

Algorithmik II

Markus Lohrey

Universität Siegen

Wintersemester 2019/2020

Konvolution von Polynomen

Betrachte zwei Polynome (mit Koeffizienten etwa aus \mathbb{C}):

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n, \quad g(x) = b_0 + b_1x + b_2x^2 + \cdots + b_mx^m$$

repräsentiert durch ihre Koeffizientenfolgen

$$f = (a_0, \dots, a_n, a_{n+1}, \dots, a_{N-1}), \quad g = (b_0, \dots, b_m, b_{m+1}, \dots, b_{N-1})$$

wobei $N = n + m + 1$, $a_{n+1} = \cdots = a_{N-1} = b_{m+1} = \cdots = b_{N-1} = 0$.

Wir wollen das Produktpolynom

$$(fg)(x) = a_0b_0 + (a_1b_0 + a_0b_1)x + \cdots + (a_0b_{N-1} + \cdots + a_{N-1}b_0)x^{N-1}$$

berechnen, welches durch die Koeffizientenfolge

$$fg = (a_0b_0, a_1b_0 + a_0b_1, \dots, a_0b_{N-1} + \cdots + a_{N-1}b_0),$$

(die **Konvolution** der Folgen f und g) repräsentiert wird.

Punktrepräsentation von Polynomen

Naive Berechnung von fg : $\mathcal{O}(N^2)$ skalare Operationen

FFT (nach James Cooley und John Tukey, 1965) reduziert die Zeit auf $\mathcal{O}(N \log(N))$

Grundidee: Punktrepräsentation von Polynomen

Ein Polynom f vom Grad $N - 1$ kann eindeutig durch die Folge der Werte

$$(f(\zeta_0), f(\zeta_1), \dots, f(\zeta_{N-1}))$$

repräsentiert werden, wobei $\zeta_0, \dots, \zeta_{N-1}$ N verschiedene Werte aus dem Grundbereich (z.B. komplexe Zahlen), sind.

Offensichtlich gilt $(fg)(\zeta) = f(\zeta)g(\zeta) \rightsquigarrow$

Die Punktrepräsentation der Konvolution von f und g kann in Zeit $\mathcal{O}(N)$ aus den Punktrepräsentationen von f und g berechnet werden.

Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):

Koeffizientenrepr. von f und g

Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):

Koeffizientenrepr. von f und g

Auswertung

Punktrepr. f und g

Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):

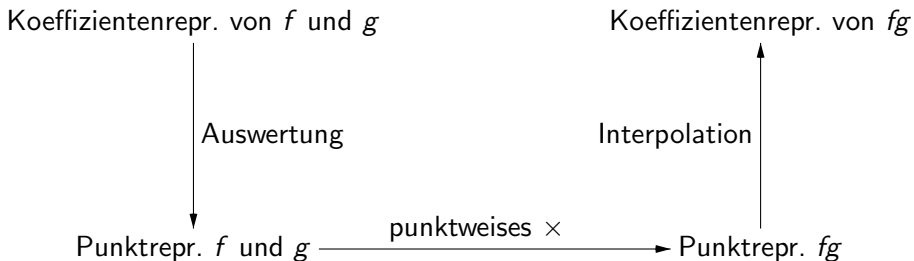
Koeffizientenrepr. von f und g

Auswertung

Punktrepr. f und g $\xrightarrow{\text{punktweises } \times}$ Punktrepr. fg

Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):



Einheitswurzeln

Der entscheidende Punkt ist die Auswahl der Punkte $\zeta_0, \zeta_1, \dots, \zeta_{N-1}$.

Annahme: Die Koeffizienten der Polynome stammen aus einem Körper \mathbb{F} , so dass gilt:

- N hat ein multiplikatives Inverses in \mathbb{F} , d.h. die Charakteristik von \mathbb{F} teilt nicht N .
- Das Polynom $X^N - 1$ hat N verschiedene Nullstellen – die **N -ten Einheitswurzeln** – welche sich als ω^i ($0 \leq i < N$) für eine Nullstelle ω schreiben lassen.

Für $\mathbb{F} = \mathbb{C}$ sind die N -ten Einheitswurzeln etwa von der Form ω^j ($0 \leq j < N$), wobei $\omega = e^{\frac{2\pi i}{N}}$.

Die Nullstelle ω wird auch als **primitive N -te Einheitswurzel** bezeichnet.

Einheitswurzeln

Einige nützliche Fakten aus der Algebra:

Sei ω eine primitive N -te Einheitswurzel.

- Für alle $i, j \in \mathbb{Z}$ gilt: $\omega^i = \omega^j$ g.d.w. $i \equiv j \pmod{N}$.
- Für $i \in \mathbb{Z}$ ist ω^i eine primitive N -te Einheitswurzel g.d.w. $\text{ggT}(i, N) = 1$ (wobei $\text{ggT}(i, N)$ der größte gemeinsame Teiler von i und N ist).
- Insbesondere ist $\omega^{-1} = \omega^{N-1}$ eine primitive N -te Einheitswurzel.

Schnelle Fourier Transformation (FFT)

Fixiere eine primitive N -te Einheitswurzel ω .

Wir wählen die Punkte $\zeta_i = \omega^i$ ($0 \leq i \leq N-1$) für die Auswertung von f und g .

Auswertung von $f = a_0 + a_1x + \dots + a_{N-1}x^{N-1}$ an den Punkten $\omega^0 = 1, \omega^1, \dots, \omega^{N-1}$ läuft auf eine Matrixmultiplikation hinaus:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} f(1) \\ f(\omega) \\ f(\omega^2) \\ \vdots \\ f(\omega^{N-1}) \end{pmatrix}$$

Die durch die Matrix $F_N(\omega) = (\omega^{ij})_{0 \leq i, j < N}$ realisierte lineare Abbildung wird als **diskrete Fourier Transformation** bezeichnet.

Inverse FFT

Lemma 1

$(F_N(\omega))^{-1} = \frac{1}{N}F_N(\omega^{-1})$, d.h. das Inverse der Matrix $(\omega^{ij})_{0 \leq i,j < N}$ ist $(\frac{\omega^{-ij}}{N})_{0 \leq i,j < N}$ (beachte: ω^{-1} ist eine primitive N -te Einheitswurzel).

Beweis: Da $x^N - 1 = (x - 1) \cdot \sum_{j=0}^{N-1} x^j$ gilt, ist jedes ω^i für $0 < i < N$ eine Nullstelle von $\sum_{j=0}^{N-1} x^j$.

Daher gilt für alle $0 \leq i < N$:

$$\sum_{j=0}^{N-1} \omega^{i \cdot j} = \begin{cases} 0 & \text{falls } i > 0 \\ N & \text{falls } i = 0 \end{cases}$$

Wir erhalten für alle $0 \leq i, j < N$:

$$\sum_{k=0}^{N-1} \omega^{ik} \omega^{-kj} = \sum_{k=0}^{N-1} \omega^{k(i-j)} = \begin{cases} 0 & \text{falls } i \neq j \\ N & \text{falls } i = j \end{cases}$$

FFT mittels Divide & Conquer

Wir müssen nun noch die diskrete Fouriertransformation

$$f \mapsto F_N(\omega)f$$

(wobei $f = (a_0, a_1, \dots, a_{N-1})^T$) in Zeit $\mathcal{O}(N \log(N))$ berechnen.

Dann kann die inverse diskrete Fouriertransformation (= Interpolation)

$$h \mapsto (F_N(\omega))^{-1}h = \frac{1}{N}F_N(\omega^{-1})h$$

in der gleichen Zeitschranke berechnet werden.

Die “Schulmethode” für die Multiplikation einer Matrix mit einem Vektor benötigt Zeit $\mathcal{O}(N^2)$: kein Gewinn gegenüber der “Schulmethode” für Polynommultiplikation.

Wir berechnen $F_N(\omega)f = (\omega^{ij})_{0 \leq i, j < N}f$ mittel Divide & Conquer.

FFT mittels Divide & Conquer

Angenommen N ist gerade. Für $f(x) = a_0 + a_1x + \dots + a_{N-1}x^{N-1}$ sei

$$f_0(x) = a_0 + a_2x^2 + a_4x^4 + \dots + a_{N-2}x^{N-2}$$

$$\widehat{f}_0(x) = a_0 + a_2x + a_4x^2 + \dots + a_{N-2}x^{\frac{N-2}{2}}$$

$$f_1(x) = a_1 + a_3x^2 + a_5x^4 + \dots + a_{N-1}x^{N-2}$$

$$\widehat{f}_1(x) = a_1 + a_3x + a_5x^2 + \dots + a_{N-1}x^{\frac{N-2}{2}}$$

Also: $f(x) = f_0(x) + xf_1(x)$, $f_0(x) = \widehat{f}_0(x^2)$ und $f_1(x) = \widehat{f}_1(x^2)$.

Die Polynome $\widehat{f}_0(x)$ und $\widehat{f}_1(x)$ haben Grad $\leq \frac{N-2}{2}$.

Sei $0 \leq i < N$. Da ω^2 eine primitive $\frac{N}{2}$ -Einheitswurzel ist, gilt:

$$\begin{aligned} (F_N(\omega)f_0)_i &= f_0(\omega^i) = \widehat{f}_0(\omega^{2i}) = \widehat{f}_0(\omega^{2i \bmod N}) = \\ &= \widehat{f}_0(\omega^{2(i \bmod \frac{N}{2})}) = \widehat{f}_0((\omega^2)^{i \bmod \frac{N}{2}}) = (F_{\frac{N}{2}}(\omega^2)\widehat{f}_0)_{i \bmod \frac{N}{2}}, \end{aligned}$$

FFT mittels Divide & Conquer

Wir erhalten

$$F_N(\omega)f_0 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2)\widehat{f}_0 \\ F_{\frac{N}{2}}(\omega^2)\widehat{f}_0 \end{pmatrix}$$

und analog

$$F_N(\omega)f_1 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2)\widehat{f}_1 \\ F_{\frac{N}{2}}(\omega^2)\widehat{f}_1 \end{pmatrix}.$$

Mit $f(x) = f_0(x) + xf_1(x)$ folgt

$$F_N(\omega)f = F_N(\omega)f_0 + (F_N(\omega)x \circ F_N(\omega)f_1),$$

wobei $F_N(\omega)x = (1, \omega, \omega^2, \dots, \omega^{N-1})^T$ und “ \circ ” die punktweise Multiplikation von Vektoren bezeichnet.

FFT mittels Divide & Conquer

Wir haben somit die Berechnung von $F_N(\omega)f$ reduziert auf:

- Die Berechnung von $F_{\frac{N}{2}}(\omega^2)\hat{f}_0$ und $F_{\frac{N}{2}}(\omega^2)\hat{f}_1$
(2 FFTs der Dimension $N/2$)
- $\mathcal{O}(N)$ viele weitere arithmetische Operationen.

Wir erhalten somit die Rekursionsgleichung

$$T_{\text{fft}}(N) = 2T_{\text{fft}}(N/2) + dN$$

für eine Konstante d .

Mastertheorem I ($a = b = 2, c = 1$):

$$T_{\text{fft}}(N) \in \theta(N \log N).$$

Schnelle Fourier Transformation (FFT)

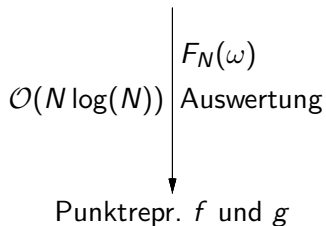
Grundprinzip der FFT:

Koeffizientenrepr. von f und g

Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g



Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

$\mathcal{O}(N \log(N))$ $F_N(\omega)$
Auswertung

Punktrepr. f und g $\xrightarrow[\mathcal{O}(N)]{\text{punktweises } \times}$ Punktrepr. fg

Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

$\mathcal{O}(N \log(N))$ Auswertung
 $F_N(\omega)$

Punktrepr. f und g

punktwises \times
 $\mathcal{O}(N)$

Punktrepr. fg

Koeffizientenrepr. von fg

Interpolation $\mathcal{O}(N \log(N))$
 $\frac{1}{N} F_N(\omega^{-1})$

FFT über endlichen Körper

Angenommen, wir wollen mittels FFT Polynome mit ganzzahligen Koeffizienten multiplizieren.

Problem, wenn wir im Körper \mathbb{C} rechnen: Rundungsfehler bei Rechnen mit N -ten Einheitswurzeln.

Erinnerung: FFT funktioniert für jeden Körper \mathbb{F} mit folgenden Eigenschaften:

- N hat ein multiplikatives Inverses in \mathbb{F} , d.h. die Charakteristik von \mathbb{F} teilt nicht N .
- Es gibt eine primitive N -te Einheitswurzel.

