Algorithmik II

Markus Lohrey

Universität Siegen

Wintersemester 2017/2018

Konvolution von Polynomen

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

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n, \; g(x) = b_0 + b_1 x + b_2 x^2 + \cdots b_m x^m$$

repräsentiert durch ihre Koeffizientenfolgen

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

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

Wir wollen das Produktpolynom

$$(fg)(x) = a_0 b_0 + (a_1 b_0 + a_0 b_1)x + \cdots + (a_0 b_{N-1} + \cdots + a_{N-1} b_0)x^{N-1}$$

berechnen, welches durch die Koeffizientenfolge

$$fg = (a_0 b_0, a_1 b_0 + a_0 b_1, \ldots, a_0 b_{N-1} + \cdots + a_{N-1} b_0),$$

(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

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

repräsentiert werden, wobei $\zeta_0, \ldots, \zeta_{N-1}$ 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):

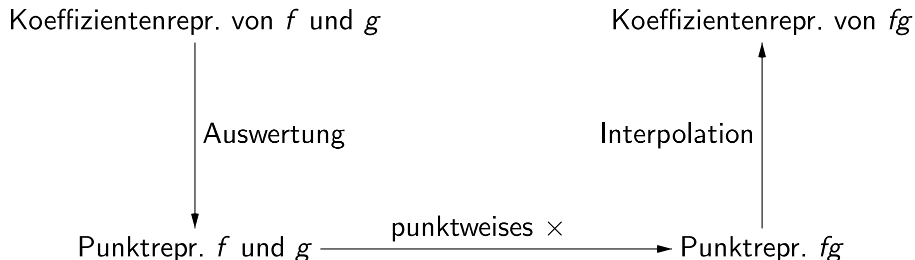
Koeffizientenrepr. von f und g

↓ Auswertung

Punktrepr. f und g —— punktweises $\times$ ——→ Punktrepr. fg

Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):



Einheitswurzeln

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

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. $i \equiv j \mod N$.
- Für $i \in \mathbb{Z}$ ist ω^i eine primitive N-te Einheitswurzel g.d.w. $\mathrm{ggT}(i, N) = 1$ (wobei $\mathrm{ggT}(i, N)$ 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 $f = a_0 + a_1 x + \cdots a_{N-1} x^{N-1}$ an den Punkten $\omega^0 = 1, \omega^1, \ldots, \omega^{N-1}$ läuft auf eine Matrixmultiplikation hinaus:

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

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* $(\omega^{ij})_{0 \le i,j < N}$ *ist* $\left(\frac{\omega^{-ij}}{N}\right)_{0 \le i,j < N}$ *(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 $0 \le i < N$:

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

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

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

FFT mittels Divide & Conquer

Wir müssen nun noch die diskrete Fouriertransformation

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

(wobei $f = (a_0, a_1, \ldots, a_{N-1})^T$) 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 $f(x) = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1}$ sei

$$f_0(x) = a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{N-2}x^{N-2}$$
$$\widehat{f_0}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{N-2}x^{\frac{N-2}{2}}$$
$$f_1(x) = a_1 + a_3 x^2 + a_5 x^4 + \cdots + a_{N-1}x^{N-2}$$
$$\widehat{f_1}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{N-1}x^{\frac{N-2}{2}}$$

Also: $f(x) = f_0(x) + x f_1(x)$, $f_0(x) = \widehat{f_0}(x^2)$ 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:

$$(F_N(\omega) f_0)_i = f_0(\omega^i) = \widehat{f_0}(\omega^{2i}) = \widehat{f_0}(\omega^{2i \bmod N}) =$$
$$\widehat{f_0}(\omega^{2(i \bmod \frac{N}{2})}) = \widehat{f_0}((\omega^2)^{i \bmod \frac{N}{2}}) = (F_{\frac{N}{2}}(\omega^2)\widehat{f_0})_{i \bmod \frac{N}{2}},$$

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 $f(x) = f_0(x) + x f_1(x)$ folgt

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

wobei $F_N(\omega)x = (1, \omega, \omega^2, \ldots, \omega^{N-1})^T$ 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 $F_{\frac{N}{2}}(\omega^2)\widehat{f_0}$ und $F_{\frac{N}{2}}(\omega^2)\widehat{f_1}$ (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 ($a = b = 2, c = 1$):

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

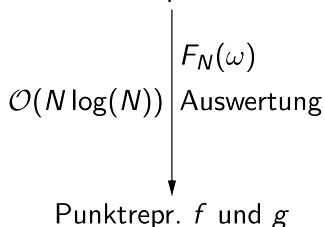
Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:



Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

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

Punktrepr. f und g —— punktweises $\times$, $\mathcal{O}(N)$ ——→ Punktrepr. fg

Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

$F_N(\omega)$

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

Punktrepr. f und g → punktweises $\times$, $\mathcal{O}(N)$ → Punktrepr. fg

Interpolation $\mathcal{O}(N\log(N))$

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

Koeffizientenrepr. von fg

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
- Boolsche 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):

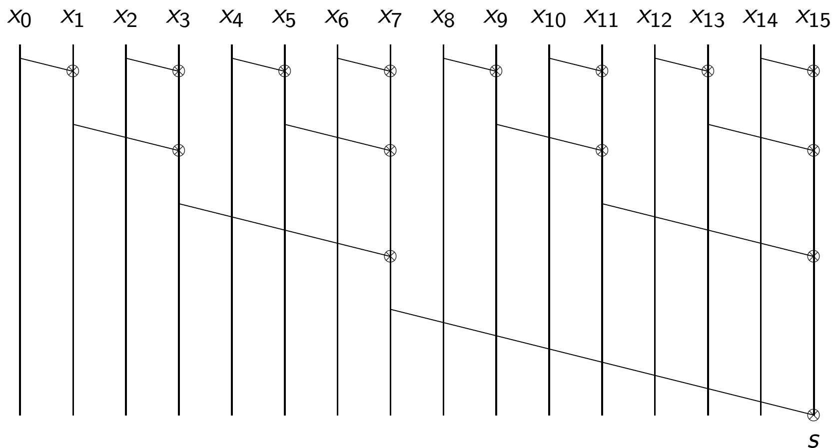
- Probleme, die mit einer PRAM mit $n^{O(1)}$ Prozessoren in Zeit $(\log n)^{O(1)}$ lösbar sind.
- Probleme, die mit einem bool'schen Schaltkreis der Tiefe $(\log n)^{O(1)}$ und Größe $n^{O(1)}$ lösbar sind.

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-Multiplikaton

Theorem 2

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 $0 \leq i \leq n-1$

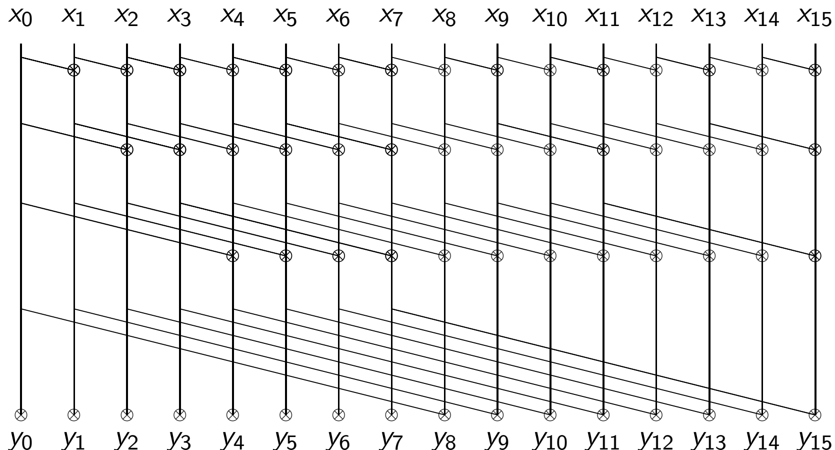
Theorem 3

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

Beweis:

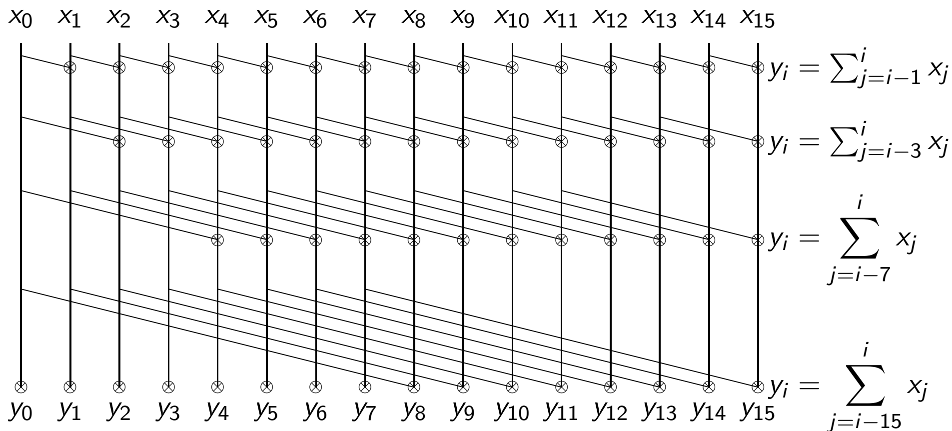
Zum Verständnis: Berechne die Summe $y_{n-1} = \sum_{j=0}^{n-1} x_j$

Parallele Präfixsummen



Parallele Präfixsummen

Sei $x_i = 0$ für alle $i < 0$.



