Zusammenfassung

Zur Diplomprüfungsvorbereitung

„Algorithmentheorie“

„Prof. Dr Thomas Ottmann“
„PD Dr. A.Heinz“

Wintersemester 2002

Zusammengefasst von Markus Krebs und Andreas Horstmann

November 2002

Vorlesung 1 – „Divide and Conquer“

Entwurfsprinzipien für Algorithmen

- Divide and Conquer
- Randomisierung
- Dynamische Programmierung
- Greedy Prinzip
- Backtracking
- Branch and Bound
- Amortisierung (Analysetechnik)

Klassen von Algorithmen

- Geometrische Algorithmen
- Mathematische Algorithmen
- Zeichenketten Verarbeitung
- Graphen Algorithmen
- Internet Algorithmen

Das Divide and Conquer Prinzip:

1. Divide: Teile das Problem in zwei oder mehr (k>=2) Teilprobleme, wenn N>c ist; sonst löse das Problem der Größe <=c direkt.

2. Conquer: Löse die Teilprobleme auf dieselbe Art (rekursiv).

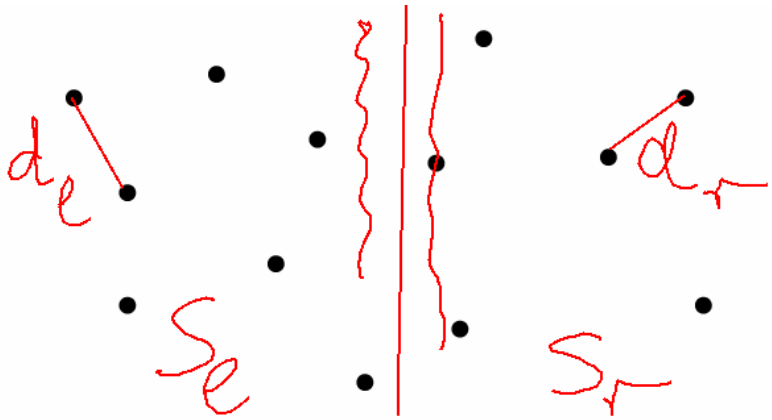
3. Merge: Füge die Teillösungen zur Geamtlösung zusammen.

Beispiel: Quicksort

Geometrisches Divide and Conquer

Algorithmus: Closest Pair Problem

Bestimme für eine Menge S von n Punkten, das Paar mit minimaler Distanz



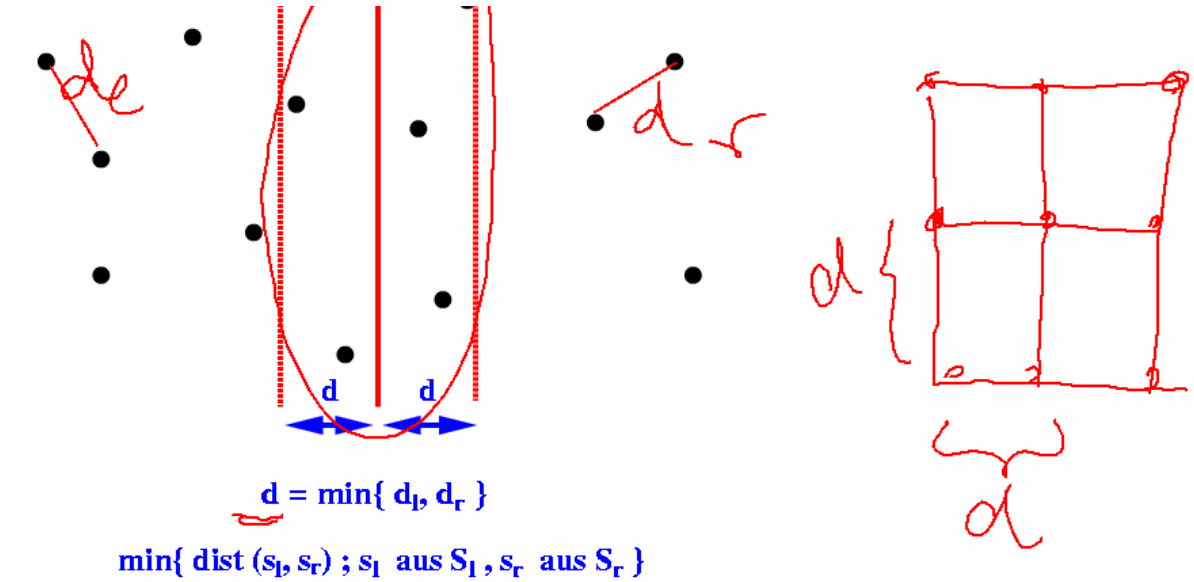
Der naive Algorithmus zur Lösung benötigt $O(n^2)$ – Paarweise Vergleiche aller Punkte

Besser geht es mit D&C in $O(n \log^2 n)$ aber Worst-Case $\Omega(n^2)$; Worst Case tritt dann ein, wenn alle Punkte sehr nahe an der Trennlinie liegen.

Divide:
Teile S in S_l und S_r

Conquer:
d_l:= mindist(S_l); d_r:= mindist(S_r);

Merge:
Mindist:=min{d_l, d_r, min{dist(s_l, s_r); s_l aus S_l, s_r aus S_r}}

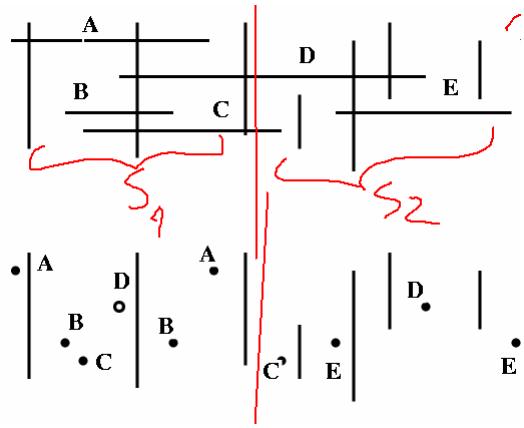


1. Sortiere Punkte im Streifen mit Breite 2*d nach aufsteigenden y-Werten
2. Betrachte zu jedem Punkt p alle Punkte q mit y-Abstand höchstens d

Zeitaufwand für Merge Schritt: $O(n \log n)$
Zeitaufwand Gesamt: $O(n \log^2 n)$

Algorithmus: Liniensegmentschnitt

Bestimme alle Paare sich schneidender Segmente



Naiver Algorithmus benötigt hier auch wieder $O(n^2)$

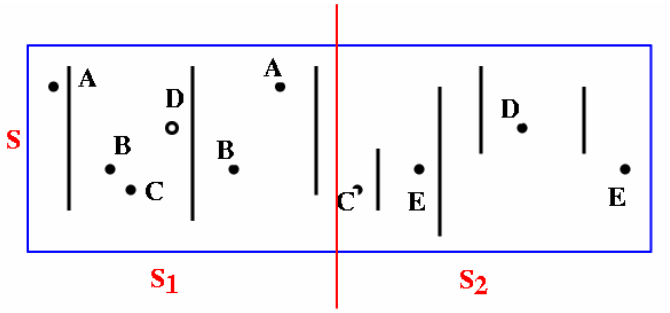
Besser Divide and Conquer (Bild) – Repräsentiere die horizontalen Segmente durch ihren Anfangs und Endpunkt, dadurch lässt sich die Menge gut aufteilen.

Funktion: ReportCuts (S)

⇨ liefert alle Paare von sich schneidenden vertikalen Segmenten in S und horizontalen Segmenten mit linkem oder rechtem Endpunkt in S

Divide:
Teile S in S_1 und S_2 , falls #(S)>1

Conquer:
ReportCuts(S_1); ReportCuts(S_2);



Merge:
Berichte alle Schnitte zwischen vertikalen Segmenten in S_1 und horizontalen Segmenten mit rechtem Endpunkt in S_2, deren linker Endpunkt weder in S_1 noch in S_2 liegt.

Aufwand
Sei k die Anzahl der Schnitte.
Somit ergibt sich:

T(N) = 2*T(n/2)+a*n+Ausgabez.
= O(n log n) +k

Die Komplexität des Algorithmus haengt nun ab, von der Ausgabe, d.h. wenn es n^2 viele Schnitte gibt, dann benötigt dieser auch $O(n^2)$.

Kompletter Algorithmus im Detail

Input: Je eine Menge horizontaler Segmente H und vertikaler Segmente V

Output: Die Menge aller sich schneidender Paare (h, v) mit h ∈H und v ∈V

Schritt 1: Sei $\overline{H}$ die Menge der von H definierten linken und rechten Endpunkte. Sei S = $\overline{H}$ ∪ V, nach x–Koordinate sortiert.

Schritt 2: LINSECT (S, LEFT, RIGHT, VERT)

end DAC–Segmentschnitt

Ein/Ausg. Paare

Basisfälle:

Fall 1: S **enthält nur ein Element** s

Fall 1.1: $s = (x,y)$ **ist ein linker Endpunkt**

$L = \{s\}$ $R = \emptyset$ $V = \emptyset$

Fall 1.2: $s = (x,y)$ **ist ein rechter Endpunkt**

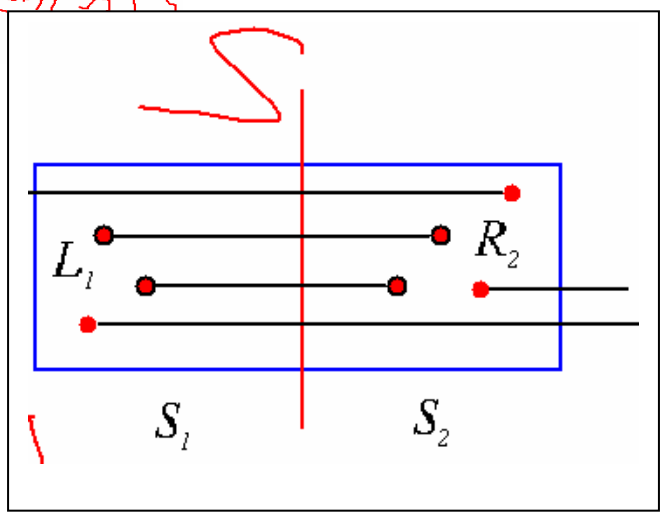
$L = \emptyset$; $R = \{y\}$; $V = \emptyset$

Fall 1.3: $s = (x,y_1,y_2)$ **ist ein vertikales Segment**

$L = \emptyset$ $R = \emptyset$ $V = \{[y_1, y_2]\}$

Fall 2: S enthält mehr als nur ein Element

Divide:
Wähle eine Koordinate x_m die S ind zwei etwas gleichgroße Teilmengen S_1 und S_2 aufteilt.

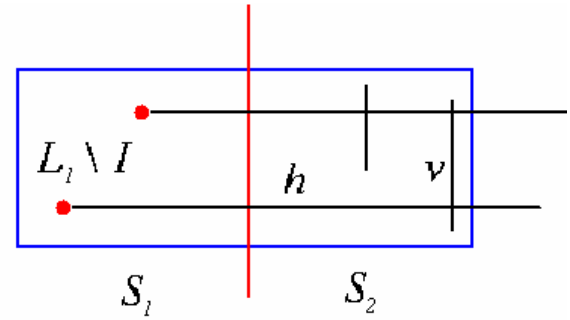


Conquer:
$(L_1,R_1,V_1) = SegSchnitt(S_1)$
$(L_2,R_2,V_2) = SegSchnitt(S_2)$

Merge:
$L = L_2 \cup (L_1 \setminus I)$
$I = L1 \cap R_2$
$R = R_1 \cup (R_2 \setminus I)$
$V = V_1 \cup V_2$

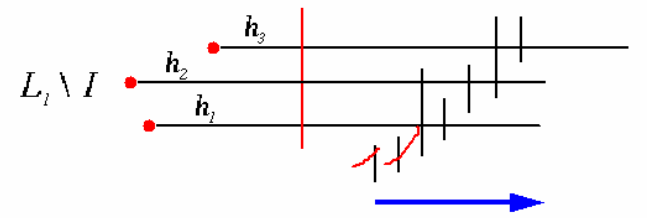
Ausgabe 1. :
Gebe alle Paare von Segmenten (h,v) aus mit

- der linke Endpunkt von h ist in $L_1 \setminus I$ enthalten
- die y-Koordinate von h ist in v enthalten
- $v \in V_2$



Annahme:

- $L_1 \setminus I$ ist nach aufsteigenden y-Koordinaten sortiert
- V_2 ist nach aufsteigenden unteren Endpunkten sortiert



Ausgabe 2.:
Gebe alle Paare von Segmenten (h,v) aus mit

- Der rechte Endpunkt von h ist in $R_2 \setminus I$ enthalten
- Die y-Koordinate von h ist in v enthalten
- $v \in V_1$

Repräsentation der Mengen:

S – nach x Koordinaten sortiertes Array
L,R,I – nach y-Koordinaten sortierte verkettete Listen
V – nach der unteren Intervallgrenze sortierte, verkettete Liste von Intervallen

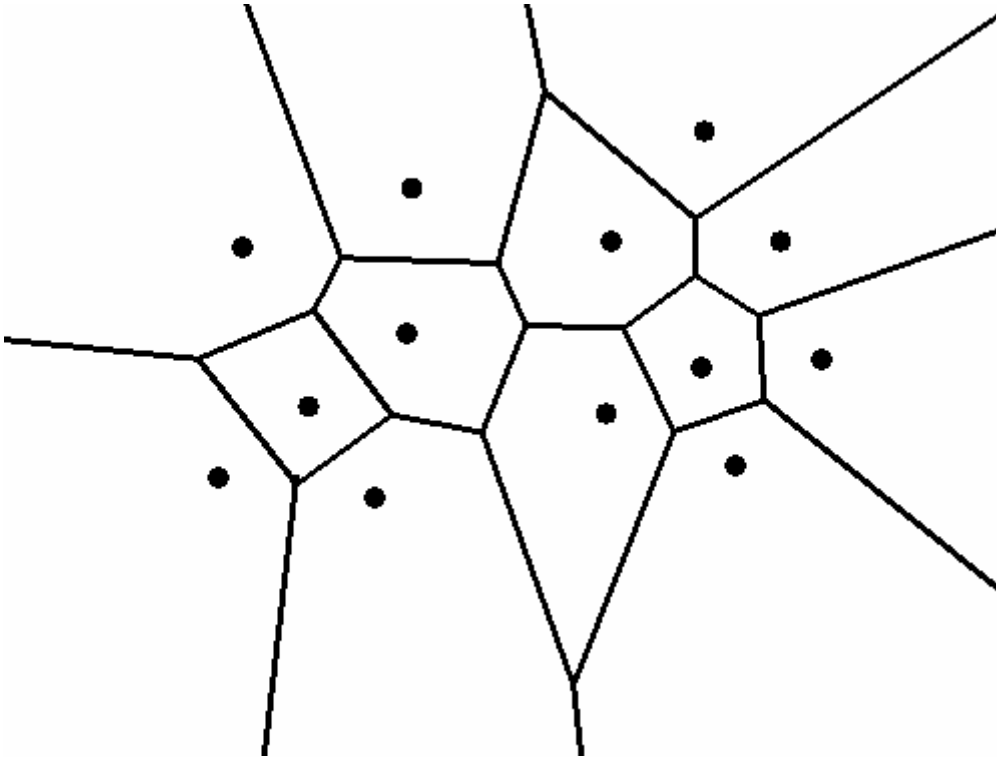
Analyse:

$T(1) = O(1)$
$T(n) = O(1) + 2\ T(n/2) + O(n)$, falls $n>1 = O((n \log n) + k)$

Das Voronoi Diagramm

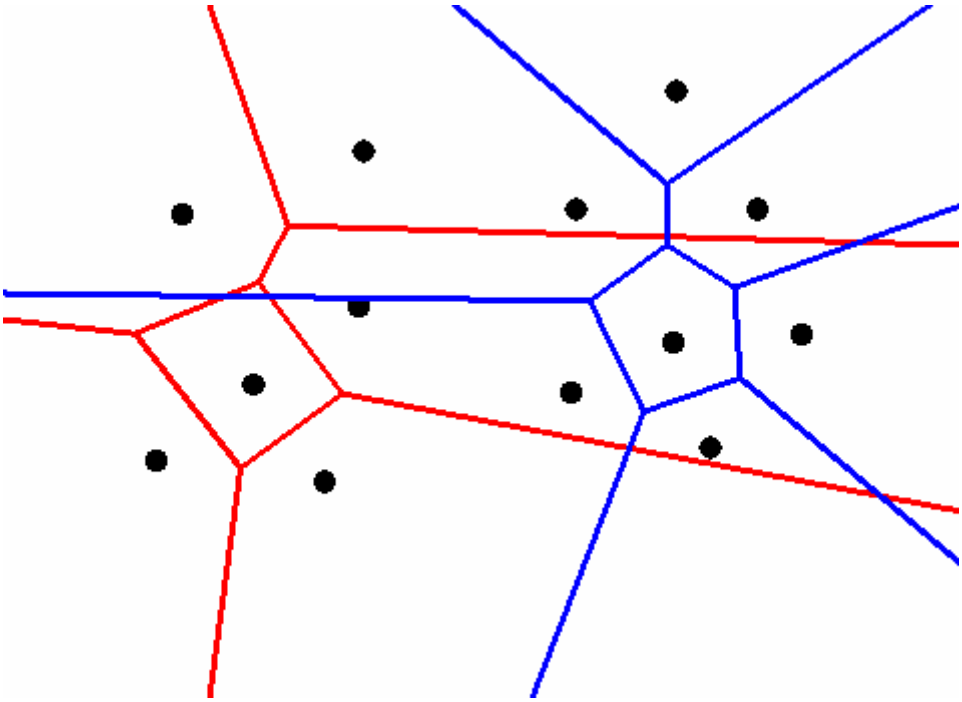
Gutes Applet unter: http://wwwpi6.fernuni-hagen.de/Geometrie-Labor/VoroGlide/

Gegeben: Eine Menge von Punkten (sites)
Gesucht: Eine Unterteilung der Ebene in Regionen gleicher nächster Nachbarn

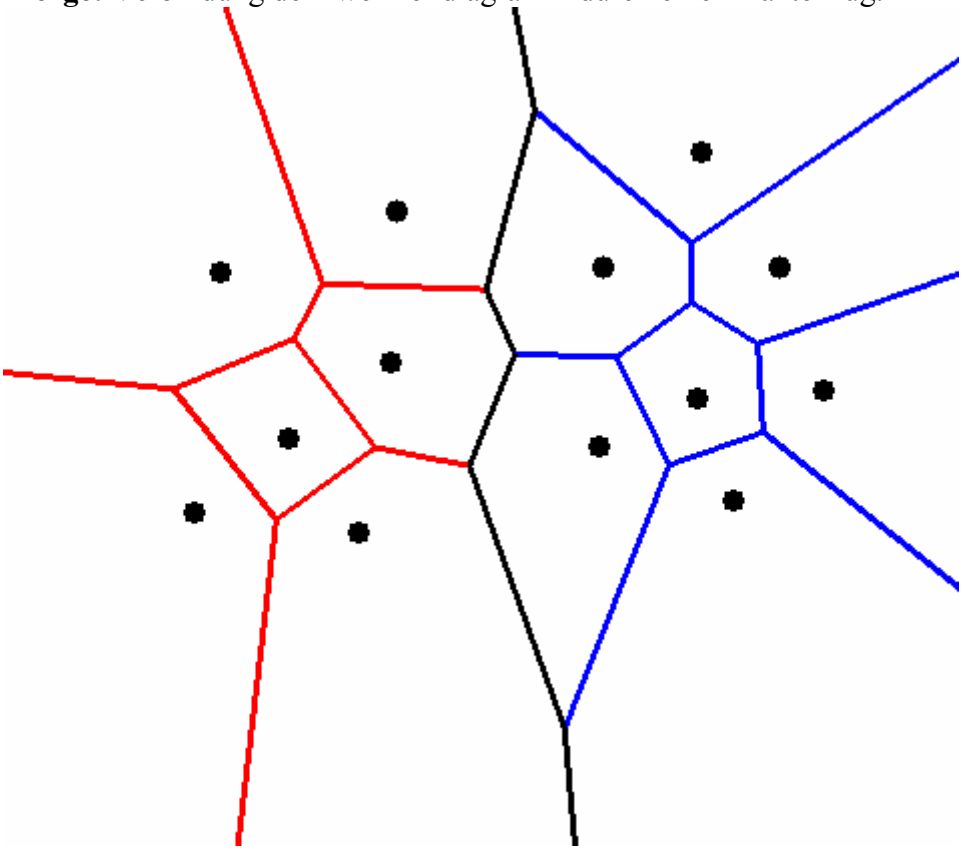


Lösung: Divide And Conquer

Divide:
Einteilung der Punktmenge in zwei Hälften
Conquer:
Rekursive Berechnung der beiden kleineren Voronoi Diagramme
Abbruchbedingung:
Voronoi Diagramm eines einzelnen Punktes ist die Gesamte Ebene



Merge: Verbindung der zwei Teildiagramm durch einen Kantenzug.



Beschreibung:

- Es wird oben im unendlichen begonnen.
- Man bildet die Senkrechte zu den beiden obersten Punkten, die genau in der Mitte der beiden Punkte liegt.
- Trifft man auf eine blaue oder eine rote Linie dann nimmt man entweder einen neuen blauen oder einen neuen roten Punkt aus einem offenen Gebiet, welches in Richtung

der anderen Farbe zeigt, um dort die Senkrechte erneut zwischen dem neuen (blauen oder roten Punkt) und dem dem alten roten bzw. blauen Punkt zu bilden.
- Die blaue bzw. rote Linie muss dann entsprechend abgeschnitten werden

Laufzeit: O(n log n)

Vorlesung 2 – Fast Fourier Transformation (FFT)

Gegeben sei ein Polynom p mit Variablen x:

$$p(x) = a_n x^n + \cdots + a_1 x^1 + a_0$$

Def.: Der Grad von p ist die höchste Potenz von p (=n)

Def.: $\mathbb{R}[x]$ ist die Menge aller reeller Polynome

Operationen die auf Polynomen angewandt werden können (mit Koeffizientendarstellung):

1. Addition zweier Polynome p und q:

Relativ einfach: Aufsummieren der korrespondierenden (mit gleicher Potenz) Koeffizienten

$$\begin{aligned} p(x) + q(x) &= (a_n x^n + \cdots + a_0) + (b_n x^n + \cdots + b_0) \\ &= (a_n + b_n)x^n + \cdots + (a_1 + b_1)x^1 + \end{aligned}$$

dies geht in O(n) Zeit

2. Multiplikation zweier Polynome p und q:

$$\begin{aligned} p(x)q(x) &= (a_n x^n + \cdots + a_0)(b_n x^n + \cdots + b_0) \\ &= c_{2n} x^{2n} + \cdots + c_1 x^1 + c_0 \end{aligned}$$

$$\Rightarrow c_i = \sum_{j=0}^{i} a_j b_{j-i}, \quad i = 0, \ldots, 2n.$$

$$a_{n+1} = \cdots = a_{2n} = 0, \; b_{n+1} = \cdots = b_{2n} = 0$$

Problem: Wenn man zwei Polynome mit Grad n miteinander multipliziert, kommt ein Polynom mit Grad 2n heraus. Deshalb setzen wir die Koeffizienten von p und q die höher sind als n auf „0“.

Das Ganze geht naiv in $O(n^2)$ bzw. $O(n^{1.58})$

Man spricht bei $\mathbb{R}[x]$ von einem Polynomring
(Erfüllt die Vorraussetzungen für einen math. Ring)

3. Auswerten eines Polynoms p an einer bestimmten Stelle x_0 mit dem „Horner Schema“

$$p(x_0) = (\cdots(a_n x_0 + a_{n-1})x_0 + \cdots + a_1)x_0 + a_0$$

Dies geht in O(n)

Möglichkeiten zur Darstellung (Repräsentation) eines Polynoms

Koeffizientendarstellung:

$$p(x) = a_n x^n + \cdots + a_1 x^1 + a_0$$

Nullstellenprodukt:

$$p(x) = a_n(x - x_1)\cdots(x - x_n)$$

Punkt-/Wertdarstellung:

Dazu benötigen wir folgendes Lemma:

Jedes Polynom p(x) aus $\mathbb{R}[x]$ vom Grad n ist eindeutig bestimmt durch n+1 Paare $(x_i,p(x_i))$ mit i=0,…,n und $x_i \neq x_j$ für $i \neq j$
Beispiel:
Die Paare (0,0),(1,6,),(2,0),(3,0) legen das Polynom:

p(x)=3x(x-2)(x-3) eindeutig fest
(Das ist jetzt relativ einfach gewesen, da wir als Wertpaare 3 Nullstellen hatten, sonst nicht so einfach)

Operationen auf Polynomen mit Hilfe der Punkt/Wertdarstellung

$$p = (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$
$$q = (x_0, z_0), (x_1, z_1), \ldots, (x_n, z_n)$$

Addition zweier Polynome p und q:

$$p + q = (x_0, y_0 + z_0), (x_1, y_1 + z_1), \ldots, (x_n, y_n + z_n)$$

dies geht in O(n)

Produkt zweier Polynome p und q:

$$p \cdot q = (x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), \ldots, (x_n, y_n \cdot z_n)$$

Vorraussetzung n≥ Grad(pq)

Die geht in O(n) (zum Vergleich bei der Koeffizientendarstellung dauerte es $O(n^2)$)

Auswertung an einer Stelle x’

Dazu muss man nun wieder in Koeffizientendarstellung umwandeln

Polynomprodukt unter Zuhilfenahme der Vorteile von beiden Darstellungen

Damit können wir die Berechnung des Polynomprodukt beshleunigen

Berechnung des Produkts zweier Polynome p und q vom Grad $\leq n$

p, q Grad n, $n+1$ Koeffizienten

↓ **Auswertung:** $x_0, x_1, \ldots, x_{2n}$

$2n+1$ Punkt/Wertpaare $(x_i, p(x_i))$ und $(x_i, q(x_i))$

↓ **Punktweise Multiplikation**

$2n+1$ Punkt/Wertpaare $(x_i, pq(x_i))$

↓ **Interpolation**

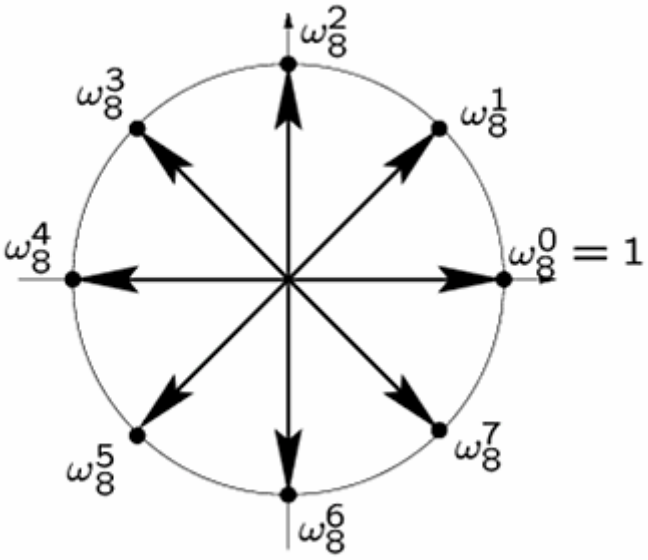
pq Grad $2n$, $2n+1$ Koeffizienten

Repräsentation von p(x)

Annahme: Grad(p)=n-1

Werte an den n Potenzen der n-ten komplexen Haupteinheitswurzel $\omega_n = e^{2\pi i/n}$

$i = \sqrt{-1} \quad e^{2\pi i} = 1$



Potenzen von wn (Einheitswurzeln):

$1 = \omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$

Diskrete Fourier Transformation (DFT)

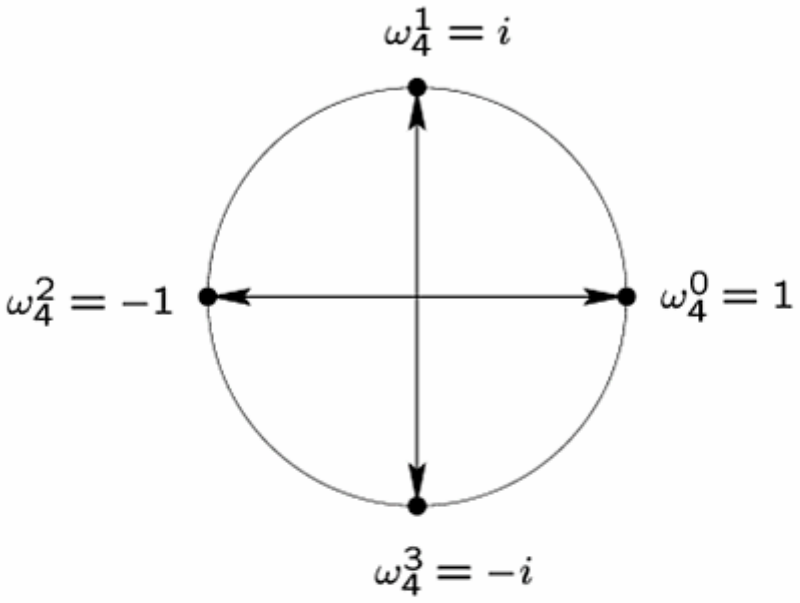
Werte von p für die n Potenzen von w_n legen p eindeutig fest, falls Grad(p)<n

$$DFT_n(p) = (p(\omega_n^0), p(\omega_n^1), \ldots, p(\omega_n^{n-1}))$$

Beispiel: n = 4

$$e^{ix} = \cos(x)+i(\sin(x))$$

$$\omega_4^0 = e^{0i} = \cos(0) + i\sin(0) = 1$$
$$\omega_4^1 = e^{2\pi i/4} = \cos(\pi/2) + i\sin(\pi/2) = i$$
$$\omega_4^2 = \left(e^{2\pi i/4}\right)^2 = \cos\pi + i\sin\pi = -1$$
$$\omega_4^3 = \left(e^{2\pi i/4}\right)^3 = \cos(3\pi/2) + i\sin(3\pi/2) = -i$$



$$p(x) = 3x^3 - 15x^2 + 18x$$

$$\begin{aligned}
(\omega_4^0, p(\omega_4^0)) &= (1, p(1)) = (1, 6) \\
(\omega_4^1, p(\omega_4^1)) &= (i, p(i)) = (i, 15 + 15i) \\
(\omega_4^2, p(\omega_4^2)) &= (-1, p(-1)) = (-1, -36) \\
(\omega_4^3, p(\omega_4^3)) &= (-i, p(-i)) = (-i, 15 - 15i)
\end{aligned}$$

$$DFT_4(p) = (6, 15 + 15i, -36, 15 - 15i)$$

Diskurs: Eigenschaften von Einheitswurzeln

$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ bilden eine multiplikative Gruppe

Kürzungslemma

Für alle $n > 0$, $0 \leq k \leq n$, und $d > 0$ gilt:

$$\omega_{dn}^{dk} = \omega_n^k$$

(wenn oben und unten der gleiche Wert d ist, kann man diesen kürzen)

Beweis (sehr intuitiv)**:**

Oben haben wir gesehen, dass: $\omega_n = e^{2\pi i/n}$

Dies wenden wir nun an: $\omega_{dn}^{dk} = e^{2\pi idk/(dn)} = e^{2\pi ik/n} = \omega_n^k$

Halbierungslemma

Die Menge der Quadrate der 2n komplexen 2n-ten Einheitswurzeln

$$\{(\omega_{2n}^0)^2, (\omega_{2n}^1)^2, \ldots, (\omega_{2n}^{2n-1})^2\}$$

ist gleich der Menge der n komplexen n-ten Einheitswurzeln

$$\{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$$

Beweis: es gilt nach dem Kürzungslemma $(\omega_{2n}^k)^2 = \omega_{2n}^{2k} = \omega_n^k$
Angewendet auf jeden Eintrag erhalten wir die Menge der n komplexen n-ten Einheitswurzeln

Summationslemma

Für alle $n > 0$, $j \geq 0$ mit $n \not| j$ gilt:

$$\sum_{k=0}^{n-1} \omega_n^{jk} = 0.$$

Dies kann man mit Hilfe der geometrischen Reihe beweisen

Fast Fourier Transformation (FFT)

Berechnung der Diskreten Fourier Transformation ($DFT_n(p)$) mit Hilfe des D&C Ansatzes

Wie können wir nun das Polynom geschickt aufteilen um D&C anwenden zu können:

Wir nehmen an n sei gerade:

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \\ &= a_0 + a_2 x^2 + \cdots + a_{n-2} x^{n-2} + \\ &\quad a_1 x + a_3 x^3 + \cdots + a_{n-1} x^{n-1} \\ &= a_0 + a_2 x^2 + \cdots + a_{n-2} (x^{(n-2)/2})^2 + \\ &\quad x\left(a_1 + a_3 x^2 + \cdots + a_{n-1} (x^{(n-2)/2})^2\right) \\ &= p_0(x^2) + x p_1(x^2) \end{aligned}$$

$$p_0(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{(n-2)/2}$$

$$p_1(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{(n-2)/2}$$

Wir teilen so auf, dass ein Teilpolynom nur aus geraden Monomen (p_0) besteht und das Andere nur aus ungeraden Monomen (p_1)

Auswertung für k=0,…,n-1

$$p(\omega_n^k) = p_0((\omega_n^k)^2) + \omega_n^k p_1((\omega_n^k)^2)$$

$$= \begin{cases} p_0(\omega_{n/2}^k) + \omega_n^k p_1(\omega_{n/2}^k), \\ \qquad \text{falls } k < n/2 \\ p_0(\omega_{n/2}^{k-n/2}) + \omega_n^k p_1(\omega_{n/2}^{k-n/2}), \\ \qquad \text{falls } k \geq n/2 \end{cases}$$

.2 Fälle

Beispiel:

$$p(\omega_4^0) = p_0(\omega_2^0) + \omega_4^0 p_1(\omega_2^0)$$
$$p(\omega_4^1) = p_0(\omega_2^1) + \omega_4^1 p_1(\omega_2^1)$$
$$p(\omega_4^2) = p_0(\omega_2^0) + \omega_4^2 p_1(\omega_2^0)$$
$$p(\omega_4^3) = p_0(\omega_2^1) + \omega_4^3 p_1(\omega_2^1)$$

Wertet man nun das Polynom aus, stellt man fest, dass bestimmte Teile doppelt vorkommen (rote Kreise). Deshalb kann man den Algorithmus an diesen Stellen verbessern. Zunächst zeigen wir aber nun den Divide and Conquer Ansatz ohne diese Verbesserung.