Lemma 2

Sei p eine Primzahl. Der Körper \mathbb{F}_p hat eine N -te primitive Einheitswurzel, falls N ein Teiler von $p - 1$ ist.

FFT über endlichen Körper

Lösung des Rundungsproblem:

- Angenommen alle Koeffizienten des Produktpolynoms fg liegen in $[-d, d]$.
- $N = 2^m$ ist vorteilhaft für die FFT.
- Suche eine Primzahl p von der Gestalt $p = 2^m \cdot k + 1$ für ein $k \geq 1$ (Fourier Primzahl), so dass $p > 2d$ (es gibt viele davon, siehe z.B. <http://www.csd.uwo.ca/~moreno/CS874/Lectures/Newton2Hensel.html/node9.html>).
- Da $\text{ggT}(N, p) = 1$ gilt, hat N ein multiplikatives Inverses in \mathbb{F}_p . Wir können also die FFT in \mathbb{F}_p machen.
- Dies liefert die Koeffizienten von fg modulo p , woraus (wegen $p > 2d$) die Koeffizienten in \mathbb{Z} berechnet werden können.

Weitere Anwendungen der FFT

- Schnelle Multiplikation von ganzen Zahlen, beginnend mit Schönhage-Strassen (Laufzeit: $\mathcal{O}(n \log n \log \log n)$).
Neueste Entwicklung (Harvey, van der Hoeven, März 2019):
Multiplikation von n -bit Zahlen in Zeit $\mathcal{O}(n \log n)$, siehe
<https://web.maths.unsw.edu.au/~davidharvey/papers/nlogn/>.
- Filter-Algorithmen
- Schnelle Algorithmen für diskrete Kosinus- und Sinustransformation (wird z.B. in JPEG und MP3/MPEG verwendet).

FFT findet sich auf einer Liste der 10 wichtigsten Algorithmen des 20. Jahrhunderts: <https://cs.gmu.edu/~henryh/483/top-10.html>.

How can we test whether $A \cdot B = C$?

The best current algorithm for multiplying two $(n \times n)$ -matrices needs approx. $\Theta(n^{2,372873\dots})$ many arithmetic operations (Virginia Vassilevska Williams 2014).

Conjecture

For every $\epsilon > 0$ there exists an algorithm for multiplying two $(n \times n)$ -matrices in time $O(n^{2+\epsilon})$.

Assume now that we have three $(n \times n)$ -matrices A , B and C .

How many arithmetic operations are needed to test whether $A \cdot B = C$ holds?

Trivial answer: $O(n^{2,372873\dots})$

But there is a better method!

How can we test whether $A \cdot B = C$?

Satz 3 (Korec, Wiedermann 2014)

Let A, B, C be $(n \times n)$ -matrices with entries from \mathbb{Z} . Using $O(n^2)$ many operations we can check whether $A \cdot B = C$ holds.

Proof: Let

$$A = (a_{i,j})_{1 \leq i,j \leq n},$$

$$B = (b_{i,j})_{1 \leq i,j \leq n},$$

$$C = (c_{i,j})_{1 \leq i,j \leq n} \text{ and}$$

$$D = (d_{i,j})_{1 \leq i,j \leq n} = A \cdot B - C.$$

Thus, we have $A \cdot B = C$ if and only if D is the zero-matrix.

Let x be real-valued variable and consider the column-vector

$$v = (1, x, x^2, \dots, x^{n-1})^T.$$

How can we test whether $A \cdot B = C$?

Hence, $D \cdot v = A \cdot B \cdot v - C \cdot v$ is a column-vector whose i -th entry is the polynomial

$$p_i(x) = d_{i,1} + d_{i,2}x + d_{i,3}x^2 + \cdots + d_{i,n}x^{n-1}.$$

We therefore have $A \cdot B = C$ if and only if $p_i(x)$ is the zero polynomial for all $1 \leq i \leq n$.

We use the following theorem:

Cauchy bound

Let $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{R}[x]$ be not the zero-polynomial and $a_n \neq 0$. For every α with $p(\alpha) = 0$ we have

$$|\alpha| < 1 + \frac{\max\{|a_i| \mid 0 \leq i \leq n-1\}}{|a_n|}.$$

How can we test whether $A \cdot B = C$?

Proof of the Cauchy bound:

By replacing $p(x)$ by the polynomial $x^n + \frac{a_{n-1}}{a_n}x^{n-1} + \dots + \frac{a_1}{a_n}x + \frac{a_0}{a_n}$, it suffices to prove the Cauchy bound for the case $a_n = 1$.

Let $p(x) = x^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ and

$$h = \max\{|a_i| \mid 0 \leq i \leq n-1\}.$$

Assume that $p(\alpha) = \alpha^n + a_{n-1}\alpha^{n-1} + \dots + a_1\alpha + a_0 = 0$, i.e.,

$$\alpha^n = -a_{n-1}\alpha^{n-1} - \dots - a_1\alpha - a_0.$$

We show that $|\alpha| < 1 + h$.

If $|\alpha| \leq 1$, we have $|\alpha| < 1 + h$ (if $h = 0$ then we must have $\alpha = 0$).

Now assume that $|\alpha| > 1$.

How can we test whether $A \cdot B = C$?

We get

$$\begin{aligned} |\alpha|^n &\leq |a_{n-1}| \cdot |\alpha|^{n-1} + \cdots + |a_1| \cdot |\alpha| + |a_0| \\ &\leq h \cdot (|\alpha|^{n-1} + \cdots + |\alpha| + 1) \\ &= h \cdot \frac{|\alpha|^n - 1}{|\alpha| - 1}. \end{aligned}$$

Since $|\alpha| > 1$, we obtain:

$$|\alpha| - 1 \leq h \cdot \frac{|\alpha|^n - 1}{|\alpha|^n} < h$$



How can we test whether $A \cdot B = C$?

Let $a = \max\{|a_{i,j}| \mid 1 \leq i, j \leq n\}$, $b = \max\{|b_{i,j}| \mid 1 \leq i, j \leq n\}$ and $c = \max\{|c_{i,j}| \mid 1 \leq i, j \leq n\}$.

The absolute values of the coefficients of the polynomials $p_i(x)$ can be bounded as follows:

$$|d_{i,j}| = \left| \sum_{k=1}^n a_{i,k} b_{k,j} - c_{i,j} \right| \leq \sum_{k=1}^n |a_{i,k}| \cdot |b_{k,j}| + |c_{i,j}| \leq n \cdot a \cdot b + c.$$

Let $d = n \cdot a \cdot b + c$ and $r = 1 + d$.

The Cauchy bound yields for all $1 \leq i \leq n$:

$$p_i(x) = 0 \Leftrightarrow p_i(r) = 0$$

Note: It is easy to get an upper bound d for the absolute values of the entries of D , but it is not so clear how to get a lower bound.

How can we test whether $A \cdot B = C$?

But: Since A, B, C are matrices over \mathbb{Z} , we have $p_i(x) \in \mathbb{Z}[x]$.

Therefore, the absolute value of the leading coefficient of $p_i(x)$ is at least 1, if $p_i(x)$ is not the zero-polynomial.

We can therefore check with the following algorithm, whether $A \cdot B = C$:

- 1 Compute $a = \max\{|a_{i,j}|\}$, $b = \max\{|b_{i,j}|\}$, $c = \max\{|c_{i,j}|\}$ and $r = 1 + n \cdot a \cdot b + c$
($3n^2$ many comparisons, 2 additions, 2 multiplications)
- 2 Compute the column vector $u = (1, r, r^2, \dots, r^{n-1})^T$.
($n - 2$ multiplications)
- 3 Compute $p := B \cdot u$, $s := A \cdot p$ and $t := C \cdot u$
($O(n^2)$ many arithmetic operations)
- 4 $A \cdot B = C$ if and only if $s = t$.

Parallele Algorithmen und NC

- 1 Einführung in parallele Architekturen
- 2 Die Klasse NC und parallele Matrix-Multiplikation
- 3 Parallele Berechnung der Präfixsummen
- 4 Parallele Integer-Addition, -Multiplikation und -Division
- 5 Parallele Berechnung der Determinante

Einführung in parallele Architekturen

- Parallele Random-Access-Maschinen (PRAM)
 - CRCW (Concurrent Read Concurrent Write)
 - CREW (Concurrent Read Exclusive Write)
 - EREW (Exclusive Read Exclusive Write)
 - ERCW (Exclusive Read Concurrent Write)
- Vektor-Maschinen
 - SIMD oder MIMD
- Boolesche und arithmetische Schaltkreise
 - DAG mit Ein- und Ausgabeknoten
 - Knoten für grundlegende bitweise und arithmetische Operationen

Die Klasse NC

NC bezeichnet die Klasse aller Probleme, die “effizient parallelisierbar” sind.

NC steht für „Nick’s Class“ (benannt nach Nick Pippenger).

Definition(en):

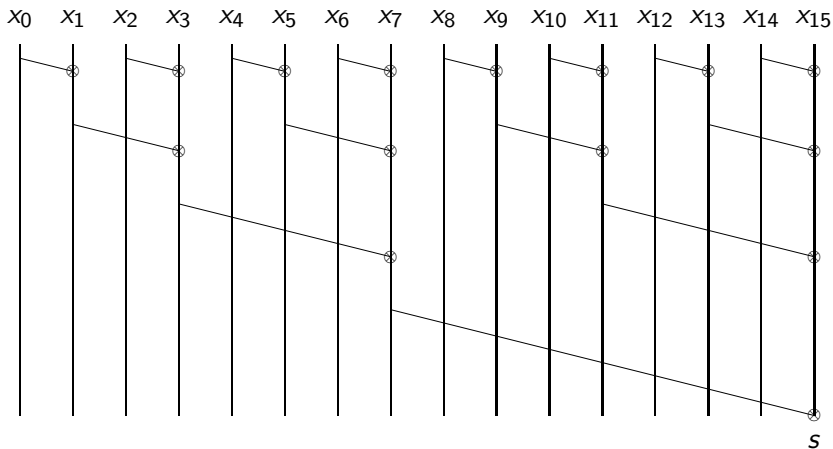
- Probleme, die mit einer PRAM mit $n^{O(1)}$ Prozessoren in Zeit $(\log n)^{O(1)}$ lösbar sind.
- Probleme, die mit einem bool’schen Schaltkreis der Tiefe $(\log n)^{O(1)}$ und Größe $n^{O(1)}$ lösbar sind.

Die Klasse ist robust gegenüber kleinen Änderungen am Maschinen-Modell.

Die Frage $NC \stackrel{?}{=} P$ ist noch offen.

Es gibt P -vollständige Probleme (z.B. das Auswerten boolescher Schaltkreise), von denen vermutet wird, dass sie nicht zu NC gehören.

Berechne die Summe $s = \sum_{i=0}^n x_i$



Summe $s = \sum_{i=0}^n x_i$ ist mit n Prozessoren in Zeit $\log n$ berechenbar.

Parallele Matrix-Multiplikation

Theorem 4

Das Produkt von zwei $n \times n$ -Matrizen ist mit n^3 Prozessoren in Zeit $1 + \log n$ berechenbar.

Beweis: Sei $A = (A_{ij})$ und $B = (B_{ij})$.

Dann gilt $(AB)_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$.

- Berechne mit n^3 Prozessoren alle n^3 Produkte.
- Teile jeder der n^2 Summen n Prozessoren zu.
- Berechne in $\log n$ Schritten alle n^2 Summen.



Parallele Präfixsummen

Das Problem:

- Eingabe: x_i ($0 \leq i \leq n - 1$)
- Ausgabe: Alle Präfixsummen $y_i = \sum_{j=0}^i x_j$ für alle $0 \leq i \leq n - 1$

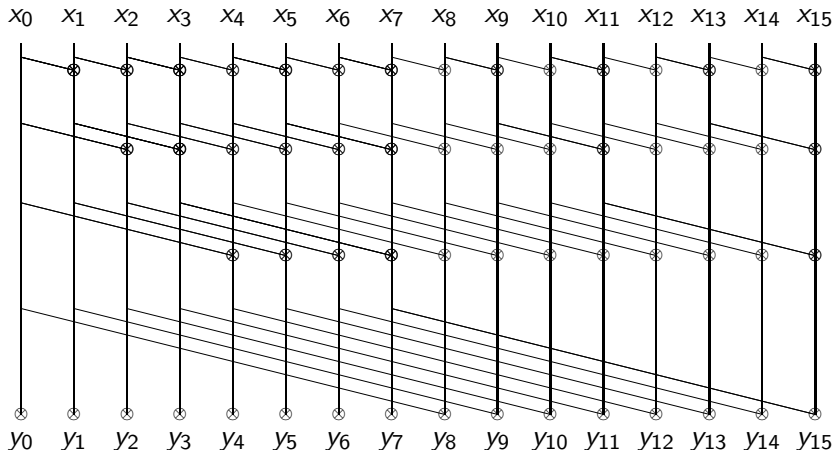
Theorem 5

Alle Präfixsummen können mit n Prozessoren in Zeit $\log n$ berechnet werden.