Integer-Addition in NC

Theorem 4

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:

$$\begin{aligned} 0 \cdot x &= 0 \\ 1 \cdot x &= 1 \\ p \cdot x &= x \end{aligned}$$

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:

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

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

Aus Theorem 4 folgt: Zwei n-bit-Binärzahlen können mit n^2 Prozessoren in Zeit $(\log n)^2$ multipliziert werden

Integer-Multiplikation in NC

Lemma 5

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:

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

Integer-Multiplikation in NC

Theorem 6

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

Beweis:

- Berechne in konstaner 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 $\mathcal{O}(\log n)$ 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 $f : \mathbb{R} \to \mathbb{R}$.

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 $\frac{1}{2^j}$ $(j > 0)$ 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 7

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

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

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 $\mathcal{O}(\log(n))$ mit $\mathcal{O}(n^4)$ Prozessoren.

Da $k \in \mathcal{O}(\log(n))$, ist $\mathcal{O}((\log n)^2)$ 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

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

unter Verwendung von n^3 Prozessoren.

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

Csanskys Algorithmus zum Invertieren von Matrizen

2. Schritt: Lösen einer linearen Gleichungssystems:

Betrachte ein Gleichungssystem der Form

$$
\begin{aligned}
x_1 &= c_1 \\
x_2 &= a_{2,1}x_1 + c_2 \\
x_3 &= a_{3,1}x_1 + a_{3,2}x_2 + c_3 \\
&\vdots \\
x_n &= a_{n,1}x_1 + a_{n,2}x_2 + \cdots a_{n,n-1}x_{n-1} + c_n
\end{aligned}
$$

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 $c = (c_1, \ldots, c_n)^T$.

Csanskys Algorithmus zum Invertieren von Matrizen

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

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

Da $\mathrm{Id} - A$ eine untere Dreiecksmatrix ist, können wir $(\mathrm{Id} - A)^{-1}$ 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 $A = (a_{i,j})_{1 \le i,j \le n}$ ist

$$\det(x \cdot \mathrm{Id} - A) = x^n - s_1 x^{n-1} + s_2 x^{n-2} - \cdots + (-1)^n s_n = \\ = \prod_{i=1}^{n} (x - \lambda_i)$$

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

Der Koeffizient s_1 ist die Spur von A:

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

Csanskys Algorithmus zum Invertieren von Matrizen

Lemma 8

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

Beweis: Übung

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

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 $s_k = \sum_{1 \leq i_1 < \cdots < i_k \leq n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k}$.

Csanskys Algorithmus zum Invertieren von Matrizen

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

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

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

Csanskys Algorithmus zum Invertieren von Matrizen

Daraus folgt, dass

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

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

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

Csanskys Algorithmus zum Invertieren von Matrizen

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

1. Berechne die Potenzen $A^1, A^2, \cdots, A^n$ in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^4 Prozessoren.
2. Berechne $\mathrm{tr}(A^1), \ldots, \mathrm{tr}(A^n)$ 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.

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

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

$$A^{-1} = \frac{(-1)^{n-1}}{s_n} \left(A^{n-1} - s_1 A^{n-2} + s_2 A^{n-3} - \cdots + (-1)^{n-1} s_{n-1} \cdot \mathrm{Id} \right).$$

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 $x_1, \ldots, x_n$ Variablen.

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

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

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

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

Satz 9

Sei $S \subseteq \mathbb{F}$ endlich und $p(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n] \setminus \{0\}$, d.h. p ist nicht das Null-Polynom. Dann hat die Gleichung $p(x_1, \ldots, x_n) = 0$ 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 $a + a_1 x_1 + \cdots + a_n x_n$.
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 $x_2, \ldots, x_n$ 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 $(a_1, \ldots, a_n) \in S^n$ 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 $\overline{x} = (x_1, \ldots, x_{n-1})$, d.h. $p = p(\overline{x}, x_n)$.
Zu jedem $s \in S$ betrache das Polynom $p(\overline{x}, s) \in \mathbb{F}[\overline{x}]$.

