sollte schnell, ausgehend von kleinen Startwerten, lange zufällige Folgen (von Bits) generieren. Am einfachsten läßt sich eine „zufällige Folge“damit beschreiben, daß sie in jeder Hinsicht scheinbar mit Folgen von (fairen) Münzwürfen erzeugt wurde. Was heißt das aber konkret?
Statistische Eigenschaften einer „zufälligen“Folge (Gleichverteilung, „gute Mischung“) sollte eine von einem Pseudo-Zufallsgenerator produzierte Folge ebenfalls besitzen. Dies läßt sich leicht nachprüfen: Man entwickelt Tests, die diese Eigenschaften überprüfen, und welche die Pseudo-Zufallsfolge bestehen muß.
Aber ein Münzwurf hat noch eine weitere wichtige Eigenschaft: Er ist nicht vorhersehbar. Eine solche Eigenschaft kann von einem „Zufallsgenerator“, der auf einer deterministischen Maschine arbeitet, nicht erwartet werden: Aus Determinismus kann kein Nichtdeterminismus produziert werden. Dies läßt sich jedoch damit umgehen, daß man etwas abweichend fordert: Ein Element einer Pseudo-Zufallsfolge darf nur unter immensem Einsatz von Rechenleistung vorausberechnet werden können. Präziser faßt dies folgende Definition:

Definition 1.1: Ein Pseudo-Zufallsgenerator heißt *in polynomialer Zeit unvorhersehbar (nach rechts / links)* genau dann, wenn für jeden endlichen Teilabschnitt einer von ihm produzierten Folge, bei dem ein (das rechts /links außen stehende) Element gelöscht wurde, dieses Element von keiner probabilistischen Turingmaschine[1] in polynomialer Zeit (bzgl. der Länge des Startwertes) besser erraten werden kann als durch einen Münzwurf.

Im folgenden werden zwei Pseudozufallsgeneratoren vorgestellt, der sogenannte $1/P$-Generator und der x^2 mod N-Generator. Dabei werden interessante Eigenschaften dieser Generatoren behandelt. Zuvor werden einige Definitionen und zahlentheoretische Grundlagen vorausgeschoben.

14.2 Grundlagen

14.2.1 Definitionen zu Pseudo-Zufallsgeneratoren

Nachfolgend wird eine präzise Definition eines Pseudo-Zufallsgenerators eingeführt. Dazu benötigt man einige Begriffe. Dabei sei $|N|_b$ die Anzahl Ziffern der b-adischen Darstellung von N. Ist b nicht angegeben, gilt $b = 2$.

Definition 2.1: Sei $W \subset \mathbb{N}$ die Menge der *Parameter-Werte.* Für jedes $N \in W$ sei $X_N \subset \{0,1\}^n$ eine Menge von *Startwerten* der Länge $n = |N|$. Damit heißt $X = \{(N,x) \mid N \in W, x \in X_N\}$ der *Startwert-Bereich.*

Im nachfolgenden Text wird $x \in X_N$ oft mit $(N,x) \in X$ identifiziert, weshalb man $X_N \subset X$ schreiben kann und somit X als disjunkte Vereinigung $\bigcup_{N \in W} X_N$.

Definition 2.2: Sei $X^n = \{(N,x) \mid N \in W, |N| = n$ und $x \in X_N\}$ die Menge der *Startwerte der Länge* n. Für alle ausreichend großen n sei μ_n eine Wahrscheinlichkeitsverteilung auf X^n. Dann heißt $U = \{\mu_n\}$ eine *zugängliche Wahrscheinlichkeitsverteilung* auf X, wenn es eine probabilistische $poly(n)$-Prozedur[2] gibt, so daß für jede ausreichend große Eingabe n Elemente aus X^n gemäß μ_n mit vernachlässigbarem Fehler ausgegeben werden. D.h. es wird ein Element aus M_n, $X^n \subset M_n$, gemäß μ'_n ausgegeben, wobei μ'_n eine Wahrscheinlichkeitsverteilung auf M_n ist und für alle t und ausreichend großen n gilt:

$$\sum_{(N,x) \in X^n} |\mu_n(N,x) - \mu'_n(N,x)| < 1/n^t.$$

---

[1] Das ist eine Turingmaschine, die pro Schritt einen Münzwurf ausführen kann, um zufällige Bits zu erhalten.

[2] Eine Turing-Maschine, die pro Schritt einen Münzwurf ausführen kann und spätestens in Zeit $poly(n)$ hält, wobei $poly$ ein Polynom und n die Eingabelänge ist.

Ein Paar (X, U), wobei X ein Startwert-Bereich und U eine zugängliche Wahrscheinlichkeitsverteilung ist, heißt *Startwert-Raum.* Im folgenden wird dafür oft vereinfachend nur X geschrieben.
Sei nun $\Sigma = \{0, 1, \ldots, b-1\}$.

Definition 2.3: Ein *Pseudo-Zufallsgenerator* G *(zur Basis* b*) auf dem Startwert-Raum* X ist eine effektive Abbildung $G : X \to \Sigma^\infty$, so daß es zu jeder ganzen Zahl $s \geq 0$ eine ganze Zahl $t \geq 0$ gibt, sodaß gilt: Für alle $(N, x) \in X$ mit $\mu_n(N, x) \neq 0$ wird $[G(N, x)]^{n^s}$, die Startfolge der Länge n^s, in Zeit $O(n^t)$ ausgegeben. (D.h. G generiert aus kleinen Startwerten (z.B. der Länge n) lange Folgen (z.B. der Länge n^s) in polynomialer Zeit.) $G(N, x)$ heißt *die von* G *mit Eingabe/Startwert* (N, x) *generierte Pseudo-Zufallsfolge.*

Repräsentiert Σ eine Menge „beobachtbarer Zustände“ für Elemente aus dem Startwert-Raum X, so ist $G(N, x)$ die Folge von Zuständen, die von x durchwandert werden. Dies motiviert folgende Formulierung für einen Pseudo-Zufallsgenerator:

Definition 2.4: Eine *Transformation* T auf dem Startwert-Bereich X ist eine in polynomialer Zeit berechenbare Abbildung $T : X \to X$, sodaß für alle ausreichend großen n gilt: $T(X^n) \subset X^n$ und T erhält μ_n (d.h. $\mu_n(A) = \mu_n(T^{-1}(A))$ für alle $A \subset X^n$). Für Startwert $x \in X_N$ heißt die Folge $x, T(x), T^2(x), \ldots$ das *Orbit von* x *unter* T. Man schreibt oft x_k für $T^k(x)$. Somit ist $x_0 = x$ und $x_{k+1} = T(x_k)$.

Definition 2.5 Eine *Partition* B vom Startwert-Bereich X nach Σ ist eine in polynomialer Zeit berechenbare Abbildung $B : X \to \Sigma$.