Beweis:

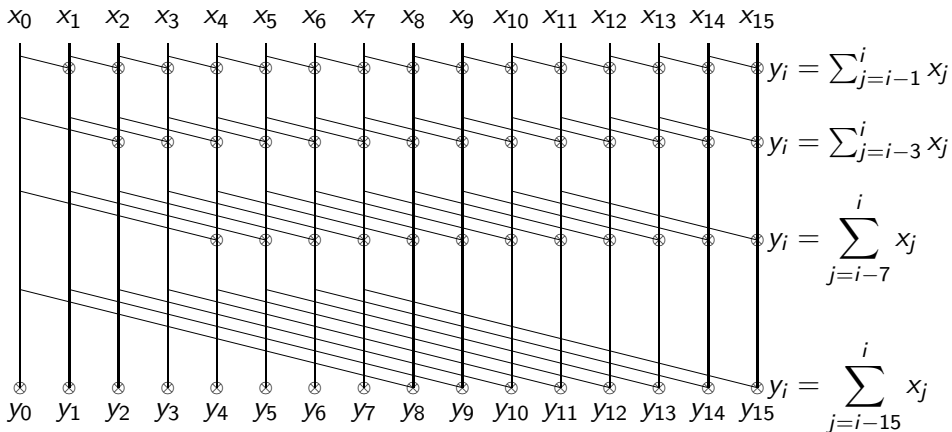
Zum Verständnis: Berechne die Summe $y_{n-1} = \sum_{j=0}^{n-1} x_j$

Parallele Präfixsummen



Parallele Präfixsummen

Sei $x_i = 0$ für alle $i < 0$.



Integer-Addition in NC

Theorem 6

Zwei n -bit-Binärzahlen können mit n Prozessoren in Zeit $\mathcal{O}(\log n)$ addiert werden.

Beweis: Seien $a_{n-1} \dots a_3 a_2 a_1 a_0$ und $b_{n-1} \dots b_3 b_2 b_1 b_0$ die Eingabezahlen (niederwertigstes Bit ist rechts).

Schritt 1: Berechne mit n Prozessoren in Zeit $\mathcal{O}(1)$ den **Carry-Propagierungs-String** $c_n c_{n-1} \dots c_3 c_2 c_1 c_0$:

$$c_i = \begin{cases} 0 & a_{i-1} = b_{i-1} = 0 \vee i = 0 \\ 1 & a_{i-1} = b_{i-1} = 1 \\ p & \text{else} \end{cases}$$

Integer-Addition in NC

Schritt 2: Berechne mit n Prozessoren in Zeit $\mathcal{O}(\log n)$ carry_i aus c_i mit dem parallelen Präfixsummen-Algorithmus unter Verwendung der folgenden assoziativen binären Operation:

$$0 \cdot x = 0$$

$$1 \cdot x = 1$$

$$p \cdot x = x$$

Beachte: Der parallele Präfixsummen-Algorithmus funktioniert für jede assoziative binäre Operation.

Schritt 3: Berechne das i -te Bit der Summe als das XOR von a_i , b_i und carry_i (wobei $a_n = b_n = 0$). □

Integer-Addition in NC

Beispiel:

$$\begin{array}{rcl}
 a & = & 100101011101011 \\
 b & = & 110101001010001 \\
 c & = & 1p01010p1ppp0p10 \\
 \text{carry} & = & 1001010110000110 \\
 \hline
 \text{sum} & = & 1011010100111100
 \end{array}$$

Multiplikation von zwei Zahlen kann auf Addition von n Zahlen zurückgeführt werden.

Aus Theorem 6 folgt: Zwei n -bit-Binärzahlen können mit n^2 Prozessoren in Zeit $(\log n)^2$ multipliziert werden

Integer-Multiplikation in NC

Lemma 7

Aus drei n -bit-Binärzahlen a, b, c können mit n Prozessoren in Zeit $\mathcal{O}(1)$ zwei $(n + 1)$ -bit Binärzahlen d, e berechnet werden mit $a + b + c = d + e$.

Beweis:

$$\begin{array}{r}
 100111 \\
 011100 \\
 +111101 \\
 \hline
 10 \\
 01 \\
 11 \quad 111101 \\
 10 \quad +000110 \\
 10 \\
 +10 \\
 \hline
 \end{array}$$

Integer-Multiplikation in NC

Theorem 8

Die Summe von n vielen n -bit Binärzahlen kann mit n^2 Prozessoren in Zeit $O(\log n)$ berechnet werden.

Beweis:

- Berechne in konstanter Zeit aus jedem Block von drei Zahlen einen neuen Block von zwei Zahlen mit einem Bit mehr.
- Wiederhole dies, bis nur noch zwei Binärzahlen übrig bleiben. Dies benötigt Zeit $O(\log n)$.
- Addiere die zwei übrig gebliebenen Zahlen in Zeit $O(\log n)$. □

Korollar

Zwei n -bit-Binärzahlen können mit n^2 Prozessoren in Zeit $O(\log n)$ multipliziert werden.

Integer-Division in NC

Ziel: Bestimme zu zwei gegebenen Binärzahlen $s, t > 0$ mit $\leq n$ Bits die eindeutigen Zahlen q und r so, dass $s = qt + r$ und $0 \leq r < t$.

Wir werden dies in Zeit $\mathcal{O}((\log n)^2)$ mit $\mathcal{O}(n^4)$ Prozessoren durchführen.

Hauptwerkzeug: Newton-Verfahren zum Approximieren von Nullstellen.

Sei $f : \mathbb{R} \rightarrow \mathbb{R}$.

Rate einen Anfangswert x_0 und berechne die Folge $(x_i)_{i \geq 0}$ mit der Rekursionsgleichung

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \text{ wobei } f' = df/dx$$

Mit Glück konvergiert die Folge $(x_i)_{i \geq 0}$ zu einer Nullstelle von f .

Integer-Division in NC

Nimm $f(x) = t - \frac{1}{x}$, also ist $\frac{1}{t}$ die eindeutige Nullstelle von f .

$f'(x) = \frac{1}{x^2}$, also wird Newtons Rekursionsgleichung zu

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} = 2x_i - tx_i^2$$

Sei x_0 die eindeutige Zahl der Form $\frac{j}{2^j}$ ($j > 0$) im Intervall $(\frac{1}{2t}, \frac{1}{t}]$.

Wir bestimmen x_0 in Zeit $\mathcal{O}(1)$ unter der Benutzung von n Prozessoren wie folgt: finde die eindeutige Zweierpotenz im Intervall $[t, 2t)$, drehe die Reihenfolge der Bits um und platziere einen Dezimalpunkt hinter der ersten 0.

Integer-Division in NC

Lemma 9

Die eindeutige Folge $(x_i)_{i \geq 0}$, die durch $x_{i+1} = 2x_i - tx_i^2$ erhalten wird (wobei x_0 die eindeutige Zahl der Form $\frac{1}{2^j}$ ($j > 0$) im Interval $(\frac{1}{2t}, \frac{1}{t}]$ ist), erfüllt $0 \leq 1 - t \cdot x_i < \frac{1}{2^{(2^i)}}$.

Beweis: Induktion über i :

Nach Definition gilt $\frac{1}{2t} < x_0 \leq \frac{1}{t}$, d.h. $0 \leq 1 - tx_0 < \frac{1}{2}$.

Für $i \geq 0$ erhalten wir

$$1 - t \cdot x_{i+1} = 1 - t(2 \cdot x_i - t \cdot x_i^2) = (1 - t \cdot x_i)^2$$

$$\text{Also } 0 \leq 1 - t \cdot x_{i+1} < \left(\frac{1}{2^{(2^i)}}\right)^2 = \frac{1}{2^{(2^{i+1})}}.$$

Integer-Division in NC

Aus dem vorangehenden Lemma erhalten wir für $k = \lceil \log(\log(s)) \rceil$:

$$0 \leq 1 - t \cdot x_k < \frac{1}{s} \leq \frac{t}{s}.$$

Also $0 \leq \frac{s}{t} - s \cdot x_k < 1$.

Es folgt, dass der ganzzahlige Anteil q von $\frac{s}{t}$ entweder $\lceil s \cdot x_k \rceil$ oder $\lfloor s \cdot x_k \rfloor$ ist (der richtige Wert kann durch einen Test bestimmt werden, bzw. durch eine einzige Multiplikation).

Der Rest r kann durch $r = s - qt$ bestimmt werden.

Integer-Division in NC

Schätzung der Laufzeit: Sei b_i die Anzahl der Bits von x_i .

Wegen $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ gilt $b_{i+1} \leq 2b_i$.

Also $b_i \in O(2^i n)$ und x_k (sowie alle x_i mit $i < k$) hat höchstens $O(2^{\lceil \log(\log(s)) \rceil} n) \leq O(n^2)$ viele Bits.

Somit benötigt die Berechnung von $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ aus x_i Zeit $O(\log(n))$ mit $O(n^4)$ Prozessoren.

Da $k \in O(\log(n))$, ist $O((\log n)^2)$ die Gesamtlaufzeit.

Csanskys Algorithmus zum Invertieren von Matrizen

Ziel: Ein NC-Algorithmus zum Invertieren einer $(n \times n)$ -Matrix (falls die Matrix invertierbar ist).

Vereinbarung: Im Folgenden nehmen wir an, dass ein einzelner Prozessor eine einzelne arithmetische Operation in Zeit $\mathcal{O}(1)$ durchführen kann. Nach vorangehenden Überlegungen wirkt sich dies nicht auf die Klasse NC aus.

Bemerkung: Falls A eine $(n \times m)$ -Matrix und B eine $(m \times p)$ -Matrix sind, dann kann $A \cdot B$ mit $n \cdot m \cdot p$ Prozessoren in Zeit $\mathcal{O}(\log(m))$ berechnet werden.

1. Schritt: Invertieren unterer Dreiecksmatrizen.

Eine Matrix ist eine **untere Dreiecksmatrix**, falls (i) alle Einträge über der Hauptdiagonale 0 sind und (ii) auf der Hauptdiagonale keine 0 vorkommt.

Csanskys Algorithmus zum Invertieren von Matrizen

Betrachte eine $(n \times n)$ große untere Dreiecksmatrix $A = \begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$.

B und D sind hier $(\frac{n}{2} \times \frac{n}{2})$ große untere Dreiecksmatrizen und C ist eine $(\frac{n}{2} \times \frac{n}{2})$ -Matrix.

Dann gilt $A^{-1} = \begin{pmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{pmatrix}$.

Diese Gleichung führt zu einem parallelen Algorithmus zum Berechnen von A^{-1} mit Zeitkomplexität

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(\log(n))$$

unter Verwendung von n^3 Prozessoren.

Also $T(n) \in \mathcal{O}(\log^2(n))$.

Csanskys Algorithmus zum Invertieren von Matrizen

2. Schritt: Lösen einer linearen Gleichungssystems:

Betrachte ein Gleichungssystem der Form

$$x_1 = c_1$$

$$x_2 = a_{2,1}x_1 + c_2$$

$$x_3 = a_{3,1}x_1 + a_{3,2}x_2 + c_3$$

$$\vdots$$

$$x_n = a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n-1}x_{n-1} + c_n$$

x_i sind Unbekannte, c_i und $a_{i,j}$ sind vorgegebene ganze Zahlen.

Sei $A = (a_{i,j})_{1 \leq i,j \leq n}$, wobei $a_{i,j} = 0$ für $i \leq j$, und sei $c = (c_1, \dots, c_n)^T$.

Csanskys Algorithmus zum Invertieren von Matrizen

Das vorangehende Gleichungssystem ist äquivalent zu $Ax + c = x$, d.h. $(A - \text{Id})x = -c$, wobei Id die $(n \times n)$ -Identitätsmatrix ist.

Also $x = (\text{Id} - A)^{-1}c$.

Da $\text{Id} - A$ eine untere Dreiecksmatrix ist, können wir $(\text{Id} - A)^{-1}$ bestimmen.

Also kann x in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^3 Prozessoren berechnet werden.