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

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

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

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

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

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

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

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

Probabilistische Tests mit Polynomen

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

$p(\overline{x}, s) = 0$ hat höchstens $\deg(p(\overline{x}, s)) \cdot |S|^{n-2} \leq \deg(p) \cdot |S|^{n-2}$ 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 $p(x_1, \ldots, x_n)$ 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 $(s_1, \ldots, s_n) \in S^n$ zufällig und gleichverteilt auswählen, gilt $\Pr(p(s_1, \ldots, s_n) = 0) \leq d/|S|$.

Anwendung: Perfekte Matchings

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

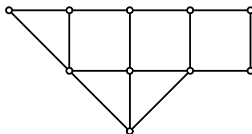
Ein Matching von G ist eine Teilmenge $M \subseteq E$ so, dass zwei verschiedene Kanten aus M keinen Knoten gemeinsam haben.

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.

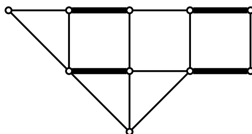
Graphentheorie: Matchings

Beispiel:



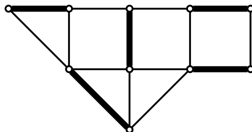
Graphentheorie: Matchings

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 $T_G = (T_{u,v})_{1 \le u,v \le n}$ 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

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

wobei S_n die Menge aller Permutationen von $\{1, \ldots, n\}$ ist, und $\mathrm{sign}(\sigma)$ 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 10

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 10:

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 $(a_1, a_2, \ldots, a_m) \in \{1, \ldots, 2n\}^m$, wobei $m = |E|$ die Anzahl der Variablen von T_G ist.
- Berechne $D = \det(T_G)(a_1, a_2, \ldots, a_m)$ 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 Tuttes Satz, dass $\det(T_G) = 0$. Somit lehnen wir mit Wahrscheinlichkeit 1 ab.

Falls G ein perfektes Matching hat, folgt nach Tuttes 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

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

Anwendung: Perfekte Matchings

Es bleibt, Tuttes 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.

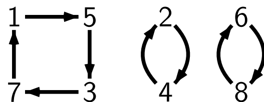
Sei $E_n = \{\sigma \in S_n \mid \sigma \text{ enthält nur Zyklen gerader Länge}\}$.

Anwendung: Perfekte Matchings

Beispiel: Sei die Permutation
$\sigma : \{1,2,3,4,5,6,7,8\} \to \{1,2,3,4,5,6,7,8\}$ 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

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

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

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

Anwendung: Perfekte Matchings

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

Lemma 11

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

Beweis: Wir zeigen

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

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

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 $\sigma_1 = (a_0, a_1, \ldots, a_{\ell-1})$ ungerade Länge $\ell \geq 3$.

Anwendung: Perfekte Matchings

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

Es gilt für alle $0 \le i \le \ell - 1$ (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 $\mathrm{sign}(\sigma) = \mathrm{sign}(\tau)$, woraus dann folgt:

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

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

(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

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

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 $\{1, \ldots, n\}$, 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:

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

Dies beweist das Lemma. □

Anwendung: Perfekte Matchings

Wir können nun den Beweis von Tuttes Satz abschließen:

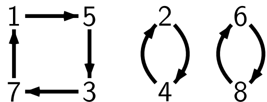
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

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 Tuttes Satz abschließen:

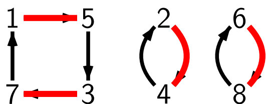
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

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 Determinate ungleich 0 ist.

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

Dies schließt den Beweis von Tuttes 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 $f : \Sigma \to \mathbb{Z}^{2\times 2}$ durch

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

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

Lemma 12

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:

$$a_1 = b_1 + b_3, \qquad a_1 + a_3 = b_3$$
$$a_2 = b_2 + b_4, \qquad a_2 + a_4 = b_4$$
$$b_1 + a_3 = 0, \text{d.h. } a_3 = b_1 = 0, \text{ ein Widerspruch zu } b_1 > 0$$
$$a_4 + b_2 = 0, \text{d.h. } a_4 = b_2 = 0, \text{ ein Widerspruch zu } a_4 > 0$$

Pattern-Matching mit Fingerprints

Nun können wir den Beweis von Lemma 12 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. $f(w) = \mathrm{Id}_2 = f(\varepsilon)$.

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.

Dann führt p zu einem ungültigen Match für (X, Y), falls $X \neq Y$ (d.h. $f(X) \neq f(Y)$, aber $f_p(X) = f_p(Y)$.