Berechnung der DFT im einzelnen:

$$DFT_n(p) = (p(\omega_n^0), p(\omega_n^1), \ldots, p(\omega_n^{n-1}))$$

Der Basisfall (dann wenn nicht mehr geteilt wird) ist:

n=1 d.h. dass Grad(p)=n-1=0

Als Ergebnis erhalten wir dann $DFT_1(p) = a_0$

Sind wir nicht auf der Ebene des Basisfalls, dann:

Folgen weitere
„Divide“ - Schritte,
wir teilen also p weiter auf in p_0 und p_1

Im **„Conquer“** Schritt folgt nun die
rekursive Berechnung von $DFT_{n/2}(p_0)$ und $DFT_{n/2}(p_1)$

im anschließenden **„Merge“** Schritt
werden die Teillösungen wieder zusammengefügt
berechne als für k=0,…n-1

$$DFT_n(p)_k = (DFT_{n/2}(p_0), DFT_{n/2}(p_0))_k + \omega_n^k \cdot (DFT_{n/2}(p_1), DFT_{n/2}(p_1))_k$$

Eine kleine Verbesserung

Nun wenden wir die oben bereits angesprochene Verbesserung an:

$$p(\omega_n^k) = \begin{cases} p_0(\omega_{n/2}^k) + \omega_n^k p_1(\omega_{n/2}^k), \\ \qquad \text{falls } k < n/2 \\ p_0(\omega_{n/2}^{k-n/2}) + \omega_n^k p_1(\omega_{n/2}^{k-n/2}), \\ \qquad \text{falls } k \geq n/2 \end{cases}$$

$$= \begin{cases} p_0(\omega_{n/2}^k) + \omega_n^k p_1(\omega_{n/2}^k), \\ \qquad \text{falls } k < n/2 \\ p_0(\omega_{n/2}^{k-n/2}) - \omega_n^{k-n/2} p_1(\omega_{n/2}^{k-n/2}), \\ \qquad \text{falls } k \geq n/2 \end{cases}$$

Im 2. Fall haben wir nun eine Umformumg, die das Ganze verbessert.

Also falls k < n/2

$$p_0(\omega_{n/2}^k) + \omega_n^k p_1(\omega_{n/2}^k) = p(\omega_n^k)$$

$$p_0(\omega_{n/2}^k) - \omega_n^k p_1(\omega_{n/2}^k) = p(\omega_n^{k+n/2})$$

Wir berechnen nun keine Werte mehr doppelt, denn wir können bereits berechnet Werte wieder verwenden (siehe Bild weiter oben)
Der 1. und der 2. Fall berechnen gleiche Teilterme, die wir uns zu Nutze machen können.
Beim 2. Fall wird immer nur n/2 abgezogen, aber, da k größer ist als n/2, kommt der berechnet Wert eigentlich doppelt vor und kann somit wieder verwendet werden.

Beispiel für die verbesserte Variante:

$$
\begin{aligned}
p(\omega_4^0) &= p_0(\omega_2^0) + \omega_4^0 p_1(\omega_2^0) \\
p(\omega_4^1) &= p_0(\omega_2^1) + \omega_4^1 p_1(\omega_2^1) \\
p(\omega_4^2) &= p_0(\omega_2^0) - \omega_4^0 p_1(\omega_2^0) \\
p(\omega_4^3) &= p_0(\omega_2^1) - \omega_4^1 p_1(\omega_2^1)
\end{aligned}
$$

Der fertige Algorithmus

Algorithmus *FFT*

Input: Ein Array a mit den n Koeffizienten eines Polynoms p und $n = 2^k$

Output: $DFT_n(p)$

```
1   if n = 1 then /* p ist konstant */
2                 return a
3   d^[0] = FFT((a_0, a_2, ..., a_{n-2}), n/2)
4   d^[1] = FFT((a_1, a_3, ..., a_{n-1}), n/2)
5   ω_n = e^{2πi/n}
6   ωnk = 1
7   for k = 0 to n/2 − 1 do
        /* ωnk = ω_n^k */
8       d_k = d_k^[0] + ωnk · d_k^[1]
9       d_{k+n/2} = d_k^[0] − ωnk · d_k^[1]
10      ωnk = ω_n · ωnk
11  return d
```

Ein komplettes Beispiel durchgerechnet:

$$p(x) = 3x^3 - 15x^2 + 18x + 0$$

$a = [0, 18, -15, 3]$

$a^{[0]} = [0, -15]$

$a^{[1]} = [18, 3]$

$FFT([0, -15], 2) = [(-15, 0), (15, 0)]$

$FFT([18, 3], 2) = [(21, 0), (15, 0)]$

$k = 0: \quad \omega nk = 1$

$d_1 = (-15,0) + 1 \cdot (21,0) = (6,0)$

$d_3 = (-15,0) - 1 \cdot (21,0) = (-36,0)$

$k = 1: \quad \omega nk = i$

$d_2 = (15,0) + i \cdot (15,0) = (15,15)$

$d_4 = (15,0) - i \cdot (15,0) = (15,-15)$

$FFT(a, 4) = [(6, 0), (15, 15), (-36, 0), (15, -15)]$

Zeitanalyse des Algorithmus

T(n)= Zeit um ein Polynom vom Grad < n an den Stellen $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ Auszuwerten.

T(1) = O(1)
T(n) = 2T(n/2)+O(n)
= O(n log n)

nun wollen wir unsere Punkt/Wert Darstellung wieder in die Koeffizientendarstellung bringen. Dazu benötigen wir die nun folgende Interpolation:

Interpolation

Gegeben sind die Punkt/Wert-Paare

$(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ mit $x_i \neq x_j$, für alle $i \neq j$

Gesucht ist das Polynom p mit den Koeffizienten $a_0,\ldots,a_{n-1}$ so dass

$$\begin{aligned} p(x_0) &= a_0 + a_1 x_0 + \cdots + a_{n-1} x_0^{n-1} = y_0 \\ p(x_1) &= a_0 + a_1 x_1 + \cdots + a_{n-1} x_1^{n-1} = y_1 \\ &\vdots \qquad\qquad \vdots \\ p(x_{n-1}) &= a_0 + a_1 x_{n-1} + \cdots + a_{n-1} x_{n-1}^{n-1} = y_{n-1} \end{aligned}$$

wir haben nun also ein Gleichungssystem mit n Gleichungen

Wir schreiben das Gleichungssystem als Matrix

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^{n-1} \\ 1 & x_1 & \cdots & x_1^{n-1} \\ & & \vdots & \\ 1 & x_{n-1} & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Diese Matrix könnten wir leicht direkt lösen, aber in $O(n^3)$

Wir haben nun aber den Spezialfall, dass alle X_i Einheitswurzeln sind, das heisst es gilt:

$x_i = \omega_n^i$

Definition:

$V_n = (\omega_n^{ij})_{i,j}, \qquad a = (a_i), \qquad y = (y_i)$

V_n ist die Matrix aus den Einheitswurzeln
a sind die gesuchten Koeffzienten
y sind die Funkionswerte zu den Einheitswurzeln

um a zu berechnen können wir nun umformen nach folgender Regel

$V_n a = y \qquad \Rightarrow \qquad a = V_n^{-1} y$

Wir müssen nun also die Inverse von V_n berechnen, für diese gilt:

Für alle $1 \leq i, j \leq n-1$ gilt:

$$(V_n^{-1})_{ij} = \frac{\omega_n^{-ij}}{n}$$

Den zugehörigen Beweis überspringen wir hier ☺

Hinweis: $V_n * V_n^{-1}$ muss die Einheitsmatrix ergeben, dies wird in dem Beweis geprüft.

Berechnen der a_i

$$
\begin{aligned}
a_i &= (V_n^{-1} y)_i \\
&= \left(\frac{1}{n}, \frac{\omega_n^{-i}}{n}, \frac{\omega_n^{-2i}}{n}, \ldots, \frac{\omega_n^{-i(n-1)}}{n} \right) \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \\
&= \sum_{k=0}^{n-1} y_k \frac{\omega_n^{-ik}}{n} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} y_k (\omega_n^{-i})^k
\end{aligned}
$$

$$
a = \frac{1}{n} \left(\sum_{k=0}^{n-1} y_k (\omega_n^{-0})^k, \sum_{k=0}^{n-1} y_k (\omega_n^{-1})^k, \ldots, \sum_{k=0}^{n-1} y_k (\omega_n^{-(n-1)})^k \right)
$$

$$
r(x) = y_0 + y_1 x + y_2 x^2 + \cdots + y_{n-1} x^{n-1}
$$

$$
a = \frac{1}{n} \left(r(\omega_n^{-0}), r(\omega_n^{-1}), \ldots, r(\omega_n^{-(n-1)}) \right)
$$

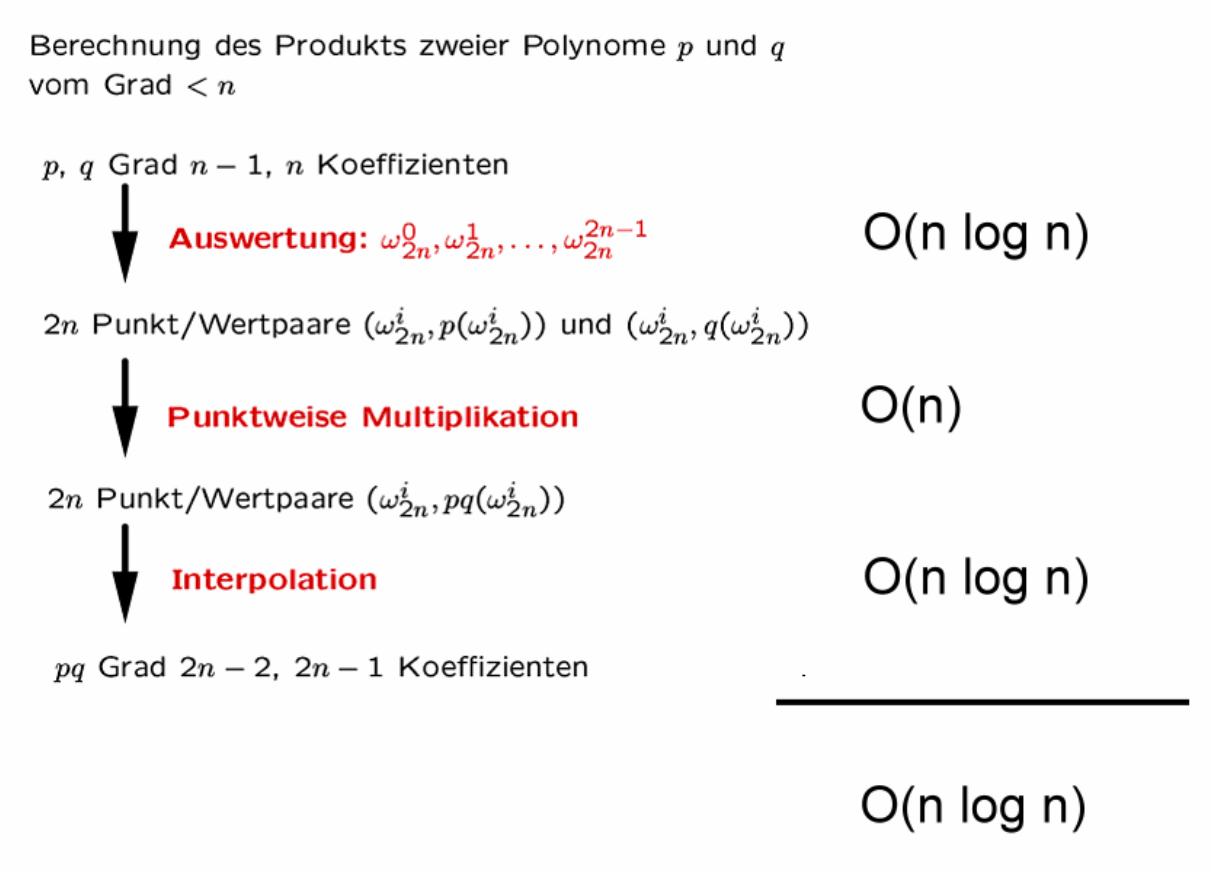
a ist ein Vektor mit den Einträgen a_i für i=0,…,n-1

r ist ein Polynom mit den Koeffizienten y_i in welches als x jeweils die Einheitswurzel eingesetzt wird. Alle Summen wenden eigentlich die gleiche Funktion an und sind sogar gleich, wenn man die w weglässt, deshalb können wir die Funktion r nehmen und sie mit den Einheitswurzeln füttern. Damit erhalten wir den Vektor a mit den Koeffizienten a_i.

unser Teilterm (in der Klammer) ist nun im Prinzip eine DFT_n. die nur noch gespiegelt werden muss.

$$
\omega_n^{-i} = \omega_n^{n-i} \quad \Rightarrow \quad a_i = \frac{1}{n} DFT_n(r)_{n-i} \qquad (n \neq 0)
$$

Komplexität



Vorlesung 3 – Zufallszahlengeneratoren

- Rechner generieren nur Pseudo-Zufallszahlen
- Systemfunktionen wie rand oder RAND0 sind nicht so gut (→ Systemabhängigkeit, unbekannte statistische Eigenschaften

Lineare Kongruenzmethode (Lehmer, 1951)

- Man spricht auch vom Lehmer-Generator

Z_0 Startwert
Z_{n+1} = $f(z_n)$, für n=1,2,…

f(z) = a * z mod m
Multiplikator: a € {2,…,m-1}
Modul:m, (Große Primzahl)

Periode von f:
→ Anzahl verschiedener Werte für z_n

Reduktion auf (0,1): $u_n=z_n/m$ (Erstellen einer Zufalls-zahl zwischen 0 und 1

Beispiele:

f(z)= 6z mod 13
→ (1,6,10,8,9,2,12,7,3,5,4,11,1…) volle Periode (alle Zahlen von 1…12 abgedeckt)

f(z)= 7z mod 13
→ (1,7,10,5,9,11, 12,6,3, 8,4,2,1) volle Periode, aber Zahlen wirken nicht zufällig

f(z)= 5z mod 13
→ (1,5,12,8,1)
→ (2,10,11,3,2)
→ (4,7,9,6,4)
→ (5^4 = 625 → 625 mod 13 = 1 → 5 ist keine Primitwurzel)
→ mehrere nicht volle Perioden → schlecht !

Beurteilung von Zufallszahlengeneratoren

Folgende Tests zur Beurteilung:
(T1) Ist f(z)=a*z mod m eine Funktion die eine volle Periode erzeugt (a muss eine Primitivwurzel sein)?
(T2) Ist die von f(z) erzeugte Zahlenfolge (z_1,z_2,z_3,…) zufällig?
(T3) Ist f effizient und korrekt implementiert? (32-Bit Rechner)

Exkurs: Primitivwurzel
a ist eine Primitivwurzel, wenn: $\mathbf{a^n \bmod m \neq 1}$ für alle n € 1,….m-2

Vorschlag von Lehmer:

m = 2^{31}-1 = 2.147.483.647 ist eine Primzahl
a = 7^5 = 16.807 ist eine Primitvwurzel bzgl. m

Beispiel: $z_0 = 1$ → $z_{10.000}$ = 1.043.618.065 (Zum testen ob T3 erfüllt ist für Lehmers Vorschlag)

Der komplette Algorithmus als Java Code:

```
class randomNumber {
  final static int a = 16807;       // 7^5
  final static int m = 2147483647; // 2^31 – 1

  private static int zn = 1;

  public static void setSeed (int seed) {
    zn = seed;
  }

  public static double rand () {
    zn = (a * zn) % m;
    return (double)zn / m;
  }
}
```

Dieser Algorithmus ist leider fehlerhaft, da a*zn > als der Integerbereich ist, somit gibt es einen Überlauf:

$Z_0 = 16807$ $z_1 = 282475249$
$Z_2 = 1622647863$ $z_3 = -1199696159$ (ÜBERLAUF!!!)

Problem ist, dass auf einem 32Bit Rechner nur Zahlen bis 2^{32} gespeichert werden können (Anmerkung: Ein Longint wird aus zwei Integern zusammengesetzt!)

Lösung des Problem mit Hilfe eines Vorschlags von Schrage

Vorschlag von Schrage (1979)

Korrekte, überlauffreie Ausführung der Operation auf ≥ 32 Bit Rechnern

Idee: Nur Multiplikationen ausführen die im Bereich $[-2^{31}, 2^{31}-1]$ liegen.

Dies wäre noch einfach falls m=a*q ist.

$Z_{n+1} = a*z_n \bmod a*q = a*(z_n \bmod q) < m$;
(Stellt sicher, dass es innerhalb des Terms keinen Überlauf gibt)

Falls m sich nicht Restfrei durch a teilen lässt, schreiben wir:

$m = a*q + r$ mit $r < q$

Durch etliche mathematische Umformungen und Substituieren erhält man nun:

$z_{n+1} = \gamma(z_n) + m*\delta(z_n)$

wobei
$\gamma(z_n) = a*(z_n \bmod q) – r* \lfloor z_n/q \rfloor$
$\delta(z_n) = \lfloor z_n/q \rfloor \lfloor a*z_n/m \rfloor$

Bedingungen:

(1) $\delta(z_n) \in \{0,1]$
(2) $a*(z_n \bmod q)$, $r* \lfloor z_n/q \rfloor \in \{0,\ldots,m-1\}$ (Sind die zwei teilterme von Gamma)
(3) $|\gamma(z_n)| \leq m-1$

Diese Bedingungen würden nun bewiesen werden

Beobachtung:
$\delta(z_n) = 0$, gdw. $\gamma(z_n) \geq 0$
$\delta(z_n) = 1$, gdw. $\gamma(z_n) < 0$

deshalb nehmen wir für $\gamma(z_n) \geq 0$
$z_{n+1} = \gamma(z_n)$
sonst $z_{n+1} = \gamma(z_n) + m$

Der modifizierte Lehmer Algorithmus sieht dann folgendermaßen aus:

Modifizierte Lehmer Algorithmus nach Schrage

```
class randomNumber2 {
  final static int a = 16807;          // 7^5
  final static int m = 2147483647;     // 2^31 - 1
  final static int q = 127773;         // m / a
  final static int r = 2836;           // m % a

  private static int zn = 1;

  public static double rand () {
    int gamma = a * (zn % q) - r * (zn / q);

    if (gamma >= 0)
      zn = gamma;
    else zn = gamma + m;

    return (double)zn / m;
  }
}
```

Wir erhalten nun mit seed=1 folgende korrekte Pseudo-Zufallszahlen:

$z_0 = 16807$,
$z_1 = 282475249$,
$z_2 = 1622650073$,
$z_3 = 984943658$,
$z_4 = 1144108930$,
$z_5 = 470211272$
$z_6 = \ldots..$

Gemischt kongruente Generatoren

z.B.

$z_{n+1} = (a*z_n + b) \bmod m$; Häufige Wahl $m=2^k$

Probleme:

- z_n wechselt zwischen gerade und ungeraden Werten
- allgemein hat der Zyklus der letzten l Bits Periode 2^l

So wurde diese Algorthmus in Java implementiert:

```
synchronized protected int next (int bits) {
  seed = (seed * 25214903917L + 11L) & ((1L << 48)
          - 1);
  return (int)(seed >>> (48 - bits));
}
```

Satz:

Ist $m = 2^k$, so erzeugt

$f(z_n) = z_{n+1} = (a*z_n + b) \bmod m$

alle ganzen Zahlen in $\{0,\ldots,2^k-1\}$, falls:

1. a mod 4 = 1
2. b ungerade ist

Das gerade behandelte Thema können wir nun direkt anwenden im folgenden Teil.

Vorlesung 4 – Randomisierte Algorithmen

Es gibt zwei große Klassen von randomisierten Algorithmen:

1. **Las Vegas Algorithmen:** wahrscheinlich schnell, immer korrekt
   - ⇨ Wir haben nur einen Erwartungswert für die Laufzeit
   - ⇨ Beispiel: Randomisiertes Quicksort

2. **Monte Carlo Algorithmen:** immer schnell, Wahrscheinlich korrekt
   - ⇨ Der Algorithmus hat eine Irrtumswahrscheinlichkeit für die richtige Ausgabe (→ erhöhen durch mehrfaches Ausführen)
   - ⇨ Beispiel: Randomisierter Primzahltest

Randomisiertes Quicksort

Normaler Quicksort siehe Anfang .

Java-Code:

```
class quickSort {
  /* sortiert das Array a zwischen den Grenzen l und
     r */
  static void sort (sequence a[], int l, int r){
    if (r > l){
      int i = random(l,r);
      a.swap(i,r);
      i = divide(a,l,r);
        /* Teile a bzgl. v = a[r] auf:
           a[l],...,a[i-1] ≤ v ≤ a[i+1],...,a[r] */
      sort (a, l, i-1);
      sort (a, i+1, r);
    } // if (r > l)
  }
}
```

Der randomisierte Quicksort-Algorithmus unterscheided sich vom normalen nur durch die Wahl des Pivotelementes. Das rechteste Element das normalerweise als Pivotelement gewählt wird, wird randomisiert, mit einem beliebigen anderen Element der Liste geswapt.

Erwartungswert für die Laufzeit:

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{k=1}^{n}(T(k-1)+T(n-k))+\Theta(n) \\ &= O(n\log n) \end{aligned}$$

Randomisierter Primzahltest

Sehen wir uns zunächst die deterministische Variante an, die zwar immer korrekte Ergebnisse liefert, aber sehr langsam ist und für große Primzahlen nicht mehr praktikabel.

Der Algorithmus teilt einfach jede zu testende Zahl n durch alle ungeraden Zahlen die kleiner als SQRT(n) sind.

Algorithmus *Deterministischer Primzahltest*
Input: eine ganze Zahl n
Output: Antwort auf die Frage: Ist n prim?
if $n = 2$ **then return** true
if n gerade **then return** false
for $i = 1$ **to** $\sqrt{n}/2$ **do**
  if $2i + 1$ teilt n
    then return false
return true

Die Laufzeit für ein d-stelliges n ist $2^{d/2}$ bzw. lineare Laufzeit

Wir suchen nun ein randomisiertes Verfahren, das:

- In polynomieller Zeit ausführbar ist,
- Falls es als Ausgabe „n nicht prim" liefert, dann soll n auch nicht prim sein
- Falls es als Ausgabe „n ist prim" liefert, dann soll n zu einer bestimmten Wahrscheinlichkeit $p > 0$ auch prim sein.
  - ⇨ Nach k Iterationen ist n nicht prim zu einer Irrtumswahrscheinlichkeit von $(1-p)^k$

Ein wesentlicher theoretischer Bestandteil des randomisierten Primzahltestes ist der „kleine Satze von Fermat":

Kleiner Fermat

Ist p prim und $0 < a < p$, dann ist $a^{p-1} \equiv 1 \pmod p$

Beispiel: p=67, $2^{66} \bmod 67 = 1$

Integrieren wir nun den kleinen Satz von Fermat in einen randomisierten Primzahltest, so erhalten wir folgende 1. Version eines Algorithmus:

Algorithmus *Randomisierter Primzahltest 1*
1 Wähle a im Bereich $[1, n-1]$ zufällig
2 Berechne $a^{n-1} \bmod n$
3 **if** $a^{n-1} \bmod n \neq 1$
4 **then** n ist definitiv *nicht prim*
5 **else** n ist möglicherweise *prim*

leider haben wir noch eine relativ große Irrtumswahrscheinlichkeit.

Ein weiteres Problem, das wir haben sind die so genannten Carmichael Zahlen:

Carmichael Zahlen

Carmichael Zahlen sind Zahlen n für die gilt:

- Der Formel von Fermat ist zwar erfüllt::
  - $a^{n-1} \bmod n = 1$
  - aber n ist die Zahl die wir testen ! und ist nicht zwingend prim

denn

- a ist Teilerfremd mit n, also:
  - ggT(a, n) = 1

solche Zahlen die vermeintliche Primzahlen sind, nennt man Carmichael Zahlen.
Die kleinste Carmichael Zahl ist 561 = 3 * 11 * 17. Wählt man nun das a also so aus, dass es gerade nicht Primfaktor oder Vielfaches von diesem von n ist, dann erhalten wir ein falsches Ergebnis. Die Wahrscheinlichkeit ein falsches a zu wählen ist sehr hoch.

Beispiel: n = 3.828.001 = 101 * 151 * 251

Außer für vielfache von 101, 151, 251 die leider nur ca. 1/50 stel aller möglichen Zahlen sind, ist

$$a^{n-1} \bmod n = 1$$

wir wissen nun, das ein randomisierter Primzahltest alleine mit dem Satz von Fermat nicht gut genug ist, da die Irrtumswahrscheinlichkeit nahezu 1 ist !

wir müssen nun die erste Bedingung, also den Satz von Fermat, mit einer zweiten Bedingung kombinieren, hierzu testen wir nun zusätzlich auf „nichttriviale Quadratwurzeln“.

Satz über nichttriviale Quadratwurzeln

Ist p prim und 0 < a < p, dann hat die Gleichung $\mathbf{a^2 \equiv 1 \pmod p}$ genau die Lösungen: a = 1 und a = p-1

Falls $a^2 \bmod n = 1$ gilt, obwohl $a \neq 1$ und $a \neq p-1$, dann heisst a „nichttriviale Quadratwurzel mod n“

Beispiel n = 35

$1^2 \equiv 1 \pmod{35}$
$34^2 \equiv 1 \pmod{35}$
aber
$6^2 \equiv 1 \pmod{35}$ → 6 ist „**Nichttriviale Quadratwurzel** mod 35“

wir müssen nun herausfinden ob es überhaupt eine „Nichttriviale Quadartwurzel mod n gibt. Die machen wir während wir a^{n-1} berechnen:

Schnelle Exponentiation (Berechnung von a^n)

$0 < a < n$

Fall 1: [n ist gerade]

$$a^n = a^{n/2} \cdot a^{n/2}$$

Fall 2: [n ist ungerade]

$$a^n = a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a$$

Beispiel:

$$a^{62} = \left(a^{31}\right)^2$$
$$a^{31} = \left(a^{15}\right)^2 \cdot a$$
$$a^{15} = \left(a^7\right)^2 \cdot a$$
$$a^7 = \left(a^3\right)^2 \cdot a$$
$$a^3 = (a)^2 \cdot a$$

Laufzeit: $O(\log^2 a^n \log n)$

Im Java-Code sieht die Kombination dann so aus:

```
private static boolean isProbablyPrime;

private static int power (int a, int p, int n) {
  /* berechnet a^p mod n und prüft, ob bei der
     Berechnung ein x auftritt mit x^2 mod n = 1 und
     x ≠ 1, n - 1 */

  if (p == 0) return 1;

  int x = power(a, p / 2, n);
  int result = (x * x) % n;

  /* prüfe, of x^2 mod n = 1 und x ≠ 1, n - 1 */
  if (result == 1 && x ≠ 1 && x ≠ n-1)
    isProbablyPrime = false;

  if (p % 2 == 1) result = (a * result) % n;

  return result;
}
```

Wir haben nun eine Laufzeit von: $O(\log^2 n \log p)$ für diesen Teil

Der gesamt randomisierte Primzahltest-Algorithmus mit beiden Bedingungen, d.h. Fermat und nichttriviale Quadratwurzeln sieht in Java nun so aus:

```
public static boolean primeTest (int n) {
  /* führt den randomisierten Primzahltest für ein
     zufälliges a aus */

  int a = random(2,n-1);

  isProbablyPrime = true;

  int result = power(a,n-1,n);

  if (result ≠ 1 || !isProbablyPrime)
    return false;
  else return true;
}
```

(auf unsere Powerfunktion von oben wird im Hauptprogramm zugegriffen)

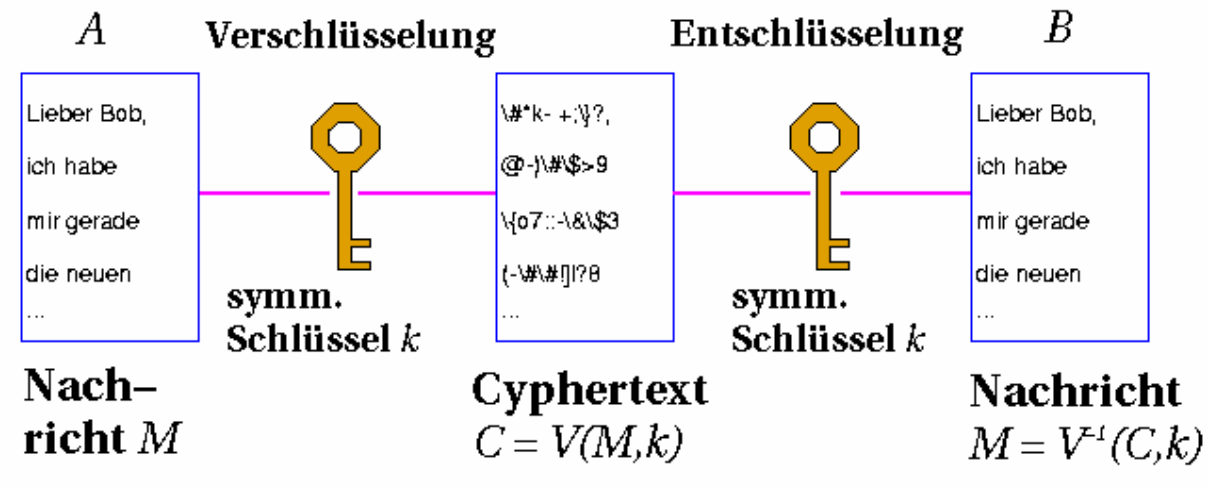
Folgender wichtiger Satz beschreibt die Irrtumswahrscheinlichkeit unseres Primzahltests:

Satz: Ist n nicht prim, so gibt es höchstens (n-9)/4 Zahlen 0 < a < n, für die der Algorithmus primeTest versagt.

⇨ Also ca. ¼, führt man den Algorithmus aber z.B. 50 mal aus so reduziert sich die Irrtumswahrscheinlichkeit auf $(¼)^{50}$,also nahezu 0, oder besser gesagt niedriger als die Wahrscheinlichkeit für einen Hardwarefehler beim Rechnen.

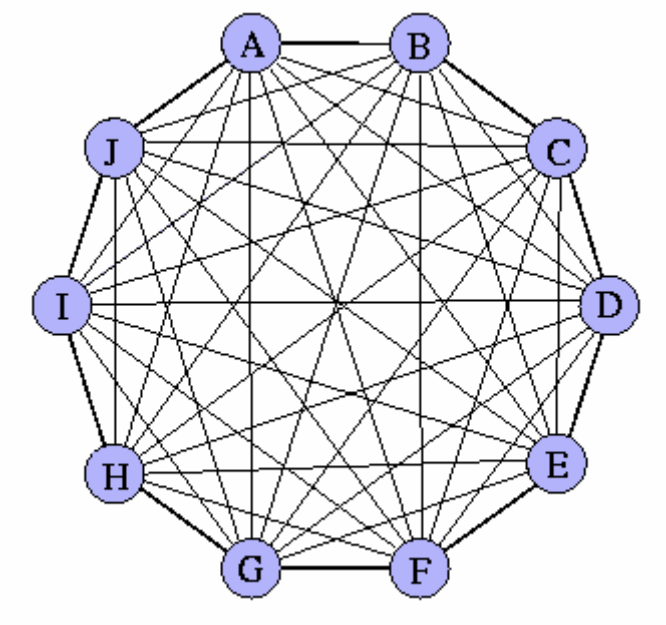
Vorlesung 5 – Kryptographie: RSA

Traditonelle Verschlüsselung mit symmetrischem Schlüssel:



Probleme:

- Schlüssel müssen irgendwann ausgetauscht werden!
- Für Nachrichten zwischen n Parteien sind n(n-1)/2 Schlüssel erforderlich



Jede Kante repräsentiert einen Schlüssel

Vorteile:

- Ver- und Entschlüsselung ohne spürbaren Geschwindigkeitsverlust möglich

Aufgaben von Sicherheitsdiensten:

Gewährleistung von

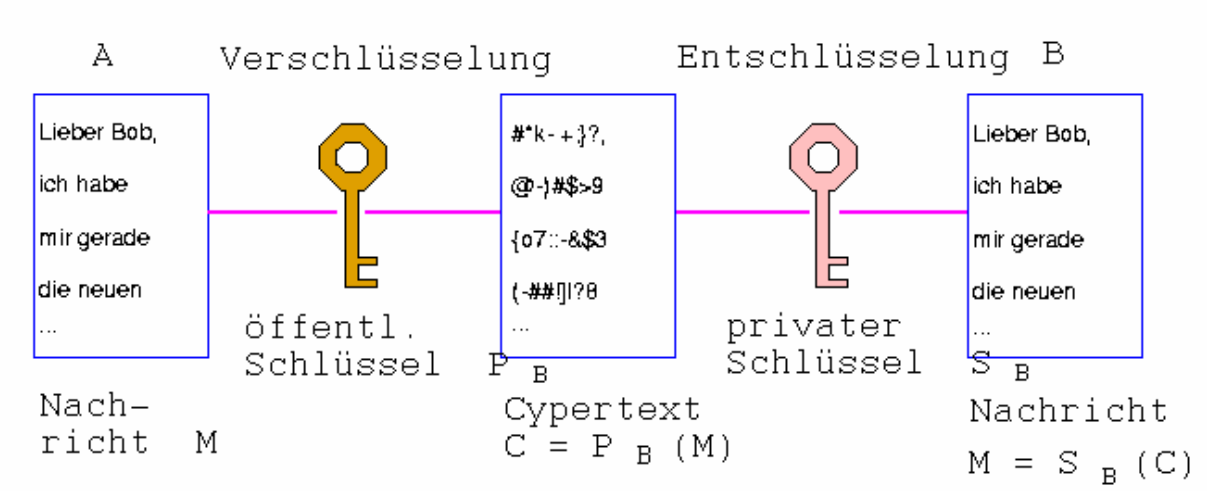
- Vertraulichkeit der Übertragung (Vor fremdem Einblick zu schützen)
- Integrität der Daten (Dass die Daten so ankommen wie sie gesendet wurden)
- Authentizität des Senders (Ist der Sender der für den er sich ausgibt)
- Verbindlichkeit der Übertragung (Durch digitale Signatur wird eine Nachricht verbindlich gemacht)

Public Key Verschlüsselungssysteme

Idee erstmals vorgestellt von Diffie Hellmann (1976), aber ohne genaue Angaben zu machen, wie das funktionieren soll.