Csanskys Algorithmus zum Invertieren von Matrizen

3. Schritt (der Hauptschritt): Berechnen des charakteristischen Polynoms

Das **charakteristische Polynom** einer $(n \times n)$ -Matrix $A = (a_{i,j})_{1 \leq i,j \leq n}$ ist

$$\begin{aligned}\det(x \cdot \text{Id} - A) &= x^n - s_1 x^{n-1} + s_2 x^{n-2} - \dots + (-1)^n s_n = \\ &= \prod_{i=1}^n (x - \lambda_i)\end{aligned}$$

wobei $\lambda_1, \dots, \lambda_n$ die mit Vielfachheiten gezählten Eigenwerte von A sind.

Der Koeffizient s_1 ist die Spur von A :

$$s_1 = \text{tr}(A) = \sum_{i=1}^n \lambda_i = \sum_{i=1}^n a_{i,i}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Lemma 10

λ_i^m ist ein Eigenwert von A^m mit derselben Vielfachheit wie λ_i von A .

Beweis: Übung

$$\text{Also } \text{tr}(A^m) = \sum_{i=1}^n \lambda_i^m.$$

Durch Einsetzen von $x = 0$ in das charakteristische Polynom erhalten wir

$$s_n = (-1)^n \det(-A) = \det(A) = \prod_{i=1}^n \lambda_i.$$

Für alle $1 \leq i \leq n$ erhalten wir $s_k = \sum_{1 \leq i_1 < \dots < i_k \leq n} \lambda_{i_1} \lambda_{i_2} \dots \lambda_{i_k}$.

Csanskys Algorithmus zum Invertieren von Matrizen

$$\text{Sei } f_k^m = \sum_{\substack{1 \leq i_1 < \dots < i_k \leq n \\ j \notin \{i_1, \dots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m$$

Also $f_k^0 = (n - k)s_k$, $f_0^m = \text{tr}(A^m)$ und

$$\begin{aligned} s_k \cdot \text{tr}(A^m) &= \left(\sum_{1 \leq i_1 < \dots < i_k \leq n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \right) \cdot \sum_{j=1}^n \lambda_j^m \\ &= \sum_{\substack{1 \leq i_1 < \dots < i_k \leq n \\ j \notin \{i_1, \dots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m + \sum_{\substack{1 \leq i_1 < \dots < i_k \leq n \\ j \in \{i_1, \dots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m \\ &= f_k^m + f_{k-1}^{m+1} \end{aligned}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Daraus folgt, dass

$$\begin{aligned}
 & s_k \cdot \text{tr}(A^0) - s_{k-1} \cdot \text{tr}(A^1) + s_{k-2} \cdot \text{tr}(A^2) - \dots \\
 & \quad + (-1)^{k-1} s_1 \cdot \text{tr}(A^{k-1}) + (-1)^k \text{tr}(A^k) \\
 & = (f_k^0 + f_{k-1}^1) - (f_{k-1}^1 + f_{k-2}^2) + \dots \\
 & \quad + (-1)^{k-1} (f_1^{k-1} + f_0^k) + (-1)^k f_0^k \\
 & = f_k^0 = (n - k) s_k
 \end{aligned}$$

und weiter, dass (zur Erinnerung: $\text{tr}(A^0) = n$)

$$\begin{aligned}
 s_k = \frac{1}{k} & (s_{k-1} \text{tr}(A^1) - s_{k-2} \text{tr}(A^2) + \dots \\
 & \quad - (-1)^{k-1} s_1 \text{tr}(A^{k-1}) - (-1)^k \text{tr}(A^k)).
 \end{aligned}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Wir können also die Koeffizienten s_k des charakteristischen Polynoms wie folgt berechnen:

- 1 Berechne die Potenzen A^1, A^2, \dots, A^n in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^4 Prozessoren.
- 2 Berechne $\text{tr}(A^1), \dots, \text{tr}(A^n)$ in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n^2 Prozessoren.
- 3 Löse das obige Gleichungssystem in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^3 Prozessoren.

Bemerkung: Dies ist nur möglich, wenn die Charakteristik p des zugrunde liegenden Körpers größer als n ist, da dann $\frac{1}{k}$ für alle $1 \leq k \leq n$ existiert.

Csanskys Algorithmus zum Invertieren von Matrizen

4. Schritt (der letzte Schritt): Invertieren beliebiger nicht-singulärer Matrizen

Satz von Cayley-Hamilton: Jede quadratische Matrix erfüllt ihre eigene charakteristische Gleichung, d.h.

$$A^n - s_1 \cdot A^{n-1} + s_2 \cdot A^{n-2} - \dots + (-1)^{n-1} s_{n-1} \cdot A + (-1)^n s_n \cdot \text{Id} = 0$$

Falls A^{-1} existiert ($\iff s_n \neq 0$) gilt also:

$$A^{-1} = \frac{(-1)^{n-1}}{s_n} (A^{n-1} - s_1 A^{n-2} + s_2 A^{n-3} - \dots + (-1)^{n-1} s_{n-1} \cdot \text{Id}) .$$

Also können wir A^{-1} in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung n^4 Prozessoren bestimmen, indem wir erst die Koeffizienten s_k berechnen und dann den obigen Ausdruck in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n^2 Prozessoren berechnen.

Randomisierte Algorithmen

Ein **randomisierter Algorithmus** (oder auch probabilistischer Algorithmus) verwendet Zufallsentscheidungen (Münzwürfe).

Beispiele:

- Quicksort mit zufällig gewählten Pivoelementen
- Quickselect für Medianbestimmung.

Bei randomisierten Algorithmen unterscheidet man zwischen

- **Las Vegas Algorithmen:** Diese liefern stets ein korrektes Ergebnis, die Rechenzeit (oder der Speicherbedarf) ist eine Zufallsgröße.
Beispiel: Bei Quicksort mit zufällig gewählten Pivoelementen ist der Erwartungswert für die Laufzeit $O(n \log n)$.
- **Monte Carlo Algorithmen:** Bei diesen gibt es eine kleine Fehlerwahrscheinlichkeit.

Probabilistische Tests mit Polynomen

Sei \mathbb{F} ein beliebiger Körper und seien x_1, \dots, x_n Variablen.

Mit $\mathbb{F}[x_1, \dots, x_n]$ bezeichnen wir den Ring der Polynome mit Variablen x_1, \dots, x_n und Koeffizienten aus \mathbb{F} .

Seien $a_1, \dots, a_k \in \mathbb{F}$ und

$$p(x_1, \dots, x_n) = \sum_{i=1}^k a_i \prod_{j=1}^n x_j^{e_{i,j}} \in \mathbb{F}[x_1, \dots, x_n].$$

Dann gilt $\deg(p) = \max\{e_{i,1} + e_{i,2} + \dots + e_{i,n} \mid 1 \leq i \leq k\}$.

Satz 11

Sei $S \subseteq \mathbb{F}$ endlich und $p(x_1, \dots, x_n) \in \mathbb{F}[x_1, \dots, x_n] \setminus \{0\}$, d.h. p ist nicht das Null-Polynom. Dann hat die Gleichung $p(x_1, \dots, x_n) = 0$ höchstens $\deg(p) \cdot |S|^{n-1}$ Lösungen in S^n .

Probabilistische Tests mit Polynomen

Beweis: Induktion über n und $\deg(p)$.

Fall 1: $n = 1$, d.h. p ist ein Polynom mit einer Variable (dies schließt den Fall $\deg(p) = 0$ mit ein, d.h. $p \in \mathbb{F} \setminus \{0\}$).

Ein Polynom mit einer Variable hat höchstens $\deg(p) = \deg(p) \cdot |S|^{n-1}$ Lösungen in S .

Fall 2: $\deg(p) = 1$, d.h. p hat die Form $a + a_1x_1 + \dots + a_nx_n$.

Da $p \neq 0$ und $\deg(p) \neq 0$, gibt es ein i mit $a_i \neq 0$. O.E.d.A. nehmen wir an, dass $a_1 \neq 0$.

Dann ist $p = 0$ äquivalent zu $x_1 = \frac{1}{a_1} \left(-a - \sum_{i=2}^n a_i x_i \right)$.

Es gibt genau $|S|^{n-1}$ Belegungen für x_2, \dots, x_n aus S . Also hat $p = 0$ höchstens $|S|^{n-1} = \deg(p) \cdot |S|^{n-1}$ Lösungen in S^n .

Probabilistische Tests mit Polynomen

Fall 3: $\deg(p) \geq 2$ und $n \geq 2$.

Fall 3.1: p ist nicht irreduzibel, d.h. $p = q \cdot r$ mit $\deg(q) < \deg(p)$ und $\deg(r) < \deg(p)$.

Weder q noch r können das Null-Polynom sein.

Nach Induktion hat $q = 0$ höchstens $\deg(q) \cdot |S|^{n-1}$ Lösungen in S^n und $r = 0$ hat höchstens $\deg(r) \cdot |S|^{n-1}$ Lösungen in S^n .

Da $(a_1, \dots, a_n) \in S^n$ genau dann eine Lösung von $q \cdot r = 0$ ist, wenn es eine Lösung von $q = 0$ oder eine Lösung von $r = 0$ ist, folgt, dass $p = 0$ höchstens

$$\deg(q) \cdot |S|^{n-1} + \deg(r) \cdot |S|^{n-1} = (\deg(q) + \deg(r)) \cdot |S|^{n-1} = \deg(p) \cdot |S|^{n-1}$$

Lösungen in S^n hat.

Probabilistische Tests mit Polynomen

Fall 3.2: p ist irreduzibel.

Sei $\bar{x} = (x_1, \dots, x_{n-1})$, d.h. $p = p(\bar{x}, x_n)$.

Zu jedem $s \in S$ betrachte das Polynom $p(\bar{x}, s) \in \mathbb{F}[\bar{x}]$.

Behauptung: $p(\bar{x}, s)$ ist nicht das Null-Polynom.

Um die Behauptung zu beweisen, nehmen wir an, dass $p(\bar{x}, s) = 0$.

Schreibe $p(\bar{x}, x_n)$ als ein Polynom mit einer einzigen Variable x_n und Koeffizienten aus $\mathbb{F}[\bar{x}]$.

Polynom-Division durch $x_n - s$ mit Rest liefert:

$$p(\bar{x}, x_n) = q(\bar{x}, x_n)(x_n - s) + r,$$

wobei r ein Polynom vom Grad 0 in x_n ist, d.h. $r \in \mathbb{F}[\bar{x}]$.

Indem man $x_n = s$ einsetzt, erhält man $r = 0$.

Also gilt $p(\bar{x}, x_n) = q(\bar{x}, x_n)(x_n - s)$, was der Irreduzibilität von p widerspricht (hier ist wichtig, dass $\deg(p) \geq 2$).

Probabilistische Tests mit Polynomen

Da $p(\bar{x}, s)$ nicht das Null-Polynom ist, können wir die Induktionsannahme auf $p(\bar{x}, s)$ anwenden:

$p(\bar{x}, s) = 0$ hat höchstens $\deg(p(\bar{x}, s)) \cdot |S|^{n-2} \leq \deg(p) \cdot |S|^{n-2}$ Lösungen in S^{n-1} .

Da es $|S|$ verschiedene Werte für s gibt, folgt, dass $p = 0$ höchstens $|S| \cdot \deg(p) \cdot |S|^{n-2} = \deg(p) \cdot |S|^{n-1}$ Lösungen in S^n hat.

Damit ist der Satz bewiesen. □

Satz von Zippel und Schwartz

Sei $p(x_1, \dots, x_n)$ ein Polynom vom Grad d , das nicht das Null-Polynom ist, und Koeffizienten aus dem Körper \mathbb{F} , und sei $S \subseteq \mathbb{F}$ endlich. Wenn wir $(s_1, \dots, s_n) \in S^n$ zufällig und gleichverteilt auswählen, gilt $\text{Prob}[p(s_1, \dots, s_n) = 0] \leq d/|S|$.

Anwendung: Perfekte Matchings

Sei $G = (V, E)$ ein ungerichteter Graph.

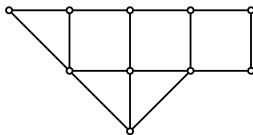
Ein **Matching** von G ist eine Teilmenge $M \subseteq E$ so, dass zwei verschiedene Kanten aus M keinen Knoten gemeinsam haben.

Ein Matching M von G ist ein **perfektes Matching**, falls $|M| = |V|/2$, d.h. jeder Knoten von G ist in genau einer Kante des Matchings enthalten.