Das System $\langle X, T, B\rangle$, wobei X ein Startwert-Raum, T eine Transformation auf X und B eine Partition ist, definiert einen Pseudo-Zufallsgenerator G zur Basis b auf X, wobei die k-te Koordinate $[G(N, x)]_k = B(T^k(x))$. Damit gilt, wenn $x_0, x_1, x_2, \ldots$ das Orbit von x unter T ist: $G(N, x) = b_0 b_1 \ldots$, wobei $b_k = B(x_k)$ der Zustand von x zur Zeit k ist.
Bemerkung: Wenn die Umkehrung von T auf X, T^{-1}, definiert und in polynomialer Zeit berechenbar ist, spricht man von G als der Abbildung von X in die zweiseitig unendlichen Folgen von Σ.

14.2.2 Nötiges aus der Zahlentheorie

Anschließend wird einiges aus der Zahlentheorie verwendet, was hier kurz erwähnt werden soll:

1. x mod N bezeichne den Rest bei der ganzzahligen Division von x durch N (und nicht die zugehörige Restklasse mod N).

2. $\varphi(N)$ ist die Eulersche Phi-Funktion mit $\varphi(N) = |\{x \in \mathbb{N} \mid 0 < x < N, \operatorname{ggT}(x, N) = 1\}|$. Ist N eine Primzahl, gilt $\varphi(N) = N - 1$.
3. $Z_N^* = \{x \in \mathbb{N} \mid 0 < x < N, \operatorname{ggT}(x, N) = 1\}$ ist eine multiplikative Gruppe mit der Ordnung $\varphi(N)$. Ist N eine Primzahl, so ist $Z_N^* = \{1, 2, \ldots, N-1\}$.
4. Die Ordnung $\operatorname{ord}_N x$ von $x \in Z_N^*$ ist die kleinste positive Zahl n, für die gilt: $x^n \equiv 1 \bmod N$.
5. $x \in Z_N^*$ heißt Primitivwurzel (oder auch Generator) von Z_N^*, wenn $\operatorname{ord}_N x = \varphi(N)$.
6. Für Carmichael's λ-Funktion, definiert in Kapitel 6.2, gilt: $\lambda(N)$ ist das kgV und Supremum der Ordnungen aller $x \in Z_N^*$
7. Folgende Sätze gelten[3]:

$x^{N-1} \equiv 1 \bmod N,$	$N \in \mathbb{P},\ x \in Z_N^*$	Kleiner Satz von Fermat
$x^{\varphi(N)} \equiv 1 \bmod N,$	$x \in Z_N^*$	Satz von Euler
$x^{\lambda(N)} \equiv 1 \bmod N,$	$x \in Z_N^*$	Carmichael's Erweiterung

8. Das Legendre-Symbol $\binom{x}{N}$ für $N \in \mathbb{P}, x \in Z_N^*$ ist 1, falls x Quadratrest $\bmod N$ ist (d.h. $\exists y \in Z_N^* : y^2 \equiv x \bmod N$) und -1, falls x Nicht-Quadratrest ist.
9. Das Jakobi-Symbol (selbe Schreibweise) ist eine Verallgemeinerung des Legendre-Symbols für alle ungeraden $N \in \mathbb{N}$. Für $N \in \mathbb{P}$ stimmt es mit dem Legendre-Symbol überein. Für die restlichen N sei $N = p_1 \cdot \ldots \cdot p_k$ mit $p_i \in \mathbb{P}$. Nun gilt: $\binom{x}{N} = \binom{x}{p_1} \cdot \ldots \cdot \binom{x}{p_k}$. Zudem ist $\binom{a}{N}\binom{b}{N} = \binom{ab}{N}$ und $\binom{-1}{N} = (-1)^{\frac{N-1}{2}}$. Eine Schreibweise soll noch eingeführt werden: $x \in Z_N^*(+1)$, falls $\binom{x}{N} = 1$ und $x \in Z_N^*(-1)$, falls $\binom{x}{N} = -1$.
10. Der chinesische Restsatz sagt im speziellen für zwei teilerfremde ganze Zahlen P und Q aus: Für $x \equiv c_P \bmod P$ und $x \equiv c_Q \bmod Q$ hat die Gleichung $x \equiv c_{PQ} \bmod PQ$ genau eine Lösung c_{PQ} (mit $c_P \in Z_P^*, c_Q \in Z_Q^*$).

Damit lassen sich nun die zwei hier zu behandelnden spezifischen Generatoren einführen.

14.3 Zwei Pseudo-Zufallsgeneratoren

14.3.1 Der $1/P$-Generator

Sei b eine fest gewählte natürliche Zahl und $\Sigma = \{0, 1, \ldots, b-1\}$.

---

[3] $\mathbb{P}$ sei die Menge der Primzahlen

Definition 3.1: Sei $W = \{P | P \in \mathbb{N}, P > 1$ und P relativ prim zu $b\}$ die Menge der Parameterwerte und der Startwert-Bereich X die disjunkte Vereinigung $\bigcup_{P \in W} Z_P^*$. Wir können X mit der Teilmenge $\{r/P | P \in W, r \in Z_P^*\}$ des Einheitsintervalls $[0, 1)$ identifizieren. μ_n sei die Verteilung auf X^n gegeben durch $\mu_n(P, r) = u_n(P) \cdot v_P(r)$, wobei u_n die gleichmäßige Wahrscheinlichkeitsverteilung auf $\{P \in W \Bigm| |P|_b = n\}$ und v_P die gleichmäßige Verteilung auf Z_P^* ist. Dann ist $U = \{\mu_n\}$ eine zugängliche Wahrscheinlichkeitsverteilung auf X.
$G : X \to \Sigma^\infty$ sei definiert durch $G(r/P) = q_1 q_2 q_3 \ldots$, der Folge von Ziffern, die in der b-adischen Darstellung von $r/P \in X$ direkt hinter dem Komma stehen. Dieser Pseudo-Zufallsgenerator heißt *$1/P$-Generator (zur Basis b).*

Bemerkung: Die Ziffernfolge von r/P läßt sich in Zeit $O(|b| \cdot |P|)$ berechnen, und die Folge $G(r/P)$ ist periodisch mit einer Periode, die $\varphi(P)$ teilt (Dies gilt, da $b^{\varphi(P)} \equiv 1 \bmod P$ — Satz von Euler).
Um den $1/P$-Generator als System $\langle X, T, B\rangle$ zu beschreiben, setzt man X wie oben, $T : X \to X$ ist definiert durch $T(x) = bx \bmod 1$ (äquivalent wäre $T(r) = br \bmod P$ für $r \in Z_P^*$, was eine Permutation von Z_P^* ist), und $B : X \to \Sigma$ durch $B(x) = \lfloor bx \rfloor$.
Beispiel: Sei b=10, P=7, r=1. Da 10 eine Primitivwurzel von 7 ist, ist die Periode der erzeugten Sequenz 6:

i	0	1	2	3	4	5	6
$x_i := b^i r \bmod 1$	1/7	3/7	2/7	6/7	4/7	5/7	1/7
$\lfloor bx_i \rfloor$	1	4	2	8	5	7	1