Idee: Jeder Teilnehmer A besitzt zwei Schlüssel

1. einen *öffentlichen Schlüssel* P_A (Public Key), der jedem anderen Teilnehmer zugänglich gemacht wird.
2. einen *geheimen Schlüssel* S_A (Secret/Private Key), der nur A bekannt ist



D = Menge der zulässigen Nachrichten , z.B. Menge aller Bitstrings endl. Länge

$$P_A(), S_A() : D \xrightarrow{1-1} D$$

Es gelten drei Bedingungen

1. $P_A()$, $S_A()$ muss effizient berechenbar sein
2. $P_A(S_A(M)) = M$ und $S_A(P_A(M)) = M$
3. $S_A()$ nicht aus 2ter Bedingung Berechenbar (in realisierbarer Zeit)

A verschickt eine Nachricht M an B:

1. A verschafft sich B's öffentlichen Schlüssel P_B, von einem öffentlichen Verzeichnis oder direkt von B
2. A berechnet die chiffrierte Nachricht $C = P_B(M)$ und sendet C an B
3. Nachdem B die Nachricht C empfangen hat, dechiffiert B die Nachricht mit seinem privaten Schlüssel S_B: $M = S_B(C)$

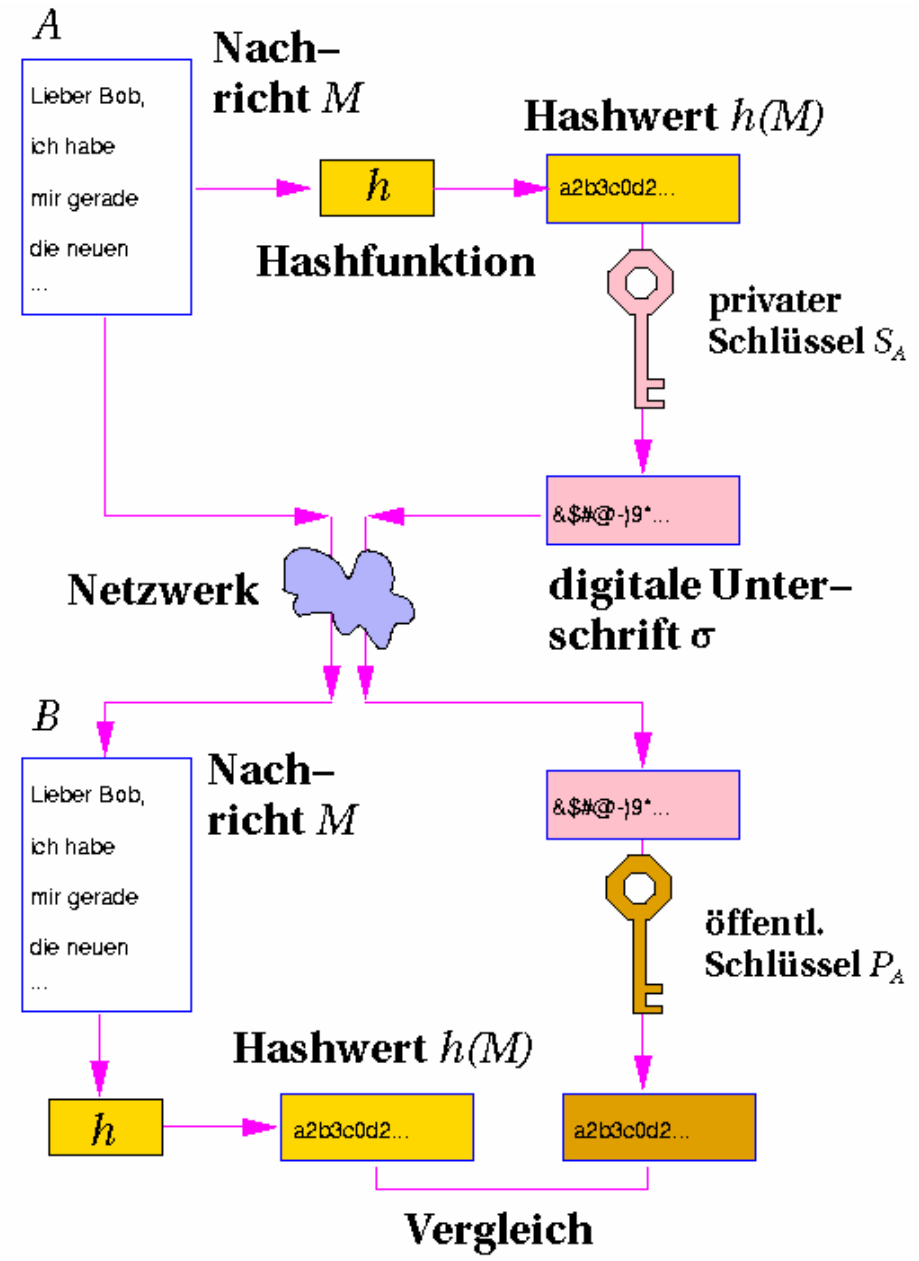
Erstellen einer digitalen Signatur - Prinzip

A schickt eine digital unterzeichnete Nachricht M' an B

1. A berechnet die digitale Unterschrift σ für M' mit Hilfe des geheimen Schlüssels:
   $$\sigma = S_A(M')$$
2. A schickt das Paar (M', σ) an B
3. Nach erhalt von (M', σ) überprüft B die digitale Signatur $P_A(\sigma) = M'$

σ kann von jedem mit P_A überprüft werden

Erstellen einer digitalen Signatur



RSA Verschlüsselungssysteme

R.**R**ivest, A. **S**chamir, L. **A**dleman

Erstellen der öffentlichen und geheimen Schlüssel:

1. Man wählt zufällig zwei große Primzahlen p und q (sehr groß: $\log_p \sim 200$)
2. sei n = pq
3. sei e eine kleine natürliche Zahl, die Teilerfremd zu (p-1)(q-1) ist
4. berechne $d = e^{-1}$

   $$d \cdot e \equiv 1 \pmod{(p-1)(q-1)}$$
5. veröffentliche P = (e, n) als öffentlichen Schlüssel
6. behalte S = (d, n) als geheimen Schlüssel

$C = P(M) = M^e \bmod n$ $S(C) = C^d \bmod n$

Ein kurzer Korrektheitsbeweis schliesst das nun ab
Beispiel:

Man wähle zwei Primzahlen p und q:
p = 17, **q = 23**

- ⇨ n = **pq = 391**
- ⇨ **(p-1)(q-1)** = 16*22 = 2^5*11 = **352**

Wähle ein e, welches Teilerfremd zu (p-1)(q-1) ist:
z.B.
e = 3 die Mutliplikative Inverse zu e ist **d = 235**

(Für die Multiplikative Inverse muss gelten 3*235 = 705 ≡ 1 (mod 352))

Die Schlüssel sind nun:

P = (3,391)
S = (235,391)

Diskurs – Multiplikative Inverse:

Betrachten wir nun die Berechnung der Multiplikativen Inversen genauer:

Satz: (ggT Rekursionssatz)
Für alle ganze Zahlen a und b mit b > 0 gilt:
ggT(a,b) = ggT(b, a mod b)

Algorithmus Euklid

```
Input:   Zwei ganze Zahlen a und b mit b ≥ 0
Output:  ggT(a, b)
1   if b = 0
2     then return(a)
3     else return(Euklid(b, a mod b))
```

Algorithmus erweiterter Euklid

```
Input:   Zwei ganze Zahlen a und b mit b ≥ 0
Output:  ggT(a, b) und zwei ganze Zahlen x und y
           mit xa + yb = ggT(a, b)
1   if b = 0
2     then return(a, 1, 0)
3   (d, x', y') := Erw-Euklid(b, a mod b)
4   x := y'; y := x' − ⌊a/b⌋ y'
5   return(d, x, y)
```

Seien
a = e, b = (p-1)(q-1)

Es wird nun mittels des Algorithmus berechnet:

xa + yb = ggT(a,b) = ggT(e, (p-1)(q-1)) = 1, da wir e Teilerfremd zu (p-1)(q-1) gewählt haben

⇨ **x*a ≡ 1 (mod (p-1)(q-1))**

Somit ist x unsere gesuchte Multiplikative Inverse

Nun käme ein Korrektheitsbeweis zum erweiterten Euklid. Algorithmus

Beispiel für Mulitplikative Inverse

$x = y'$

$y = x' - \lfloor a/b \rfloor\, y'$

$a = 99, \qquad b = 78$

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	3 - 1 · ·
78	21	3	3	3	· 2 - 3 · 3 = -11
21	15	1	3	-2	1 - 1 · (-2) = 3
15	6	2	3	1	0 - 2 · 1 = -2
6	3	2	3	0	1 - 2 · 0 = 1
3	0	–	3	1	0

Zuerst werden die a's und b's alle berechnet (von oben nach unten), danach die x's und y's (von unten nach oben). ACHTUNG: d ist hier nicht die Multiplikative Inverse wie zuvor !!!

Anmerkung: Das Beispiel hat nicht direkt was mit dem RSA zu tun, es werden einfach nur der ggT von a und b berechnet und danach die x und y dazu berechnet so dass gilt xa + yb = ggt(a,b)

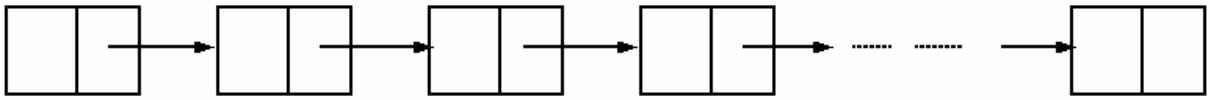
Wählt man ein korrektes (a=) e und (b=)(p-1)(q-1), so ist der ggT(a,b) = d = 1

Dann **erhält man x als Multiplikative Inverse**

Vorlesung 6 – Randomisierte Datenstrukturen

Mögliche Implementationen von Wörterbüchern:

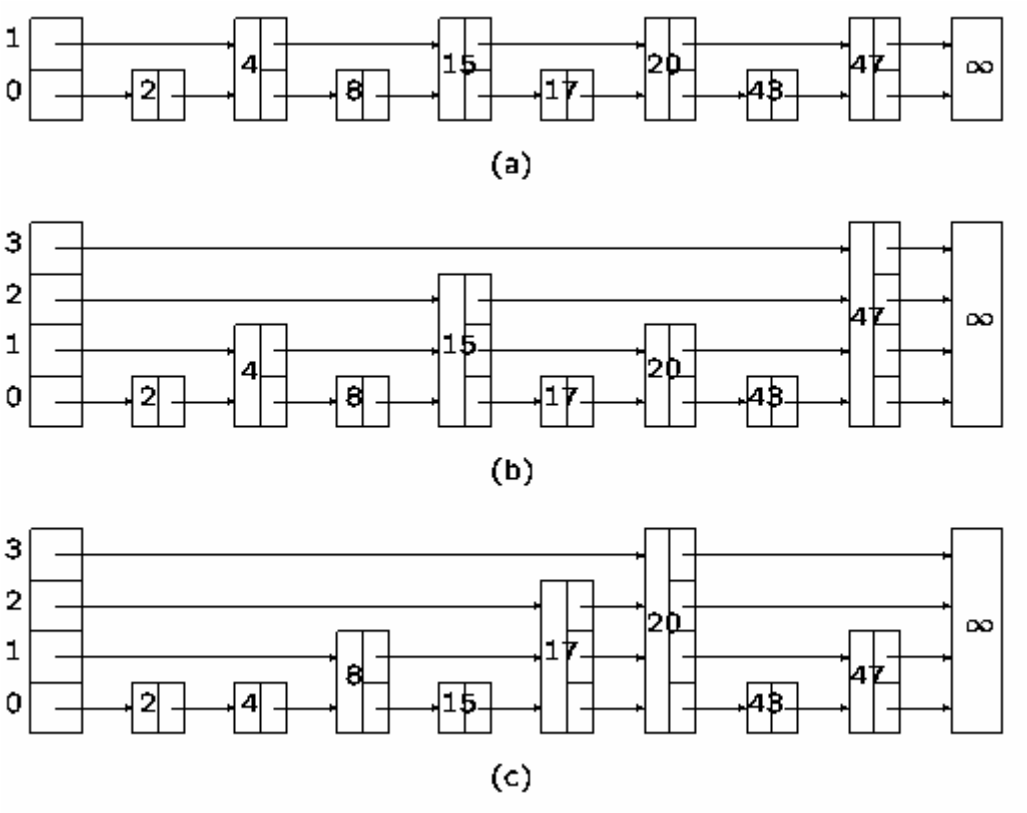
- Verkettete Lineare Listen
  - ⇨ Suchen, Einfügen, Entfernen in O(n) Schritten



- (Balancierte) Suchbäume
  - ⇨ Suchen, Einfügen, Entfernen in O(log n) Schritten

Perfekte Skiplisten

Idee: Hinzufügen von zusätzlichen Zeigern in die Liste um die Suche in der sortierten Liste zu beschleunigen



(a) Beispiel für eine Skipliste mit Höhe 1 (2 Ebenen). In der zweiten Ebene wird nur jedes zweite Element angelinkt, d.h. die Suchgeschwindigkeit verdoppelt sich !
(b) Perfekte Skipliste mit Höhe O(log n)
(c) Randomisierte Skipliste: Die Niveaus werden nicht strickt nach Regel gesetzt, sondern „ausgewürfelt“

Die Head und Tail Elemente werden auf einen sehr niedrigen Wert (z.B. -∞) bzw. einen sehr hohen Wert (z.B. +∞) gesetzt.

Def. Eine Perfekte Skipliste ist nun eine sortierte verkettet gespeicherte Liste mit Kopfelement ohne Schlüssel und Endelement mit Schlüssel ∞ und:

- Jeder 2^0-te Knoten hat Zeiger auf nächsten Knoten auf Niveau 0
- Jeder 2^1-te Knoten hat Zeiger auf 2^{1-}entfernten Knoten auf Niveau 1
- Jeder 2^2-te Knoten hat Zeiger auf 2^{2-}entfernten Knoten auf Niveau 2
- ….
- Jeder 2^k-te Knoten hat Zeiger auf 2^{k-}entfernten Knoten auf Niveau k

$k = \lceil \log n \rceil - 1$

Die Anzahl der Zeiger in einer perfekten Skipliste halbiert sich in jeder Ebene, die Summe daraus entspricht der geometrischen Reihe, die gegen 2 konvergiert. Wir erhalten also 2n Zeiger.

Die Listenhöhe ist log n

⇨ Die Anzahl der Zeiger pro Listenelement ist höchstens log n, nämlich dann wenn ein Element an einer stelle mit Höhe log n steht

Datenstruktur zur Repräsentation eines Knotens aus der Liste (Java Code)

```
class skipListNode {
  /* Knotenklasse für Skip-Listen */

  protected int key;
  protected Object information;

  protected skipListNode [] next;


  /* Konstruktor */
  skipListNode (int key, int height) {
     this.key = key;
     this.next = new skipListNode [height+1];
  }
}
```

Datenstruktur zum Erstellen einer (noch) leeren Skiplist (Java Code)

```
class skipList {
  /* Implementiert eine Skip-Liste */

  public int maxHeight = 0;     // Max. Höhe der Liste

  private skipListNode head;    // Kopf der Liste
  private skipListNode tail;    // Ende der Liste

  private int height;           // Akt. Höhe der Liste

  private skipListNode[] update;  // Hilfsarray


  /* Konstruktor */
  skipList (int maxHeight) {
     this.maxHeight = maxHeight;
     height = 0;

     head = new skipListNode (Integer.MIN_VALUE,
                              maxHeight);
     tail = new skipListNode (Integer.MAX_VALUE,
                              maxHeight);
     for (int i = 0; i <= maxHeight; i++)
       head.next[i] = tail;

     update = new skipListNode[maxHeight+1];
  }
```

Algorithmus zum Suchen in Skip-Listen (Java Code)

```
public skipListNode search (int key) {
  /* liefert den Knoten p der Liste mit p.key = key,
     falls es ihn gibt, und null sonst */

  skipListNode p = head;

  for (int i = height; i >= 0; i--)
    /* folge den Niveau-i Zeigern */
    while (p.next[i].key < key) p = p.next[i];

  /* p.key < x <= p.next[0].key */

  p = p.next[0];
  if (p.key == key && p != tail) return p;
  else return null;
}
```

Das Suchen in der Liste geht in O(log n) Zeit, aber

Einfügen und Entfernen bei perfekten Skiplisten benötigt Ω(n) Zeit, weil die gesamte Struktur nach jedem Einfüge- bzw. Entfernenprozess neu generiert werden muss, damit diese Liste wieder perfekt ist !

⇨ **Randomisierung**

Randomisierte Skiplisten

Besonderheiten:

- Aufgabe der starren Verteilung der Höhen der Listenelemten (Keine starre Struktur mehr)
- Aber der Anteil der Elemente mit gleicher Höhe wird beibehalten (Randomisierung durch Münze werfen - später)
- Höhen werden gleichmäßig und zufällig über die Liste verteilt

Einfügen in „Randomisierte Skiplisten“

Einfügen von k:

1. Suchen der Einfügeposition von k. Die Suche endet bei dem größten Knoten q der gerade noch kleiner ist als k.
2. Füge neuen Knoten p mit Schlüssel k und *zufällig gewählter Höhe* nach Knoten q ein
3. Wähle die Höhe von p so, dass für alle i gilt:

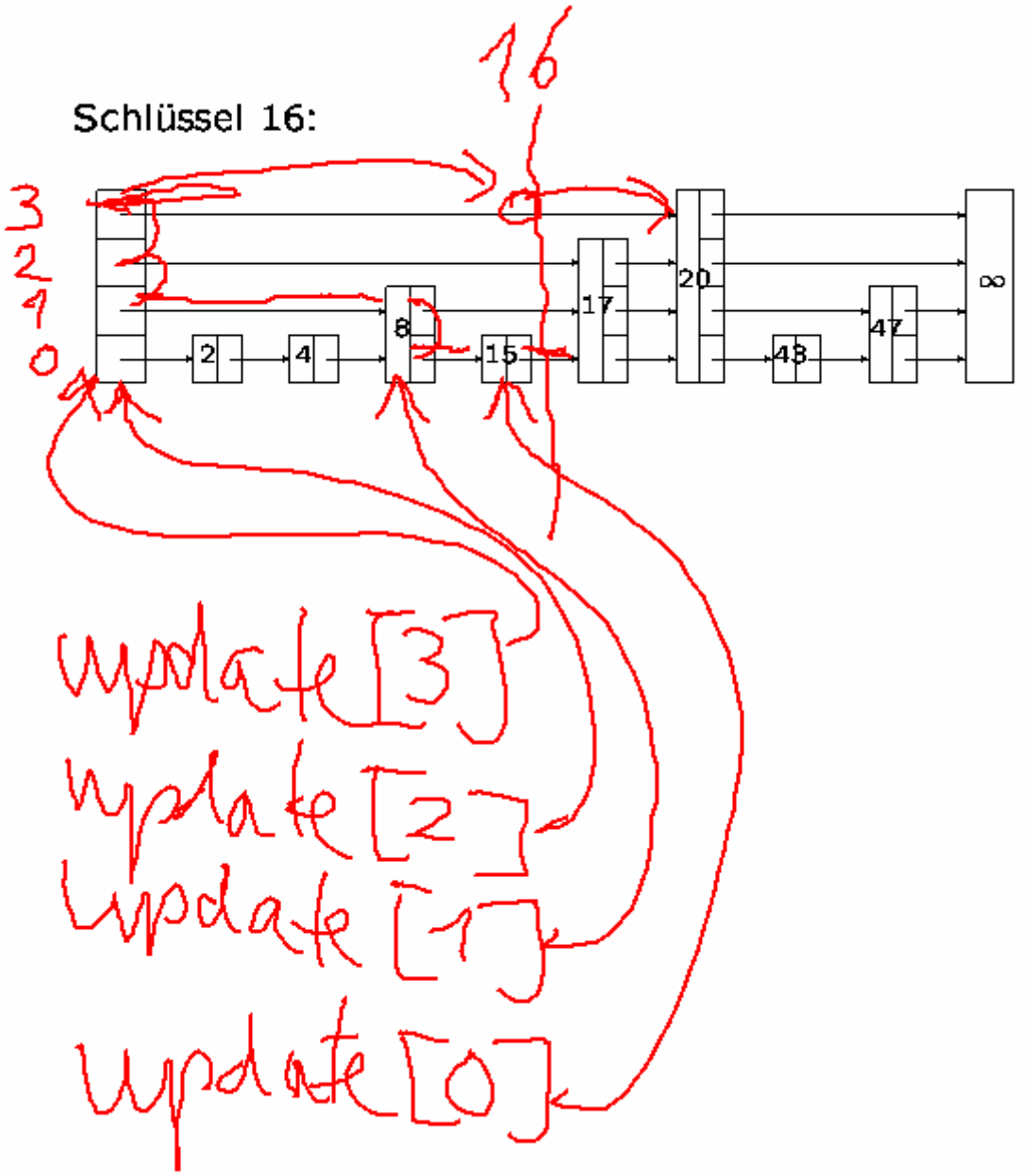
$$P(\text{Höhe von } p = i) = \frac{1}{2^{i+1}}$$

4. Sammle alle Zeiger, die über die Einfügestelle hinweg weisen in einem Array und verkette alle Nachbarn mit Niveau ≤ i mit p. (Anmerkung: Der Satz war auf der Folie so kompliziert und wohl auch falsch geschriebn, dass man ihn nicht kapiert, für die Richtigkeit dieses Satze übernehmen wir keine Haftung)

Der zugehörige Algorithmus zum Einfügen (in Java Code)

```
public void insert (int key) {
  /* fügt den Schlüssel key in die Skip-Liste ein */

  skipListNode p = head;
  for (int i = height; i >= 0; i--) {
    while (p.next[i].key < key) p = p.next[i];
    update[i] = p;
  }

  p = p.next[0];
  if (p.key == key) return; // Schüssel vorhanden

  int newheight = randheight ();
  if (newheight > height) {
    /* Höhe der Skip-Liste anpassen */
    for (int i = height + 1; i <= newheight; i++)
      update[i] = head;
    height = newheight;
  }

  p = new skipListNode (key, newheight);

  for (int i = 0; i <= newheight; i++) {
    /* füge p in Niveau i nach update[i] ein */
    p.next[i] = update[i].next[i];
    update[i].next[i] = p;
  }
}
```

Beispiel für das Einfügen in eine randomisierte Skipliste:
Mit Angabe des Updatearrays (siehe unverständlicher Satz oben)



Einfügen geht nun in gleicher Zeit wie Suchen, also O(log n), weil die Zeigerupdates in konstanter Zeit gehen, wir müssen also im Prinzip nur noch suchen.

Das Erzeugen der randomisierten Höhen ist nun nur noch ein kleines Problem. Man kann sich dabei einfach vorstellen eine Münze zu werfen. Jedesmal wenn ein Kopf kommt, dann wird die Höhe um eins erhöht, ansonsten wird abgebrochen, also die Höhe für das Element beibehalten. Für das Niveau 0 haben wir dann also Wahrscheinlichkeit ½ für Niveau 1 haben wir ¼ und für Niveau 2 haben wir 1/8 usw…

Der zugehörige

Algorithmus zum randomisierten Erzeugen von Höhen

sieht dann so aus:

```
private int randheight () {
  /* liefert eine zufällige Höhe zwischen 0 und
     maxHeight */

  int height = 0;

  while (rand () % 2 == 1 && height < maxHeight)
    height++;

  return height;
}
```

$$P(\text{randheight} = i) = \frac{1}{2^{i+1}}$$

Entfernen eines Schlüssel aus einer randomisierten Skipliste

1. Suche erfolgreich nach k
2. Entferne Knoten p mit p.key aus allen Niveau i Listen, mit $0 \leq i \leq$ Höhe von p, und adjustiere ggf. die Listenhöhe

Zugehöriger Algorithmus (in Java Code)

```
public void delete (int key) {
  /* enfernt den Schlüssel key aus der Skip-Liste */
  skipListNode p = head;
  for (int i = height; i >= 0; i--) {
    /* folge den Niveau-i Zeigern */
    while (p.next[i].key < key) p = p.next[i];
    update[i] = p;
  }

  p = p.next[0];
  if (p.key != key) return; /* Schüssel nicht
                               vorhanden */

  for (int i = 0; i < p.next.length; i++) {
    /* entferne p aus Niveau i */
    update[i].next[i] = update[i].next[i].next[i];
  }

  /* Passe die Höhe der Liste an */
  while (height >= 0 && head.next[height] == tail)
    height--;
}
```

Die rot markierte Zeile ist der Entfernungsschritt, d.h. das Umleiten der Zeiger

Verhalten von randomisierten Skiplisten

- Unabhängig von der Erzeugungshistorie (degeneriert nicht, es gibt keine schlechte Eingabefolge)
- Erwartete Anzahl von Zeigern in einer Liste der Länge n ist 2n
- Erwartete Höhe einer Liste mit n Elementen ist in O(log n) (genauer $\leq 2 \log n + 2$) (Langen Beweis hat Ottman ausgeführt (ca. 15 Minuten lang)
- Erwartete Suchkosten für die Suche nach einem Element in einer Liste mit n Elementen sind in O(log n) (ohne Beweis)
- Einfügen und Entfernen von Elementen in Liste mit n Elementen in erwarteter Zeit O(log n) möglich.

Randomisierte Suchbäume

Idee:
Nutze Randomisierung um das degenerieren natürlicher Suchbäume zu linearen Listen zu verhindern.
Ordne Schlüsseln zufällig gewählte Zeitmarken zu, die eine fiktive Einfügereihenfolge simulieren.

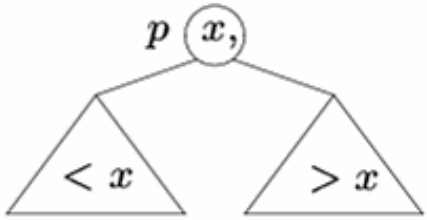
Was sind nun Treaps ?

Treap = Tree + Heap

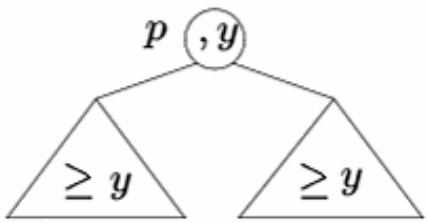
Ein Knoten x hat 2 statt einer Komponente:

- x.key (=x)
- x.priority (=y)

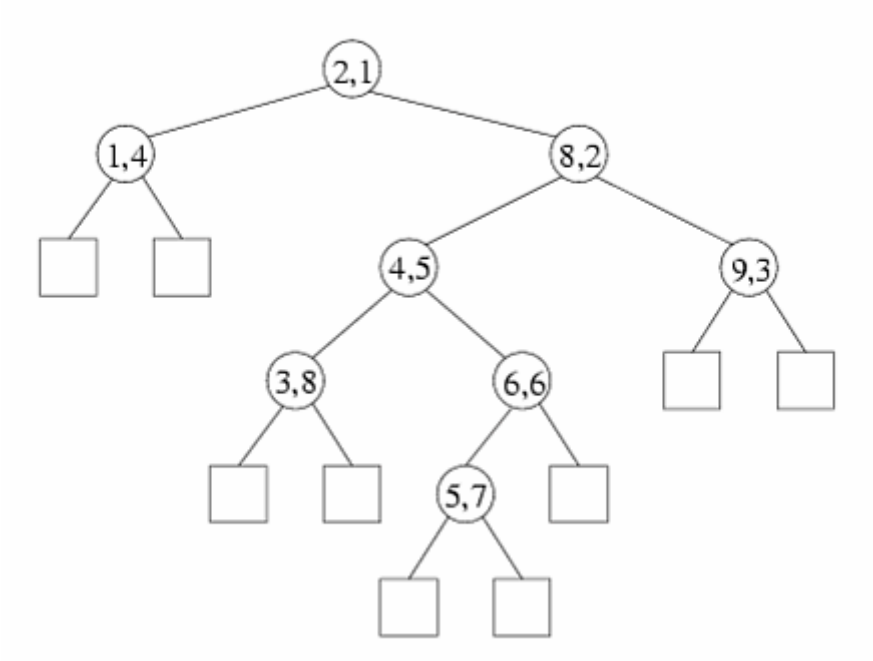
(a) Schüsselkomponenten: Suchbaumbedingung



(b) Prioritäten: Heapbedingung



Beispiel für einen Treap:



Für jede Menge

$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

mit paarweise verschiedenen Schlüsseln x_i und Prioritäten y_i, gibt es genau einen Treap der S speichert.

Das liegt daran, dass das Wurzelelement durch die Heapbedinung (kleinstes (bei minHeap)) festgelegt ist. Die weiteren Aufteilungen erfolgen nach Regel !

Ein Element einfügen in einen Treap

Füge x = (x.key,x.priority) in Treap T ein:

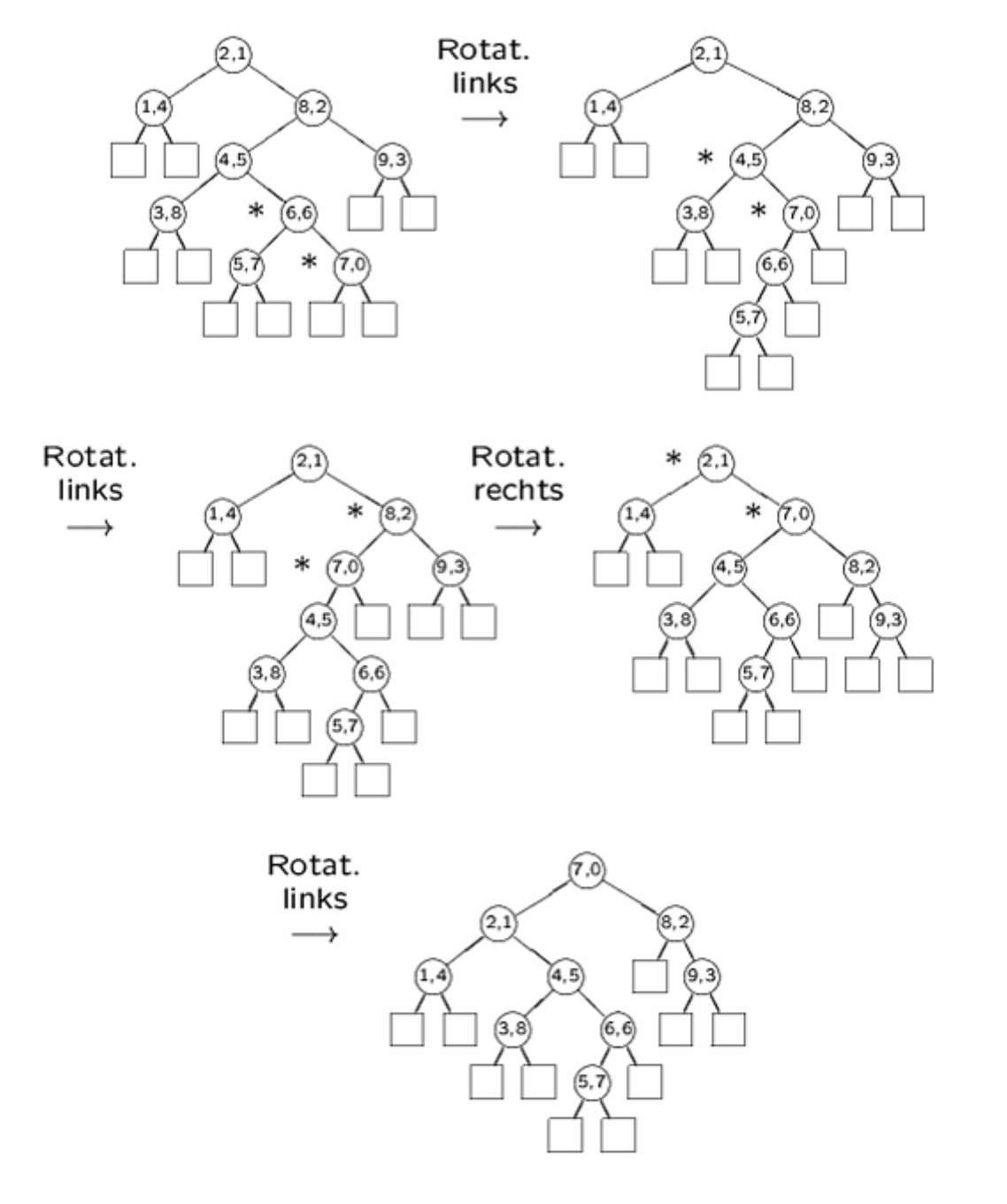
1. Einfügen in dem Blatt in dem die erfolglose Suche nach x.key endet (wie bei natürlichen Suchbäumen)
2. (x.key,x.priority) solange hinaufrotieren bis die Heapbedingung wieder erfüllt ist

Ein Element aus einem Treap entfernen

Umgekehrt wie einfügen

1. Den zu entfernenden Schlüssel bis auf Blattebene hinunterrotieren (nur Schlüssel mit kleinerer Priorität dürfen im Gegenzug hochrotiert werden
2. Knoten entfernen

Beispiel: Einfügen des Elementes (7,0) in den Treap



Die Sterne (*) geben jeweils an, an welchen stellen die Heapbedingung verletzt ist, diese Knoten müssen rotiert werden.