Beachte: Ein perfektes Matching kann nur existieren, falls die Anzahl der Knoten von G gerade ist.

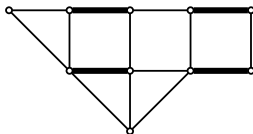
Graphentheorie: Matchings

Beispiel:



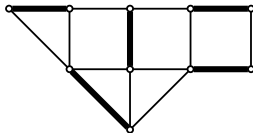
Graphentheorie: Matchings

Beispiel: ein Matching, welches jedoch nicht perfekt ist.



Graphentheorie: Matchings

Beispiel: ein perfektes Matching.



Anwendung: Perfekte Matchings

Wir suchen nach einem randomisierten NC-Algorithmus zum Testen, ob ein Graph ein perfektes Matching hat.

Dazu werden wir ein Polynom konstruieren, das genau dann nicht null ist, wenn G ein perfektes Matching hat.

Bemerkung: Es gibt einen deterministischen Algorithmus in Polynomialzeit zum Testen, ob ein Graph ein perfektes Matching hat, aber es ist nicht bekannt, ob dieses Problem in (deterministischen) NC liegt.

Anwendung: Perfekte Matchings

Sei $G = (V, E)$ ein ungerichteter Graph mit Knotenmenge $V = \{1, 2, \dots, n\}$.

Zu jedem $\{u, v\} \in E$ mit $u < v$ sei $x_{u,v}$ eine Variable.

Die **Tutte-Matrix** von G ist die Matrix $T_G = (T_{u,v})_{1 \leq u, v \leq n}$ mit

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u, v\} \in E \text{ und } u < v \\ -x_{v,u} & \text{falls } \{u, v\} \in E \text{ und } u > v \\ 0 & \text{sonst} \end{cases}$$

Anwendung: Perfekte Matchings

Wir interessieren uns für die Determinante von T_G , und verwenden hierfür die Leibniz-Formel:

Leibniz-Formel für die Determinante

Sei $A = (A_{i,j})$ eine $(n \times n)$ -Matrix. Dann gilt

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

wobei S_n die Menge aller Permutationen von $\{1, \dots, n\}$ ist, und $\text{sign}(\sigma)$ ist $+1$ (-1), falls σ das Produkt einer geraden (ungeraden) Anzahl von Transpositionen ist.

Hintergrund zu Permutationen kommt später.

Insbesondere ist $\det(T_G)$ ein Polynom mit $|E| \leq n^2$ Variablen vom Grad höchstens n .

Anwendung: Perfekte Matchings

Tuttes Satz

G hat ein perfektes Matching genau dann, wenn $\det(T_G)$ nicht das Nullpolynom ist.

Der Beweis kommt später.

Vorsicht: Wir können die Leibniz-Formel nicht dazu verwenden, das Polynom $\det(T_G)$ effizient auszurechnen, da die Summe über S_n aus $n!$ Summanden besteht.

Aber wir können probabilistisch testen, ob $\det(T_G)$ das Nullpolynom ist.

Anwendung: Perfekte Matchings

Satz 12

Es gibt einen randomisierten NC-Algorithmus (der Zeit $(\log(n))^{\mathcal{O}(1)}$ unter Verwendung von $n^{\mathcal{O}(1)}$ Prozessoren benötigt), der einen ungerichteten Graphen G als Eingabe erhält und für den gilt:

- Falls G kein perfektes Matching hat, lehnt der Algorithmus G mit Wahrscheinlichkeit 1 ab.
- Falls G ein perfektes Matching hat, akzeptiert der Algorithmus G mit Wahrscheinlichkeit $\geq 1/2$.

Bemerkung: Die Wahrscheinlichkeit $\frac{1}{2}$ kann auf $1 - \frac{1}{2^k}$ erhöht werden, indem der Algorithmus k mal wiederholt wird.

Anwendung: Perfekte Matchings

Beweis von Satz 12:

Der Algorithmus funktioniert wie folgt auf einem Graph $G = (V, E)$ mit $|V| = n$:

- Konstruiere die Tutte-Matrix T_G in Zeit $\mathcal{O}(1)$ unter Verwendung $|V|^2$ Prozessoren.
- Wähle zufällig einen Vektor $(a_1, a_2, \dots, a_m) \in \{1, \dots, 2n\}^m$, wobei $m = |E|$ die Anzahl der Variablen von T_G ist.
- Berechne $D = \det(T_G)(a_1, a_2, \dots, a_m)$ in NC unter Benutzung von Csanskys Algorithmus.
- Falls $D \neq 0$ akzeptiere, ansonsten lehne ab.

Anwendung: Perfekte Matchings

Falls G kein perfektes Matching hat, folgt nach Tutte's Satz, dass $\det(T_G) = 0$. Somit lehnen wir mit Wahrscheinlichkeit 1 ab.

Falls G ein perfektes Matching hat, folgt nach Tutte's Satz, dass $\det(T_G)$ nicht das Null-Polynom ist.

Also folgt nach dem Satz von Zippel und Schwartz (mit $S = \{1, \dots, 2n\}$), dass

$$\text{Prob}[\text{Algorithmus akzeptiert nicht}] \leq \frac{1}{2n} \deg(\det(T_G)) \leq \frac{1}{2}$$



Anwendung: Perfekte Matchings

Es bleibt, Tutte's Satz zu beweisen:

$G = (\{1, \dots, n\}, E)$ hat ein perfektes Matching $\Leftrightarrow \det(T_G) \neq 0$.

Hierbei ist $T_G = (T_{u,v})_{1 \leq u, v \leq n}$ mit

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u, v\} \in E \text{ und } u < v \\ -x_{v,u} & \text{falls } \{u, v\} \in E \text{ und } u > v \\ 0 & \text{sonst} \end{cases}$$

Hintergrund zu Permutationen: Sei σ eine Permutation von $\{1, \dots, n\}$. Dann kann σ eindeutig als Produkt von disjunkten Zyklen geschrieben werden.

Sei $E_n = \{\sigma \in S_n \mid \sigma \text{ enthält nur Zyklen gerader Länge}\}$.

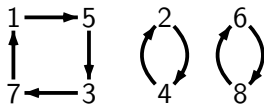
Anwendung: Perfekte Matchings

Beispiel: Sei die Permutation

$\sigma : \{1, 2, 3, 4, 5, 6, 7, 8\} \rightarrow \{1, 2, 3, 4, 5, 6, 7, 8\}$ wie folgt definiert:

a	1	2	3	4	5	6	7	8
$\sigma(a)$	5	4	7	2	3	8	1	6

Dann sieht σ wie folgt aus:



Wir schreiben auch

$$\sigma = (1, 5, 3, 7)(2, 4)(6, 8) = (6, 8)(2, 4)(1, 5, 3, 7) = \dots$$

Es gilt $\sigma \in E_n$, da alle Zyklen gerade Länge haben.

Anwendung: Perfekte Matchings

Eine Transposition ist eine Permutation der Form (a, b) , d.h. sie besteht aus genau einem Kreis der Länge 2 und Fixpunkten (Kreisen der Länge 1).

Jede Permutation kann als ein Produkt von (nicht unbedingt disjunkten) Transpositionen geschrieben werden.

Im Beispiel der vorangehenden Folie haben wir

$$\sigma = (1, 7)(3, 5)(5, 7)(2, 4)(6, 8)$$

(das Produkt muss von links nach rechts ausgewertet werden).

Das Vorzeichen einer Permutation σ ist

$$\text{sign}(\sigma) = \begin{cases} +1 & \text{falls } \sigma \text{ das Produkt einer geraden Anzahl} \\ & \text{von Transpositionen ist} \\ -1 & \text{falls } \sigma \text{ das Produkt einer ungeraden Anzahl} \\ & \text{von Transpositionen ist} \end{cases}$$

Anwendung: Perfekte Matchings

Zur Erinnerung: $\det(T_G) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)}$.

Lemma 13

$$\det(T_G) = \sum_{\sigma \in E_n} \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)}$$

Beweis: Wir zeigen

$$\sum_{\sigma \in S_n \setminus E_n} \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)} = 0.$$

Anwendung: Perfekte Matchings

Zunächst gilt: Für jede Permutation $\sigma \in S_n$, die einen Fixpunkt hat, d.h. $\sigma(j) = j$ für ein $1 \leq j \leq n$ (jede solche Permutation gehört zu $S_n \setminus E_n$) gilt $\prod_{i=1}^n T_{i,\sigma(i)} = 0$ (denn $T_{j,\sigma(j)} = 0$).

Wir betrachten nun noch alle Permutationen aus $S_n \setminus E_n$, die keinen Fixpunkt haben.

Sei $U_n = \{\sigma \in S_n \setminus E_n \mid \forall i : \sigma(i) \neq i\}$.

Es genügt zu zeigen: $\sum_{\sigma \in U_n} \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)} = 0$.

Sei $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k \in U_n$, wobei die σ_i paarweise disjunkte Zyklen sind.

O.E.d.A. hat $\sigma_1 = (a_0, a_1, \dots, a_{\ell-1})$ ungerade Länge $\ell \geq 3$.

Anwendung: Perfekte Matchings

Sei $\tau = \sigma_1^{-1} \sigma_2 \cdots \sigma_k = (a_{\ell-1}, \dots, a_1, a_0) \sigma_2 \cdots \sigma_k \in U_n$.

Es gilt für alle $0 \leq i \leq \ell - 1$ (wobei $i + 1$ also $i + 1 \bmod \ell$ zu lesen ist):

$$T_{a_i, \sigma(a_i)} = T_{a_i, a_{i+1}} = -T_{a_{i+1}, a_i} = -T_{a_{i+1}, \tau(a_{i+1})}$$

Also gilt:

$$\prod_{i=1}^n T_{i, \sigma(i)} = (-1)^\ell \prod_{i=1}^n T_{i, \tau(i)} = - \prod_{i=1}^n T_{i, \tau(i)}$$

Wir behaupten, dass $\text{sign}(\sigma) = \text{sign}(\tau)$, woraus dann folgt:

$$\text{sign}(\sigma) \prod_{i=1}^n T_{i, \sigma(i)} = -\text{sign}(\tau) \prod_{i=1}^n T_{i, \tau(i)}.$$

Anwendung: Perfekte Matchings

Es gilt $\sigma\tau = \sigma_1\sigma_2\cdots\sigma_k\sigma_1^{-1}\sigma_2\cdots\sigma_k = (\sigma_2\cdots\sigma_k)^2$.

Also $\text{sign}(\sigma\tau) = +1$ (jede Permutation der Form ρ^2 ist ein Produkt einer geraden Anzahl von Transpositionen) und somit $\text{sign}(\sigma) = \text{sign}(\tau)$.

(Falls σ (τ) Produkt einer ungeraden Anzahl (geraden) Anzahl von Transpositionen wäre, dann wäre $\sigma\tau$ ein Produkt einer ungeraden Anzahl von Transpositionen.)

Wir können die Paarung zwischen σ und τ auf alle Permutationen in U_n erweitern, was zeigt, dass

$$\sum_{\sigma \in U_n} \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)} = 0.$$

Anwendung: Perfekte Matchings

Genauer: Definiere auf der Menge U_n eine Involution g (d.h. $g(\rho) \neq \rho$ und $g^2(\rho) = \rho$) wie folgt:

Sei $\sigma = \sigma_1\sigma_2 \cdots \sigma_k \in U_n$, wobei die σ_i paarweise disjunkte Zyklen sind.

Nimm den eindeutigen Zyklus σ_j ungerader Länge, welcher das Minimum aller Elemente aus $\{1, \dots, n\}$, welche auf ungeraden Zyklen liegen, enthält.

Definiere $g(\sigma) = \sigma_1 \cdots \sigma_{j-1} \sigma_j^{-1} \sigma_{j+1} \cdots \sigma_k$.

Nach den Argumentationen der vorangehenden Folie gilt:

$$\forall \sigma \in U_n : \text{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)} = -\text{sign}(g(\sigma)) \prod_{i=1}^n T_{i,g(\sigma)(i)}.$$

Dies beweist das Lemma. □

Anwendung: Perfekte Matchings

Wir können nun den Beweis von Tutte's Satz abschließen:

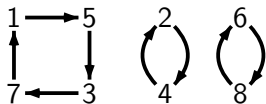
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

Nach dem vorangehenden Lemma existiert ein $\sigma \in E_n$ so, dass $\prod_{i=1}^n T_{i,\sigma(i)} \neq 0$.

Also gilt $\{i, \sigma(i)\} \in E$ für alle $1 \leq i \leq n$.

Wir erhalten ein perfektes Matching für G , indem wir jede zweite Kante von jedem Zyklus von σ auswählen.