Zu einer natürlichen Zahl k bezeichnet $\pi(k)$ die Anzahl der Primzahlen im Intervall $\{1, \dots, k\}$.

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 13

Für $1 \leq i \leq t$ seien X_i und Y_i Strings der Länge m.

Falls m groß genug ist, führt eine zufällig gewählte Primzahl p aus dem Intervall $\{1, \ldots, M\}$ zu einem ungültigen Match mindestens eines Paars (X_i, Y_i) mit einer Wahrscheinlichkeit von höchstens $\frac{\pi(4mt)}{\pi(M)}$.

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:

$$\exists 1 \leq i \leq t : p \text{ führt zu einem ungültigen Match von } (X_i, Y_i)$$

$$\iff$$

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

$$\Longrightarrow$$

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

$$\Longrightarrow$$

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

Die Anzahl verschiedener Primfaktoren von 2^{4tm} ist höchstens $\pi(4mt)$ (falls $4mt \geq 29$).

$\leadsto$ Wahrscheinlichkeit für ein ungültiges Match $\leq \dfrac{\pi(4mt)}{\pi(M)}$. □

Pattern-Matching mit Fingerprints

Lemma 14

Seien (X_i, Y_i) Paare von Strings der Länge m ($1 \leq i \leq t$).

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

Dann führt eine zufällig gewählte Primzahl $p \in \{1, \ldots, M\}$ zu einem ungültigen Match für mindestens ein Paar (X_i, Y_i) mit einer Wahrscheinlichkeit von höchstens $\mathcal{O}\left(\frac{1}{t^{k-1}}\right)$, falls m groß genug ist.

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

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

$$\rightsquigarrow \Pr(\exists \text{ ungült. Match}) \leq \mathcal{O}\left(\frac{m \cdot t \cdot (\ln(m) + k\ln(t))}{\ln(mt) \cdot m \cdot t^k}\right) \leq \mathcal{O}\left(\frac{1}{t^{k-1}}\right)$$

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 15

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

- *Falls $T[i, i+m-1] = P$, dann $MATCH[i] = 1$ mit Wahrscheinlichkeit 1.*
- *Die Wahrscheinlichkeit, dass ein i existiert mit $MATCH[i] = 1$ 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 14 ist die Wahrscheinlichkeit, dass ein Eintrag $\text{MATCH}[i]$ fälschlicherweise auf 1 gesetzt ist, beschränkt durch $\mathcal{O}\left(\frac{1}{n^k}\right)$.

Mittlere Höhe von Suchbäumen

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

Sei $\mathcal{B}_n$ die Menge aller binären Suchbäume auf den Knoten $\{1, \ldots, n\}$ ($\mathcal{B}_0$ 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 $(\pi_1, \pi_2, \ldots, \pi_n)$ von $(1, 2, \ldots, n)$ aus. Jede der $n!$ vielen Permutationen wird mit Wahrscheinlichkeit $1/n!$ ausgewählt.
- Dann wird ein Suchbaum aufgebaut, indem die Elemente $1, \ldots, n$ in der Reihnfolge $\pi_1, \pi_2, \ldots, \pi_n$ 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 $i \in \{1, \ldots, n\}$ 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 16

Für den Erwartungswert

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

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

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

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.

Optimale Suchbäume

Wiederholung (Algorithmik I):

- Knotenmenge des Suchbaums $= \{1, \dots, n\}$.
- $\gamma(k)$: Wahrscheinlichkeit für Zugriff auf Knoten k.
- $P[i,j] = \sum_{v \in [i,j]} \ell_B(v) \cdot \gamma(v)$: gewichtete innere Pfadlänge (weighted inner path length) eines optimalen Suchbaums B für die Knotenmenge $\{i, \dots, j\}$.
- $R[i,j]$: größte Wurzel eines optimalen Suchbaums für $\{i, \dots, j\}$.
- $r[i,j]$: kleinste Wurzel eines optimalen Suchbaums für $\{i, \dots, j\}$.
- $\Gamma[i,j] := \sum_{k=i}^{j} \gamma(k)$: Gesamtgewicht der Knotenmenge $\{i, \dots, j\}$.

In Algorithmik I haben wir einen $O(n^3)$-Algorithmus zur Berechnung optimaler Suchbäume kennengelernt.

Knuth: Verbesserung auf $O(n^2)$.