Die Frage ist nun, wie wählt man die Priorität geschickt

- ⇨ Randomisierung
- ⇨ Man redet nun von einem randomisierten Suchbaum

Randomisierte Suchbäume

⇨ Treaps mit zufälligen Prioritäten

Definition:
Ein randomisierter Suchbaum für eine Menge S von Schlüsseln ist ein Treap mit Knoten,

- deren Schlüssel genau die Schlüssel in S sind
- deren Prioritäten unabhängig und gleich verteilt aus [0,1) gewählt sind.

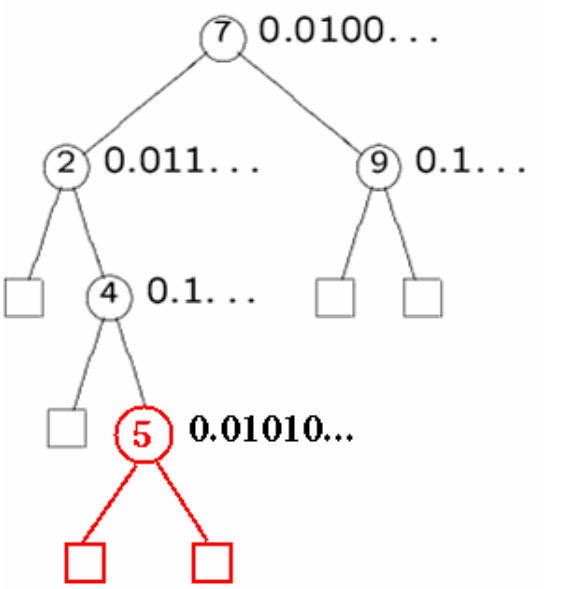
Die Prioritäten können als „Zeitmarken“ aufgefasst werden.

Die erwartetet Struktur eines randomisierten Suchbaumes für n Schlüssel (bei zufälliger Wahl der Prioritäten)
=
Erwartete Struktur eines natürlichen Suchbaumes für n Schlüssel (bei zufälliger Wahl der Permutation)

⇨ Erwartungswert für die Höhe des Baumes = $O(\log_2 n)$ bzw. $1.44 \log_2 n$
⇨ Suchkosten O(log n)
⇨ Kosten zum Entfernen /Einfügen eines Schlüssels x
  - Kosten(Zugriff(x)) + Rotationskosten
  - ≤O(log n) +2

Rotationskosten sind deshalb 2, weil man davon ausgeht, das ¾ der Knoten auf Blattebene sind und man im Schnitt dann nur 2 Rotationen benötigt.

Praktische Realisierung der Prioritäten
Erzeuge Dualdarstellung der den Schlüsseln zugewiesenen Prioritäten nach Bedarf Bitweise, Stück für Stück:



Einfügen der (5)

- Priorität 5: 0.01010…
- Für den Vergleich mit 4 genügt 0.0

- Für den Vergleich mit 2 genügt 0.010
- Für den Vergleich mit 7 genügt 0.0101

Vorlesung 7 – Amortisierte Analyse

- Wir betrachten eine Folge von n Operationen $o_1,\ldots,o_n$ auf eine Datenstruktur D
- Die Ausführungszeit einer einzelnen Operation o_i nennen wir T_i
- Die Gesamtlaufzeit $T = T_1,T_2,\ldots,T_n$
- Die Laufzeit einer einzelnen Operation kann in einem großen Bereich schwanken, z.B. zwischen 1,…,n . Aber nicht bei allen Operationen der Folge kann der schlechteste Fall eintreten

⇨ Amortisierte Analyse wird benötigt

Bei der Analyse von Algorithmen spricht man von folgenden Werten:

- Best Case
- Worst Case
- Average Case
- Amortisierter Worst Case
    - Was sind die durchschnittlichen Kosten einer schlechtest möglichen Folge von Operationen?

Was ist die Idee der Amotisierung ?

- Wir bezahlen für billige Operationen lieber etwas mehr um das Ersparte für teurere Operationen zu verwenden

Dabei unterscheidet man drei verschiedene Methoden:

- Aggregatmethode
- Bankkontomethode
- Potentialfunktion-Methode

Aggregatmethode: am Beispiel Dualzähler

Wir bestimmen die Bitwechselkosten eines Dualzählers

Opera-tion	Zähler-stand	Kosten
0	00000	
1	0000 1	
2	000 10	
3	0001 1	
4	00 100	
5	0010 1	
6	001 10	
7	0011 1	
8	0 1000	
9	0100 1	
10	010 10	
11	0101 1	
12	01 100	
13	0110 1	
	⋮	⋮
n		

Die Bitwechselkosten entsprechen der #endender_einsen + 1

Wir haben n Operationen, wobei es

- n/2 Operationen mit Bitwechselkosten „1“
- n/4Operationen mit Bitwechselkosten „2“
- n/8 Operationen mit Bitwechselkosten „3“
- …
- log n Operationen mit Bitwechselkosten n/($2^{\log n}$)

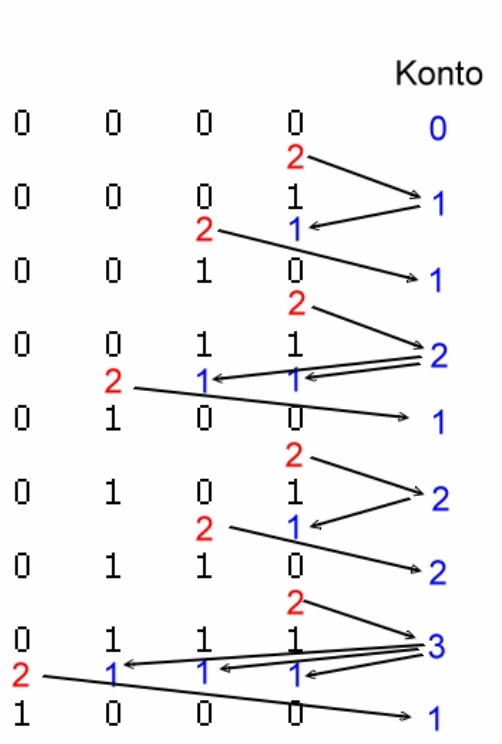
gibt

Summiert man dies auf, so ergibt sich als **Gesamtkosten 2n**

Bankkontomethode: am Beispiel Dualzähler

Idee: bezahle 2 KE (Kosteneinheiten) für das Verwandeln einer 0 in eine 1

⇨ Jede 1 hat dann eine KE auf dem Konto



Beobachtung: In jedem Schritt wird genau eine „0“ in eine „1“ umgewandelt

Die dritte Methode ist die

Potentialfunktion-Methode: am Beispiel Dualzähler

- **Potentialfunktion Φ**
- Datenstruktur $D \mapsto \Phi(D)$
- t_l sind die wirklichen Kosten den Operation l
- Φ_l ist das Potential nach Ausführen der l-ten Operation$^{(=\Phi(D_l))}$
- a_l sind die amortisierten Kosten der l-ten Operation

Definition:

$$a_l = t_l + \Phi_l - \Phi_{l-1}$$

Beispiel – Dualzähler:

D_i: Zustand nach der i-ten Operation (also die Folge von 0en und 1en nach o_i)

$\Phi_i = \Phi(D_i) =$ **# von Einsen in** D_i (z.b. Φ(101101)=4)

Es muss für Φ gelten (was es hier auch tut):
Φ(000….000) = 0
Φ(…..) ≥ 0

i-te Operation

- D_{i-1} ….0/1….0111…1
  - B_{i-1} ist die Gesamtanzahl der Einsen in D_{i-1}

- b_i ist die Anzahl der endenden Einsen in D_{i-1} (ROTE ZAHLEN)

- D_i .....0/1….1000…0
  - $B_i = B_{i-1} – b_i + 1$

t_i sind die wirklichen Bitwechselkosten der Operation i
a_i sind die amortisierten Bitwechselkosten der Operation i

somit ergibt sich für a_i
$a_i = (b_i+1) + (B_{i-1}- b_i + 1) - B_{i-1} = 2$

⇨ $\Sigma t_l \leq 2n$

Dynamische Tabellen

Problem :
Verwaltung einer Tabelle unter den Operationen „Einfügen“ und „Entfernen“, so dass:

- Die Tabellengröße der Anzahl der Elementen angepasst werden kann
- Immer ein konstanter Anteil der Tabelle mit Elementen belegt ist
- Die Kosten für n Einfüge oder Entfernen Operationen O(n) sind

Organisation der Tabelle z.B. durch: Hashtabelle, Heap, Stack etc.

Belegungsfaktor α_T Anteil der Tabellenplätze von T die belegt sind

Algorithmus für Einfügen

```
class dynamicTable {

  int [] table;

  int size;          // Größe der Tabelle
  int num;           // Anz. der Elemente

  dynamicTable () {       // Initialisierung der leeren Tabelle
    table = new int [1];
    size = 1;
    num = 0;
  }

  insert (int x) {
    if (num == size) {
      newTable = new int[2*size];
      for (i=0; i < size; i++)
        füge table[i] in newTable ein;
      table = newTable;
      size = 2*size;
    }
    füge x in table ein;
    num = num + 1;
  }
```

Kosten von n – Einfügeoperationen in eine anfangs leere Tabelle

t_i Kosten der i-ten Einfügeoperation

Worst Case

- t_i=1, falls die Tabelle vor Operation i nicht voll ist
- t_i= (i-1)+1, falls die Tabelle vor Operation i voll ist

Also verursachen n-Einfügeoperationen höchstens Gesamtkosten von:

$O(n^2)$

Sei T eine Tabell mit
k= T.num Elementen
s= T.size Größe

Unsere Potentialfunktion (die vom Himmel fällt → nichttrivial eine geeignet zu finden)

$$\Phi(T) = 2k - s$$

Eigenschaften

- $\Phi_0 = \Phi(T_0) = \Phi(\text{leere Tabelle}) = -1$
- **Für alle** $i \geq 1$**:** $\Phi_i = \Phi(T_i) \geq 0$

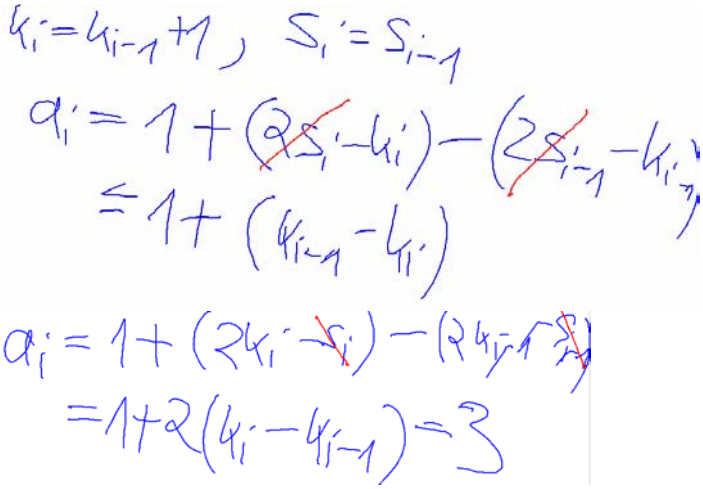
  Weil $\Phi_n - \Phi_0 \geq 0$ **gilt, ist**

  $\sum a_i$ **obere Schranke für** $\sum t_i$
- **Unmittelbar vor einer Tabellenexpansion ist** $k = s$**,**
  also $\Phi(T) = k = s$
- **Unmittelbar nach einer Tabellenexpansion ist** $k = s/2$**,**
  also $\Phi(T) = 2k - s = 0$

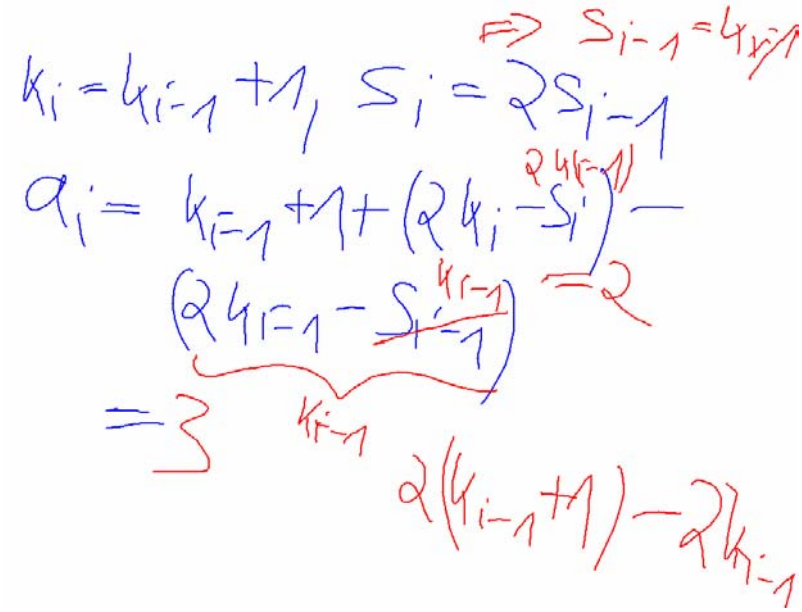
k_i = # Elemente in T nach der i–ten Operation

s_i = Tabellengröße von T nach der i–ten Operation

Fall 1: i-te Operation löst keine Expansion aus



Fall 2: i-te Operation löst Expansion aus



Entfernen von Elementen

Kontrahieren der Tabelle, falls die Belegung zu gering wird

Ziele:

1. Belegungsfaktor bleibt durch eine Konstante nach unten beschränkt
2. Amortisierte Kosten einer einzelnen einfüge oder entfernen Operation sind Konstant

1er Versuch:

- Expansion wie vorher
- Kontraktion: Halbiere Tabellengröße, sobald die Tabelle weniger als ½ voll ist

Das Problem könnte nun sein, dass die Eingabefolge von Entfernen und Einfügenoperationen die Tabelle immer abwechseln expandieren und kontrahieren lässt.

Also z.B. folgende Folge von Operationen

$I^{n/2}, I, D, D, I, I, D, D, \ldots$

dadurch erhalten wir eine Laufzeit von $O(n^2)$

2er Versuch

- Expansion wie vorher
- Kontraktion: Halbiere Tabellengröße, sobald die Tabelle weniger als ¼ voll ist

Folgerung:

Die Tabelle ist stets wenigstens zu ¼ voll, d.h.:

$$1/4 \leq \alpha(T) \leq 1$$

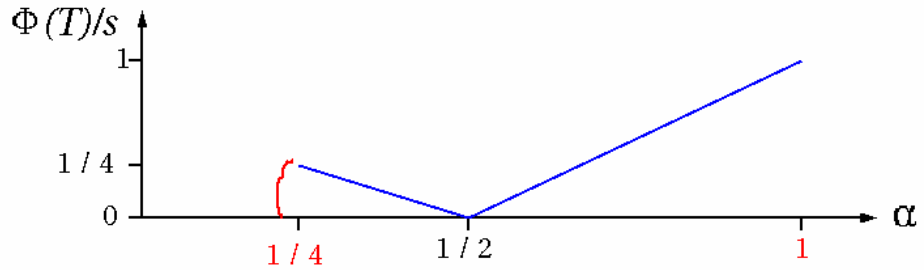
Was sind die Kosten einer Folge von Einfüge und Entferneoperationen?

k = T.num
s = T.size
α = k/s

Potentialfunktion Φ

$$\Phi(T) = \begin{cases} 2k - s, & \textbf{falls } \alpha \geq 1/2 \\ s/2 - k, & \textbf{falls } \alpha < 1/2 \end{cases}$$

unsere Belegung sieht also folgendermassen aus



Unmittelbar nach einer Kontraktion oder Expansion gilt immer:

s=2k, also Φ(T)=0

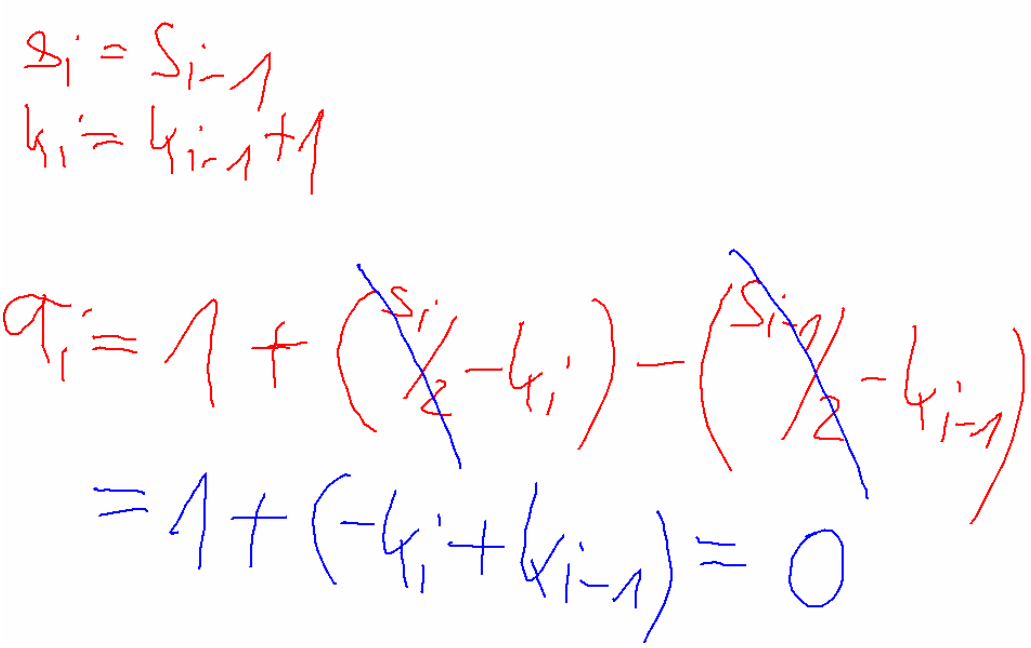
Einfügen

i-te Operation $k_i=K_{i-1}+1$

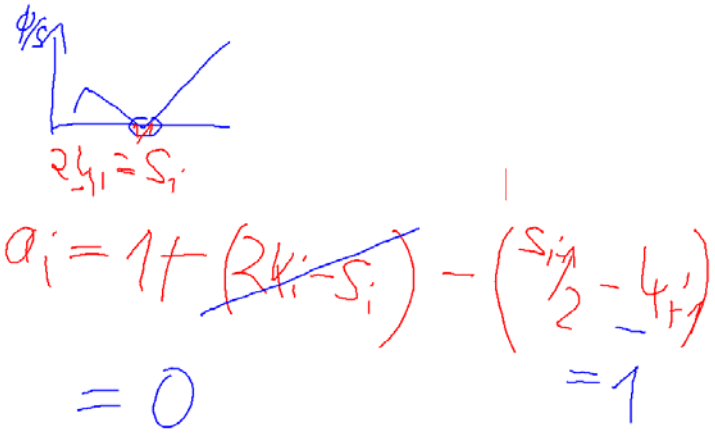
Fall 1: $\alpha_{i-1} \geq ½ \Rightarrow \alpha_i \geq ½$; $a_i = 3$ → O(1)

Fall 2: $\alpha_{i-1} \leq ½$

Fall 2.1: $\alpha_i < ½$: → Keine Expansion



Fall 2.1: $\alpha_i \geq ½$: → Keine Expansion



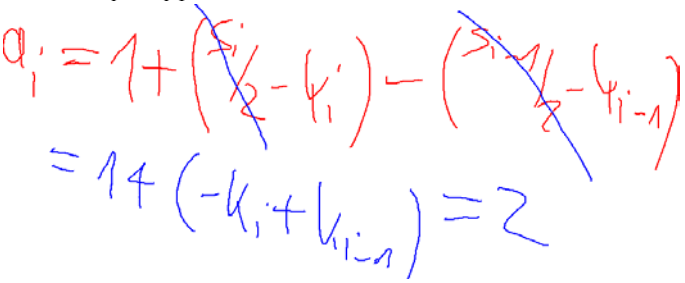
Entfernen

$k_i = k_{i-1} - 1$

Fall 1: $\alpha_{i-1} < ½$

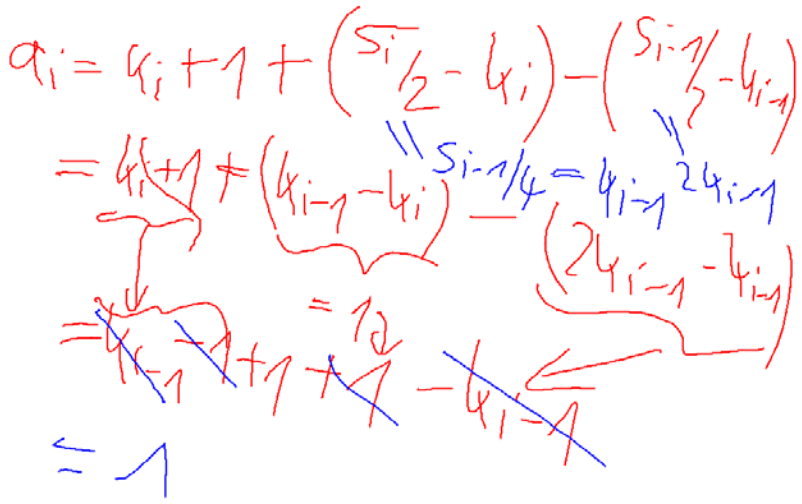
Fall 1.1: Entfernen verursacht keine Kontraktion

$s_i = s_{i-1}$



Fall 1.2: Entfernen verursacht Kontraktion

$2s_i = s_{i-1}$ $\quad k_{i-1} = s_{i-1}/4$

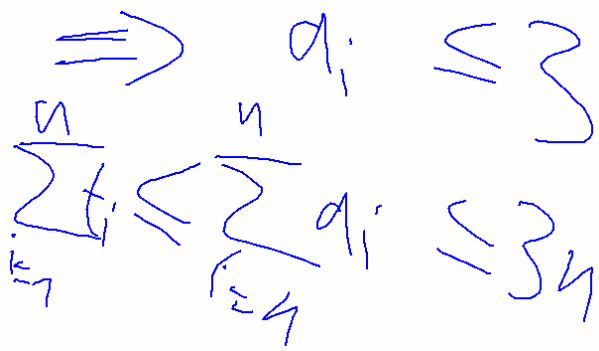


Fall 2: $\alpha_{i-1} \geq ½$ Entfernen verursacht keine Kontraktion
$s_i = s_{i-1}$ $k_i = k_{i-1} - 1$

Fall 2.1: $\alpha_i \geq ½$
Berechnungen nicht durchgeführt, aber analog wie zuvor

Fall 2.2: $\alpha_i < ½$
Berechnungen nicht durchgeführt, aber analog wie zuvor

Als Fazit kann man nun folgendes sagen:



Die amortisierten Kosten sind also Konstant

ZUM THEMA AMORTISIERTE ANALYSE GAB ES NOCH EINEN 3. VORTRAG, DER ABER NICHT OFFIZIELL ZU ALGORITHMENTHEORIE GEHÖRT UND DESHALB HIER NICHT BESPROCHEN WIRD.

Vorlesung 8 – Vorrangswarteschlangen (Priority Queues)

Operationen auf Vorrangswarteschlangen

$Q.initialize()$: erstelle die leere Menge Q

$Q.isEmpty()$: *true* gdw. Q ist leer

$Q.insert(e)$: füge Eintrag e in Q ein und gebe einen Zeiger auf den Knoten, der Eintrag e enthält, zurück

$Q.deletemin()$: liefere den Eintrag aus Q mit minimalen Schlüssel und entferne ihn

$Q.min()$: liefere den Eintrag aus Q mit minimalen Schlüssel

$Q.decreasekey(v,k)$: verringere den Schlüssel von Knoten v auf k

zusätzliche Operationen:

$Q.delete(v)$: entferne Knoten v mit Eintrag aus Q (ohne v zu suchen)

$Q.meld(Q')$: vereinige Q und Q' (*concatenable queue*)

$Q.search(k)$: suche den Eintrag mit Schlüssel k in Q (*searchable queue*)

u.v.a., z.B. $predecessor$, $successor$, max, $deletemax$

Implementationen von Vorrangswarteschlangen:

	Liste	Heap	Bin.-Q.	Fib.-Hp.
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
delete-min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
meld ($m \leq n$)	$O(1)$	$O(n)$ od. $O(m \log n)$	$O(\log n)$	$O(1)$
decr.-key	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$

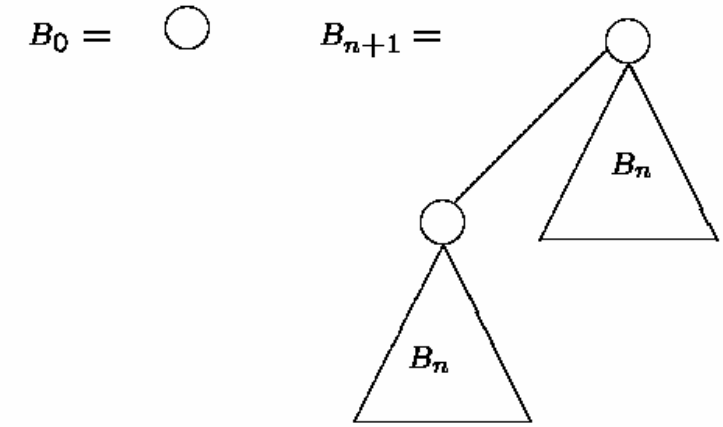
$*$ = amortisierte Zeit

$delete(e,Q) = decreasekey(e,-\infty,Q) + deletemin(Q)$

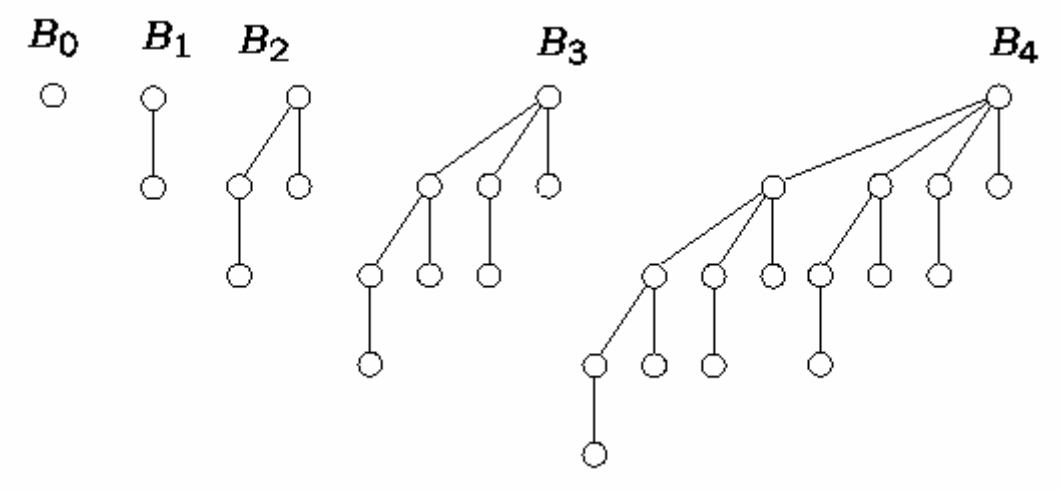
Binomialqueues

Binomialbäume

Rekursive Definition eines Binomialbaumes Bn, n ≥ 0

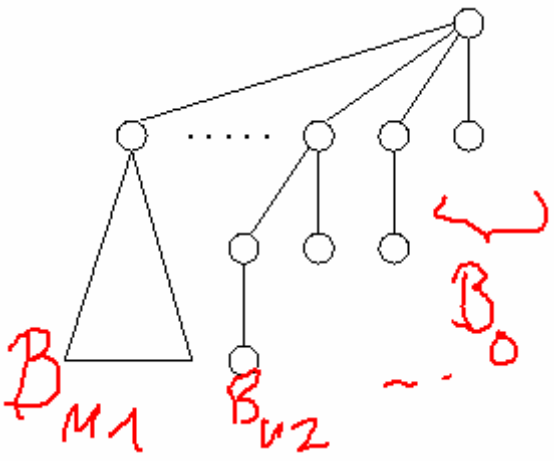


Beispiel für die ersten 5 Binomialbäume



Folgerungen daraus:

- B_n hat 2^n Knoten
- B_n hat Höhe n
- Wurzel hat Grad n (=Ordnung)
- B_n =



- Es gibt genau $\binom{n}{i}$ Knoten mit Tiefe i in B_n

Was Binomialbäume sind, haben wir hier eben definiert. Was sind aber nun Binomialqueues?

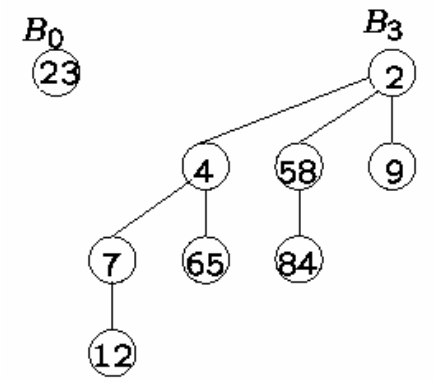
Binomialqueues

Eine Binomialqueue Q ist eine Vereinigung Heapgeordneter Binomialbäume verschiedener Ordnung

n Schlüssel: $B_i \in Q \Leftrightarrow i\text{-tes Bit in } (n)_2 = 1$

Beispiel 1:

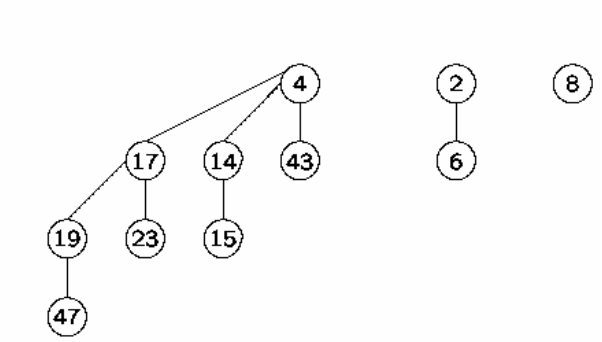
9 Schlüssel: {2,4,7,9,12,23,58,65,84}
$9 = (1001)_2$



Bestimmen des Minimums kostet höchstens O(log n) Zeit, da man hierzu nur alle Wurzeln der Bäume betrachten muss.

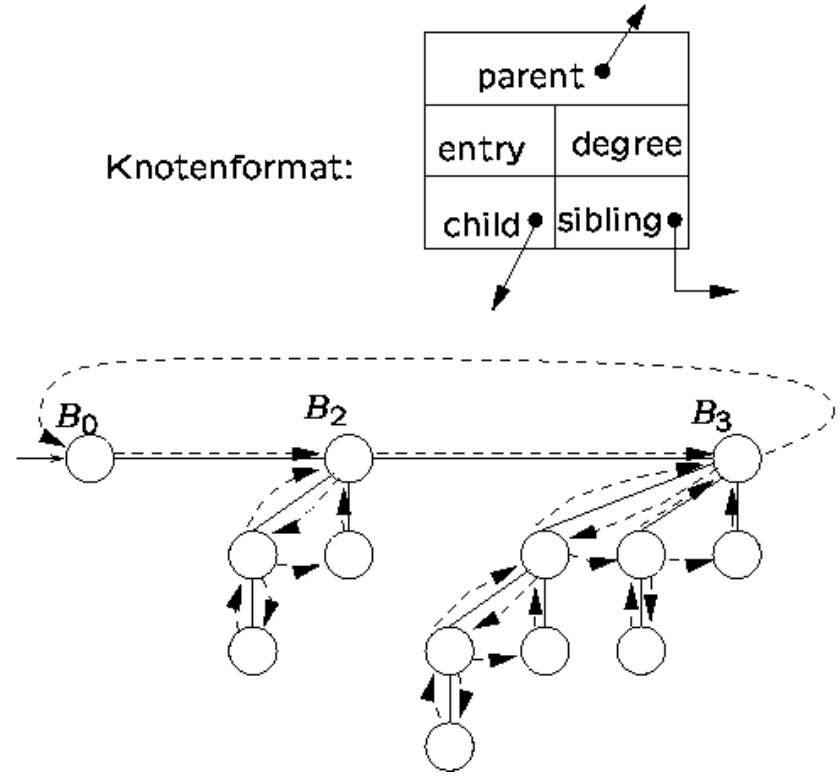
Beispiel 1:
11 Schlüssel: {2,4,6,8,14,15,17,19,23,43,47}
$11 = (1011)_2$ → 3 Binomialbäume B_0, B_1 und B_3
B_3, B_1, B_0

Zeiger auf Minimum



Was eine Binomialqueue ist, wissen wir nun, aber in welcher Datenstruktur wird sie abgelegt?