Beispiel:



Eine Permutation $\sigma \in E_8$

Anwendung: Perfekte Matchings

Wir können nun den Beweis von Tutte's Satz abschließen:

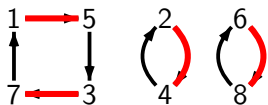
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

Nach dem vorangehenden Lemma existiert ein $\sigma \in E_n$ so, dass $\prod_{i=1}^n T_{i,\sigma(i)} \neq 0$.

Also gilt $\{i, \sigma(i)\} \in E$ für alle $1 \leq i \leq n$.

Wir erhalten ein perfektes Matching für G , indem wir jede zweite Kante von jedem Zyklus von σ auswählen.

Beispiel:



Eine Permutation $\sigma \in E_8$
 Ein perfektes Matching, das durch
 Auswählen jeder zweiten Kante
 jedes Kreises erhalten wird

Anwendung: Perfekte Matchings

(2) Nimm an, dass G ein perfektes Matching $M \subseteq E$ hat.

Weise den Variablen $x_{u,v}$ ($\{u, v\} \in E$, $u < v$) Werte zu durch

$$x_{u,v} = \begin{cases} 1 & \text{falls } \{u, v\} \in M \\ 0 & \text{sonst} \end{cases}$$

Mit dieser Substitution wird T_G eine Matrix, die in jeder Zeile und in jeder Spalte genau einen von 0 verschiedenen Eintrag (1 oder -1) enthält.

Aus Leibniz Formel folgt, dass die Determinante ungleich 0 ist.

Also kann $\det(T_G)$ nicht das Null-Polynom sein.

Dies schließt den Beweis von Tutte's Satz ab.

Pattern-Matching mit Fingerprints

Sei Σ ein endliches Alphabet.

Sei $T = a_1 a_2 \cdots a_n$ ein Text und $P = b_1 b_2 \cdots b_m$ ein Pattern
($a_i, b_j \in \Sigma$, $m \leq n$).

Ziel: Finde alle Vorkommen von P in T , d.h. alle Positionen
 $1 \leq i \leq n - m + 1$ so, dass

$$T[i, i + m - 1] := a_i a_{i+1} \cdots a_{i+m-1} = P.$$

Der Algorithmus von Knuth, Morris und Pratt erreicht dies in sequentieller
Zeit $\mathcal{O}(m + n)$.

Wir wollen hier einen einfachen randomisierten parallelen Algorithmus
entwickeln.

Pattern-Matching mit Fingerprints

Wir nehmen im Folgenden an, dass $\Sigma = \{0, 1\}$.

Definiere $f : \Sigma \rightarrow \mathbb{Z}^{2 \times 2}$ durch

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Wir erweitern f zu einem Homomorphismus $f : \Sigma^* \rightarrow \mathbb{Z}^{2 \times 2}$

Lemma 14

Der Homomorphismus f ist injektiv. Falls $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ und $|w| = \ell$, dann $a_i \leq 2^\ell$.

Pattern-Matching mit Fingerprints

Beweis: Es gilt

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix}$$

und

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 + a_3 & a_2 + a_4 \\ a_3 & a_4 \end{pmatrix}$$

(1) Wenn $w \in \{0, 1\}^*$ und $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$, dann $a_1, a_4 > 0$, $a_2, a_3 \geq 0$.

Beweis von (1): Induktion über $|w|$.

Pattern-Matching mit Fingerprints

(2) Falls $w \neq \varepsilon$, dann $f(w) \neq f(\varepsilon) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Beweis von (2):

Falls $w = 0u$ und $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$, dann $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix}$.

Falls $f(w) = f(\varepsilon)$, dann würde $a_1 = a_2 = a_3 = 0$ gelten, was ein Widerspruch zu $a_1 > 0$ ist (siehe (1)).

Für $w = 1u$ können wir ähnlich argumentieren.

Pattern-Matching mit Fingerprints

(3) $f(0u) \neq f(1v)$ für alle $u, v \in \{0, 1\}^*$.

Beweis von (3): Sei $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ und $f(v) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$.

Falls $f(0u) = f(1v)$, erhalten wir:

$$\begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix} = \begin{pmatrix} b_1 + b_3 & b_2 + b_4 \\ b_3 & b_4 \end{pmatrix}$$

Also:

$$a_1 = b_1 + b_3, \quad a_1 + a_3 = b_3$$

$$a_2 = b_2 + b_4, \quad a_2 + a_4 = b_4$$

$$b_1 + a_3 = 0, \text{ d.h. } a_3 = b_1 = 0, \text{ ein Widerspruch zu } b_1 > 0$$

$$a_4 + b_2 = 0, \text{ d.h. } a_4 = b_2 = 0, \text{ ein Widerspruch zu } a_4 > 0$$

Pattern-Matching mit Fingerprints

Nun können wir den Beweis von Lemma 14 beenden.

Angenommen $u, v \in \{0, 1\}^*$, $u \neq v$ und $f(u) = f(v)$.

Wir leiten einen Widerspruch ab.

Die Matrizen $f(0)$ und $f(1)$ sind invertierbar ($\det(f(0)) = \det(f(1)) = 1$), also ist jede Matrix $f(x)$ invertierbar.

1. Fall: $u = vw$ mit $w \neq \varepsilon$.

Dann gilt $f(v)f(w) = f(u) = f(v)$, d.h. $f(w) = \text{Id}_2 = f(\varepsilon)$.

Widerspruch zu (2)!

2. Fall: $v = uw$ mit $w \neq \varepsilon$: Analog

Pattern-Matching mit Fingerprints

3. Fall: Es existieren u', v', w existieren mit $u = w0u'$ und $v = w1v'$.

Dann gilt $f(w)f(0u') = f(w)f(1v')$, d.h. $f(0u') = f(1v')$.

Widerspruch zu (3).

4. Fall: Es existieren $u'v', w$ existieren mit $u = w1u'$ und $v = w0v'$:
Analog.

Dies beweist die erste Aussage des Lemmas.

Die zweite Aussage über die Größe der a_i folgt durch Induktion über $\ell = |w|$.

Pattern-Matching mit Fingerprints

Erste Idee zum Pattern-Matching: Vergleiche $f(P)$ mit $f(T[i, i + m - 1])$ für alle $1 \leq i \leq n - m + 1$.

Problem: $f(P)$ kann Einträge der Größe F_{m+1} haben, die $\mathcal{O}(m)$ Bits benötigen. Also benötigt das Vergleichen von $f(P)$ mit $f(T[i, i + m - 1])$ Zeit $\mathcal{O}(m)$ und wir gewinnen nichts gegenüber dem direkten Vergleichen von P und $T[i, i + m - 1]$.

Lösung: Rechne modulo einer Primzahl, die groß genug ist.

Zu einem Wort $w \in \{0, 1\}^*$ und einer Primzahl p sei

$$f_p(w) = \begin{pmatrix} a_1 \bmod p & a_2 \bmod p \\ a_3 \bmod p & a_4 \bmod p \end{pmatrix}, \text{ wobei } f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$$

Die Matrix $f_p(w)$ wird **Fingerprint** des Strings w (bezüglich der Primzahl p) genannt.

Pattern-Matching mit Fingerprints

Sei p eine Primzahl und seien $X, Y \in \{0, 1\}^*$ zwei Strings.

Wir sagen, dass p **schlecht** für (X, Y) ist, falls gilt:

- $X \neq Y$ (d.h. $f(X) \neq f(Y)$) und
- $f_p(X) = f_p(Y)$.

Zu einer natürlichen Zahl k sei $\text{Primes}(k)$ die Menge der Primzahlen p mit $2 \leq p \leq k$ und $\pi(k) = |\text{Primes}(k)|$.

Dann gilt:

- $\frac{k}{\ln(k)} \leq \pi(k) \leq 1.2551 \frac{k}{\ln(k)}$
- Zu $u \leq 2^k$ ist die Anzahl verschiedener Primfaktoren von u höchstens $\pi(k)$, falls k groß genug ist ($k \geq 29$).

Pattern-Matching mit Fingerprints

Lemma 15

Seien $X_1, Y_1, \dots, X_t, Y_t$ Strings der Länge m mit m genügend groß und sei $M \geq 1$.

Für eine zufällig aus $\text{Primes}(M)$ gewählte Primzahl p gilt:

$$\text{Prob}[\exists 1 \leq i \leq t : p \text{ schlecht für } (X_i, Y_i)] \leq \frac{\pi(4mt)}{\pi(M)}$$

Beweis: Für $1 \leq i \leq t$ sei

$$f(X_i) = \begin{pmatrix} a_{i,1} & a_{i,2} \\ a_{i,3} & a_{i,4} \end{pmatrix} \text{ und } f(Y_i) = \begin{pmatrix} b_{i,1} & b_{i,2} \\ b_{i,3} & b_{i,4} \end{pmatrix}.$$

Pattern-Matching mit Fingerprints

Dann gilt:

$$\exists 1 \leq i \leq t : p \text{ ist schlecht für } (X_i, Y_i)$$

$$\iff$$

$$\exists 1 \leq i \leq t : f(X_i) \neq f(Y_i) \text{ und } f_p(X_i) = f_p(Y_i)$$

$$\implies$$

$$p \text{ teilt das Produkt } \prod \{|a_{i,j} - b_{i,j}| \mid 1 \leq i \leq t, 1 \leq j \leq 4, a_{i,j} \neq b_{i,j}\}$$

$$\implies$$

$$p \text{ teilt eine Zahl } u \leq 2^{4tm}$$

Die Anzahl verschiedener Primfaktoren von u ist höchstens $\pi(4mt)$ (für $4mt \geq 29$).

$$\rightsquigarrow \text{Prob}[\exists 1 \leq i \leq t : p \text{ schlecht für } (X_i, Y_i)] \leq \frac{\pi(4mt)}{\pi(M)}.$$

Pattern-Matching mit Fingerprints

Lemma 16

Seien $X_1, Y_1, \dots, X_t, Y_t$ Strings der Länge m mit m genügend groß.

Sei $k \geq 1$ eine beliebig gewählte Konstante und $M = mt^k$.

Für eine zufällig aus $\text{Primes}(M)$ gewählte Primzahl p gilt:

$$\text{Prob}[\exists 1 \leq i \leq t : p \text{ schlecht für } (X_i, Y_i)] \leq \mathcal{O}\left(\frac{1}{t^{k-1}}\right)$$

Pattern-Matching mit Fingerprints

Beweis: Es gilt $\pi(4mt) \leq \mathcal{O}\left(\frac{m \cdot t}{\ln(mt)}\right)$ und

$$\pi(M) = \pi(mt^k) \geq \frac{mt^k}{\ln(mt^k)} = \frac{mt^k}{\ln(m) + k \ln(t)}.$$

Mit Lemma 15 folgt:

$$\begin{aligned} & \text{Prob}[\exists 1 \leq i \leq t : p \text{ schlecht für } (X_i, Y_i)] \\ & \leq \mathcal{O}\left(\frac{m \cdot t \cdot (\ln(m) + k \ln(t))}{\ln(mt) \cdot m \cdot t^k}\right) \\ & \leq \mathcal{O}\left(\frac{1}{t^{k-1}}\right) \end{aligned}$$

Pattern-Matching mit Fingerprints

Zur Erinnerung: $T = a_1 a_2 \cdots a_n$ (Text), $P = b_1 b_2 \cdots b_m$ (Pattern)

Zu einer Primzahl $p \in n^{\mathcal{O}(1)}$ können wir mit n Prozessoren in Zeit $\mathcal{O}(\log(n))$ alle Fingerprints $f_p(T[i, i + m - 1])$ ($1 \leq i \leq n - m + 1$) berechnen:

Dazu berechnen wir mit dem Präfix-Summen-Algorithmus alle Produkte

$$\begin{aligned} R_i &= f_p(T[1, i]) = f_p(a_1) f_p(a_2) \cdots f_p(a_i) \\ R_i^{-1} &= f_p(T[1, i])^{-1} = f_p(a_i)^{-1} f_p(a_{i-1})^{-1} \cdots f_p(a_1)^{-1} \end{aligned}$$

($1 \leq i \leq n$) in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n Prozessoren.

Da wir modulo einer Primzahl der Größe $n^{\mathcal{O}(1)}$ rechnen, haben alle Zahlen, die in der Berechnung vorkommen, höchstens $\mathcal{O}(\log(n))$ viele Bit.

Berechne zuletzt $f_p(T[i, i + m - 1]) = R_{i-1}^{-1} \cdot R_{i+m-1}$ in Zeit $\mathcal{O}(1)$ unter Verwendung von n Prozessoren.