14.3.2 Der x^2 mod N-Generator

Definition 3.2: Sei $W = \{N \in \mathbb{N} \mid N = P \cdot Q$ mit $|P| = |Q|, P \equiv Q \equiv 3 \bmod 4\}$ die Menge der Parameterwerte und $X_N = \{x^2 \bmod N | x \in Z_N^*\}$ für alle $N \in W$ die Quadratreste mod N. Der Startwert-Bereich X sei die disjunkte Vereinigung $\bigcup_{N \in W} X_N$. Für $(N, x) \in X^n$ sei $\mu_n(N, x) = u_n(N) \cdot v_N(x)$, wobei u_n die gleichmäßige Wahrscheinlichkeitsverteilung auf $\{N \in W \Bigm| |N| = n\}$ und v_N die gleichmäßige Verteilung auf X_N ist. Dann ist $\{\mu_n\}$ eine zugängliche Wahrscheinlichkeitsverteilung auf X, da die Anzahl der wie oben beschriebenen N mit $|N| = n$ ebenso wie $|X_N|$ in polynomialer Zeit berechenbar ist[4]. $T : X \to X$ sei definiert durch $T(x) = x^2 \bmod N$ für $x \in X_N$ und $B : X \to \{0, 1\}$ durch $B(x) = x \bmod 2$ (die Parität von x). Dann heißt der durch $\langle X, T, B\rangle$ definierte Pseudo-Zufallsgenerator der *x^2 mod N-Generator.*

---

[4] $1/(k \ln 2)$ aller k-Bit Zahlen sind prim und davon ist die Hälfte $\equiv 3 \bmod 4$; die Primzahleigenschaft läßt sich in polyn. Zeit ebenso entscheiden wie ggT's (für X_N).

Dieser Generator gibt also die Pseudo-Zufallsfolge $b_0 b_1 b_2 \ldots$ mit Eingabe (N, x_0) aus, wobei $b_i = x_i \bmod 2$ mit $x_i = x_{i-1}^2 \bmod N$ für $i > 0$. Die Periode ist gewöhnlich $\lambda(\lambda(x))$ (Siehe Kapitel 6.2), und mit $x_i = x_0^{2^i} \bmod N = x_0^{2^i \, mod \, \lambda(N)} \bmod N$ läßt sich die Folge effizient vorwärts und rückwärts – wegen der bekannten Periodizität – berechnen. Allerdings muß zur Berechnung von $\lambda(N)$ die Primfaktorzerlegung von N bekannt sein.

Beispiel: Sei $N = 7 \cdot 19 = 133, x_0 = 4$. Da $\lambda(N) = 18$ und damit $\lambda(\lambda(N)) = 6$, hat die erzeugte Folge Periode 6:

i	0	1	2	3	4	5	6
x_i	4	16	123	100	25	93	4
b_i	0	0	1	0	1	1	0

14.4 Annahmen für die weitere Behandlung

Für einige der weiteren Beweise wird von Annahmen ausgegangen, die hier aufgeführt sind. Diese sind als große Probleme der Zahlentheorie bekannt, konnten aber bisher nicht bewiesen werden.

Die erste Annahme bezieht sich auf das *Problem des diskreten Logarithmus:* Sei P eine Primzahl, b eine Primitivwurzel von P. Die Funktion $f_{b,P} : Z_P^* \to Z_P^*$ mit $f_{b,P} = b^x \bmod P$ ist eine Permutation von Z_P^* und in polynomialer Zeit berechenbar. Das *Problem des diskreten Logarithmus* mit Parameter b und P besteht darin, für jedes $y \in Z_P^*$ den Index $x \in Z_P^*$ mit $b^x \bmod P = y$ zu finden. Eine (probabilistische) Prozedur $\mathbf{P}[b, P, y]$ löst das Problem des diskreten Logarithmus, wenn $\forall P \in \mathbb{P}, \forall b \in Z_P^*, \forall y \in Z_P^*$ gilt: $\mathbf{P}[b, P, y] = x$ mit $b^x \bmod P = y$.

Die Annahme zum diskreten Logarithmus besteht darin, daß eine Prozedur, die dieses Problem löst, für einen Anteil der Eingaben ineffizient sein wird. D.h. für diese Eingaben ist die Zeitkomplexität höher als polynomial.

Die zweite Annahme gilt dem *Quadratrestproblem:* Sei N ein Produkt zweier unterschiedlicher, ungerader Primzahlen. Genau die Hälfte der Elemente von Z_N^* sind in $Z_N^*(+1)$, die andere in $Z_N^*(-1)$. Keines der Elemente von $Z_N^*(-1)$ ebensowenig wie die Hälfte der Elemente von $Z_N^*(+1)$ ist Quadradrest; die andere Hälfte von $Z_N^*(+1)$ besteht aus Quadratresten. Das *Quadratrestproblem* besteht darin, zu entscheiden, ob ein Element aus $Z_N^*(+1)$ Quadratrest ist oder nicht. $\mathbf{P}[N, x]$ löst dieses Problem, wenn für alle wie oben beschriebenen N und $x \in Z_N^*$ 0 oder 1 ausgegeben wird, je nachdem, ob x Quadratrest $\bmod N$ ist oder nicht.

Die Quadratrest-Annahme (QRA) besteht darin, daß jede (probabilistische) $poly(n)$-Prozedur zur Lösung des Quadratrestproblems für einen nicht zu vernachlässigbaren Teil der Eingaben falsche Ausgaben liefert.

14.5 Eigenschaften des $1/P$-Generators

Wesentliches Ergebnis dieses Abschnittes wird es sein, zu zeigen, daß die Folgen, die der $1/P$-Generator liefert, vollständig vorhersagbar sind. Die oben eingeführten Bezeichnungen für den $1/P$-Generator werden noch etwas erweitert:
Sei $P \in \mathbb{P}$ und $b \in Z_P^*$. Für $0 < r_0 < P$ ist die b-adische Darstellung von $r_0/P = 0, q_1q_2q_3\ldots$ mit $0 \le q_i < b$. r_m sei der „Rest“ nach Berechnung der ersten m Ziffern; d.h. $0 < r_m = b^m r_0 \bmod P < P$.
Die erzeugten Folgen besitzen einen guten „Verteilungscharakter“. Dazu folgende Definition :

Definition 5.1: Seien $P, b \in \mathbb{N}, P$ beliebig. Eine Folge $q_1q_2q_3\ldots$ von b-adischen Ziffern der Periode $P-1$ heißt *(verallgemeinerte) De Bruijn-Folge mit Periode $P-1$ zur Basis b*, wenn jede b-adische Zeichenkette der Länge $|P|_b - 1$ mindestens einmal, und jede b-adische Zeichenkette der Länge $|P|_b$ höchstens einmal in einer Periode der Folge vorkommt.