Monotonie der Wurzel

Knuths Verbesserung ($n^3 \rightsquigarrow n^2$) beruht auf folgendem Satz:

Theorem 17 (Monotonie der Wurzel)

Sei $r[i,j]$ (bzw. $R[i,j]$) die kleinste (bzw. größte) Wurzel eines optimalen Suchbaumes für die Knoten $\{i,\dots,j\}$. Dann gilt für $n \geq 2$:

$$\begin{aligned} r[1,n-1] &\leq r[1,n] \\ R[1,n-1] &\leq R[1,n] \end{aligned}$$

Wegen Links-Rechts-Symmetrie gilt analog:

Theorem 18 (Monotonie der Wurzel)

Für $n \geq 2$ gilt:

$$\begin{aligned} r[1,n] &\leq r[2,n] \\ R[1,n] &\leq R[2,n] \end{aligned}$$

Monotonie der Wurzel

Beweis: Wir beweisen den Satz durch Induktion über die Zahl der Knoten, d. h. wir können ihn für kleineres n bereits als bewiesen annehmen.

Zunächst zeigen wir folgendes Lemma:

Lemma 19 (große Wurzel & minimaler Level für n)

Sei B_j ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit minimalem Level j von Knoten n. Sei y_1 die Wurzel von B_j.

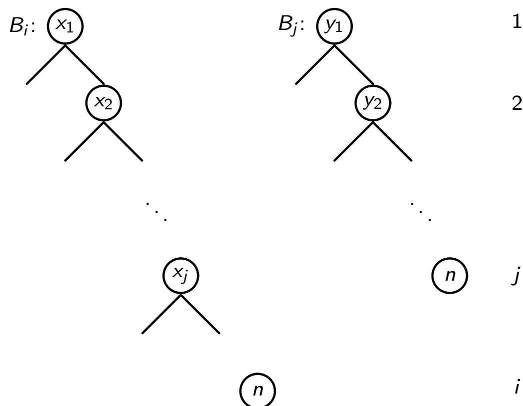
Sei B_i ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel $x_1 > y_1$.

Dann existiert ein optimaler Suchbaum B' für $\{1, \ldots, n\}$ mit Wurzel x_1 und Knoten n auf Level j.

Wir werden sehen, wie diese Verbindung der Eigenschaften minimales Level für n und große Wurzel für den Beweis des Satzes von Nutzen ist.

Monotonie der Wurzel

Beweis des Lemmas: Wir betrachten die rechten Äste der Bäume B_i und B_j, wobei die Knoten von B_i mit x_k und die Knoten von B_j mit y_k bezeichnet sind (siehe folgende Abbildung).



Monotonie der Wurzel

Bei festem x_1 maximieren wir x_2, dann maximieren wir x_3 usw, d.h. $x_{k+1} = R[x_k + 1, n]$.

Der neue Baum wird weiterhin mit B_i bezeichnet.

Falls in B_i und B_j der Knoten n auf gleichem Level j liegt, sind wir fertig.

Andernfalls können wir annehmen, dass der Knoten n in B_i auf Level i liegt und $i > j$ gilt, weil j minimal gewählt wurde.

Sei k maximal mit $x_k > y_k$.

Dann gilt $1 \leq k < j$.

Wegen $y_k + 1 \leq x_k + 1$ und Induktion (Theorem 18) folgt

$$y_{k+1} \leq R[y_k + 1, n] \leq R[x_k + 1, n] = x_{k+1}.$$

Also folgt $y_{k+1} = x_{k+1}$, da k maximal gewählt wurde.

Monotonie der Wurzel

Ausserdem muss $k+1 < j$ gelten, denn sonst würde $n = y_j = y_{k+1} = x_{k+1}$ gelten, d.h. n würde in B_i und B_j auf dem gleichen Level liegen.

Also existiert das rechte Kind von x_{k+1} bzw. y_{k+1}.

Sei nun R_i (R_j) der Unterbaum von B_i (B_j) unterhalb des rechten Kinds von x_{k+1} (y_{k+1}).

Wegen $x_{k+1} = y_{k+1}$ haben R_i und R_j dieselbe Knotenmenge $\{x_{k+1}+1, \ldots, n\}$ und sind optimale Suchbäume.

Wir bilden einen Baum B' durch Ersetzen von R_i in B_i durch R_j.

Da $P(R_i) = P(R_j)$ gilt, ergibt sich auch $P(B') = P(B_i) = P(B_j)$.

Also ist B' ist optimal für $\{1, \ldots, n\}$, hat x_1 als Wurzel und den Knoten n auf Level j. □

Monotonie der Wurzel

Symmetrisch zu Lemma 19 lässt sich zeigen:

Lemma 20 (kleine Wurzel & maximaler Level für n)

Sei B_i ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit maximalem Level i von Knoten n. Sei x_1 die Wurzel von B_i.