Pattern-Matching mit Fingerprints

Satz 17

Sei k eine Konstante. Mittels $\mathcal{O}(n)$ vielen Prozessoren kann in Zeit $\mathcal{O}(\log(n))$ ein Array $\text{MATCH}[1, \dots, n]$ mit folgenden Eigenschaften berechnet werden:

- Falls $T[i, i + m - 1] = P$, dann $\text{MATCH}[i] = 1$ mit Wahrscheinlichkeit 1.
- Die Wahrscheinlichkeit, dass ein i existiert mit $\text{MATCH}[i] = 1$ und $T[i, i + m - 1] \neq P$, ist durch $\mathcal{O}\left(\frac{1}{n^k}\right)$ beschränkt.

Pattern-Matching mit Fingerprints

Beweis:

- 1 Sei $M = m \cdot n^{k+1} \leq n^{k+2}$.
- 2 Wähle eine zufällige Primzahl $p \in \{1, \dots, M\}$.
- 3 Berechne $f_p(P)$ in Zeit $\mathcal{O}(\log(m))$ unter der Verwendung von $m \leq n$ Prozessoren.
- 4 Für $1 \leq i \leq n - m + 1$ berechne parallel $L_i := f_p(T[i, i + m - 1])$ unter Verwendung des Algorithmus der vorangehenden Folie.
- 5 Für $1 \leq i \leq n - m + 1$ setze parallel $\text{MATCH}[i] = 1$ genau dann, wenn $L_i = f_p(P)$.

Nach Lemma 16 ist die Wahrscheinlichkeit, dass ein Eintrag $\text{MATCH}[i]$ fälschlicherweise auf 1 gesetzt ist, beschränkt durch $\mathcal{O}\left(\frac{1}{n^k}\right)$.

Streaming algorithms

A **streaming algorithm** receives a sequence $S = (a_1, a_2, \dots, a_m)$ (a so called stream) of element from some universe of size n .

At time t the algorithm only has direct access to a_t and its internal memory state.

In particular: no direct access to the previous values a_1, a_2, \dots, a_{t-1} unless the algorithm explicitly stores these values.

Goal: Compute important statistical data without storing all values from the stream.

Often, a good approximation of such statistical data suffices.

Streaming algorithms

Let $S = (a_1, a_2, \dots, a_m)$ be a stream of elements $a_i \in \{1, \dots, n\}$.

For $1 \leq i \leq n$ let $m_i(S) = |\{t \mid a_t = i\}|$ be the number of occurrences of i in the stream S .

For $k \geq 0$ we define the F_k -norm (or the k -th moment) of S as

$$F_k(S) = \sum_{i=1}^n m_i(S)^k$$

Note that:

- $F_0(S)$ is the number of different elements that appear in S .
- $F_1(S) = m$ is the length of the stream.

Streaming algorithms

Goal: Space efficient algorithm for computing $F_0(S)$.

A naive solution can do this by storing n bits:

- For each $1 \leq i \leq n$ we store a bit s_i which is initially zero and set to one, once the data value i appears in the stream.
- At the end we output $\sum_{i=1}^n s_i$.

Can we do better, i.e., compute $F_0(S)$ with $o(n)$ space?

Depends on what we want:

- The exact value $F_0(S)$ cannot be computed in space $o(n)$ — even if we allow randomized streaming algorithms with an error probability of $< 1/2$. \rightsquigarrow **randomized communication complexity**
- But: a good approximation of $F_0(S)$ can be computed in space $\mathcal{O}(\log n)$ with a randomized streaming algorithm with a small error probability. \rightsquigarrow **algorithm of Alon, Matias and Szegedy (AMS)**

Pairwise independent hash functions

Main tool of the AMS-algorithm: **pairwise independent hashing**.

Family of pairwise independent hash functions

Let A and B be finite sets, $|A| \geq 2$.

A set $\mathcal{H} \subseteq \{h \mid h : A \rightarrow B\}$ is a family of **pairwise independent hash functions** if the following holds for all $a_1, a_2 \in A$ with $a_1 \neq a_2$ and $b_1, b_2 \in B$:

If we choose a mapping $h \in \mathcal{H}$ uniformly at random (every $h \in \mathcal{H}$ is chosen with probability $1/|\mathcal{H}|$) then

$$\text{Prob}[h(a_1) = b_1 \wedge h(a_2) = b_2] = 1/|B|^2.$$

Pairwise independent hash functions

Let $\mathcal{H} \subseteq \{h \mid h : A \rightarrow B\}$ be a family of pairwise independent hash functions and choose a mapping $h \in \mathcal{H}$ uniformly at random. We have:

- $\text{Prob}[h(a) = b] = 1/|B|$ for all fixed $a \in A$ and $b \in B$.

Take an $a' \in A$ with $a \neq a'$ (recall that $|A| \geq 2$). Then we have

$$\begin{aligned}
 \text{Prob}[h(a) = b] &= \text{Prob} \left[\bigvee_{c \in B} (h(a) = b \wedge h(a') = c) \right] \\
 &= \sum_{c \in B} \text{Prob}[h(a) = b \wedge h(a') = c] \\
 &= \frac{|B|}{|B|^2} \\
 &= 1/|B|
 \end{aligned}$$

Hence, for a fixed $a \in A$ the value $h(a)$ is uniformly distributed over the set B .

Pairwise independent hash functions

- We get for $a_1 \neq a_2$ the **pairwise independence property**:

$$\text{Prob}[h(a_1) = b_1 \wedge h(a_2) = b_2] = \text{Prob}[h(a_1) = b_1] \cdot \text{Prob}[h(a_2) = b_2].$$

- For all $a_1 \neq a_2$ we have $\text{Prob}[h(a_1) = h(a_2)] = 1/|B|$:

$$\begin{aligned} \text{Prob}[h(a_1) = h(a_2)] &= \text{Prob} \left[\bigvee_{b \in B} (h(a_1) = b \wedge h(a_2) = b) \right] \\ &= \sum_{b \in B} \text{Prob}[h(a_1) = b \wedge h(a_2) = b] \\ &= 1/|B| \end{aligned}$$

One says that \mathcal{H} is a **universal family of hash functions**.

Pairwise independent hash functions

We now construct a family of pairwise independent hash functions on the finite field \mathbb{F}_p ($\{0, 1, \dots, p-1\}$ with addition and multiplication modulo p) for a prime number p .

For $x, y \in \mathbb{F}_p$ define the mapping $h_{x,y} : \mathbb{F}_p \rightarrow \mathbb{F}_p$ by

$$h_{x,y}(a) = (ax + y) \bmod p \text{ for } a \in \mathbb{F}_p.$$

Let $\mathcal{H}_p = \{h_{x,y} \mid x, y \in \mathbb{F}_p\}$.

Theorem 18

\mathcal{H}_p is a family of pairwise independent hash functions on \mathbb{F}_p .

Pairwise independent hash functions

Proof: We have $|\mathcal{H}_p| = p^2$.

Fix $a_1, a_2, b_1, b_2 \in \mathbb{F}_p$ with $a_1 \neq a_2$.

Then the system

$$a_1x + y = b_1 \pmod{p}$$

$$a_2x + y = b_2 \pmod{p}$$

has a unique solution $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$.

In other words: there is a unique $h_{x,y} \in \mathcal{H}_p$ such that $h_{x,y}(a_1) = b_1 \wedge h_{x,y}(a_2) = b_2$.

Thus: If $x, y \in \mathbb{F}_p$ are chosen uniformly at random, then

$$\text{Prob}[h_{x,y}(a_1) = b_1 \wedge h_{x,y}(a_2) = b_2] = 1/p^2. \quad \square$$

Remark: The above proof works for every finite field \mathbb{F}_{p^m} instead of \mathbb{F}_p . For the AMS algorithm it is convenient to take a finite field \mathbb{F}_{2^m} .

The AMS algorithm

Fix the universe $\{0, 1\}^m$ (bit strings of length m) of size $n = 2^m$.

We identify elements of $\{0, 1\}^m$ with elements of the field \mathbb{F}_{2^m} .

For $u \in \{0, 1\}^m$ define $\rho(u) = \max\{k \mid u = 0^k v \text{ for some } v \in \{0, 1\}^{m-k}\}$.

Note that $\rho(w) \leq m \leq \log_2 n$.

The AMS algorithm

- randomly choose $h = h_{x,y} \in \mathcal{H}_n$
- initialize $z := 0$
- for every new data value $u \in \{0, 1\}^m$ in the stream set
 $z := \max\{z, \rho(h(u))\}$
- return $2^{z+1/2}$

Each of the numbers x, y, z fits into $m = \log_2 n$ bits.

The AMS algorithm therefore needs space $\mathcal{O}(\log n)$.

Analysis of the AMS algorithm

Let $S = (u_1, u_2, \dots, u_l)$ be the input stream with $u_1, \dots, u_l \in \{0, 1\}^m$.

Let $A = \{u_1, u_2, \dots, u_l\} \subseteq \{0, 1\}^m$ be the corresponding set.

The goal is to approximate the size of A .

In the following h denotes the randomly chosen hash function and z_f the final value of z computed by the AMS algorithm.

For $0 \leq r \leq m$ and $u \in \{0, 1\}^m$ we define the random variable $X_{r,u}$ by

$$X_{r,u} = \begin{cases} 1 & \text{if } \rho(h(u)) \geq r \\ 0 & \text{else} \end{cases}$$

Let $Y_r = \sum_{u \in A} X_{r,u}$.

Note: $Y_r > 0$ if and only if $z_f \geq r$.

Analysis of the AMS algorithm

Since for every $u \in \{0, 1\}^m$, $h(u)$ is uniformly distributed over $\{0, 1\}^m$ (i.e., $\text{Prob}[h(u) = v] = 1/2^m$ for every $v \in \{0, 1\}^m$) we have

$$\begin{aligned}
 E[X_{r,u}] &= \text{Prob}[\rho(h(u)) \geq r] \\
 &= \text{Prob}[\text{a randomly chosen } v \in \{0, 1\}^m \text{ starts with } 0^r] \\
 &= \frac{2^{m-r}}{2^m} \\
 &= \frac{1}{2^r}.
 \end{aligned}$$

Linearity of expectation yields

$$E[Y_r] = \sum_{u \in A} E[X_{r,u}] = \frac{|A|}{2^r}.$$

Claim: For every $0 \leq r \leq m$ the random variables $X_{r,u}$ ($u \in \{0, 1\}^m$) are pairwise independent.

Analysis of the AMS algorithm

Proof of the claim: Let $u, v \in \{0, 1\}^m$ with $u \neq v$ and $a, b \in \{0, 1\}$.

With $L_1 = \{w \in \{0, 1\}^m \mid \rho(w) \geq r\}$ and $L_0 = \{0, 1\}^m \setminus L_1$ we get

$$\begin{aligned}
 \text{Prob}[X_{r,u} = a \wedge X_{r,v} = b] &= \text{Prob}[h(u) \in L_a \wedge h(v) \in L_b] \\
 &= \text{Prob} \left[\bigvee_{x \in L_a} \bigvee_{y \in L_b} (h(u) = x \wedge h(v) = y) \right] \\
 &= \sum_{x \in L_a} \sum_{y \in L_b} \text{Prob}[h(u) = x \wedge h(v) = y] \\
 &= \sum_{x \in L_a} \sum_{y \in L_b} \text{Prob}[h(u) = x] \cdot \text{Prob}[h(v) = y] \\
 &= \sum_{x \in L_a} \text{Prob}[h(u) = x] \cdot \sum_{y \in L_b} \text{Prob}[h(v) = y] \\
 &= \text{Prob}[X_{r,u} = a] \cdot \text{Prob}[X_{r,v} = b]
 \end{aligned}$$

Analysis of the AMS algorithm

The claim implies

$$\text{Var}[Y_r] = \sum_{u \in A} \text{Var}[X_{r,u}] \leq \sum_{u \in A} E[X_{r,u}^2] = \sum_{u \in A} E[X_{r,u}] = \frac{|A|}{2^r}.$$

Applying Markov's inequality yields

$$\text{Prob}[Y_r > 0] = \text{Prob}[Y_r \geq 1] \leq E[Y_r] = \frac{|A|}{2^r}.$$

Chebyshev's inequality gives

$$\begin{aligned} \text{Prob}[Y_r = 0] &\leq \text{Prob}[|Y_r - E[Y_r]| \geq |A|/2^r] \\ &\leq \frac{\text{Var}[Y_r]}{(|A|/2^r)^2} \\ &\leq \frac{2^r}{|A|}. \end{aligned}$$

Analysis of the AMS algorithm

Recall that the AMS-algorithm outputs the value $\alpha := 2^{z_f+1/2}$.