Satz 5.1: Sei $P \in \mathbb{P}, b$ eine Primitivwurzel mod$P$. Sei $r_0 \in Z_P^*$. Dann ist die vom $1/P$-Generator zur Basis b erzeugte Pseudo-Zufallsfolge mit Eingabe r_0/P eine (verallgemeinerte) De Bruijn-Folge mit Periode $P-1$ zur Basis b.

Beweis:
Da $r_m = b^m \cdot r_0 \bmod P$ und b eine Primitivwurzel mod$P$ ist, hat die Folge der Reste r_m Periode $P-1$, die Reste einer Periode sind paarweise verschieden und $\{r_m \mid 1 \le m \le P-1\} = Z_P^*$. Gleiches gilt für die Folge der Quotienten r_m/P :

$$\{r_m/P \mid 1 \le m \le P-1\} = \{1/P, 2/P, \ldots, (P-1)/P\}. \tag{14.1}$$

Deshalb ist die Folge der erzeugten Ziffern q_m höchstens von Periode $P-1$. Wäre die Periode kleiner, also $0, q_{m_1+1}q_{m_1+2}\ldots = 0, q_{m_2+1}q_{m_2+2}\ldots$, würde $r_{m_1} = r_{m_2}$ gelten, da $r_m/P = 0, q_{m+1}q_{m+2}\ldots$. Dies steht jedoch im Widerspruch zur paarweisen Verschiedenheit der Reste einer Folgenperiode. Also ist die Periode $P-1$.
Eine Zeichenkette $a_1 \cdots a_s$ von s b-adischen Ziffern erscheint in der Darstellung von r_0/P genau dann, wenn es Anfang der Darstellung eines r_m/P ist mit $1 \le m \le P-1$, und nach (1) genau dann, wenn es Anfang der Darstellung eines $k/P, 1 \le k \le P-1$ ist. Genauso entsprechen die b-adischen Zeichenketten der Länge s jeweils einem Subintervall von [0,1), nämlich $[l/b^s, (l+1)/b^s)$ mit ganzem $l, 0 \le l < b^s$.
Da $1/P < 1/b^{|P|_b - 1}$, gibt es für jedes l mindestens ein $k, 1 \le k \le P-1$ mit $k/P \in [l/b^{|P|_b-1}, (l+1)/b^{|P|_b-1})$, woraus die erste Behauptung folgt. Andererseits, da $1/b^{|P|_b} < 1/P$, gibt es für jedes l höchstens ein $k, 1 \le k \le P-1$ mit $k/P \in [l/b^{|P|_b}, (l+1)/b^{|P|_b})$, woraus die zweite Behauptung folgt.

qed.

Man mag wegen Satz 5.1 glauben, daß der $1/P$-Generator ein guter Pseudo-Zufallsgenerator ist. Auch, da man aus einem während des Generierens auftretenden Restes r_m nicht auf seinen Index m schließen kann (Nach der Annahme zum diskreten Logarithmus).
Nach Satz 5.1 lassen sich aus $|P| - 1$ aufeinanderfolgenden Ziffern nicht die nachfolgenden berechnen. Allerdings reichen $2|P| + 1$ Ziffern ohne Kenntnis von P schon aus, um P zu berechnen und die Folge nach vorne und hinten zu erweitern. Dies zeigt folgender Satz:

Satz 5.2: Seien P, b ganze Zahlen > 1 und zueinander relativ prim (d.h. $\mathrm{ggT}(P, b) = 1$). Sei r_0 eine ganze Zahl mit $0 < r_0 < P$. Folgende Probleme lassen sich in polynomialer Zeit bzgl. $|P|$ lösen:

1. Sei *poly* ein fest gewähltes Polynom
   Eingabe: P, b, Rest r_m und $k \in \mathbb{N}, k \leq poly(|P|)$
   Ausgabe: $r_{m-1}, r_{m+k}; q_m q_{m+1} \cdots q_{m+k}$
2. Eingabe: $P, b, |P|$ aufeinanderfolgende Ziffern $q_{m+1} q_{m+2} \cdots q_{m+|P|}$
   Ausgabe: r_m (und damit nach 1. $r_{m+|P|}$ und $q_m, q_{m+|P|+1}$)
3. Sei P nun relativ prim zu 1, zu 2 ..., zu b (um sicherzustellen, daß es nur ein P gibt, das r_m und r_{m+1} erzeugt haben kann)
   Eingabe: b, r_m, r_{m+1}, so daß $r_m \cdot b \neq r_{m+1}$
   Ausgabe: P (und damit nach 1. $q_m q_{m+1} \dots q_{m+|P|}$)
4. Sei r_0 nun relativ prim zu P (z.B. $r_0 = 1$)
   Eingabe: $b; k$ aufeinanderfolgende Ziffern $q_{m+1} q_{m+2} \cdots q_{m+k}$, wobei $k = \lceil log_b(2P^2) \rceil, m$ beliebig (Bemerkung: $k \leq 2|P| + 1$)
   Ausgabe: $P; r_m$

Beweis:
Zu 1.) $r_{m+k} = b^k r_m \bmod P$ und $r_{m-1} = b^{-1} r_m \bmod P$, wobei b^{-1} das Inverse zu $b \bmod P$ bezeichnet. Damit gilt

$$(b^k r_m)/P = q_{m+1} \cdots q_{m+k} + r_{m+k}/P \tag{14.2}$$

Somit ist $q_m \cdots q_{m+k} = \lfloor (b^{k+1} r_{m-1})/P \rfloor$.
Zu 2.) Nach (2) gilt $r_m = (q_{m+1} \cdots q_{m+|P|}) \cdot P/b^{|P|} + (r_{m+|P|})/b^{|P|}$. Deshalb gilt $r_{m+|P|} < P < b^{|P|}, r_m = \lceil (q_{m+1} \cdots q_{m+|P|}) \cdot P/b^{|P|} \rceil$.
Zu 3.) Nach (2) mit $k = 1$ gilt: $b \cdot r_m - r_{m+1} = q_{m+1} \cdot P$ mit $0 \leq q_{m+1} < b$. Es gilt sogar $0 < q_{m+1}$, da $b \cdot r_m \neq r_{m+1}$. Also ist P gleich einer ganzen Zahl in $(b \cdot r_m - r_{m+1})/1, (b \cdot r_m - r_{m+1})/2, \dots, (b \cdot r_m - r_{m+1})/(b - 1)$. Man wähle P so, daß es relativ prim zu $1, 2, \dots, b$ ist. Davon gibt es nur eines: um dies zu zeigen seien P, Q zwei solche ganze Zahlen. Dann gilt $P \cdot i = Q \cdot j$ für $0 < i, j < b$.