Child-Sibling Darstellung von Binomialqueues



Die Parentzeiger (der Wurzeln) sind zu einem Kreis verbunden. Es ist möglich, alle Kinder eines bestimmten Knotens zu durchlaufen, indem man zuerst zum Kind (Childzeiger) des aktuellen Knoten geht und dann die Geschwisterknoten (Siblingzeiger) durchläuft.

Nun folgt eine äußerst wichtige Funktion die als Grundlage für viele andere Operationen verwendet wird, das Verschmelzen:

Vereinigung (verschmelzen) zweier Binomialbäume B, B' gleicher Ordnung

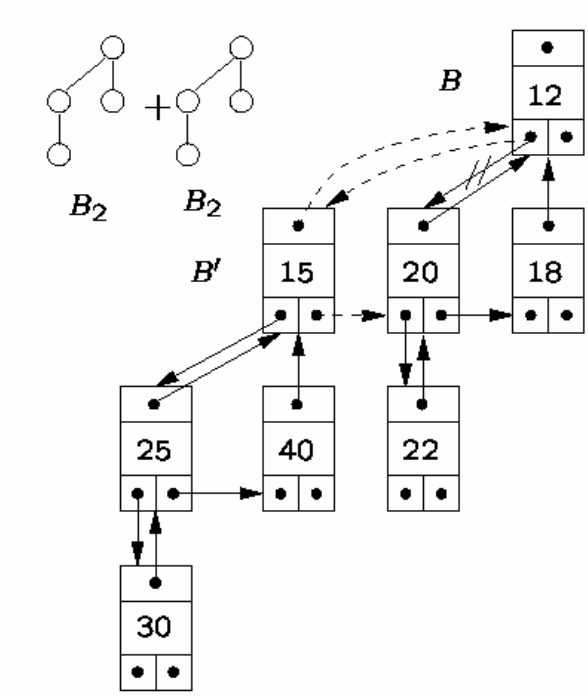
Link Operation:

```
B.Link(B')
1  if B.key > B'.key
2     then B'.Link(B)
   /* B.key ≤ B'.key */
3  B'.parent := B
4  B'.sibling := B.child
5  B.child := B'
6  B.degree := B.degree + 1
```

Mach Wurzel des Baumes mit größerem Schlüssel zum Sohn des Baumes mit kleinerem Schlüssel.

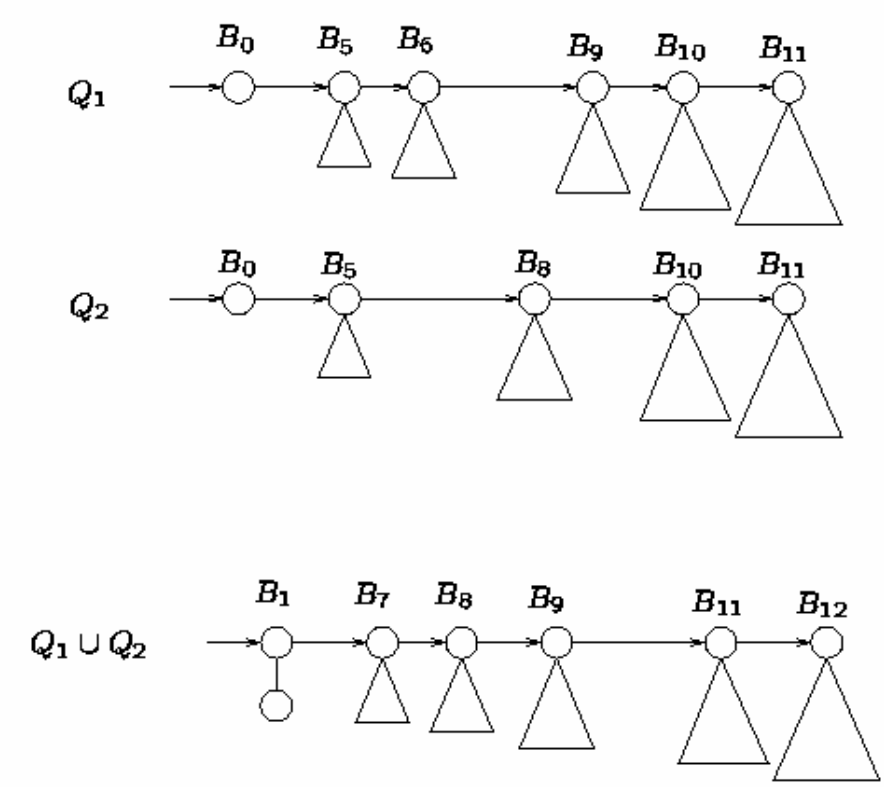
Die Link Operation geht in konstanter Zeit O(1)

Beispiel einer Linkoperation zweier Bäume der Ordnung B_2



Die gestrichelten Pfeile sind die neuen Zeiger. Der Baum mit der Wurzel „15“ wird als Sohn des anderen Baumes angehängt. Der Childlink von „12“ nach „20“ wird gestrichen und nach „15“ gesetzt. Der Sibling Zeiger von „15“ wird auf Child(20) gesetzt.

Vereinigung (verschmelzung) zweier Binomialqueues Q_1 und Q_2



Das Verschmelzen der zwei Binomialqueues entspricht einfach der Binären Addition.

Die beiden B_0 Bäume addieren sich zum Baum B_1 → Übertrag. Weitere B_1 gibt es nun nicht, deshalb wird B_1 eingetragen. Der nächste Baum ist nun erst B_5, diesen gibt es zweimal, wir erhalten einen Übertrag B_6, dieser ergibt mit dem weiteren B_6 aus Q_1 einen weiteren Übertrag B_7, B_6 kommt also nicht vor im Ergebnis…etc.

Die Verschmelzung der beiden Binomialqueues geht in O(log n) Zeit

$Q.initialize$: $Q.root = null$

$Q.insert(e)$: new B_0; $B_0.entry := e$; $Q.meld(B_0)$
Zeit: $O(\log n)$

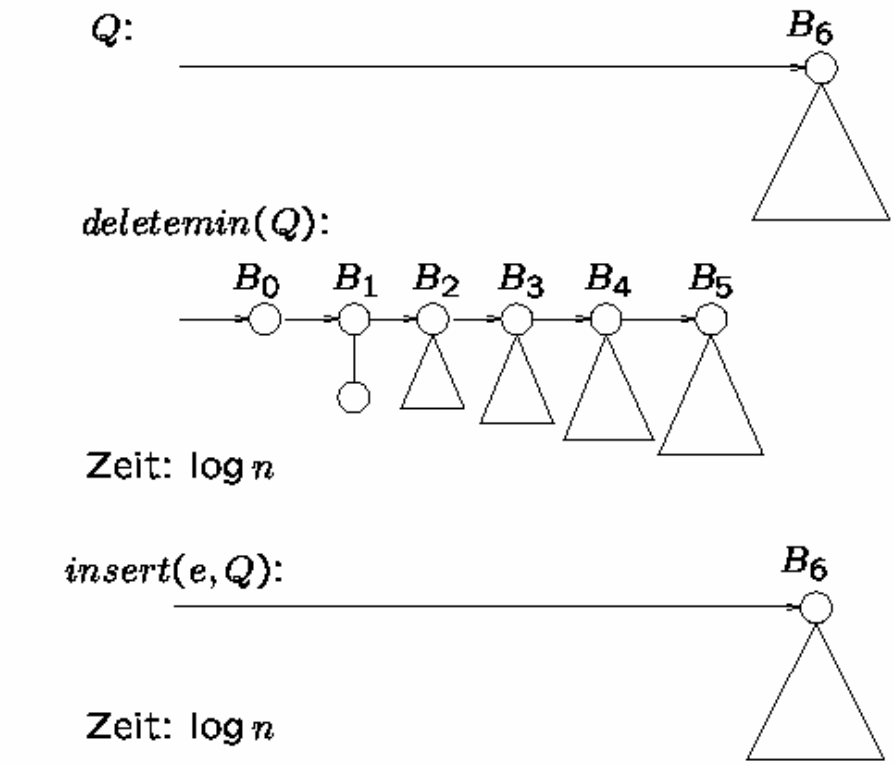
$Q.deletemin()$:
1. bestimme B_i mit minimalem Schlüssel in der Wurzelliste und entferne B_i aus Q
2. drehe die Reihenfolge der Söhne von B_i um, also zu $B_0, B_1, \ldots, B_{i-1} \Rightarrow Q'$
3. $Q.meld(Q')$

Zeit: $O(\log n)$

$Q.decreasekey(v, k)$:
1. $v.entry.key := k$
2. $v.entry$ nach oben steigen lassen in dem geg. Baum, bis die Heapbedingung erfüllt ist

Beim Einfügen eines Elementes wird zunächst ein neuer Baum B_0 mit dem einzufügenden Schlüssel als Inhalt erzeugt. Dieser Baum wird dann mit der Binomialqueue verschmolzen

Beispiel für eine Worst-Case Folge von Operationen:



$\Phi(Q) = \#$ Bäume in Q

Zusammenfassender Vergleich:

	Liste	Heap	Bin.-Q.	Fib.-Hp.
initial.	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
delete-min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
meld ($m \leq n$)	$O(1)$	$O(n)$ od. $O(m \log n)$	$O(\log n)$	$O(1)$
decr.-key	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$
delete	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$

$*$ = amortisierte Zeit

Vorlesung 9 – Fibonacci Heaps

Ein richtige ausführliche Definition von Fibonacci Heaps gibt es nicht, man kann sagen:

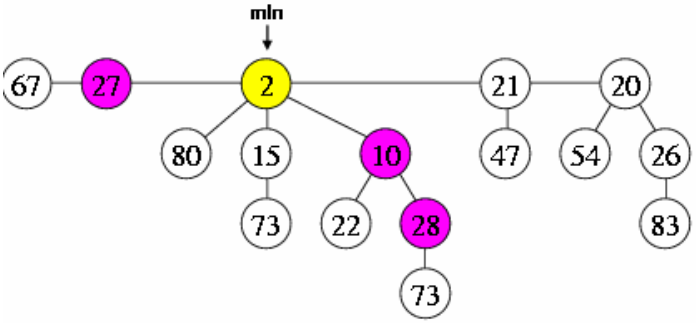
Ein Fibonacci Heap Q ist eine Kollektion heapgeordneter Bäume

Man bezeichnet die Fibonacci Heaps auch als Lazy-Meld Version der Binomialqueues → d.h. Die Vereinigung von Bäumen gleiche Ordnung wird bis zur nächsten Deletemin Operation aufgeschoben.

Man definiert folgende Variablen:

Q.min
→ Wurzel des Baumes mit kleinstem Schlüssel
Q.rootlist
→ Zirkuläre, doppeltverkettete und ungeordnete Liste der Wurzeln der Bäume
Q.size:
→ Anzahl der Knoten in Q

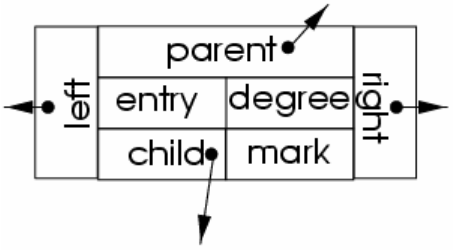
Beispiel für einen Fibonacci Heap:



B ist also ein heapgeordneter Baum der in der Liste der Wurzeln (Q.rootlist) steht.

B.childlist:
→ zirkuläre, doppeltverkettete und ungeordnete Liste der Söhne von B

Das Knotenformat sieht dann so aus:



Im Unterschied zum Binomialqueue Knotenformat haben wir nun statt eines Siblings (rechter Nachbar) einen rechten und einen linken Nachbarn.

Degree ist nun die Anzahl der Kinder (Söhne) des Knotens

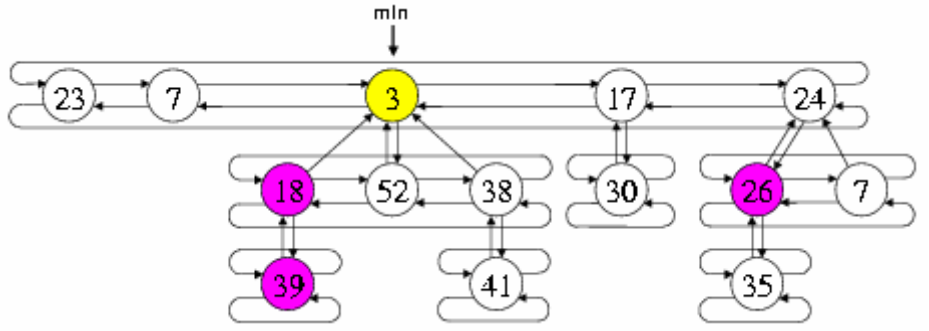
Child zeigt nur auf eines der Kinder des Knotens, diese sind aber ihrerseits auch wieder doppeltverkettet, so dass man indirekt auch auf die anderen Kinder zugreifen kann

Mark (Boolean) wird gesetzt, wenn eines der Kinder abgeschnitten wird.

Vorteile zirkulärer, doppeltverketteter Listen

1. Entfernen eines Elements in konstanter Zeit
2. Vereinigen zweier Listen in konstanter Zeit

So sieht der Fibonacci Heap mit allen Zeigern aus



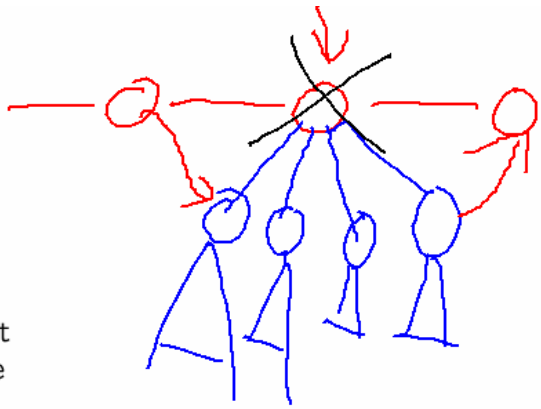
Operationen auf Fibonacci Heaps

Q.initialize
→ 1. Q.rootlist = Q.min = null
→ 2. Q.size = 0
Q.meld(Q')
→ 1. Verkette Q.rootlist und Q'.rootlist
→ 2. update Q.min
→ 3. Q.size=Q.size+Q'.size
Q.insert(e)
→ 1. Bilde einen Knoten für einen neuen Schlüssel => Q'
→ 2. Q.meld(Q')
Q.min()
→ 1. Gebe Q.min.entry zurück

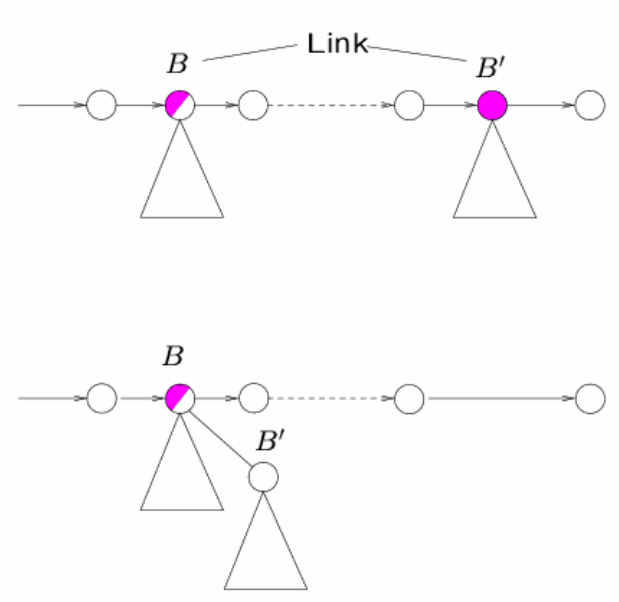
All diese Operationen gehen in O(1)

Q.deletemin()
→ Lösche den Knoten mit dem kleinsten Schlüssel aus Q und gebe den Eintrag des Knotens zurück

```
1  m = Q.min()
2  if Q.size() > 0
3     then entferne Q.min aus Q.rootlist
4          füge Q.min.childlist in Q.rootlist ein
5          Q.consolidate()
           /* Verschmelze solange Knoten mit
              gleichem Rang, bis das nicht geht
              und bestimme dann das minimale
              Element */
6  return m
```



link
→ Verbinden zweier Bäume gleichen rangs
→ rang (B) = Grad der Wurzel von B
→ 2 Heapgeordnete Bäume B,B' mit rang(B) = rang (B')
→ Der Baum mit dem größeren Wurzelschlüssel wird unter den anderen gehängt (B.key<B'.key)

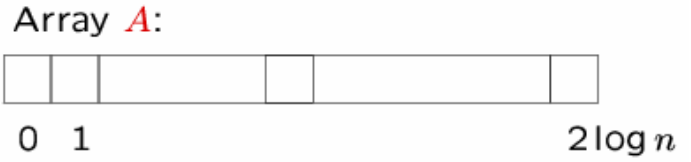


→ rang(B) = rang(B) +1
→ B'.mark = false (markierung wird aufgehoben)

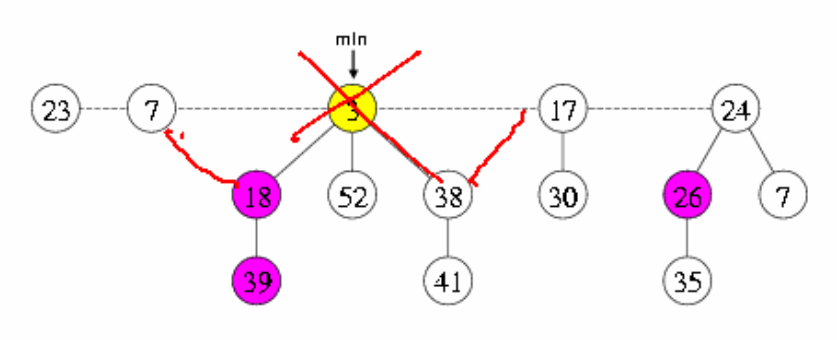
deletemin

- ⇨ Um die Funktion deletemin zu beschränken müssen wir zunächst wissen, wie groß der maximale Grad des Heaps ist, also aus wievielen Wurzeln die Rootliste besteht.
- ⇨ Annahme: rang(Q) ≤ 2 log n, wenn Q.size = n

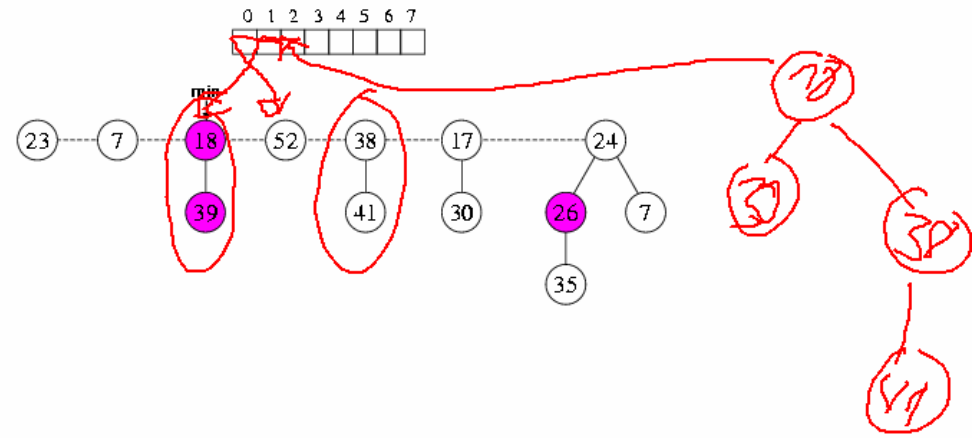
- ⇨ Finde Wurzeln mit gleichem Rang



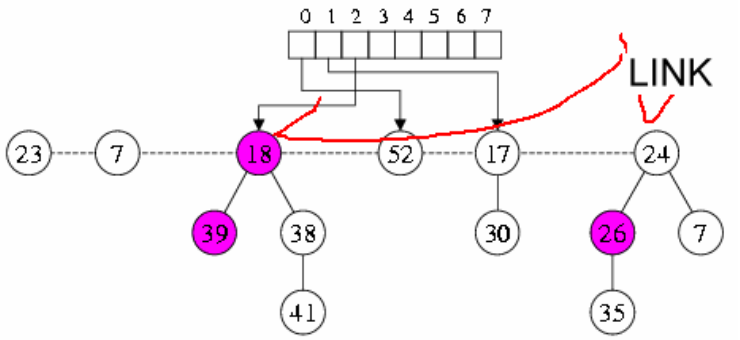
- ⇨ Erstelle ein Array mit Größe 2 log n. Jetzt wird die Wurzelliste sequentiell durchlaufen und Wurzeln mit Grad k werden an k-ter stelle im Array A eingetragen. Ist dieses schon belegt, dann muss verschmolzen werden. Das k-te Feld wird dann freigegeben und der neue verschmolzene Baum ins k+1-te Feld eingetragen, soweit dies frei ist, ansonsten wir erneut verschmolzen.



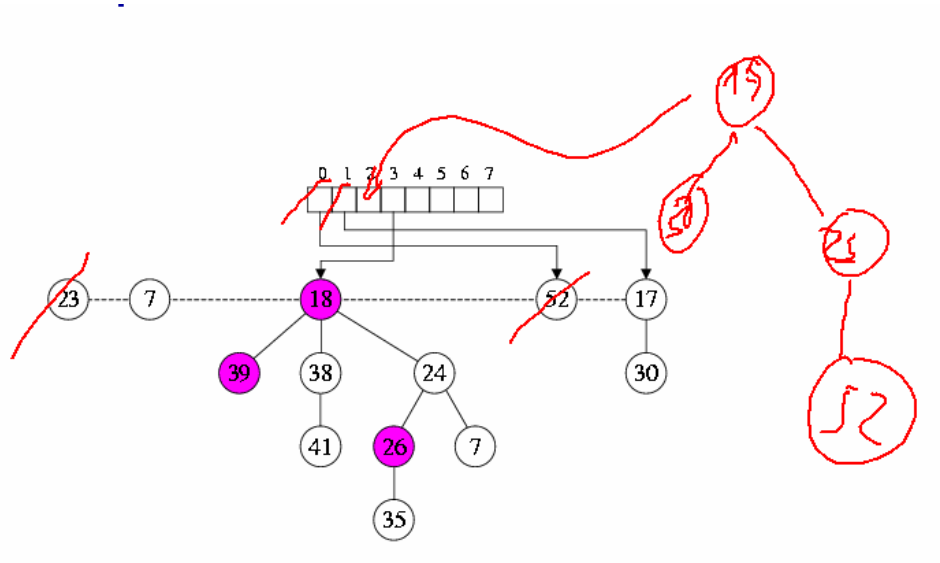
- ⇨ Auf obigem Bild haben wir nun den kleinsten Knoten (3) entfernt und die Kinder in die Wurzelliste übernommen
- ⇨ Wir beginnen nun die Wurzelliste ab der Stelle an der Q.min war sequentiell zu durchlaufen



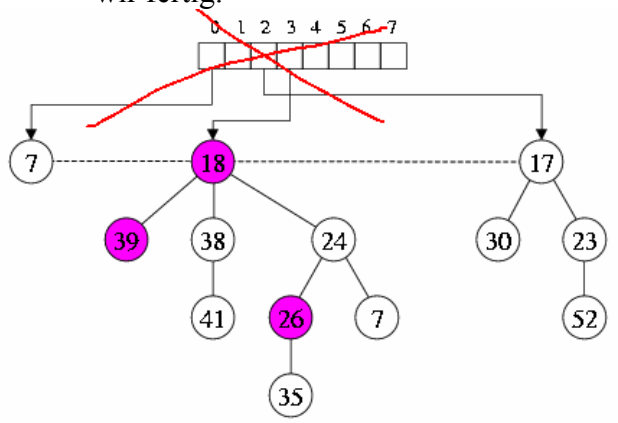
- ⇨ Es sind nun zwei neue Bäume vom Grad 1 enstanden, die miteinander verschmolzen werden müssen. Der neue, verschmolzene Baum (roter Baum) hat dann den Grad 2 und wird dann im 2. Feld des Arrays A eingetragen.



- ⇨ Nun werden die nächsten 2 Bäume mit Wurzel (18) und (24) verschmolzen.



⇨ Jetzt sind noch die beiden Knoten (23) und (52) vom gleichen Grad und werden verschmolzen. Nach dem Verschmelt entsteht nun wieder ein Baum mit Grad 1, welcher nun mit dem Baum mit Wurzel (17) verschmolzen werden muss. Dann sind wir fertig.



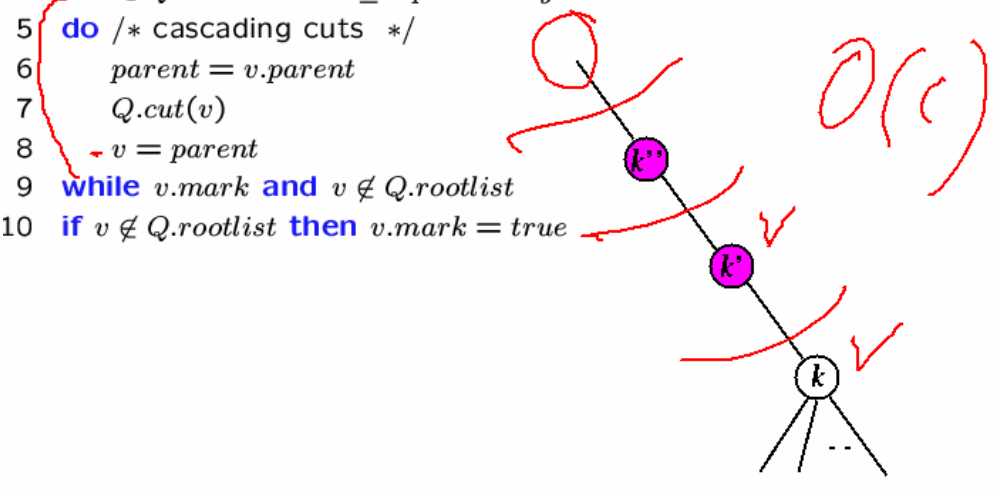
⇨ Das Array A kann nun gelöscht werden

Der ganze Algorithmus sieht nun so aus:

```
Q.consolidate()
1   A = Array von Fibonacci-Heap Knoten der
        Länge 2 log n
2   for i = 0 to 2 log n do A[i] = frei
3   while Q.rootlist ≠ ∅ do
4       B = Q.delete-first()
5       while A[rang(B)] ist nicht frei do
6           B' = A[rang(B)]
7           A[rang(B)] = frei
8           B = Link(B,B')
        end while
9       A[rang(B)] = B
    end while
10  Q.rootlist = A.list()
11  bestimme Q.min
```

Algorithmus Decrease Key

```
Q.decrease-key(v, k)
 1  if k > v.key then return
 2  v.key = k
 3  update Q.min
 4  if v ∈ Q.rootlist or k ≥ v.parent.key then return
 5  do /* cascading cuts */
 6      parent = v.parent
 7      Q.cut(v)
 8      v = parent
 9  while v.mark and v ∉ Q.rootlist
10  if v ∉ Q.rootlist then v.mark = true
```



- ⇨ Zum Ersetzen eines Schlüssels durch einen kleineren Schlüssel
- ⇨ Die Heapbedingung wird gesichert in dem der verringerte Knoten und seine nachfolger in die Wurzelliste mitaufgenommen werden (KEIN HEAPSORT)
- ⇨ Der Vater des entfernten Knoten wird nun markiert, sollte er schon markiert sein, wird er selbst auch in Wurzelliste mit aufgenommen und der Knoten darüber wird markiert → Cascading Cuts !
- ⇨ Die Laufzeit des Algorithmus ist durch die Anzahl der benötigten Cuts bestimmt also O(Cuts)

Algorthmus Q.cut

```
Q.cut(v)
1  if v ∉ Q.rootlist
2    then /* trenne v von seinem Vater */
3         rang(v.parent) = rang(v.parent) − 1
4         v.parent = null
5         entferne v aus v.parent.childlist
6         füge v in Q.rootlist ein
```

- ⇨ Der oben beschriebene Cut Algorithmus für die Cascading Cuts
- ⇨ Cut gehe ihn konstanter Zeit O(1)

Algorithmus Q.delete

- ⇨ zum entfernen eines beliebigen Knotens

```
Q.delete(v)
1  if v = Q.min
2     then return Q.delete-min
3  Q.cut(v) und kaskadiere
4  entferne v aus Q.rootlist
5  füge v.childlist in Q.rootlist ein
6  return v.key
```

- ⇨ Zunächst wird überprüft ob der zu entfernende Knoten v evtl. ein der min-Knoten ist, falls ja wird deletemin aufgerufen
- ⇨ Sonst wird mittels Cut der Knoten v in die Wurzelliste übernommen und Cascading Cuts ausgeführt (siehe oben)
- ⇨ Der Knoten v wird nun aus der Wurzelliste entfernt und sein Kinder in diese eingefügt
- ⇨ Die Laufzeit des Algorithmus ist durch die Anzahl der benötigten Cuts bestimmt also O(Cuts)

Wir wollen nun noch einmal genau erklären wann ein Knoten markiert wird

Markieren von Knoten

- ⇨ wenn v an einen Knoten angehängt wird, dann wird v demarkiert
  - o also v.mark = false
- ⇨ wenn v einen seiner Söhne verliert, dann wird v markiert
  - o also v.mark = true
- ⇨ wenn v nun einen zweiten Sohn verliert, dann wird v abgetrennt und in die Wurzelliste übernommen

Die Markierung gibt also an, ob v bereits einen Sohn verloren hat, seitdem v zuletzt Sohn eines anderen Knotens geworden ist.

Analyse der Operationen von Fibonacci Heaps

Mit Hilfe der Potentialfunktionsmethode.

Folgende Funktion verwenden wir: $\Phi_Q = r_Q + 2m_Q$

r_Q

- ⇨ ist die Anzahl der Knoten in Q.rootlist

m_Q

- ⇨ ist die Anzahl der markierten Knoten in Q, die sich nicht in der Wurzelliste befinden

Die Analyse der Operationen ergab folgendes:

	Liste	Heap	Bin.-Q.	Fib.-Hp.
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
delete-min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
meld ($m \leq n$)	$O(1)$	$O(n)$ od. $O(m \log n)$	$O(\log n)$	$O(1)$
decr.-key	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$
delete	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$

$*$ = amortisierte Zeit

Erkenntnis: Eine der Operationen des Fibonacci Heaps muss logarithmisch, da man sonst die untere Schranke des Sortieren unterschreiten und man somit in O(n) sortieren könnte, die untere Schranke für Sortieren ist aber n log n. Sortieren kann man hier simulieren durch n mal insert und dann n mal deletemin. Ergibt O(n log n)

Diskurs über die Annahme, dass der rang(Q) ≤ 2 log n ist

Betrachten wir zunächst folgendes Lemma
Sei v ein Knoten in dem Fibonacci Heap Q. Ordnet man die Söhne $u_1,\ldots,u_k$ von v nach dem Zeitpunkt des Anhängens an v, dann gilt:

$$\text{rang}(u_i) \geq i - 2$$

Beweis:

Zeitpunkt des Anhängens von u_i
- #Söhne von v (rang(v)) $\geq i - 1$
  - (→ zum Zeitpunkt i wurden bereits mindestens i-1 Knoten an v gelinkt)
- #Söhne von ui (rang(u_i)) $\geq i - 1$
  - (→ weil der Rang von v und u_i gleich sein muss zum verlinken)

Söhne die u_i in der Zwischenzeit verloren hat: höchstens 1, da u_i sonst abgehängt würde. (Stichwort: Markierungen)

⇨ $\text{rang}(u_i) \geq i-2$ q.e.d.

Maximaler Rang eines Knotens

Satz: Sei v ein Knoten in einem Fibonacci Heap Q mit Rang k. Dann ist v die Wurzel eines Teilbaumes mit mindesten F_{k+2} Knoten. F_k ist die k-te Fibonacci Zahl. Deswegen heissen die Fibonacci Heaps so wie sie heissen.

Fibonacci Zahlen
$F_0 = 0$
$F_1 = 1$
$F_k = F_{k-1} + F_{k-2}$
→ {0,1,1,2,3,5,8,13,21,34…}

- ⇨ Die Fibonacci Zahlen steigen approximativ wie der „goldene Schnitt“, also ca 1.61^k ((SQRT(5)+1)/2)=1.61…)

Die Anzahl der Nachkommen eines Knotens ist exponentiell in der Zahl seiner Söhne

- ⇨ Für den maximalen Rang k eines Knotens v in einem Fibonacci Heap Q mit n Knoten gilt:

  Bei k Söhnen gilt $F_{k+2} \leq n$ → $k = O(\log n)$

Beweis:

S_k = Mindestgröße eines Teilbaumes, dessen Wurzel k Söhne hat
$S_0 = 1 \geq F_2$
$S_1 = 2 \geq F_3$

Es gilt:

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \qquad \text{für } k \geq 2 \qquad (1)$$

Fibonacci Zahlen:

$$F_{k+2} = 2 + \sum_{i=2}^{k} F_i \qquad (2)$$

$$= 2 + F_2 + F_3 + \cdots + F_k$$

$$\leq S_0 + \leq S_1 + \ldots + \leq S_{k-2} \leq \mathbf{S_k}$$

aus (1) und (2) und Induktion folgt: $S_k \geq F_{k+2}$

Vorlesung 10 – Union Find Strukturen

Problem:
Verwaltung einer Familie von disjunkten Mengen unter den Operationen.