Let a be the smallest number such that $2^{a+1/2} \geq 3|A|$. We get

$$\text{Prob}[\alpha \geq 3|A|] = \text{Prob}[z_f \geq a] = \text{Prob}[Y_a > 0] \leq \frac{|A|}{2^a} \leq \frac{\sqrt{2}}{3}.$$

Let b be the largest number such that $2^{b+1/2} \leq |A|/3$. We get

$$\text{Prob}[\alpha \leq |A|/3] = \text{Prob}[z_f \leq b] = \text{Prob}[Y_{b+1} = 0] \leq \frac{2^{b+1}}{|A|} \leq \frac{\sqrt{2}}{3}.$$

We have $\frac{\sqrt{2}}{3} \approx 0.4714$.

Let us fix now an arbitrary (small) $\epsilon > 0$.

We show how to reduce the above error probability 0.4714 to ϵ .

Probability amplification for the AMS algorithm

- We run k (odd) many independent copies of the AMS algorithm on the input stream.
- In other words: each copy of the algorithm randomly chooses its own hash function and these choices are made independent from each other.
- Let α_i be the output of the i -th copy of the algorithm.
- At the end we output the median $\hat{\alpha}$ of $\alpha_1, \dots, \alpha_k$.

Probability amplification for the AMS algorithm

Let us analyze the error:

Define a Bernoulli random variable F_i :

$$F_i = \begin{cases} 1 & \text{if } \alpha_i \geq 3|A| \\ 0 & \text{else} \end{cases}$$

Note: $\text{Prob}[F_i = 1] = \text{Prob}[F_j = 1]$ for all $1 \leq i, j \leq k$.

Let $p = \text{Prob}[F_i = 1]$.

Our previous analysis yields $p \leq \frac{\sqrt{2}}{3} < 1/2$.

Let $\delta = \frac{1}{2p} - 1 \geq \frac{3}{2\sqrt{2}} - 1 > 0$.

Also note: If $\hat{\alpha} \geq 3|A|$ then $\sum_{i=1}^k F_i \geq k/2$.

Probability amplification for the AMS algorithm

With the first Chernoff bound we get:

$$\begin{aligned}
 \text{Prob}[\hat{\alpha} \geq 3|A|] &\leq \text{Prob}\left[\sum_{i=1}^k F_i \geq k/2\right] \\
 &= \text{Prob}\left[\sum_{i=1}^k F_i \geq (1 + \delta)pk\right] \\
 &\leq e^{-\delta^2 pk/3}
 \end{aligned}$$

For $k \geq \frac{3}{\delta^2 p} \cdot \ln(2/\epsilon) \in \mathcal{O}(\log(1/\epsilon))$ we get $\text{Prob}[\hat{\alpha} \geq 3|A|] \leq \epsilon/2$.

Analogously we obtain with the second Chernoff bound:

$$\text{Prob}[\hat{\alpha} \leq |A|/3] \leq \epsilon/2.$$

In total: $\text{Prob}[|A|/3 < \hat{\alpha} < 3|A|] \geq 1 - \epsilon$.

Crash course on random variables: (pairwise) independence

Let X be a **random variable**, which takes values from a finite set $A \subseteq \mathbb{R}$.
 For every $a \in A$ we have the probability $\text{Prob}[X = a]$ that X takes the value a .

Independent and pairwise independent random variables

Let X_1, X_2, \dots, X_n be random variables.

X_1, X_2, \dots, X_n are **independent** if:

$$\text{Prob} \left[\bigwedge_{i=1}^n X_i = a_i \right] = \prod_{i=1}^n \text{Prob}[X_i = a_i].$$

X_1, X_2, \dots, X_n are **pairwise independent** if for all $1 \leq i, j \leq n$ with $i \neq j$:

$$\text{Prob}[X_i = a \wedge X_j = b] = \text{Prob}[X_i = a] \cdot \text{Prob}[X_j = b].$$

Crash course on random variables: expected value, variance

Expected value and variance

The **expected value** of X is $E[X] = \sum_{a \in A} \text{Prob}[X = a] \cdot a$.

The **variance** of X is $\text{Var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$.

Linearity of expectation

For random variables X and Y and $a \in \mathbb{R}$ we have:

$$E[X + Y] = E[X] + E[Y]$$

$$E[aX] = a \cdot E[X]$$

Crash course on random variables:

Let X_1, X_2, \dots, X_n be random variables.

Linearity of expectation implies $E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i]$.

In general, $\text{Var}[\sum_{i=1}^n X_i] = \sum_{i=1}^n \text{Var}[X_i]$ is **wrong**.

Lemma 19

If X_1, X_2, \dots, X_n are pairwise independent random variables then

$$\text{Var} \left[\sum_{i=1}^n X_i \right] = \sum_{i=1}^n \text{Var}[X_i].$$

Crash course on random variables: tail inequalities

Markov's inequality

Let X be a non-negative random variable. For any real number $t > 0$, we have

$$\text{Prob}[X \geq t] \leq \frac{E[X]}{t}.$$

Chebyshev's inequality

Let X be a random variable. For any real number $t > 0$, we have

$$\text{Prob}[|X - E[X]| \geq t] \leq \frac{\text{Var}[X]}{t^2}.$$

Crash course on random variables: tail inequalities

A **Bernoulli random variable** X is a random variable that only takes the two values 0 and 1.

Note: $E[X] = \text{Prob}[X = 1]$

Chernoff bound

Let X_1, X_2, \dots, X_n be independent Bernoulli random variables with the same distribution: $\text{Prob}[X_i = 1] = p$ for all $1 \leq i \leq n$.

For every $0 < \delta < 1$ we have:

$$\text{Prob} \left[\sum_{i=1}^n X_i \geq (1 + \delta)pn \right] \leq e^{-\delta^2 pn/3}$$

$$\text{Prob} \left[\sum_{i=1}^n X_i \leq (1 - \delta)pn \right] \leq e^{-\delta^2 pn/2} \leq e^{-\delta^2 pn/3}$$

Mittlere Höhe von Suchbäumen

Wir werden zeigen, dass ein Suchbaum auf den Knoten $\{1, \dots, n\}$ im Mittel die Höhe $\mathcal{O}(\log n)$ hat.

Sei \mathcal{B}_n die Menge aller binären Suchbäume auf den Knoten $\{1, \dots, n\}$ (\mathcal{B}_0 besteht nur aus dem leeren Baum \emptyset und wir setzen $\text{height}(\emptyset) = -\infty$).

Wir erzeugen ein $B \in \mathcal{B}_n$ mittels folgenden Zufallsexperiment:

- Wir wählen zufällig und gleichverteilt eine Permutation $(\pi_1, \pi_2, \dots, \pi_n)$ von $(1, 2, \dots, n)$ aus. Jede der $n!$ vielen Permutationen wird mit Wahrscheinlichkeit $1/n!$ ausgewählt.
- Dann wird ein Suchbaum aufgebaut, indem die Elemente $1, \dots, n$ in der Reihenfolge $\pi_1, \pi_2, \dots, \pi_n$ in den Suchbaum eingefügt werden.
- Beachte: Verschiedene Permutationen können den gleichen Suchbaum erzeugen.

Beispiel: $(2, 1, 3)$ und $(2, 3, 1)$ erzeugen den gleichen Suchbaum (mit Wurzel 2).

Mittlere Höhe von Suchbäumen

Folgendes Zufallsexperiment liefert für jeden Suchbaum die gleiche Wahrscheinlichkeit wie oben:

- Falls $n \geq 2$, wähle zufällig und gleichverteilt ein Element $i \in \{1, \dots, n\}$ aus.
Jedes Element wird mit Wahrscheinlichkeit $1/n$ gezogen.
- Erzeuge dann rekursiv nach dem gleichen Experiment einen Suchbaum $L \in \mathcal{B}_{i-1}$ (bzw. $R \in \mathcal{B}_{n-i}$)
- Ersetze in R jeden Knoten j durch $i + j$.
- Der erzeugte Suchbaum hat i als Wurzel und L (bzw. R) als linken (bzw. rechten) Teilbaum.

Beachte: Dieses Zufallsexperiment ergibt **nicht** die Gleichverteilung auf Binärbäumen mit n Knoten.

Mittlere Höhe von Suchbäumen

Wir definieren folgende Zufallsvariablen:

- H_n ist die Höhe eines zufällig erzeugten Suchbaums $B \in \mathcal{B}_n$.
- $X_n = 2^{H_n}$

Theorem 20

Für den Erwartungswert

$$E[H_n] = \sum_{B \in \mathcal{B}_n} \text{Prob}[B] \cdot \text{height}(B)$$

gilt: $E[H_n] \leq 3 \cdot \log_2(n)$.

Beweis: Wir zeigen zunächst, dass $E[X_n]$ durch ein Polynom $p(n)$ beschränkt ist.

Sei B ein Suchbaum mit Wurzel $1 \leq i \leq n$ und linken (bzw. rechten) Teilbaum L (R).

Dann gilt $\text{height}(B) = 1 + \max\{\text{height}(L), \text{height}(R)\}$ und daher:

Mittlere Höhe von Suchbäumen

$$\begin{aligned}
E[X_n] &= \sum_{B \in \mathcal{B}_n} \text{Prob}[B] \cdot 2^{\text{height}(B)} \\
&= \sum_{i=1}^n \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \frac{1}{n} \text{Prob}(L) \text{Prob}(R) 2^{1+\max\{\text{height}(L), \text{height}(R)\}} \\
&= \frac{2}{n} \sum_{i=1}^n \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \text{Prob}(L) \text{Prob}(R) \max\{2^{\text{height}(L)}, 2^{\text{height}(R)}\} \\
&\leq \frac{2}{n} \sum_{i=1}^n \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \text{Prob}(L) \text{Prob}(R) (2^{\text{height}(L)} + 2^{\text{height}(R)}) \\
&= \frac{2}{n} \sum_{i=1}^n \left(\sum_{L \in \mathcal{B}_{i-1}} \text{Prob}(L) 2^{\text{height}(L)} + \sum_{R \in \mathcal{B}_{n-i}} \text{Prob}(R) 2^{\text{height}(R)} \right) \\
&= \frac{2}{n} \sum_{i=1}^n (E[X_{i-1}] + E[X_{n-i}]) = \frac{4}{n} \sum_{i=0}^{n-1} E[X_i]
\end{aligned}$$

Mittlere Höhe von Suchbäumen

Behauptung 1: $\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$ für $n \geq 1$.

Beweis durch Induktion:

$$n = 1: \sum_{i=0}^0 \binom{i+3}{3} = \binom{3}{3} = 1 = \binom{1+3}{4}.$$

Sei nun $n \geq 2$:

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \sum_{i=0}^{n-2} \binom{i+3}{3} + \binom{n+2}{3} = \binom{n+2}{4} + \binom{n+2}{3} = \binom{n+3}{4}.$$

Behauptung 2: $E[X_n] \leq \frac{1}{4} \binom{n+3}{3}$.

Beweis durch Induktion:

$$n = 0: E[X_0] = 2^{-\infty} = 0 \leq \frac{1}{4} \binom{3}{3}$$

$$n = 1: E[X_1] = 2^0 = 1 = \frac{1}{4} \binom{4}{3}$$

Sei nun $n \geq 2$:

Mittlere Höhe von Suchbäumen

$$\begin{aligned}
E[X_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[X_i] \\
&\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\
&= \frac{1}{n} \binom{n+3}{4} \\
&= \frac{1}{n} \frac{(n+3)!}{4!(n-1)!} \\
&= \frac{1}{4} \frac{(n+3)!}{3!n!} \\
&= \frac{1}{4} \binom{n+3}{3}
\end{aligned}$$

Mittlere Höhe von Suchbäumen

Ausserdem gilt $\frac{1}{4} \binom{n+3}{3} \leq n^3$ für $n \geq 1$ und damit $E[X_n] \leq n^3$.

Die Funktion $x \mapsto 2^x$ ist konvex.

Also folgt aus Jensen's Ungleichung (siehe Algorithmik I):

$$2^{E[H_n]} \leq E[2^{H_n}] = E[X_n] \leq n^3.$$

Es folgt: $E[H_n] \leq 3 \log_2(n)$. □

Bemerkung: Bei einer Gleichverteilung (jeder binäre Suchbaum tritt mit der gleichen Wahrscheinlichkeit auf) erhält man $\Theta(\sqrt{n})$ für die erwartete Höhe.