OBdA. sei $P < Q$. Q ist relativ prim zu $1, 2, \dots b$, also ist $\mathrm{ggT}(Q, i) = 1$, somit $Q|P$, d.h. $Q \leq P$, was den Widerspruch bringt.
Zu 4.) Dies läßt sich sehr gut mit Kettenbrüchen beweisen. Dazu ein paar Vorbemerkungen:
Sei $\alpha = a_0 + 1/(a_1 + 1/(a_2 + \dots + 1/(a_{k-1} + 1/a_k) \dots)) =: [a_0; a_1, a_2, \dots, a_k]$. Die Konvergenten von α sind $[a_0; a_1], [a_0; a_1, a_2], \dots [a_0; a_1 \dots, a_k]$. Als Bruch geschrieben lauten die Konvergenten für $a_0 = 0$:\$A_1/B_1 := [0; a_1] = 1/a_1$, $A_2/B_2 := [0; a_1, a_2] = a_2/(a_1 a_2 + 1)$, …, $A_i/B_i = (a_i A_{i-1} + A_{i-2})/(a_i B_{i-1} + B_{i-2})$. Letzeres läßt sich durch vollständige Induktion zeigen.
Nach (2) ist $r_m/P = q_{m+1} \cdots q_{m+k}/b^k + \varepsilon$ mit $0 \leq \varepsilon < 1/b^k$. Nach [LeVeque77, S.237] gilt, daß die Kettenbruchentwicklung von $q_{m+1} \cdots q_{m+k}/b^k$ eine Konvergente r_m/P besitzt, wenn $1/b^k \leq 1/2P^2 \Leftrightarrow 2P^2 \leq b^k \Leftrightarrow log_b(2P^2) \leq k$, wie gefordert. Da b und r_0 relativ prim zu P sind, gilt wegen $r_m = b^m r_0 \bmod P$: $\mathrm{ggT}(r_m, P)$=1 und somit $r_m = A_i, P = B_i$ für eine Konvergente von $q_{m+1} \cdots q_{m+k}/b^k$.
Stoppt man die Kettenbruchentwicklung, wenn die ersten k Ziffern von A_i/B_i mit $q_{m+1} \dots q_{m+k}$ übereinstimmen, muß $A_i = r_m$ und $B_i = P$ gelten, da B_i streng monoton wächst und dann der Fehler ausreichend klein ist, um dies sicherzustellen. Da A_i und B_i exponentiell wachsen, kann r_m und P in polynomialer Zeit zu $|B_i|$ und damit $|P|$ berechnet werden.

qed.

Ein größeres Beispiel (mit Primzahl P=503) wird im Vortrag gezeigt. Dabei hat die erzeugte Folge eine Periode von 502. Erstaunlich wirkt das Ergebnis von Satz 5.2(4): Mit beliebigen 6 Ziffern der Folge ergibt die 5. Konvergente der zugehörigen Kettenbruchentwicklung das richtige Ergebnis $r_m/503$.

14.6 Eigenschaften des x^2 mod N-Generators

14.6.1 Nicht-Vorhersagbarkeit des x^2 mod N-Generators

Im folgenden sei $N = PQ$ Produkt zweier verschiedener Primzahlen $\equiv 3 \bmod 4$. x_i sei jeweils ein Quadratrest mod$N$, d.h. $x_i \in \mathrm{QR}_N$, $x_{i+1} = x_i^2 \bmod N$ und $b_i = x_i \bmod 2$. Folgende Behauptungen werden aufgestellt:

1. Die Kenntnis von N ist ausreichend, um die Folge $x_0, x_1, x_2 \dots$ (und damit $b_0 b_1 b_2 \dots$) effizient in Vorwärts-Richtung zu berechnen mit gegebenem Startwert x_0. Die Anzahl der Schritte pro Ausgabe ist unter Benutzung von „schneller Multiplikation“ $O(|N|^{1+\varepsilon})$.

2. Die Faktoren P und Q von N sind ausreichend und notwendig, um die Folge effizient in Rückwärtsrichtung zu berechnen von gegebenem x_0 aus.

3. Die Faktoren von N sind notwendig (unter der Annahme, daß sie notwendig für das Lösen des Quadratrestproblemes für $x \in Z_N^*(+1)$ sind), um auch nur einen ε-Vorteil im Schätzen von b_{-1} in polynomialer Zeit zu haben, wenn N und x_0, zufällig ausgewählt aus QR_N, gegeben sind.

Behauptung 1 ist klar ersichtlich.
Um Behauptung 2 zu zeigen, beweisen wir zuerst

Satz 6.1: Wenn $N = P \cdot Q$, wobei P, Q verschiedene Primzahlen $\equiv 3 \bmod 4$ sind, dann hat jeder Quadratrest $\bmod N$ genau eine Wurzel, die wieder Quadratrest $\bmod N$ ist.

Beweis:
Wenn N Produkt zweier verschiedener Primzahlen P, Q ist, hat jeder Quadratrest genau 4 Wurzeln, hier mit $\pm x$ und $\pm y$ bezeichnet (Hälfte der Z_P^* mit $P \in \mathbb{P}$ sind Quadratreste und 2.2(10)). Da $N \equiv 1 \bmod 4$, ist $\left(\frac{-1}{N}\right) = 1$ (siehe 2.2(9)), und somit $\left(\frac{x}{N}\right) = \left(\frac{-x}{N}\right)$ und $\left(\frac{y}{N}\right) = \left(\frac{-y}{N}\right)$.
Wegen $x^2 - y^2 = (x+y)(x-y) \equiv 0 \bmod PQ$ muß gelten: OBdA $P|(x+y)$. Damit $Q|(x-y)$, sonst entsteht ein Widerspruch zu 4 unterschiedlichen Wurzeln. Damit hat man jedoch $x \equiv y \bmod Q$ und damit $\left(\frac{x}{Q}\right) = \left(\frac{y}{Q}\right)$ und entsprechend $\left(\frac{x}{P}\right) = \left(\frac{-y}{P}\right)$. Somit gilt nach 2.2(9) $\left(\frac{x}{N}\right) = \left(\frac{-x}{N}\right) \neq \left(\frac{y}{N}\right) = \left(\frac{-y}{N}\right)$.
Entfernt man die beiden Wurzeln $\in Z_N^*(-1)$, bleiben zwei übrig, von denen genau eine in $Z_P^*(+1)$ und $Z_Q^*(+1)$ ist. Diese und nur diese Wurzel ist Quadratrest $\bmod N$.

qed.

Die Notwendigkeit, die Primfaktorzerlegung von N zu kennen, folgt nun: Angenommen, man könnte die Folge effizient rückwärts berechnen. Um N zu zerlegen, wählen wir ein $x \in Z_N^*(-1)$. Damit berechnen wir $x_0 = x^2 \bmod N$ und anschließend x_{-1}. Da sich der $\mathrm{ggT}(x + x_{-1}, N)$ effizient ausrechnen läßt, bekommen wir P oder Q (nach Beweis von Satz 6.1). Dann ließe sich das Quadratrestproblem effizient lösen, was im Widerspruch zur entsprechenden Annahme ist.
Mit Kenntnis der Zerlegung von N können wir effizient die Folge rückwärts berechnen:

Satz 6.2: Es gibt einen effizienten Algorithmus $\mathbf{A}$, der bei Eingabe N (von vorgeschriebener Form), den Primfaktoren P und Q von N mit $N = PQ$und beliebigem Quadratrest $x_0 \in Z_N^*$ den einzigen Quadratrest $x_{-1} \bmod N$ mit $(x_{-1})^2 \bmod N = x_0$ berechnet. D.h. $\mathbf{A}(P, Q, x_0) = x_{-1}$.