Sei B_j ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel $y_1 < x_1$.

Dann existiert ein optimaler Suchbaum B' für $\{1, \ldots, n\}$ mit Wurzel y_1 und Knoten n auf Level i.

Monotonie der Wurzel

Nun zurück zum Beweis von Theorem 17:

Im folgenden bezeichnen wir mit α das Gewicht des größten Knotens n, d.h. $\alpha := \gamma(n)$.

Der Wert α variiert zwischen 0 und ∞.

Sei r_α (R_α) die kleinste (größte) Wurzel eines optimalen Suchbaums für die Knoten $\{1, \dots, n\}$ unter der Bedingung $\alpha = \gamma(n)$.

Monotonie der Wurzel

Behauptung 1: $r[1, n-1] \leq r_0$ bzw. $R[1, n-1] \leq R_0$.

Beachte: Ist B (B') optimaler Suchbaum für $\{1, \ldots, n\}$ ($\{1, \ldots, n-1\}$), so gilt $P(B) = P(B')$ ($\alpha = 0$ ist hier wichtig).

Für $R[1, n-1] \leq R_0$:

- Sei B optimaler Suchbaum für $\{1, \ldots, n-1\}$ mit Wurzel $R[1, n-1]$.
- Füge n ganz rechts zu B hinzu.
- Der resultierende Suchbaum für $\{1, \ldots, n\}$ ist optimal und hat Wurzel $R[1, n-1]$.

Für $r[1, n-1] \leq r_0$:

- Sei B' optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel r_0.
- Entferne n aus B'.
- Der resultierende Suchbaum für $\{1, \ldots, n-1\}$ ist optimal und hat Wurzel r_0.

Optimale Suchbäume

Zusammen mit Behauptung 1 folgt Theorem 17 aus:

Behauptung 2: Wenn $\alpha < \beta$, dann $r_\alpha \leq r_\beta$ und $R_\alpha \leq R_\beta$.

Wir zeigen zunächst die Behauptung $R_\alpha \leq R_\beta$.

Für $i \in \{1, \ldots, n\}$ sei B_i ein optimaler Suchbaum unter der Nebenbedingung, dass der Knoten n auf dem Level i liegt.

Beachte:

- B_i muss nicht insgesamt optimal sein: Es kann sein, dass durch Verschieben von n auf einen anderen Level $\neq i$ ein besserer Suchbaum entsteht.
- B_i ist nicht unbedingt eindeutig.

Wir wählen nun B_i so, dass die Wurzel maximal wird (und weiterhin n auf Level i liegt).

Optimale Suchbäume

Mit $P_\alpha(B_i)$ bezeichnen wir die gewichtete innere Pfadlänge in Abhängigkeit vom Gewicht α des Knotens n.

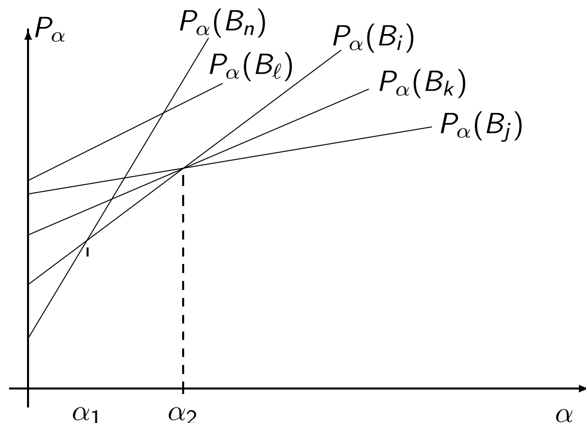
Dann gibt es eine Konstante $c(i)$ mit

$$P_\alpha(B_i) = \alpha \cdot i + c(i),$$

d.h. der Graph $P_\alpha(B_i)$ ist eine Gerade mit Steigung i.

Aufgrund der Linearität erhalten wir das folgende (vertikal gestauchte) Bild, bei der jede Steigung $1 \leq i \leq n$ genau einmal vorkommt.

Optimale Suchbäume



Optimale Suchbäume

Für jedes α ist dann

$$P(\alpha) := \min\{P_\alpha(B_i) \mid 1 \leq i \leq n\}$$

das Gewicht eines optimalen Suchbaums unter der Bedingung $\gamma(n) = \alpha$.

Dann ist $P(\alpha)$ ein konkaver Linienzug mit den Ecken $\alpha_1 < \alpha_2 < \cdots < \alpha_m$. Setze $\alpha_0 = 0$ und $\alpha_{m+1} = \infty$.