Repräsentation der Mengen M_i:

M_i wird durch ein repräsentatives Element aus M_i identifiziert

e.make-set():

erzeugt eine neue Menge mit kanonischem Element e. Die Menge wird durch e repräsentiert

e.find-set():

Liefert den Namen des Repräsentanten derjenigen Menge, die das Element e enthält.

e.union(f):

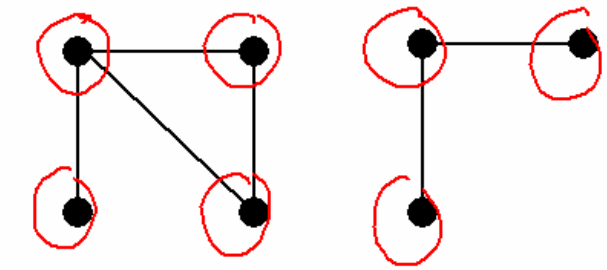
Vereinigt die Mengen M_e und M_f, die die Elemente e und f enthalten, zu einer neuen Menge M und liefert ein Element aus M_e U M_f als Repräsentanten von M. M_e und M_f werden zerstört.

Beispiel: Zusammenhangstest

Input: Graph G=(V,E)
Output: Die Familie der Zusammenhangskomponenten von G

Algorithmus: Connected Components

```
for all v in V do v.makeset()
for all (u,v) in E do
  if u.findset() ≠ v.findset()
    then u.union(v)
```



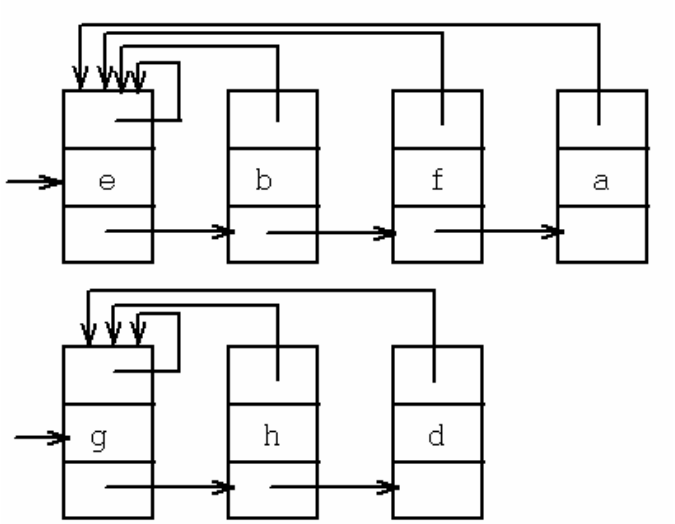
- Erzeuge für jeden Knoten ein Set
- Wenn es eine Kante gibt die die Knoten u und v verbindet, aber u und v noch in verschiedenen Sets sind, dann werden diese beiden Sets vereinigt

Same Component (u,v):

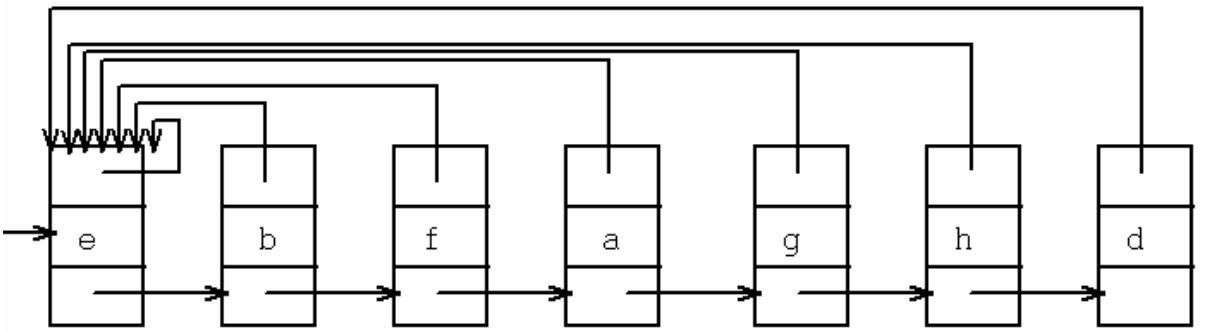
```
if   u.findset()   =   v.findset()
  then return   true
  else return   false
```

- testen ob zwei Knoten in der gleichen Zusammenhangskomponente liegen

Repräsentation durch verkettete Listen:



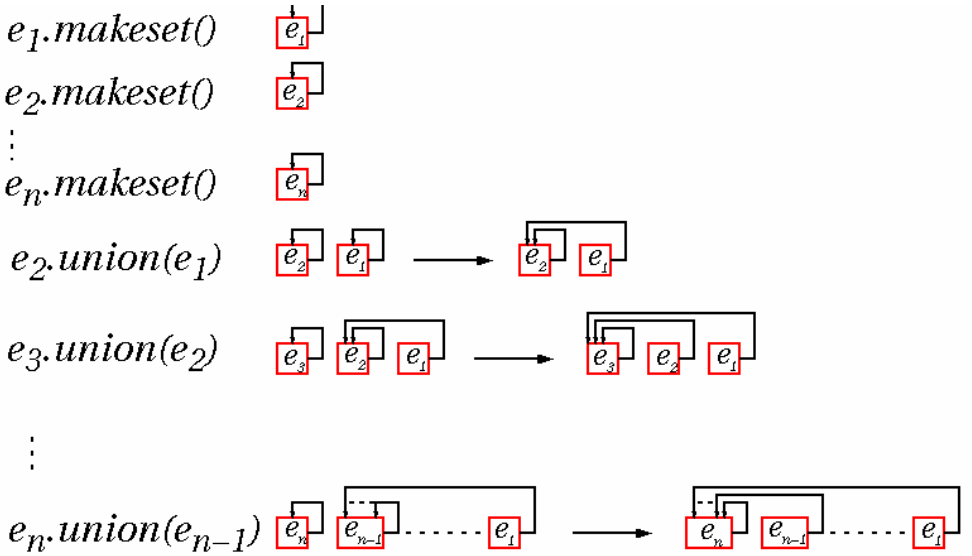
b.union(d)→ g wird an Knoten a angehängt (z.B. a.union(g) würde dasselbe machen)



Laufzeit der verwendeten Operationen:

- x.makeset() → O(1)
- x.findset() → O(1)
- x.union(y) → O(Größe der angehängten Liste)

Worst Case - Operationenfolgen:



Problem: Längere Liste wird stets an die kürzere angehängt

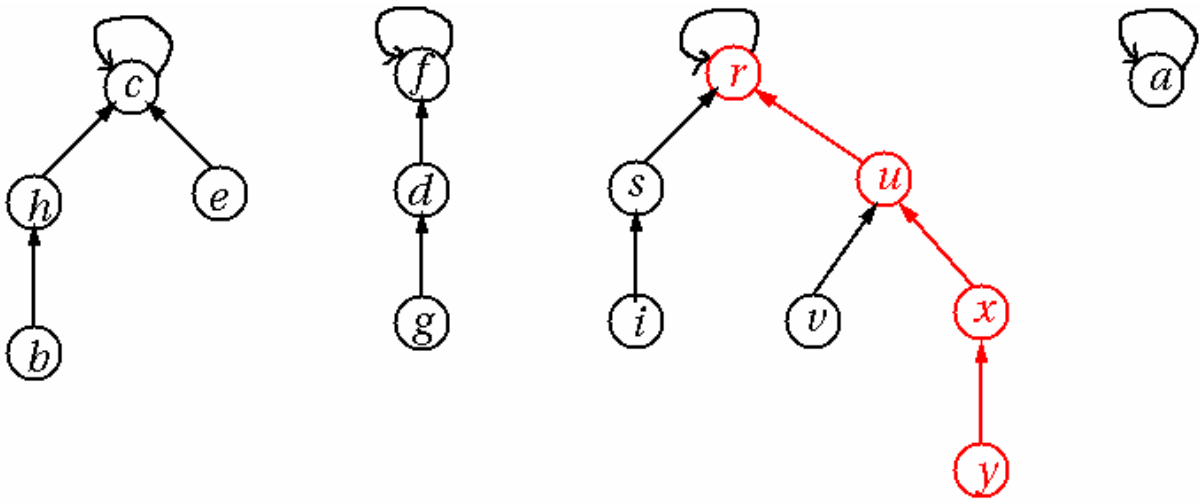
- wir haben dann im ungünstigsten Fall i-1 Zeigerzuweisungen im i-ten Schritt (e_i.union(e_{i-1}))
- 2n Operationen kosten also: O(n) + O($\sum_{i=2}^{n} i-1$)

→ Verbesserung: Immer kleinere an größere Liste anhängen !
man nennt dies gewichtete Union-Heuristik, man muss dann allerdings die Listenlänge als Parameter mitführen.

Satz:
Bei Verwendung der gewichteten Union-Heuristik kann jede Folge von m *makeset()*, *find()* und *union()* Operationen, die höchsten n *makeset()*-Operationen enthält in Zeit **O(m+n log n)** ausgeführt werden.

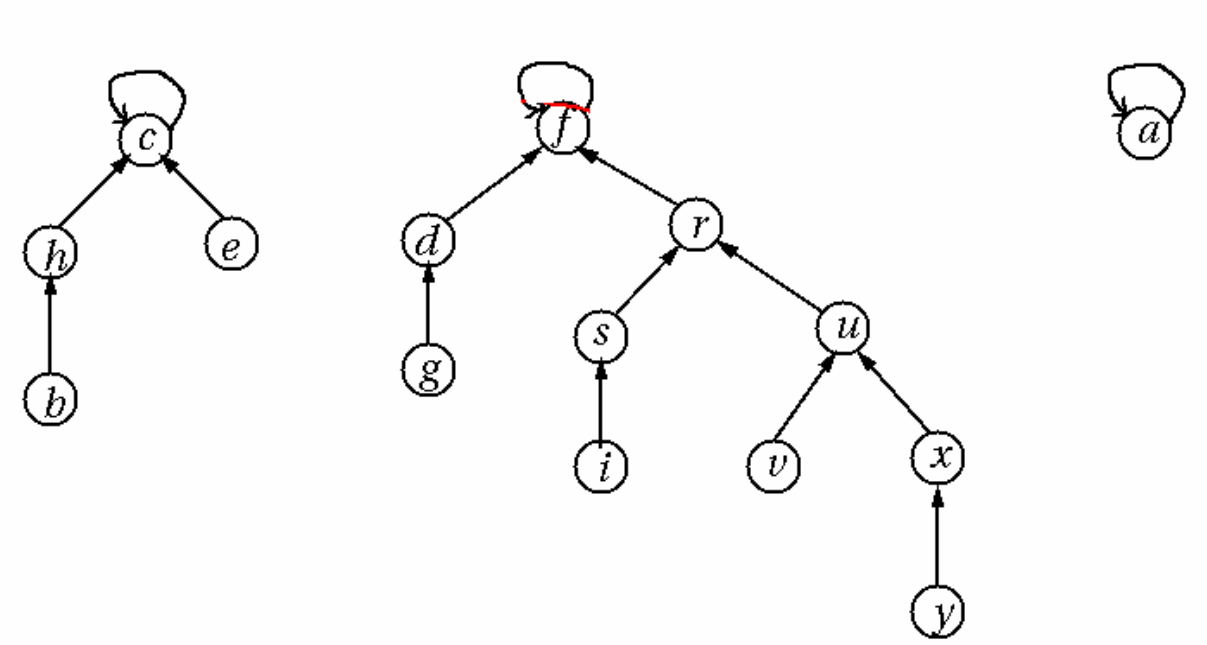
Beweis:
Betrachte Element e
Anzahl der Änderungen des Zeigers von e auf das repräsentative Element: log n

Repräsentation durch Wald von Bäumen



- a.makeset() → siehe rechtester Baum

- y.findset() → siehe roten Pfad (liefert r als Ergebnis
- d.union(i) → Mache den Repräsentanten der einen Menge (z.B. f) zum direkten Vater des Repräsentanten der anderen Menge (r)



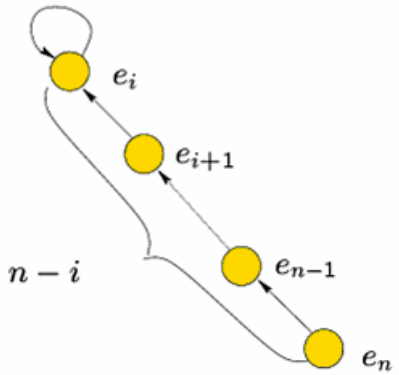
Worst-Case Beispiel

for $i = 1$ to n do $e_i.make\text{-}set()$

for $i = n$ to 2 do $e_{i-1}.union(e_i)$

for $i = 1$ to f do $e_n.find\text{-}set()$

i-ter Schritt



- Erzeuge n Bäume mit einem Element (Knoten)
- Vereinige diese Bäume nun in umgekehrte Reihenfolge

⇨ Somit entsteht eine lineare Liste

- man benötigt nun für die f findset() Operationen O(f n) Zeit

→ SCHLECHT

Verbesserung (ähnlich wie zuvor bei den verketteten Listen)

Vereinigung nach Größe

zusätzliches Feld:
$e.size$ = (#Knoten im Teilbaum von e)

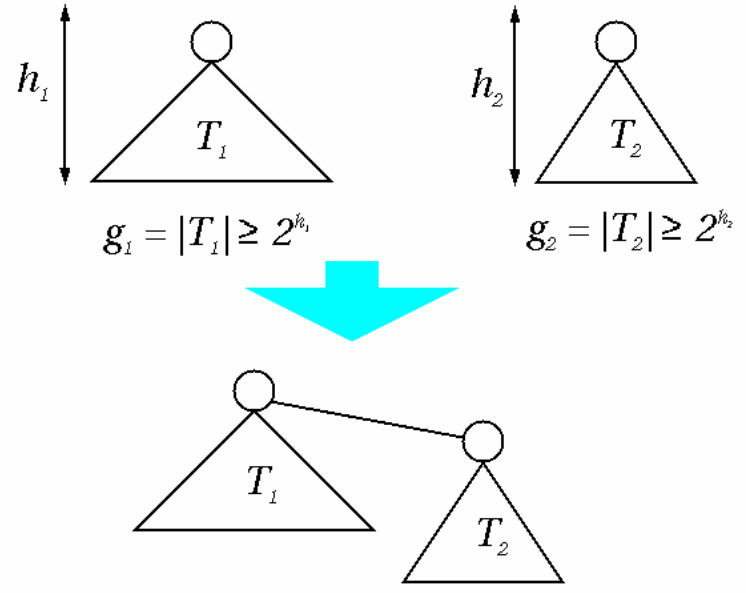
```
e.make-Set()
1  e.parent = e
2  e.size = 1

e.union(f)
1  Link(e.find-set(), f.find-set())

Link(e, f)
1  if e.size ≥ f.size
2     then f.parent = e
3          e.size = e.size + f.size
4     else /* e.size < f.size */
5          e.parent = f
6          f.size = e.size + f.size
```

- Der kleinere Baum wird nun an den Größeren angehängt

Satz:
Das Verfahren „Vereinigung nach Größe“ hat folgende Invariante:
⇨ Ein Baum mit Höhe h hat wenigstens 2^h Knoten

Beweis:
Induktionsanfang: Bei einem Baum mit einem Knoten gilt die Behauptung weil $2^0=1$

Induktionsschritt:



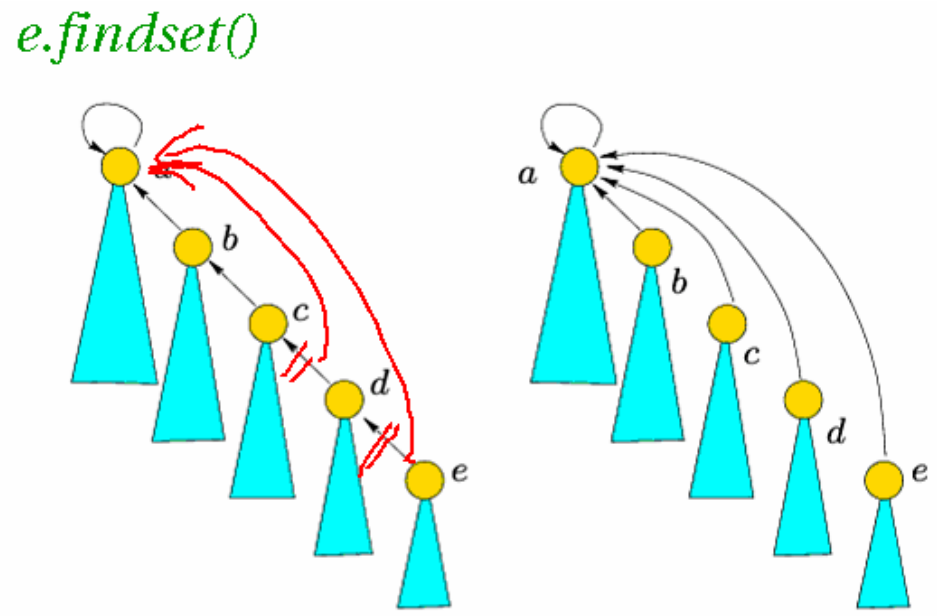
$g_1 \geq g_2$

Fall1: *Der neue Baum ist genauso hoch wie T_1*
$g_1+g_2 \geq g_1 \geq 2^{h1}$

Fall2: *Der neue Baum ist gewachsen*
Höhe von T: $h_2 + 1$
$g = g_1+g_2 \geq 2^{h2} + 2^{h2} = 2^{h2+1} \geq g_2$

Konsequenz:
Eine findset()-Operation kostet höchstens O(log n) viele Schritte, falls die Anzahl der makeset()-Operationen gleich n ist

Weitere Verbesserung durch Pfadverkürzung




```
e.find-set()
1  if e ≠ e.parent
2     then e.parent = e.parent.find-set()
3  return  e.parent
```

Beim durchlaufen der Knoten während findset() setzt man einfach alle Zeiger direkt auf die Wurzel

Analyse der Laufzeit

m Gesamtanzahl der Operationen, davon sind

f find-Set() Operationen und
n make-Set() Operationen.
⇨ höchstens n-1 union-Operationen

Vereinigung nach Größe:
⇨ O(n + f log n)

Vereinigung mit Pfadverkürzung
⇨ $\text{Falls } f < n,\ \Theta(n + f \log n)$
$\text{Falls } f \geq n,\ \Theta(f \log_{1+f/n} n)$

Anmerkung: Die Basis des Logarithmus in der Laufzeit verändert sich.

Satz: (Vereinigung nach Größe und Pfadverkürzung)
m union-find-operationen über n Elementen benötigen bei der Verwendung der Strategie Vereinigung nach Größe und Pfadverkürzung $\Theta(m \cdot \alpha(m,n))$ Schritte.

Wobei $\alpha(m,n)$ die inverse der Ackermann Funktion ist !!!

Exkurs: Ackermann Funktion und Inverse

Ackermann Funktion

$$\begin{aligned} A(1,j) &= 2^j, && \text{für } j \geq 1 \\ A(i,1) &= A(i-1,2) && \text{für } i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)) && \text{für } i,j \geq 2 \end{aligned}$$

Inverse Ackermann Funktion

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$$

$$\begin{aligned} A(i, \lfloor m/n \rfloor) &\geq A(i,1) \\ A(2,1) &= A(1,2) = 2^2 = 4 \\ A(3,1) &= A(2,2) = A(1, A(2,1)) = 2^4 = 16 \\ A(4,1) &= A(3,2) = A(2, A(3,1)) = A(2,16) \\ &\geq 2^{2^{2^2}} = 2^{65536} \end{aligned}$$

$\alpha(m,n) \leq 4$, für $\log n < 2^{65536}$

Vorlesung 11 – Greedy Verfahren

Greedy Verfahren kann man zur Lösung eines Optimierungsproblems:verwenden
⇨ Treffe in jedem Verfahrensschritt diejenige Entscheidung die im Moment am besten ist

Es gibt 3 Möglichkeiten des Ergebnisses:

- Wir erhalten stets die optimale Gesamtlösung
- Wir erhalten eine Lösung, die zwar nicht optimal ist, aber vom Optimum stets nur wenig abweicht
- Die berechnete Lösung kann beliebig schlecht werden

Zwei einfache Beispiele bei denen Greedy sehr schlecht werden kann.

Beispiel 1: Münzwechselproblem

Euro Bargeld Werte:

{500, 200, 100, 50, 20,10,5,2,1}

Beobachtung:
Jeder Euro Betrag kann durch Münzen und Banknoten mit diesen Werten bezahlt werden.

Ziel:
Bezahlung eines Betrages n mit möglichst wenig Münzen und Banknoten

Das Greedy Verfahren geht nun folgendermaßen vor: Wähle die maximale Anzahl von Banknoten und Münzen mit jeweils größtmöglichem Wert, bis der gewünschte Betrag n erreicht ist.

Beispiel: n=487

500	**200**	**100**	**50**	**20**	**10**	**5**	**2**	**1**
-	2x	-	1x	1x	1x	1x	1x	-

Allgemeines Münzwechselproblem:

Werte von Münzen und Banknoten $n_1,\ldots,n_k$ mit
$n_1 > n_2 > n_3 > n_4 > \ldots > n_k$ und $n_k = 1$

Das Greedy Zahlungsverfahren als Algorithmus

1. $w = n$

2. for $i = 1$ **to** k **do**

 # Münzen mit Wert $n_i = \lfloor w / n_i \rfloor$

$w = w - n_i \lfloor w / n_i \rfloor$

Jeder Geldbetrag kann bezahlt werden

Beispiel für ein beliebig schlechtes Abschneiden des Greedy Verfahrens
Im Land Absurdia

Es gibt nur drei Münzen: $n_3 = 1$; $n_2 > 1$ beliebig; $n_1 = 2n_2 + 1$

Beispiel: 41,20,1

Zu zahlender Betrag $n = 3n_2$ (z.B. n=60)

Optimale Zahlungsweise: 3 Münzen à n_2

Greedy Zahlungsverfahren: → eine Münze vom Wert n_3
Rest ist nun $n - n_3$
→ Greedy wählt n_2-1 Münzen vom Wert $n_1=1$

Wie man sieht wählt Greedy hier sehr schlecht, weil n_2 beliebig ist

Beispiel 2: Travelling Salesman Problem (TSP)

Gegeben: n Orte und Kosten c(i,j) um von i nach j zu reisen
Gesucht: Eine billigste Rundreise, die alle Orte genau einmal besucht

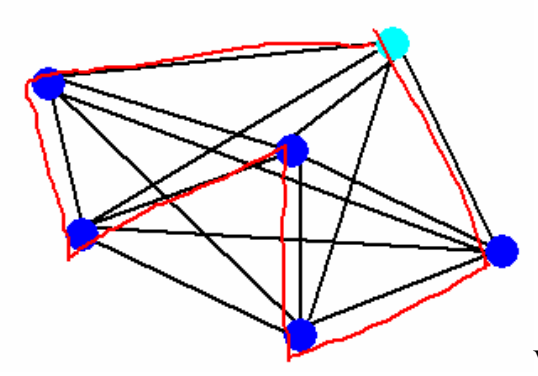
Formal: Eine Permutation p von {1,..,n} so dass

$$c(p(1),p(2)) + \cdots + c(p(n-1),p(n)) + c(p(n),p(1))$$

mininmal ist

Greedy Verfahren zur Lösung von TSP:

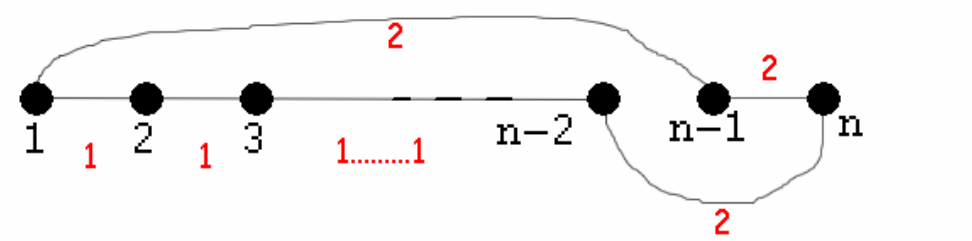
Beginne mit Ort 1 und gehe jeweils zum nächsten noch nicht besuchten Ort . Wenn alle Orte besucht sind, kehre zum Ausgangsort 1 zurück.



Vollständiger Graph

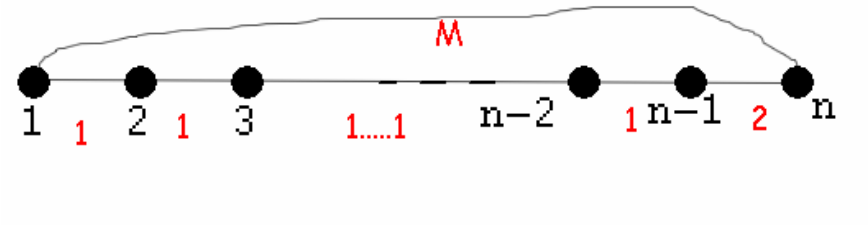
Beispiel für beliebig schlechtes Abschneiden des Greedy Verfahren:

c(i,i+1) = 1, für i = 1,2,…,n-1
c(n,1) = M, für eine sehr große Zahl M
c(i,j) = 2, sonst



Optimale Lösung (c(T)=n+3)

Vom Greedy Verfahren berechnete Tour:



Sehr schlechte Lösung (c(T)=n-1+M)

Das Aktivitäten Auswahlproblem

Gegeben:

- ⇨ $S=\{a_1,a_2,a_3,\ldots,a_n\}$, Menge von n Aktivitäten, die alle eine Ressource benötigen, z.B. einen Hörsaal

Aktivität a_i:

- ⇨ Beginn b_i und Ende e_i
- ⇨ Aktivitäten a_i und a_j heißen kompatibel, falls:

$$[b_i,e_i)\cap[b_j,e_j)=\emptyset$$

Gesucht:

- ⇨ Eine größtmögliche Menge paarweise kompatibler Aktivitäten

Idee:

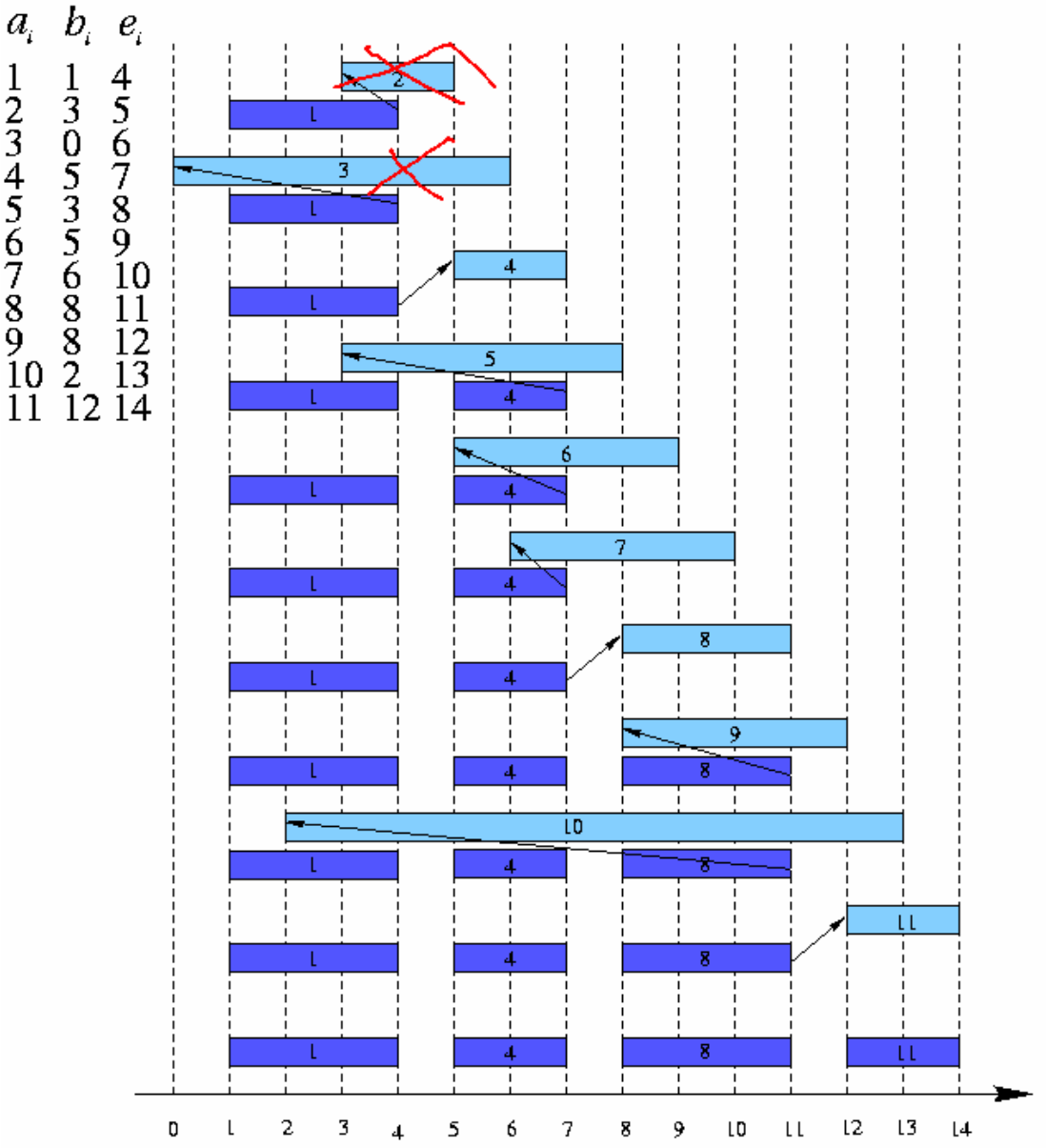
- ⇨ Maximiere die nicht verwendete Zeit

Annahme:

- ⇨ Aktivitäten sind nach aufsteigender Zeit des Endes sortiert:
- ⇨ $e_1\leq e_2\leq e_3\leq\ldots\leq e_n$

Greedy Strategie:

- ⇨ Wähle immer die Aktivität mit frühestem Endzeitpunkt, die legal eingeplant werden kann
- ⇨ Insbesondere ist die erste Aktivität die mit dem frühesten Endzeitpunkt



Der Zugehörige Algorithmus

```
Algorithmus Greedy-Aktivitäten
Input:   n Aktivitätenintervalle [b_i, e_i), 1 ≤ i ≤ n
         mit e_i ≤ e_{i+1};
Output:  Eine maximal große Menge von paarweise
         kompatiblen Aktivitäten;
1  A_1 = {a_1}
2  last = 1
3  for i = 2 to n do
       /* last ist die zuletzt zu A_{i-1} hinzugefügte
          Aktivität */
4      if b_i < e_last
5        then A_i = A_{i-1}
6        else /* b_i ≥ e_last */
7             A_i = A_{i-1} ∪ {a_i}
8             last = i
9  return A_n
```

Laufzeit des Algorithmus: O(n)
(Schleife von 2 bis n, jedes Element muss einmal angesehen werden)

Invarianten:

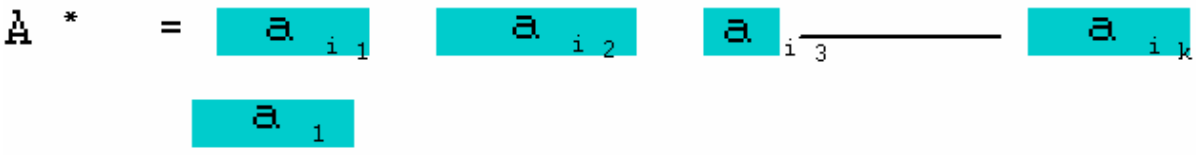
Es gilt: $e_{last} = \max\{e_k | a_k \in A_i\}$
Es gibt eine optimale Lösung A* mit
$A^* \cap \{a_1,\ldots,a_i\} = A_i$

Satz:

- ⇨ Das Greedy Verfahren zur Auswahl der Aktivitäten liefert eine optimale Lösung des Aktivitäten Auswahlproblems

Beweis:

- ⇨ Wir zeigen: Für alle $1 \leq i \leq n$ gilt:
  - Es gibt eine optimale Lösung A* mit $A^* \cap \{a_1,\ldots,a_i\} = A_i$
- ⇨ Induktionsanfang: i=1:
  - wähle $A^* \subseteq \{a_1,\ldots,a_n\}$, A* ist optimal $A^* = \{A_{i1},\ldots,A_{ik}\}$



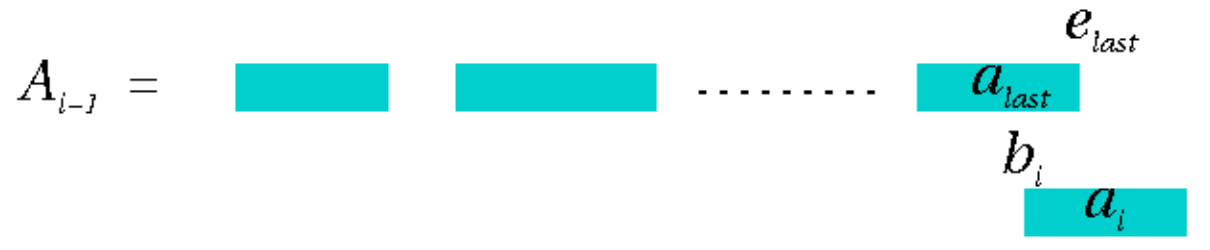
$A_2^* = A^* \setminus \{a_{i1}\} \cup \{a_1\}$ (d.h. a_{i1} wird durch a_1 ersetzt) → Der Induktionsanfang stimmt

- ⇨ Induktionsschritt: i - 1 → i
- ⇨ Wähle $A^* \subseteq \{a_1,\ldots,a_n\}$, A* ist optimal mit $A^* \cap \{a_1,\ldots,a_{i-1}\} = A_{i-1}$
- ⇨ Betrachte $R = A^* \setminus A_{i-1}$

Beobachtung:

⇨ R ist eine optimale Lösung für die Menge der Aktivitäten in $[a_i,\ldots,a_n\}$ die zu den Aktivitäten in A_{i-1} kompatibel sind.