Beweis:
Nach Satz 6.1 ist die Abbildung $f : x \mapsto x^2 \bmod N$ der Quadratreste mod$N$ in sich selbst eine Permutation. A kann wie folgt beschrieben werden:
Berechne $x_P = \sqrt{x_0} \bmod P$, so daß $\left(\frac{x_P}{P}\right) = 1$. Dabei beschreibt $\sqrt{x_0} \bmod P$ diejenige Zahl, dessen Quadrat $x_0 \bmod N$ ist.
$\sqrt{x_0} \bmod P = \pm x_0^{(P+1)/4} \bmod P$ (für $P \equiv 3 \bmod 4$), da $(\pm x_0^{(P+1)/4})^2 = x_0 \cdot (x_0^{1/2})^{P-1} \equiv x_0 \cdot 1 \bmod P$ wegen Existenz von $x_0^{1/2}$ und kleinem Satz von Fermat 2.2(7). Genauso sei $x_Q = \sqrt{x_0} \bmod Q$ mit $\left(\frac{x_Q}{Q}\right) = 1$.
Konstruiere mithilfe des erweiterten euklidischen Algorithmus [Bund91, S.31] ganze Zahlen u und v so, daß $P \cdot u + Q \cdot v = 1$, und daraus $x_N = \pm x_P \cdot Q \cdot v \pm x_Q \cdot P \cdot u = \sqrt{x_0} \bmod N$, was mithilfe des chinesischen Restsatzes aus $x_N \equiv x_P \bmod P$ und $x_N \equiv x_Q \bmod Q$ folgt. x_N ist somit eindeutig, und da es sowohl Quadratrest mod$P$ als auch mod$Q$ ist, folgt, daß x_N die gesuchte Wurzel von $x_0 \bmod N$ ist.

qed.

Zum Beweis von Behauptung 3 folgende

Definition 6.1: Sei *poly* ein Polynom und $0 < \varepsilon \leq 1/2$ mit nicht vernachlässigbar kleinem ε. Eine *poly*-Prozedur **P** hat einen ε-Vorteil für N im Schätzen der Parität (von x_{-1}, gegeben $x_0 \in$QR$_N$) genau dann, wenn mit die Parität von x_{-1} für mindestens $1/2 + \varepsilon$ der Elemente von QR$_N$ als Eingabe x_0 korrekt geschätzt werden kann.

Satz 6.3: Ein ε-Vorteil zum Schätzen der Parität kann effizient und gleichmäßig in einen ε-Vorteil zum Schätzen der Quadratrest-Eigenschaft umgewandelt werden.

Beweis:
Sei $x \in Z_N^*(+1)$ ein Element, dessen Quadratrest-Eigenschaft mod$N$ herausgefunden werden soll. Sei $x_0 = x^2 \bmod N$. Da $P \equiv Q \equiv 3 \bmod 4$, sind die Wurzeln von x_0 aus $Z_N^*(+1)$ x und $N - x$ (nach Satz 6.1). Da N ungerade ist, haben die Wurzeln entgegengesetze Parität. Genau eine dieser Wurzeln ist Quadratrest (d.h. gleich x_{-1}) und nur diese hat dieselbe Parität wie x_{-1}. Deshalb entspricht das Schätzen der Parität dem Schätzen der Quadratrest-Eigenschaft.

qed.

Satz 6.4: Ein ε-Vorteil mit nicht vernachlässigbar kleinem ε für das Schätzen der Quadratrest-Eigenschaft kann effizient und gleichmäßig zu einem $(1/2 - \delta)$ -Vorteil mit nicht vernachlässigbar kleinem δ ausgebaut werden.

Dazu soll hier nur eine Beweisskizze gegeben werden [GoldMic82].
Sei **P** die probabilistische *poly*-Prozedur für das Schätzen der Quadratrest-Eigenschaft. In polynomialer Zeit läßt sich die durchschnittliche Wahrscheinlichkeit A, daß **P** einen Quadratrest aus QR_N richtig, und die durchschnittliche Wahrscheinlichkeit B, daß **P** einen Nicht-Quadratrest falsch schätzt, ungefähr berechnen. Dabei muß $A - B \geq 2\varepsilon$ gelten. Sei $x \in Z_N^*(+1)$ das Element, dessen QR-Eigenschaft herausgefunden werden soll. In Abhängigkeit von ε und δ bestimmt man k (dem schwachen Gesetz der großen Zahlen folgend), und wählt k-mal zufällig ein $r \in Z_N^*$ aus, berechnet $x \cdot r^2 \bmod N$ und läßt von **P** die QR-Eigenschaft bestimmen (die gleich der von x ist). Nach den k Durchläufen vergleicht man den Bruchteil, für den **P** einen Quadratrest ausgegeben hat, mit A und B und erhält damit den Vorteil $(1/2 - \delta)$ im Schätzen.
Behauptung 3 folgt: Gäbe es eine Prozedur **P**, die die Parität mit ε-Vorteil schätzen kann (ohne die Primfaktoren P und Q zu kennen), läßt sich diese in **P'** umwandeln, die das Quadratrestproblem mit Vorteil $1/2 - \delta$ löst. Dies widerspricht QRA aus Abschnitt 4.
Dem Satz 6.5 wird vorausgeschickt

Definition 6.2: Ein *Schätzer* **P** ist eine *poly*-Prozedur, die auf Eingabe N und $b_1 \ldots b_k$ mit $b_i \in \{0,1\}$ und $k < poly(|N|)$ entweder 0 oder 1 ausgibt. **P** hat einen *ε-Vorteil* für das Voraussagen von Folgen, produziert vom $x^2 \bmod N$-Generator, nach links genau dann, wenn mit $k < poly(|N|)$ für mehr als $1/2 + \varepsilon$ aller Elemente x von QR_N auf Eingabe $N, b_1(x) \ldots b_k(x)$ Parität $b_0(x)$ korrekt geschätzt werden kann (mit $b_i(x) = (x^{2^i} \bmod N) \bmod 2$).

Satz 6.5: Gemäß der Quadratrest-Annahme ist der $x^2 \bmod N$ - Generator ein nicht vorhersagbarer Pseudo-Zufallsgenerator. D.h. jeder Schätzer **P** hat höchstens für einen beliebig kleinen Bruchteil der N wie vorgeschrieben einen $1/n^t$ -Vorteil für ausreichend große n und beliebigem $t \in \mathbb{N}$, Folgen des $x^2 \bmod N$ - Generator nach links vorherzusagen.