Sei i_k so, dass für alle $\alpha \in [\alpha_k, \alpha_{k+1}]$ gilt: $P(\alpha) = P_\alpha(B_{i_k})$, d.h. B_{i_k} ist optimal, falls $\gamma(n) \in [\alpha_k, \alpha_{k+1}]$.

Beachte: Dann gilt für alle $\alpha \in (\alpha_k, \alpha_{k+1})$: $R_\alpha =$ Wurzel von B_{i_k}.

Es genügt, die beiden folgenden Aussagen zu zeigen:

1. Wenn $\alpha_k < \alpha < \alpha_{k+1} = \beta$ dann gilt $R_\alpha \leq R_\beta$.
2. Wenn $\alpha = \alpha_k < \beta < \alpha_{k+1}$ dann gilt $R_\alpha = R_\beta$.

Optimale Suchbäume

Zu 1: Für α gilt: $R_\alpha =$ Wurzel von B_{i_k}.

Ausserdem ist B_{i_k} für $\gamma(n) = \beta$ noch immer ein optimaler Suchbaum.

Also gilt $R_\beta =$ größte Wurzel eines für $\gamma(n) = \beta$ optimalen Suchbaum $\geq R_\alpha$.

Zu 2: Für β gilt: $R_\beta =$ Wurzel von B_{i_k}.

Ausserdem ist B_{i_k} für $\gamma(n) = \alpha$ noch immer ein optimaler Suchbaum.

Wegen Lemma 19 gibt es einen Suchbaum B mit $P_\alpha(B) = P(\alpha)$ (das Optimum bei α) und Wurzel R_α, wo ausserdem n auf dem kleinstmöglichen Level liegt.

Dieser kleinstmögliche Level ist aber i_k.

Also ist B ein optimaler Suchbaum unter der Einschränkung, dass n auf Level i_k liegt, und bei dem ausserdem die Wurzel ($= R_\alpha$) maximal ist.

Also haben B und B_{i_k} die gleiche Wurzel, d.h. $R_\alpha = R_\beta$.

Optimale Suchbäume

Analog zeigt man: Wenn $\alpha < \beta$, dann $r_\alpha \leq r_\beta$.

Man wählt hier die Wurzel von B_i minimal.

Dann gilt für alle $\alpha \in (\alpha_k, \alpha_{k+1})$: $r_\alpha =$ Wurzel von B_{i_k}.

Man zeigt nun die beiden folgenden Aussagen:

1. Wenn $\alpha_k < \alpha < \alpha_{k+1} = \beta$ dann gilt $r_\alpha = r_\beta$.
2. Wenn $\alpha = \alpha_k < \beta < \alpha_{k+1}$ dann gilt $r_\alpha \leq r_\beta$.

Für den Beweis von 1 verwedet man Lemma 20. □

Korollar

Es gilt: $r[i, j-1] \leq r[i, j] \leq r[i+1, j]$.

Optimale Suchbäume

```
cost[n, n+1] := 0;
for i := 1 to n do
   cost[i, i-1] := 0;
   cost[i, i] := γ(i);
   Γ[i, i] := γ(i);
   r[i, i] := i;
endfor
```

```
for d := 1 to n-1 do
   for i := 1 to n-d do
      j := i+d;
      left := r[i, j-1]; right := r[i+1, j];
      root := left;
      t := cost[i, left-1] + cost[left+1, j];
      for k := left+1 to right do
         if cost[i, k-1] + cost[k+1, j] < t then
            t := cost[i, k-1] + cost[k+1, j];
            root := k;
         endif
      endfor
      Γ[i, j] := Γ[i, j-1] + γ(j);
      cost[i, j] := t + Γ[i, j];
      r[i, j] := root;
   endfor
endfor
```

Optimale Suchbäume

Laufzeit: $\sum_{d=1}^{n} \sum_{i=1}^{n-d} (1 + r[i+1, i+d] - r[i, i+d-1]) =$

$$\sum_{d=1}^{n} (n - d + r[n-d+1, n] - r[1, d]) \in \Theta(n^2).$$

Bemerkung: Es wurde ein linear geordnetes Feld $v_1, \ldots, v_n$ von Knoten vorausgesetzt. Ein solches Feld erhält man aus einem ungeordneten Feld in Zeit $\mathcal{O}(n \log n)$ Schritten.

Damit gilt: Aus einem beliebigen Feld mit n Elementen kann ein optimaler Suchbaum in Zeit $\Theta(n^2 + n \log n) = \Theta(n^2)$ Schritten erzeugt werden.