Beweis:
Angenommen, **P** sei ein Schätzer mit ε-Vorteil für den $x^2 \bmod N$-Generator. Dieser kann in eine Prozedur mit ε-Vorteil für das Schätzen der Parität wie in Def. 6.1 umgewandelt werden und diese anschließend in eine Prozedur zum Schätzen der Quadratrest-Eigenschaft. Schließlich läßt sich der Vorteil vergrößern nach Satz 6.4. Damit erhält man den Widerspruch zu QRA.

qed.

Nach einem Fundamentalsatz von Yao [Yao82] besteht eine vom $x^2 \bmod N$-Generator erzeugt Folge damit jeden in polynomialer Zeit laufenden statistischen Test auf Zufälligkeit.

Hier soll nur noch angemerkt werden, daß - würde dies nicht der Fall sein - ein solcher Test in einen Schätzer – wie oben – umgewandelt werden könnte, der nach Satz 6.5 der Quadratrest-Annahme widerspricht. Exakte Ausführungen dazu in [Blum86].

14.6.2 Periodenlänge der erzeugten Folgen

$\pi(x_0)$ gebe hier die Periode der erzeugten Folge $x_0, x_1, \ldots$ mit $x_i = x_0^{2^i} \bmod N$ an. Um die genaue Länge einer Periode angeben zu können, zeigen ich zunächst, daß $\pi(x_0)|\lambda(\lambda(N))$. Dazu die Definition der λ-Funktion:

Definition 6.3: Sei $M = 2^e \cdot P_1^{e_1} \cdot \ldots \cdot P_k^{e_k}$, wobei $P_1, \ldots P_k$ unterschiedliche, ungerade Primzahlen sind. Carmichael's λ-Funktion ist definiert durch

$$\lambda(2^e) = \begin{cases} 2^{e-1} \text{ für } e = 1, e = 2 \\ 2^{e-2} \text{ für } e > 2 \end{cases}$$

und $\lambda(M) = \text{kgV}[\ \lambda(2^e), (P_1 - 1) \cdot P_1^{e_1-1}, \ldots, (P_k - 1) \cdot P_k^{e_k-1}\ ]$.

$\lambda(M)$ ist kgV und Supremum der Ordnungen von Z_M^* [Knuth81] und $a^{\lambda(M)} \equiv 1 \bmod M$ für $a \in Z_M^*$.

Satz 6.6: Sei N von vorgeschriebener Form und x_0 Quadratrest mod$N$. $\pi = \pi(x_0)$ sei Periode der Folge $x_0, x_1, \ldots$. Dann gilt: $\pi|\lambda(\lambda(N))$.

Beweis:
Für $x \in Z_N^*(+1)$ ist $\text{ord}_N x$ ungerade, da
(1) $\text{ord}_N x_i = \text{ord}_N x_{i+1}$, da $\text{ord}_N x_{i+1}|\text{ord}_N x_i$ und $x_0, x_1, \ldots$ istzyklisch;
(2) für alle $u \in \mathbb{N}$ gilt: $2^u \| \text{ord}_N x_i \Rightarrow 2^{u-1} \| \text{ord}_N x_{i+1}$. (Hier bedeutet $2^u \| \text{ord}_N x_i$, daß $2^u | \text{ord}_N x_i$, aber nicht $2^{u+1} | \text{ord}_N x_i$). Dies kann aber nach (1) für kein u der Fall sein.
Deshalb gilt: $2^{\lambda(\text{ord}_N x_0)} \equiv 1 \bmod (\text{ord}_N x_0)$. Aber π ist die kleinste positive Ganzzahl mit $2^\pi \equiv 1 \bmod (\text{ord}_N x_0)$, da π die kleinste positive Ganzzahl ist, so daß $x_0 = x_0^{2^\pi} \bmod N$. Deshalb gilt $\pi|\lambda(\text{ord}_N x_0)$.
Aber $\lambda(\text{ord}_N x_0)|\lambda(\lambda(N))$, da $\text{ord}_N x_0|\lambda(N)$ für $x_0 \in Z_N^*$. Aus diesem Grunde gilt $\pi|\lambda(\lambda(N))$.

qed.

Jetzt fehlt noch, daß $\pi|\lambda(\lambda(N))$. Das gilt jedoch nur unter bestimmten Voraussetzungen:

Satz 6.7: Sei N von vorgeschriebener Form und x_0 Quadratrest mod$N$. $\pi = \pi(x_0)$ sei Periode der Folge $x_0, x_1, \ldots$. (1) Wähle N so, daß $\operatorname{ord}_{\lambda(N)/2} 2 = \lambda(\lambda(N))$. (2) Wähle x_0 so, daß $\operatorname{ord}_N x_0 = \lambda(N)/2$. Dann gilt $\pi | \lambda(\lambda(N))$ und nach Satz 6.6 $\pi = \lambda(\lambda(N))$.

Beweis:
$\lambda(N)/2$ ist die kleinste positive Ganzzahl, sodaß $x_0^{\lambda(N)/2} \equiv 1 \bmod N$ nach (2). Aber π ist die kleinste positive Ganzzahl, sodaß $x_0 = x_0^{2^\pi} \bmod N$, womit $x_0^{2^\pi - 1} \equiv 1 \bmod N$. Damit gilt $\lambda(N)/2 | 2^\pi - 1$, und somit ist $2^\pi \equiv 1 \bmod \lambda(N)/2$. Nach (1) ist $\lambda(\lambda(N))$ die kleinste positive Ganzzahl, sodaß $2^{\lambda(\lambda(N))} \equiv 1 \bmod \lambda(N)/2$. Daraus folgt, daß $\lambda(\lambda(N)) | \pi$.

qed.

Bedingung (2) gilt für einen Großteil der Quadratreste in Z_N^*, genauer für mindestens $\Omega(N/4(\ln \ln N)^2)$ [Blum86, S.378]. Für Bedingung 1 möchte ich nur Voraussetzungen angeben, unter denen diese gilt.

Definition 6.4: P ist eine *besondere Primzahl*, wenn $P = 2P_1 + 1$ und $P_1 = 2P_2 + 1$ mit $P1, P2 \in \mathbb{P}$.

Satz 6.8: Ist $N = PQ$ mit verschiedenen, besonderen Primzahlen $\equiv 3 \bmod 4$ und ist 2 Quadratrest mod P_1 *oder* mod Q_1, so gilt $\operatorname{ord}_{\lambda(N)/2} 2 = \lambda(\lambda(N))$.

Der Beweis dieses Satzes ist ebenfalls in [Blum86] nachzulesen.

14.6.3 Zugehörige Algorithmen

Im Satz 6.7 wurden gewisse Bedingungen gestellt, damit $\pi = \lambda(\lambda(N))$ gilt; folgender Algorithmus berechnet π, ohne daß Vorbedingungen gelten müssen.

Satz 6.9: Es existiert ein effizienter Algorithmus **A**, der bei Eingabe von N in vorgeschriebener Form[5], $\lambda(N)$, $\lambda(\lambda(N))$ und der Faktorisierung von $\lambda(\lambda(N))$ die Periode $\pi(x_0)$ für jedes $x_0 \in \mathrm{QR}_N$ liefert.

Beweis:
Bekanntlich gilt $x_i = x_0^{2^i} \bmod N$, $x_\pi = x_0^{2^\pi} mod N = x_0$ und $\pi |\ \lambda(\lambda(N))$ nach Satz 6.6. Deshalb ist $x_0^{2^{\lambda(\lambda(N))}} \equiv x_0 \bmod N$.
Nach Carmichael's Erweiterung des Eulerschen Satzes ist $x_0^{\lambda(N)} \equiv 1 \bmod N$ und damit ist $x_0^{2^{\lambda(\lambda(N))} + k\lambda(N)} \equiv x_0 \bmod N$ für jede ganze Zahl k.
Deshalb $x_0^{2^{\lambda(\lambda(N))} mod \lambda(N)} \equiv x_0 \bmod N$. Damit läßt sich mit den Eingaben von **A** effizient auf π schließen: Suche nach dem größten $d | \lambda(\lambda(N))$, sodaß
$x_0^{2^{\lambda(\lambda(N))/d} mod \lambda(N)} \equiv x_0 \bmod N$. Dann ist $\pi = \lambda(\lambda(N))/d$.

---

[5] $N = P \cdot Q$, wobei P, Q Primzahlen $\equiv 3 \bmod 4$ sind

qed.

Nun folgen noch zwei Algorithmen, um die Folge nach vorwärts und rückwärts zu erweitern.

Satz 6.10a: Es existiert ein effizienter Algorithmus **A**, der bei Eingabe von N (wie vorgeschrieben), $\lambda(N)$, $x_0 \in \mathrm{QR}_N$ und $i \in \mathbb{N}$ das Folgeglied x_i ausgibt.

Beweis:
$x_i = x_0^{2^i \, mod \, \lambda(N)} \bmod N$.
Bemerkung: Die Anzahl der Schritte beträgt bei Benutzung der „schnellen Multiplikation“ $O(|N|^{2+\varepsilon})$.

Satz 6.10b: Es existiert ein effizienter Algorithmus **A**, der bei Eingabe von N (wie vorgeschrieben), $\lambda(N)$, $x_0 \in \mathrm{QR}_N$ und $i \in \mathbb{N}$ das Folgeglied x_{-i} ausgibt (man beachte den negativen Index !).

Beweis:
Nach [Miller75] läßt sich N effizient faktorisieren, wenn man $\lambda(N)$ kennt ($\lambda(N) = \mathrm{kgV}\,[P-1, Q-1]$). Nun läßt sich $\sqrt{x_0} \bmod P = \pm x_0^{(P+1)/4} \bmod P$ berechnen, wie im Beweis von Satz 6.2 gezeigt. Genau eine dieser Wurzeln ist Quadratrest wegen 2.2(9) und $\left(\frac{-1}{P}\right) = -1$. Also ist $x_{-1} \bmod P$ entweder $+x_0^{(P+1)/4}$ oder $-x_0^{(P+1)/4}$.
Ähnlich gilt wegen $(-1)^{(P+1)/4} = \pm 1$: $x_{-2} \bmod P$ ist entweder $+x_0^{((P+1)/4)^2}$ oder $-x_0^{((P+1)/4)^2}$. Fährt man so fort, erhält man, daß $x_{-i} \bmod P$ entweder $+x_0^{((P+1)/4)^i}$ oder $-x_0^{((P+1)/4)^i}$, was effizient berechnet werden kann (über das Legendre-Symbol).
In gleicher Weise erhält man $x_{-i} \bmod Q$, und der chinesische Restsatz erlaubt es (mit Hilfe des erweiterten Euklidischen Algorithmus), $x_{-i} \bmod N$ effizient zu berechnen.

qed.

Damit will ich die ausführliche Behandlung des $x^2 \bmod N$-Generators abschließen.

14.7 Schlußbemerkungen

Hier wurden zwei Pseudo-Zufallsgeneratoren vorgestellt, die auf den ersten Blick „gute“ Folgen liefern, nach eingehender Betrachtung jedoch völlig verschiedene Eigenschaften besitzen.
Begnügt man sich damit, eine gute Verteilung im Sinne von De Bruijn-Folgen zu bekommen, ist aber nicht an Unvorhersagbarkeit interessiert, erweist sich

der $1/P$-Generator als gute Wahl. Dies kann durchaus ausreichend als Pseudo-Zufallsquelle für probabilistische Algorithmen sein. „Hintergrundrauschen" läßt sich mit diesem Generator ebenfalls perfekt generieren.
Der $x^2 \bmod N$-Generator hat wegen seiner Nicht-Vorhersehbarkeit deutlich mehr Anwendungsfelder. Hier will ich wegen seiner speziellen Eigenschaft vor allem die Verwendung in der Kryptologie, im speziellen in Public-Key-Systemen erwähnen. Öffentlicher Schlüssel von A sei N_A, die Faktorisierung von N_A sei nur A selbst bekannt. Will B eine Nachricht m an A schicken, erzeugt er mit zufälligem $x_0 \in Z^*_{N_A}$ ein „One-Time-Pad", indem er die Folge $b_0, b_1, \ldots b_k$ berechnet, wobei k die Länge der Nachricht m ist. Anschließend sendet er die Exklusiv-Oder Verknüpfung seiner Nachricht mit dem One-Time-Pad, zusammen mit x_{k+1}. Nur A kann die Folge rückwärts expandieren und damit $b_0, b_1, \ldots b_k$ berechnen, womit er die ursprüngliche Nachricht m erhält.

Das war's ... und noch viel Spaß in der Ferienakademie.

Weidendorfer Josef

Literaturverzeichnis

[Blum86] L.Blum, M.Blum, M.Shub, *A simple unpredictable pseudo-random number generator*, SIAM Computing Vol.15 No.2 May 1986

[Bund91] P.Bundschuh, *Einführung in die Zahlentheorie*, 2.Auflage, Springer-Verlag 1991

[GoldMic82] S.Goldwasser, S.Micali, *Probabilistic encryption and how to play mental poker keeping secret all partial inforamtion* , 14th STOC 1982, S.365-377

[Gundlach72] K.-B. Gundlach, *Einführung in die Zahlentheorie*, B.I., Mannheim 1972

[Knuth81] D.Knuth, *The Art of Computer Programming, Vol.2, Seminumerical Algorithms*, Addison-Wesley, MA, 1981

[LeVeque77] W.LeVeque, *Fundamentals of Number Theory*, Addison-Wesley, MA, 1977

[Miller75] G.Miller, *Riemann's hypothesis and tests for primality*, Ph.D. thesis, Univ.California, Berkeley, 1975

[Shanks76] D.Shanks, *Solved und Unsolved Problems in Number Theory*, Chelsea, New York 1976

[Yao82] A. Yao, *Theory and applications of trapdoor functions*, IEEE 23rd Symp. on Foundations of Computer Science 1982, S. 80-91