Fall 1: $b_i < e_{last}$



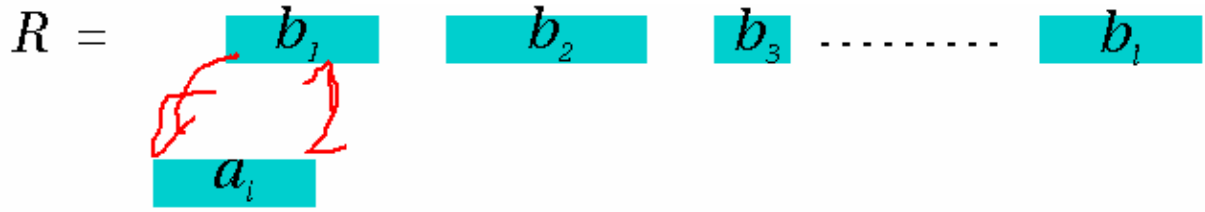
⇨ a_i ist nicht kompatibel zu a_{i-1}
⇨ a_i ist nicht in R enthalten und auch nicht in A*
⇨ $A^* \cap \{a_1,\ldots,a_i\} = A_{i-1} = A_i$

Fall 2: $b_i \geq e_{last}$

⇨ a_i ist kompatibel zu a_{i-1}

Es gilt:

$R \subseteq \{a_1,\ldots,a_n\}$



⇨ $B^* = A_{i-1} \cup (R \setminus \{b_1\}) \cup \{a_i\}$ ist optimal

$B^* \cap \{a_1,\ldots,a_i\} = A_{i-1} \cup \{a_i\} = A_i$

$A^* \cap \{a_1,\ldots,a_n) = A^* = A_n$

Wann ist ein Greedy Verfahren Optimal ?

Dazu muss man folgende zwei Dinge zeigen:

1. Greedy Wahl Eigenschaft
   - Wann man eine optimale Teillösung hat, und man trifft eine lokal optimale Wahl, dann gibt es eine global optimale Lösung, die diese Wahl enthält
   - G* ist Obermenge von T* U {l-opt}

2. Optimalität von Teillösungen:
   - Eine Teillösung einer optimalen Lösung ist eine optimale Lösung des Teilproblems
   - T* U {l-opt} = T_2*
   - G* ist Obermenge von T_2* U {l-opt}
   - Usw.

⇨ Nach jeder lokal optimalen Wahl erhalten wir ein zur Ausgangssituation analoges Problem

Binäre Zeichen Codes

ASCII Code, UNICODE

Jeder Buchstabe einer Datei wird durch eine eindeutig bestimmte Bitsequenz (Codewort) repräsentiert.

Codeworte können
a. feste
b. variable
Länge haben

Codes variabler und fester Länge

Beispiel: Datei D mit 100 000 Zeichen aus $\{a, b, c, d, e, f\}$

Zeichen	a	b	c	d	e	f
Häufigkeit (in 1000)	45	13	12	16	9	5
Codewort						
fester Län.	000	001	010	011	100	101
variab. Län.	0	101	100	111	1101	1100

a. feste Länge: |Kodifikat (D)| = 300.000 Bits
b. variable Länge: |Kodifikat (D)| = (45*1 + (13+12+16)*3 + (9+5)*4)*1000 =224.000 Bits

Präfix Codes

Entschlüsselung
a. Code fester Länge: Pro Gruppe von Bits ein Zeichen
b. Code variable Länge: Präfix Codes

Präfix-Code:
→ Kein Codewiort ist Präfix eines anderen Codeworts

Dekodierungsbeispiel:

Der obige Code variabler Länge ist ein Präfix-Code

Entschlüsselung von 0 0 101 1101… = aabe…
a a b e

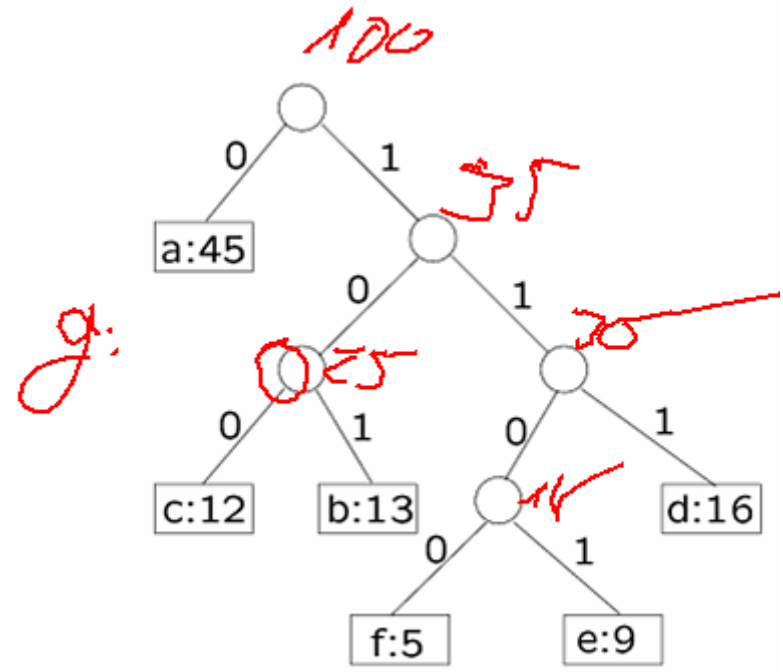
Man muss immer von Anfang an entschlüsseln. Man darf nicht mittendrin anfangen, da sonst falsche Wörter gelesene werden können.

Darstellung von Präfix-Codes

Binärbaume:

Trie von Retrieval

Beispiel:



Code Bäume

- ⇨ Jedes Blatt repräsentiert einen Buchstaben (mit Häufigkeit)
- ⇨ Jeder innere Knoten v enthält das Gewicht (Summe der Häufigkeiten) der Blätter des Teilbaumes mit Wurzel v
- ⇨ Jede Kante erhält ein Label € {0,1}
- ⇨ Jeder Weg von der Wurzel zu einem Blatt repräsentiert ein Codewort

Optimalität von Präfixcodes

Optimaler Präfixcode C (für eine Datei D):
Kodierung von D mit C hat minimale Gesamtlänge

Lemma:
Der einem optimalen Präfixcode für eine Datei entsprechende Code-Baum ist ein vollständiger Binärbaum, d.h. ein Binärbaum für den jeder innere Knoten zwei Söhne hat.

Besteht die Datei D aus den Buchstaben von A, so hat der Codebaum eines optimalen Präfixcodes für die Elemente von A genau

- ⇨ |A| Blätter
- ⇨ |A| -1 innere Knoten

Kosten eines Code-Baumes

Es sei

T		Code-Baum eines Präfix Codes
f(c)	=	Häufigkeit des Buchstabens c € A in der Datei D
$d_T(c)$	=	Tiefe des c repräsentierenden Blattes in T
	=	Länge des Codeworts für c

Dann ist:

$$B(T) = \sum_{c \in A} f(c) \cdot d_T(c) = \text{Kosten von } T$$

= Anzahl der Bits, um die Datei D durch den mit T beschriebenen Code zu kodieren

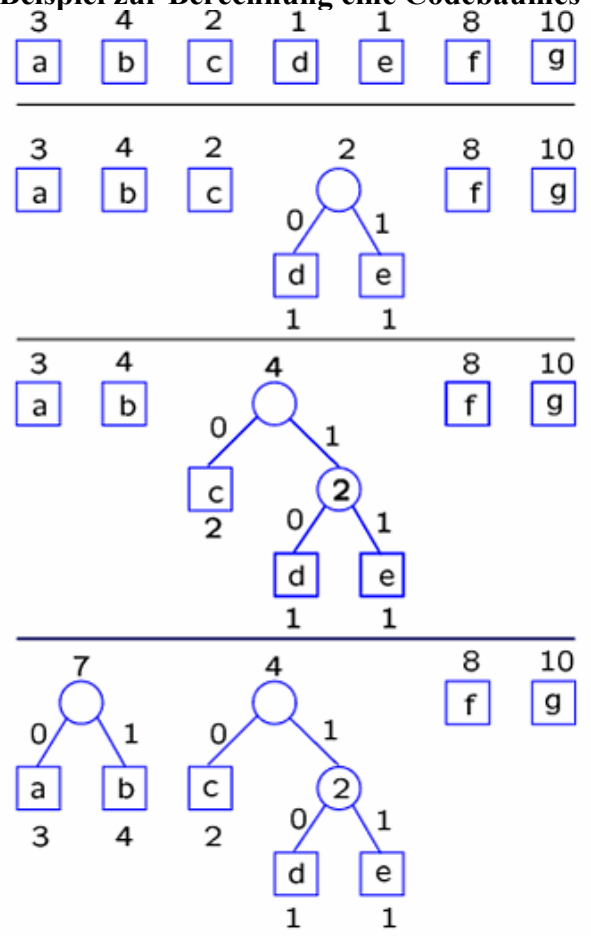
Huffman Codes

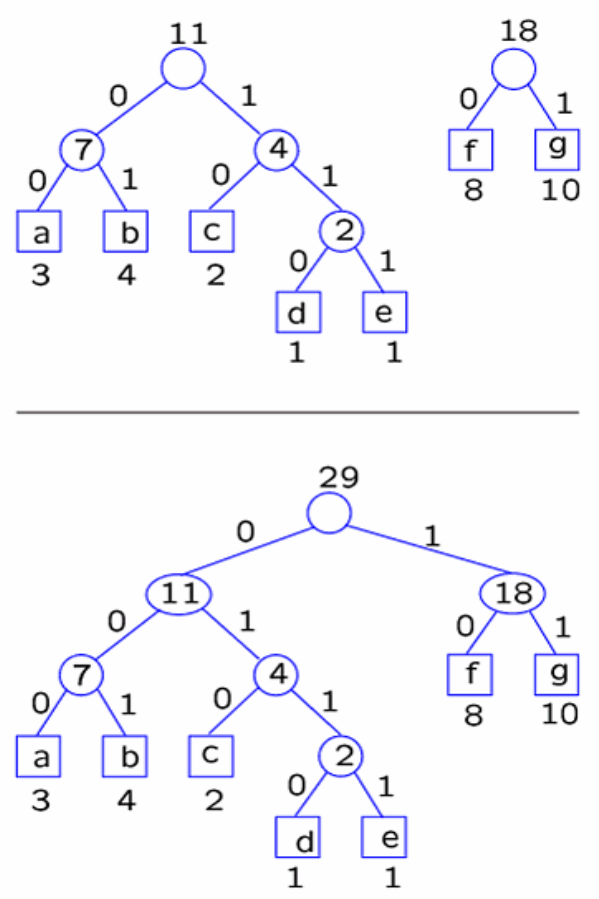
Vorschlag zur Konstruktion eines Code Baumes

Algorithmus *Huffman-Code*
Input: Eine Menge von Zeichen A mit Häufigkeiten $c.f$ für $c \in A$;
Output: Der Huffman Code von A

```
1  n = |A|
2  bilde die Priority Queue Q von Knoten, die mit
     den Zeichen von A initialisiert sind und den
     Häufigkeiten c.f als Prioritätsordnung
3  for i = 1 to n − 1 do
4     schaffe neuen Knoten z mit Söhnen z.right
        und z.left
5     x = Q.delete-min(); z.left = x
6     y = Q.delete-min(); z.right = y
7     z.f = x.f + y.f
8     Q.insert(z)
9  return Q.delete-min()
```

Beispiel zur Berechnung eine Codebaumes





Bemerkungen:

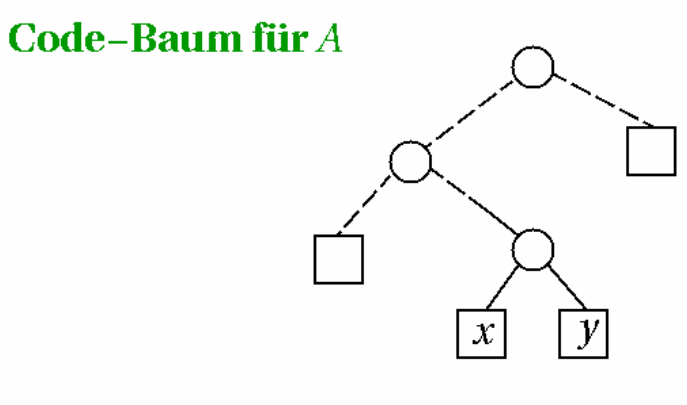
- ⇨ Zur Konstruktion des Huffman Codes sind in der Regel zwei durchgänge durch den Text erforderlich
  - Ermittelung der Häufigkeiten der Buchstaben
  - Bau des Code-Baumes und Verschlüsselung

- ⇨ Decodierung verlangt Austausch des Code-Baumes (bei Lempel-Ziv entfällt dies)
- ⇨ Entschlüsselung muss stets am Textanfang beginnen

Korrektheit des Huffman Verfahrens

Lemma 1:
Sei A ein Alphabet, so dass jeder Buchstabe c aus A die Häufigkeit f(c) hat. Seien x und y zwie Buchstaben aus A mit minimaler Häufigkeit. Dann gibt es einen optimalen Präfixcode , in dem die Codewörter für x und y beide maximale Länge haben und sich nur im letzten Bit unterscheiden.

Die vom Huffman Verfahren getroffene Erste Wahl ist korrekt

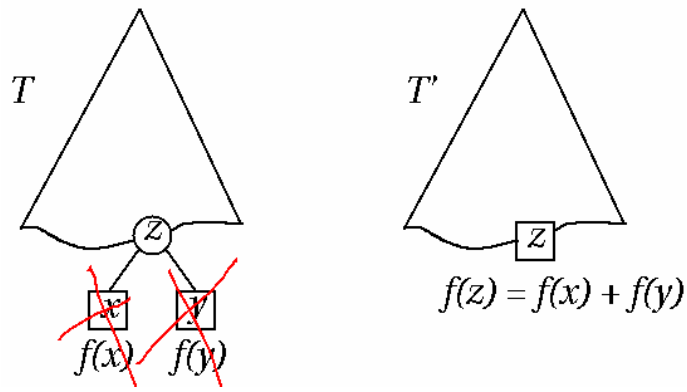


Lemma 2:
Sei T ein vollständiger Binärbaum der einen optimalen Präfixcode für ein Alphabet A repräsentiert. Jeder Buchstabe c aus A trete mit Häufigkeit f(c) auf. Seien x und y zwei benachbarte Blätter in T und z ihr Vater.

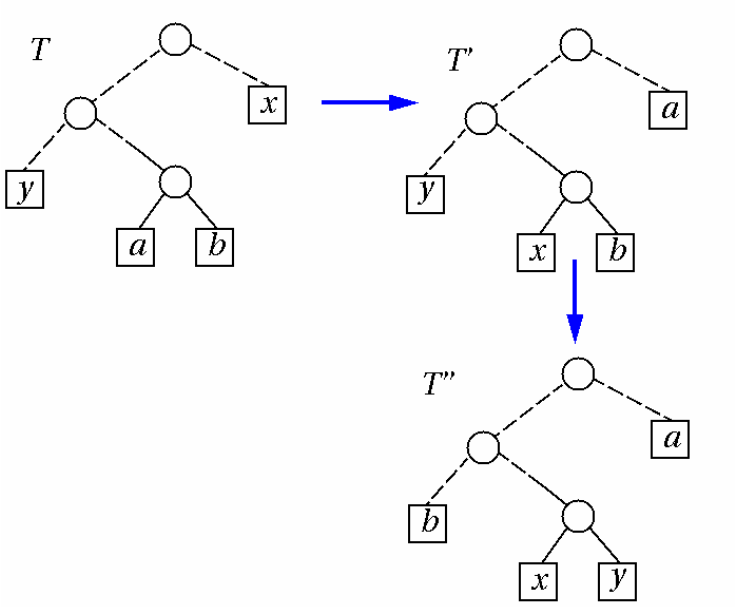
Fasst man z als neuen Buchstaben mit Häufigkeit f(z)=f(x)+f(y) auf, so ist der Baum T' = T\{x,y} ein optimaler Code-Baum für das Alphabet

$$A' = (A - \{x, y\}) \cup \{z\}$$

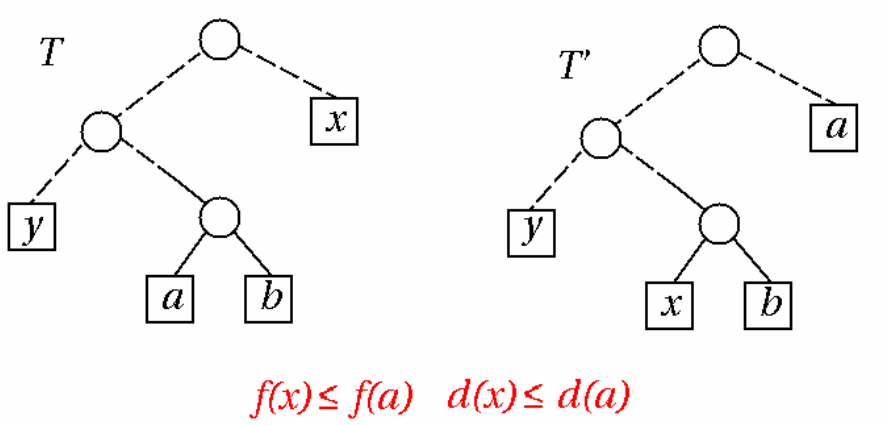
Die weiteren Zusammenfassungen im Huffman Verfahren sind korrekt.



Beweis von Lemma 1:
T sei ein optimaler Code-Baum von A
a,b seien zwei benachbarte Blätter maximaler Tiefe
x,y seien zwei Buchstaben mit minimaler Häufigkeit

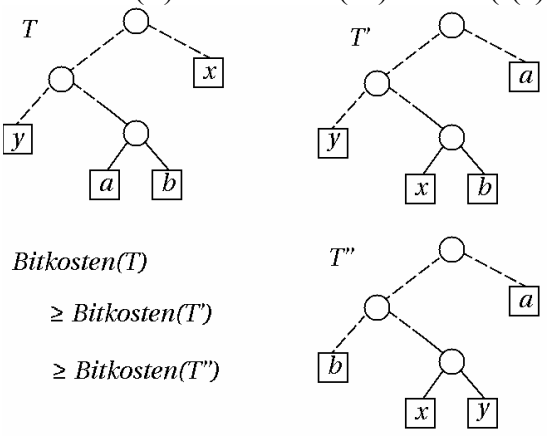


Wir tauschen x gegen a und y gegen b aus, und betrachten die Bitkosten des ursprünglichen Baumes und des Neuen

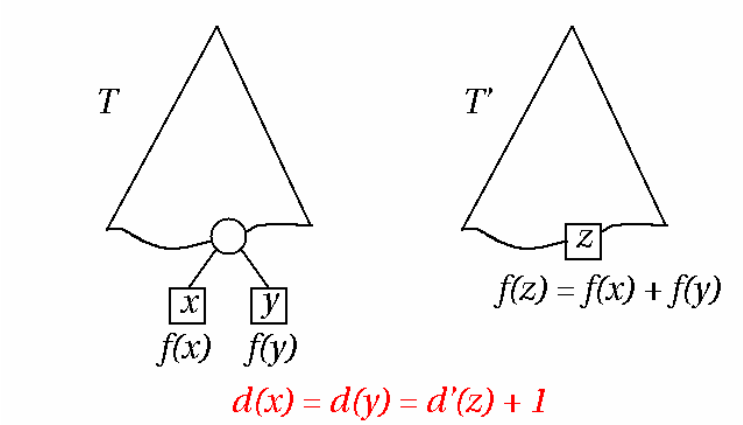


Es genügt nur einen Fall zu betrachten, der andere geht analog

Bitkosten (T) – Bitkosten (T') = … = (f(a)-f(x)) * (d(a)-d(x)) ≥ 0



Beweis von Lemma 2:



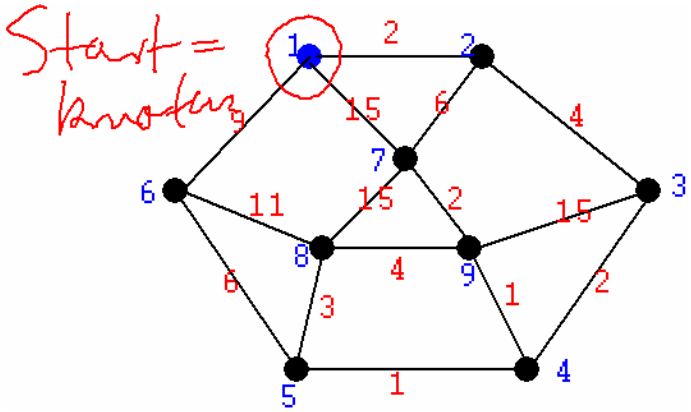
Bitkosten (T) – Bitkosten (T') = … = f(x)+f(y)

Vorlesung 12 – Dijkstra's kürzeste Wege

Kürzesteste Wege in Distanzgraphen

Gegeben: Graph G(V,E)
c: E→R_+, Kantenlänge
Startknoten s aus V

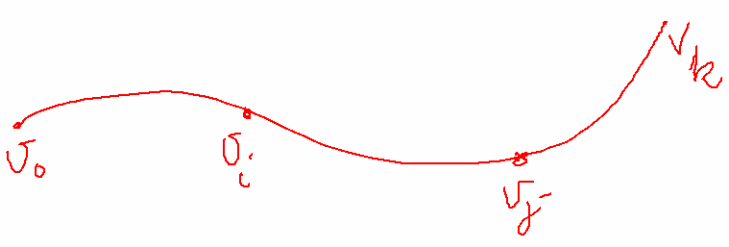
Gesucht: Zu jedem Knoten v aus V ein kürzester Weg von s nach v in G



Shortest Path: Sp(1,3)=[1,2,3]
Kosten: c(sp(1,8))=12

Optimalitätsprinzip

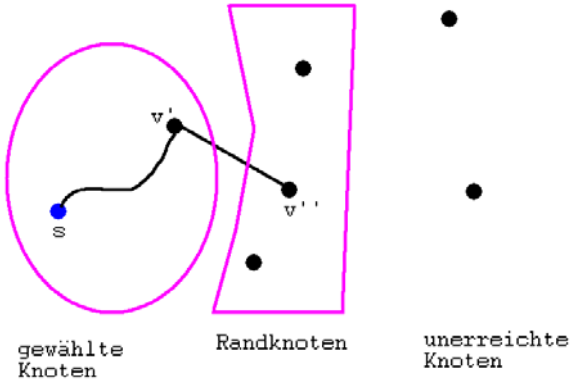
Sei P=[v_0,v_1,…,v_k] ein kürzester Weg von v_0 nach v_k. Dann ist jeder Teilweg P'=[$v_{i,...,}v_j$] mit $0 \leq i < j \leq k$ ein kürzester Weh von v_i nach v_j.



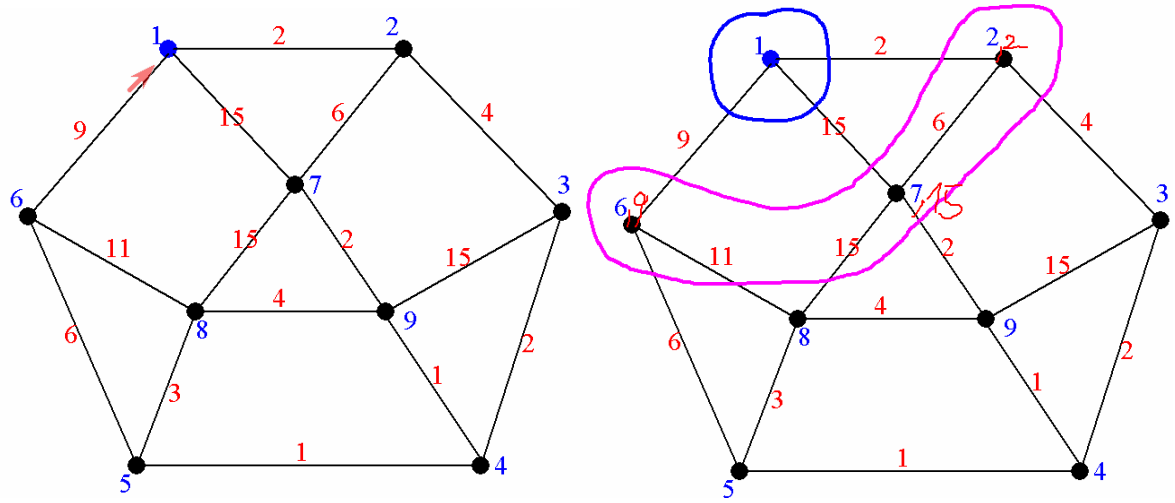
Fortsetzung kürzeste Wege:

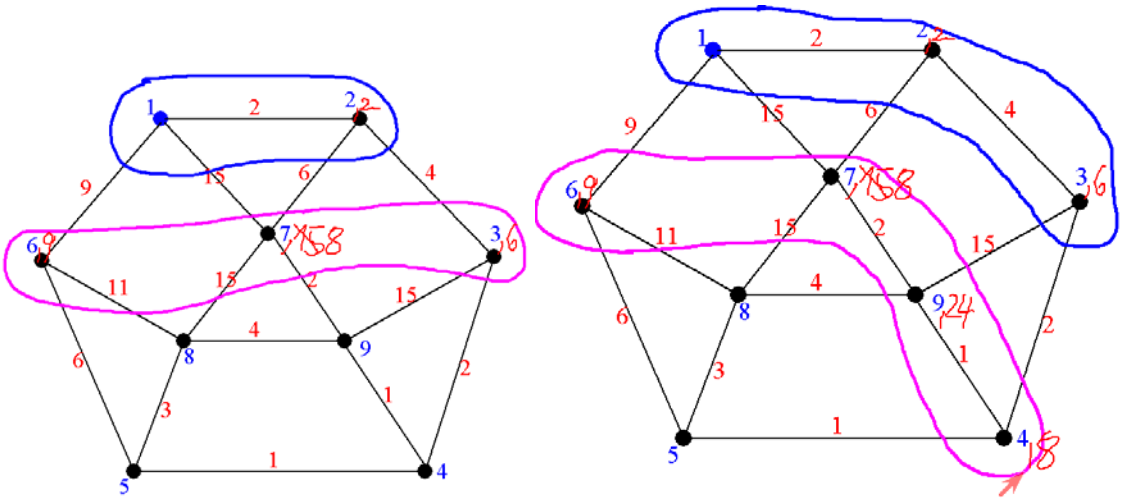
1. Für alle kürzesten Wege sp(s,v) und Kanten (v,v') gilt:
   $c(sp(s,v))+c(sp(v,v'))\geq c(sp(s,v'))$
2. Greedy Methode zur Berechnung kürzester Wege: Erzeuge die kürzesten Pfade von s zu den übrigen Knoten so, dass die Längen der Pfade nicht abnehmen. Ist S eine Menge von Knoten, für die man die kürzesten Pfade von s aus schon kennt, so muss der nächst längere Pfad von s zu einem Knoten u, der nicht zu S gehört, ausschließlich über Knoten aus S führen.

Betrachten wir nun den Algorithmus von Dijkstra (1959)

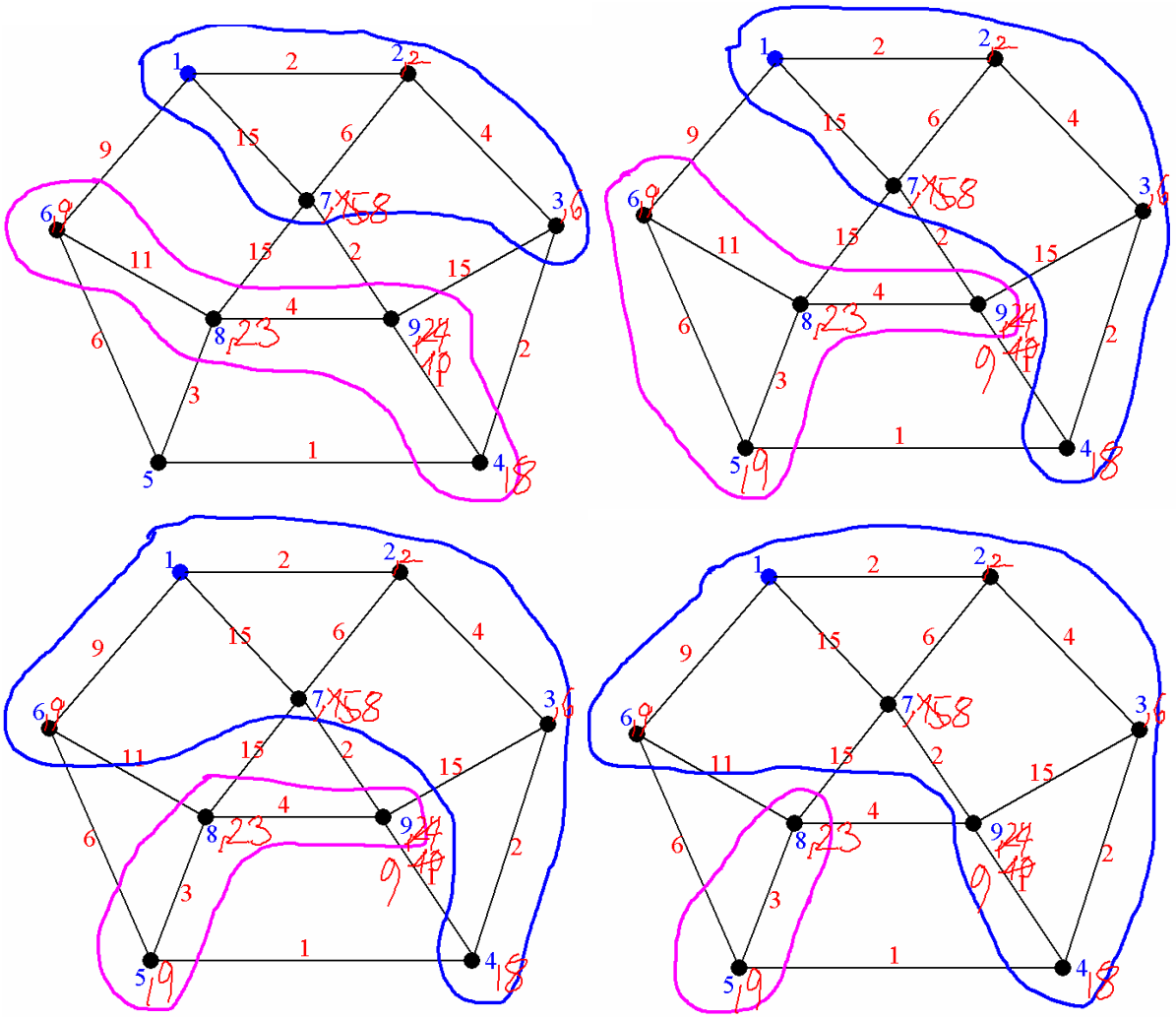


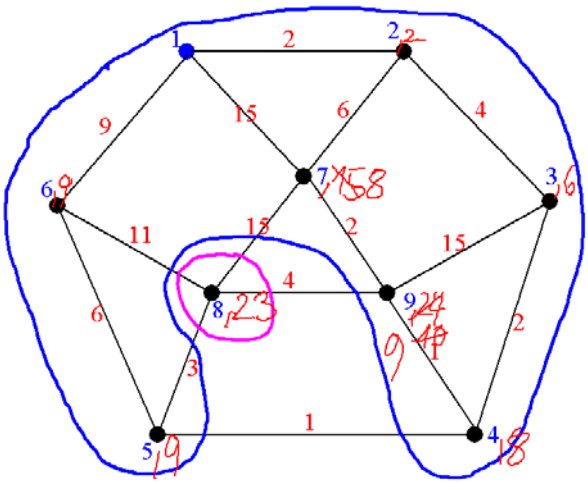
Gewählte Knoten: Der kürzeste Weg von s nach v' ist schon bekannt
Randknoten: Weg von v' nach v'' bekannt (aber nicht unbedingt kürzester)





Die 24 als Entfernung Bei Knoten 9 ist falsch → Richtig ist 21





Dijsktra's Algorithmus als Code

Input: Graph G(V,E)
c: E→R_+, Kantenlänge
Startknoten s aus V

Output: Zu jedem Knoten v aus V ein kürzester Weg von s nach v in G

Inititalisierung:

```
/* Anfangs sind alle Knoten außer s unerreicht */
for all  v ∈ V – {s} do
    { v.Vorgänger = undefiniert ;
      v.Entfernung = ∞ ;
      v.gewählt = false ; }
s.Vorgänger = s ;
s.Entfernung = 0 ;
s.gewählt = true ;
/* Alle zu s adjazenten Kanten gehören zum Rand */
R = Ø ;
ergänze R bei s ;
```

Berechne Wege ab s:

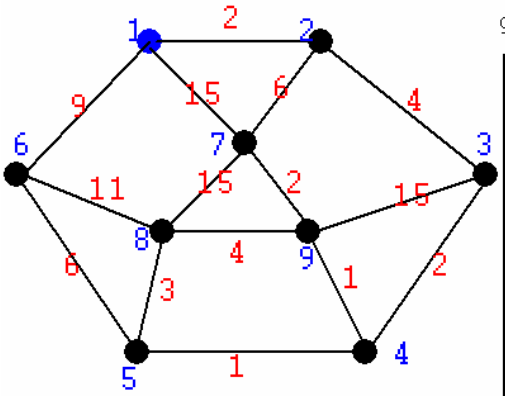
```
while R ≠ Ø do
  /* Wähle zu s nächstgelegenen Randknoten */
  { wähle v aus R mit v.Entfernung minimal und
    entferne v aus R ;
    v.gewählt = true ;
    ergänze R bei v }
```

Ergänze R bei v:

```
for all (v, v') aus E do
    if not v'.gewählt = true and
       (v.Entfernung + c((v,v')) < v'.Entfernung) then
        { v'.Vorgänger = v ;
          v'.Entfernung = v.Entfernung + c((v,v')) ;
          vermerke v' in R ; }
```

decrease

Ein Beispiel komplett durchgerechnet:



Knoten:

(Nr., Entfernung, Vorgänger)

gewählt	Randknoten
(1,0,1)	(2,2,1), (6,9,1), (7,15,1)
(2,2,1)	(6,9,1), (7,8,2), (3,6,2)
(3,6,2)	(6,9,1), (7,8,2), (4,8,3), (9,21,3)
(7,8,2)	(6,9,1), (4,8,3), (9,10,7), (8,23,7)
(4,8,3)	(6,9,1), (8,23,7), (9,9,4), (5,9,4)
(6,9,1)	(9,9,4), (5,9,4), (8,20,6)
(9,9,4)	(5,9,4), (8,13,9)
(5,9,4)	(8,12,5)
(8,12,5)	

Operationen auf R:

(1) Initialisieren: R = ∅

(2) Prüfen, ob R = ∅

(3) wählen und entfernen des Knotens mit minimaler Entfernung

(4) neuen oder geänderten Eintrag im Rand vermerken

Beispiel: Auslesen des kürzesten Pfades vom Startknoten (1) zur 8 (dazu Vorgängerzeigern folgen):

[8,5,4,3,2,1]

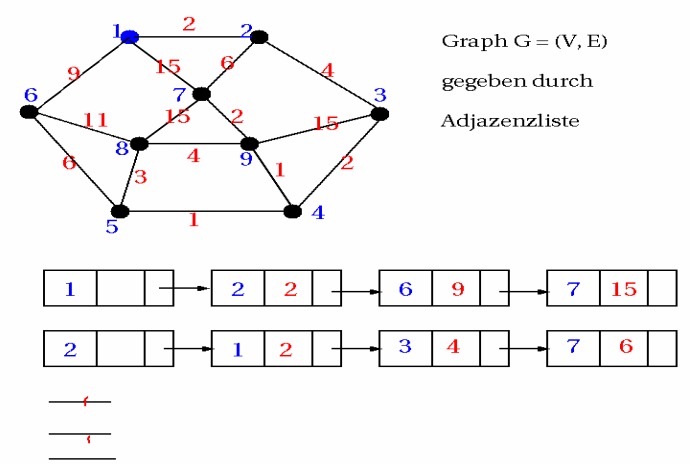
Verwendete Operationen hierbei:

- Einfügen → Hinzufügen eines Knoten in der Randliste

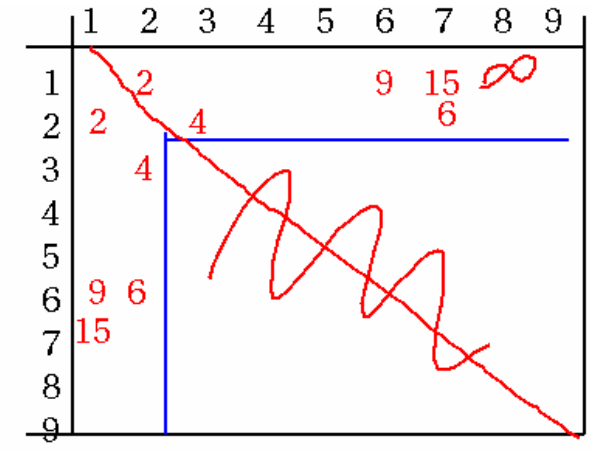
- Min_entfernen → Entfernen des Knotens mit der kürzesten Strecke zu den bereits gewählten Knoten.
- Decreasekey → Wen ein kürzerer Weg gefunden wurde
- Prüfen ob Randliste leer ist → Abbruchbedingung

Darstellungsformen eine Graphen

Adjazenliste→ Zu jedem Knoten die Nachbarn in der Liste



Adjazenzmatrix



d(i,j)= c((i,j))

Zwei verschiedene Implementierungen

1. Version: Keine explizite Speicherung des Randes

Wir speichern ein 4-Tupel: (Knotennr, Entfernung, Vorgänger, gewählt)

1. Initialisiere: R = Ø
2. Prüfen ob R = Ø
   → Für alle Knoten v prüfe ob not v.gewählt=true → O(|V|)

3. Wählen und Entfernen des Knotens mit minimaler Entfernung
   Bestimme unter allen Knoten v mit v.gewählt=false einen Knoten v mit v.Entfernung minimal, setze v.gewählt=true → O(|V|)

4. Neuen oder geänderten Eintrag im Rand R vermerken → $O(|V|^2)$

Gesamtlaufzeit = $O(n^2)$ → Gut für Graphen mit vielen Kanten

2. Version: Verwaltung der Randknoten in einem Fibonacci Heap

Operation auf R:

1. Initialisiere: R = Ø → O(1)
2. Prüfen ob R = Ø → O(1)

3. Wählen und Entfernen des Knotens mit minimaler Entfernung → O(log n)
4. Neuen oder geänderten Eintrag im Rand R vermerken, Decreasekey → O(1)

FibHeap speichert die Knoten aus V mit aktueller Entfernung zu s als Heapordnung.
→ $|Rand| \leq |V| = n$

Sämtlicher Änderungen und Neueinträge im FibHeap sind in Ziet O(|E|) ausführbar.

Gesamtlaufzeit: $O(|E| + |V| \log |V|)$ → gut, wenn Kantenanzahl klein !

Vorlesung 13 – Minimalspannende Bäume (MST)

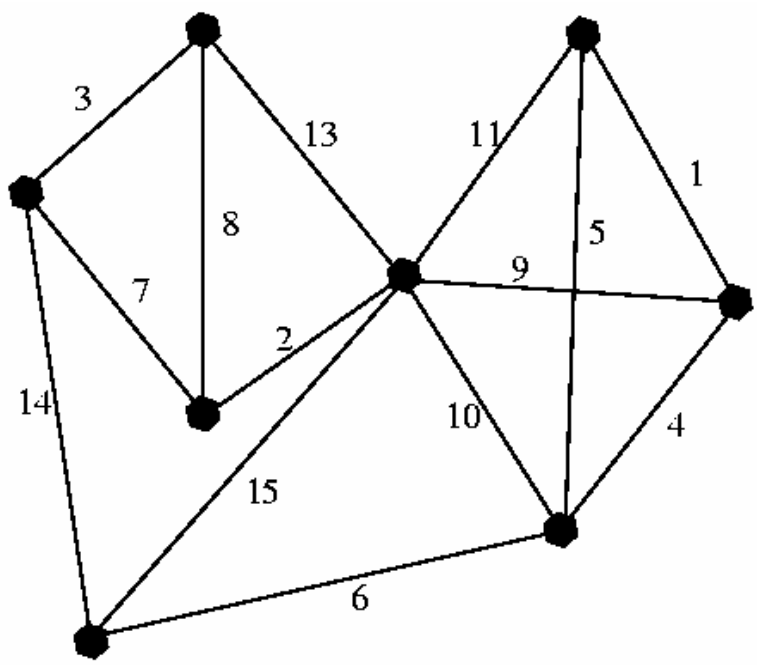
Aufspannende Bäume, minimalen Gewichts in Graphen

Das MST Problem

Gegeben: Zusammenhängender Graph G = (V,E)
V Knotenmenge
$E \subseteq \{\{u,v\} : u,v \in V, u \neq v\}$ Kantenmenge
Gewichtsfunktion c auf E

Gesucht: aufspannender Baum B = (V(B),E(B)) in G mit minimalem Gewicht; also Teilgraph B mit
V(B) = V
$E(B) \subset E$
B ist ein Baum
$\sum_{e \in E(B)} c(e)$ ist minimal

Ein Graph B = (V(B),E(B)) heißt Baum, wenn es zwischen je zwei Knoten aus V(B) genau einen Weg in B gibt.



Färbungsverfahren von Tarjan - Greedy Methode

Kanten werden nacheinander grün oder rot eingefärbt

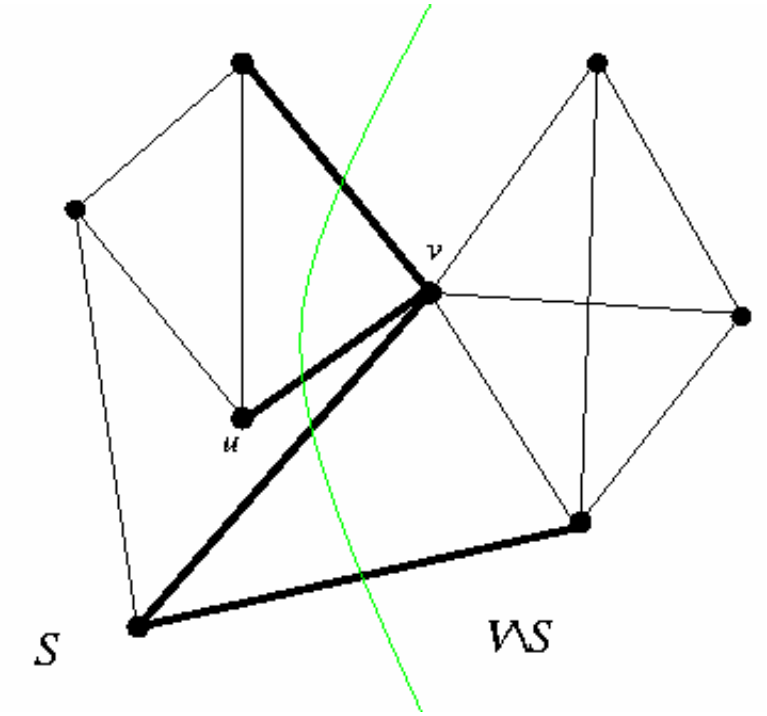
Am Ende bildet die Menge der grünen Kanten einen aufspannenden Baum minimalen Gewichts

Färbungen der Kanten sind Anwendungen von Regeln, einer grünen Regel oder einer roten Regel

Notwendige Begriffe zur Formulierung der Regeln sind:

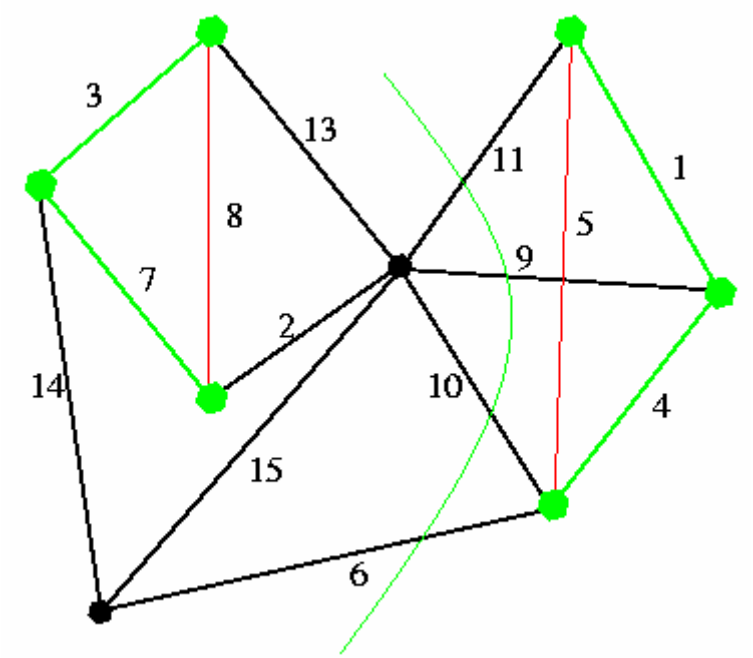
- Schnitt
- Kreis

Ein Schnitt in einem Graph G=(V,E) ist eine Partition (S,V \S) der Knotenmenge V von G.
Eine Kante {u,v} von G kreuzt den Schnitt (S,V \ S), wenn u € S und v € V\S ist.
Oft wird auch die Menge der Kanten aus G, die den Schnitt (S,V\S) kreuzen mit diesem Schnitt identifiziert.

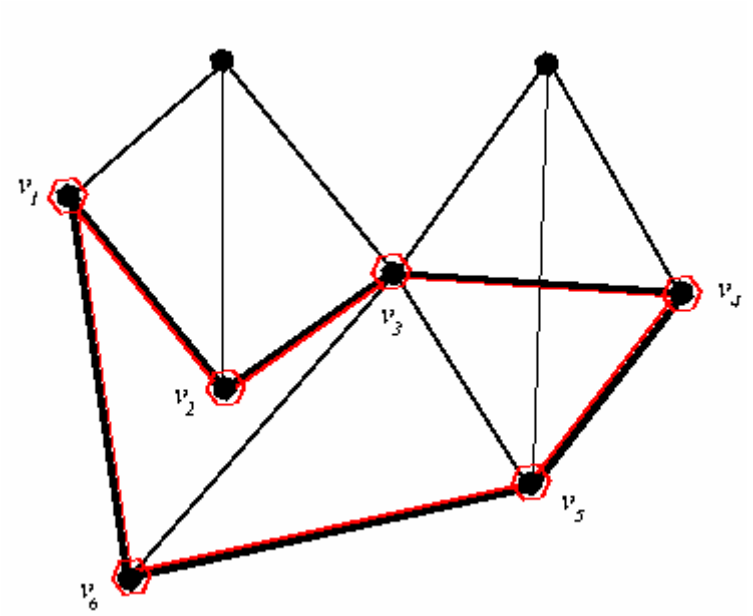


Grüne Regel

Voraussetzung: Wenn ein Schnitt im Graph exisitiert, der von keiner grünen Kante und mindestens einer ungefärbten Kante gekreuzt wird, dann folgt ein
Färbungsschritt: färbe eine ungefärbte Kante minimalen Gewichts, die diesen Schnitt kreuzt, grün.

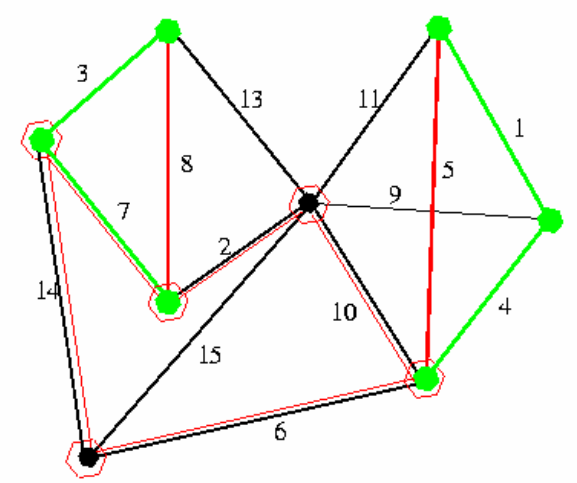


Ein Kreis in einem Graph ist eine Folge $v_1,v_2,\ldots,v_k = v_1$ von Knoten aus G, in der zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind, und kein Knoten ausser dem Anfangs- und Endknoten zweimal auftritt.



Rote Regel

Voraussetzung: Wenn ein Kreis im Graph existiert, der keine rote und mindestens eine ungefärbte Kante enthält, dann folgt ein
Färbungsschritt: färbe eine ungefärbte Kante maximalen Gewichts, die auf diesem Kreis liegt rot.



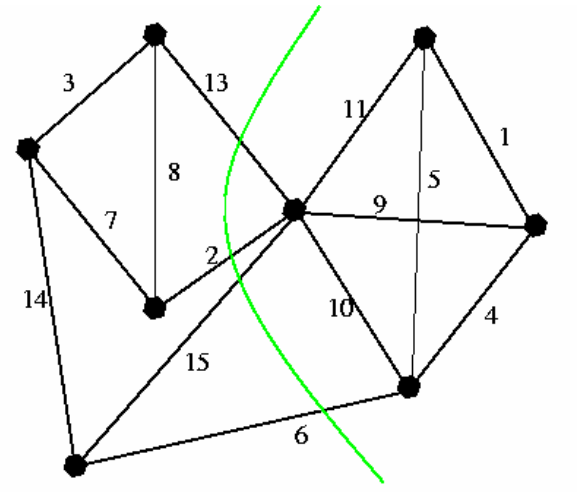
Färbungsalgorithmus im Einzelnen

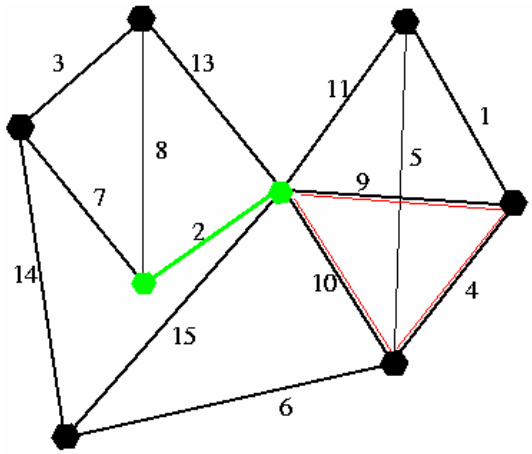
Solange noch eine der beiden Regeln anwendbar ist, wende die grüne oder die rote Regel an

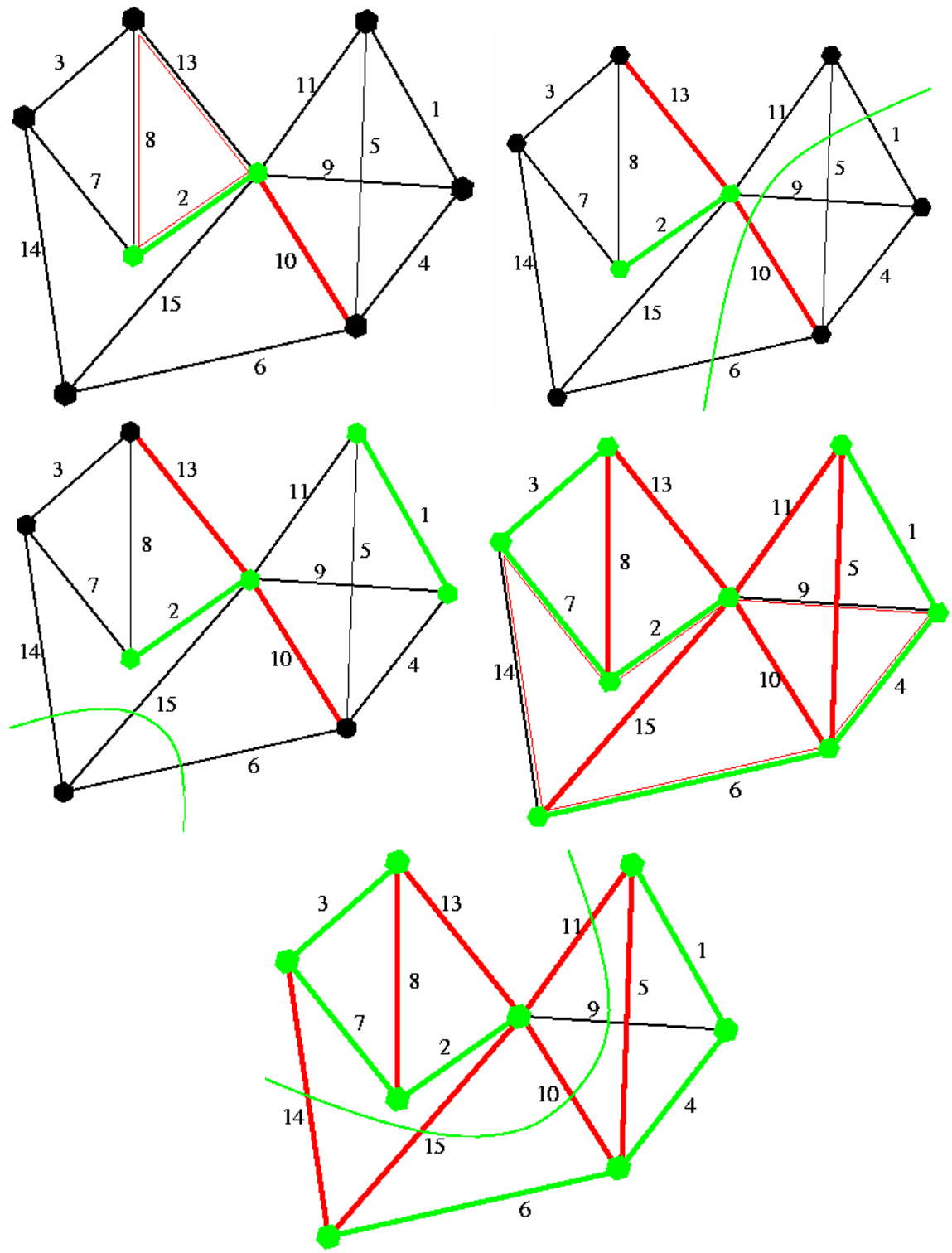
Bemerkung: Algorithmus ist nicht deterministisch!

Behauptung: Am Ende bildet die Menge der grünen Kanten einen Baum minimalen Gewichts. Alle anderen Kanten sind rot gefärbt.

Beobachtung: Der Färbungsalgorithmus kann bereits abgebrochen werden, wenn die grüne Regel nicht mehr anwendbar ist.







Korrektheit des Färbungsalgorithmus

Färbungsinvariante:
Nach jedem Färbungsschritt existiert im Graph ein aufspannender Baum minimalen Gewichts, der alle grünen und keine rote Kante enthält.

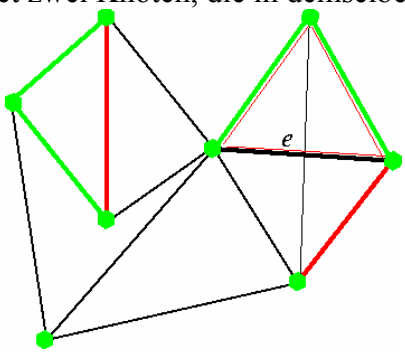
Wir beweisen:
Der Färbungsalgorithmus erhält die Färbungsinvariante

- Bewiesen wird mittels Induktion über die Färbungsschritte → Wenn vor Ausführungeine Färbungsschrittes ein aufspannender Baum minimalen Gewichts

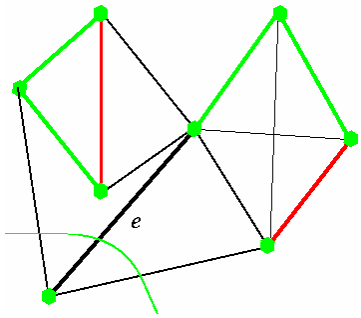
existiert, der alle grünen und keine rote Kante enthält, so auch nach Ausführung eines Färbungsschritts. → Hier würde nun der ausführliche Beweis folgen

Solange noch nicht alle Kanten gefärbt sind, ist noch mindestens eine Regel anwendbar.

- Beweisidee: Betrachte eine ungefärbte Kante e. Da die Färbungsinvariante gilt, bilden die Knoten des Graphen zusammen mit den grünen Kanten eine Menge „grüner Bäume“. Für die Kante e sind zwei Fälle zu unterscheiden:
  - Fall 1: e verbindet zwei Knoten, die in demselben grünen Baum liegen



  - Fall 2: e verbindet zwei Knoten, die in verschiedenen grünen Bäumen liegen



- Beide Fälle wurden in der Vorlesung bewiesen

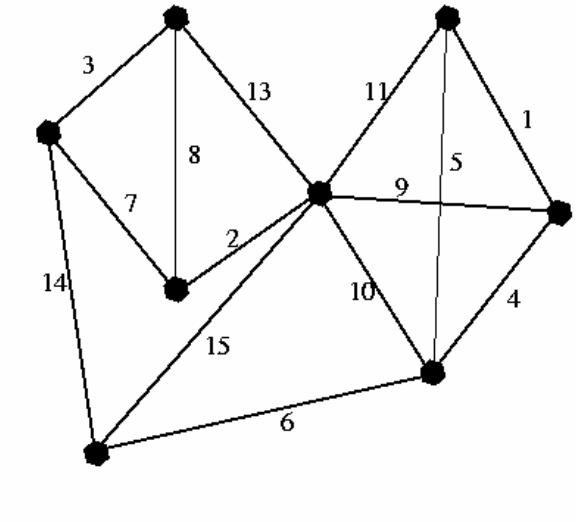
Algorithmus von Kruskal

1. Sortiere die Kanten nach ihrem Gewicht in aufsteigender Reihenfolge
2. Durchlaufe die sortierten Kanten der Reihe nach, und wende folgenden Färbungsschritt an:
   a. Wenn beide Endknoten der Kante in demselben grünen Baum liegen, so färbe sie rot;
   b. Ansonsten färbe sie grün

Der Algorithmus von Kruskal endet, wenn alle Kanten durchlaufen sind. Er ist eine spezielle Version des Färbungsalgorithmus von Tarjan (Die wurde dann noch ausführlich gezeigt, ist aber eigentlich klar).

Für diesen Algorithmus existiert eine Animation:

Animation: Algorithmus von Kruskal



Folge sortierter Kantengewichte:

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Laufzeit des Algorithmus von Kruskal:
Hängt ab von

- Aufwand für sortieren
- Aufwand für Test ob Kante beide Endknoten in demselben grünen Baum hat

Implementation mit Hilfe einer UNION-FIND Struktur

- Kanten werden in einem Preprocessing sortiert
- Oganisation der grünen Bäume wird als Folge von UNION und FIND-Operationen realisiert
    - FIND: Finde die grünen Bäume, in denen die beiden Endknoten der zu färbenden Knate liegen
    - UNION: Vereinige die beiden grünen Bäume, in denen die beiden Endknoten einer Kante liegen, die grün gefärbt wird

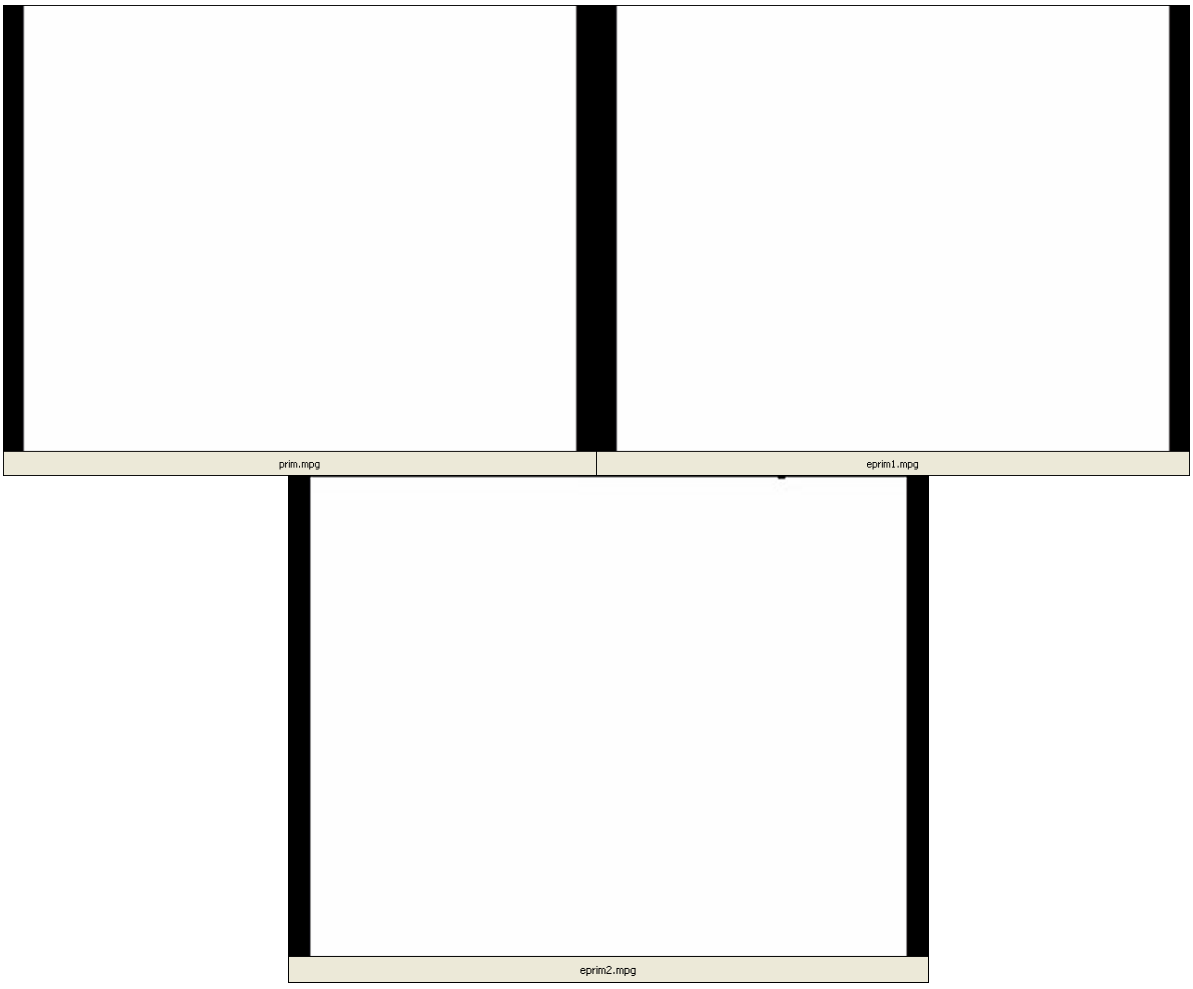
Der Algorithmus von Prim

1. Wähle einen beliebigen Startknoten und betrachte diesen als einen grünen Baum
2. Wiederhole den folgenden Färbungsschritt (|V|-1)mal

a. Wähle eine ungefärbte Kante minimalen Gewichts, die genau einen Endknoten in dem grünen Baum hat, der den Startknoten enthält und färbe sie grün

Auch dieser Algorithmus ist ein spezielle Version des Färbungsalgorithmus von Tarjan. Die wurde ebenfalls ausführlich gezeigt.

Animationen zum Prim Algorithmus



Laufzeit des Algorithmus von Prim
Hängt ab von:

- Aufwand für die Wahl der Kante minimalen Gewichts, die genau einen Endknoten in dem grünen Baum hat.

Implementation mit einem HEAP

- Der HEAP wird zur Organisation derjenigen Kanten benutzt, die genau einen Endknoten in dem aktuellen grünen Baum haben, und daher „Kandidaten“ dafür sind grün gefärbt zu werden.

Greedy Algorithmen und Matroide

Beobachtung:
Wir haben in dem Korrektheitsbeweis zu dem Färbungsalgorithmus mehrfach ein Austauschargument benutzt.

Dieses Austauschargument basiert auf einer interessanten strukturellen Eigenschaft aufspannenden Bäume in Graphen.

Die Mengen von Kanten eines Graphen, welche Bäume induzieren, bilden einen Matroid

Matroide sind Mengensysteme, die eine gewisse Austauscheigenschaft erfüllen.

Ein Mengensystem U über einer endlichen Menge M heisst Unabhängigkeitssystem, wenn

- $\emptyset \in \mathcal{U}$
- $I_1 \in \mathcal{U}, I_2 \subseteq I_1$ induziert $I_2 \in \mathcal{U}$

Ein Unabhängigkeitssystem U ist ein Matroid, falls es folgende „Austauschbedingung“ erfüllt:

Falls $I, J \subseteq M$, $I, J \in \mathcal{U}$ mit $|I| = |J|$, dann existiert ein $e \in I$ und ein $e' \in J$ mit $I \setminus \{e\} \cup \{e'\} \in \mathcal{U}$.

Es gilt nun: Ein Greedy Algorithmus liefert eine optimale Lösung über einem Mengensystem, genau dann, wenn das Mengensystem ein Matroid ist.

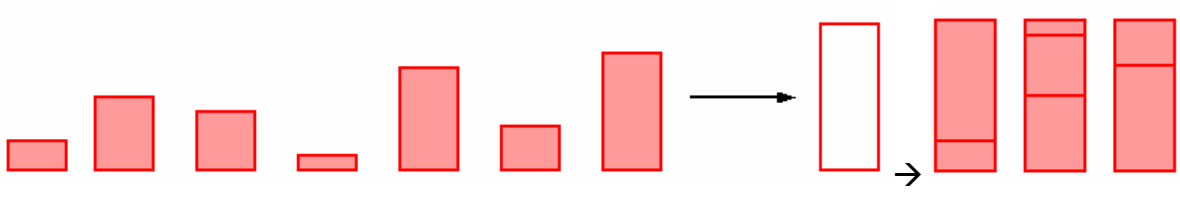
Vorlesung 14 – Bin Packing

Gegeben: n Objekte der Größen $s_1,\ldots,s_n$
Mit $0<s_i\leq 1$, für $1\leq i\leq n$

Gesucht: Die kleinst mögliche Anzahl von Kisten (Bins) der Größe 1, mit der alle Objekte verpackt werden.

Beispiel:

7 Objekte mit den Größen:
0.2., 0.5, 0.4, 0.7, 0.1, 0.3 , 0.8



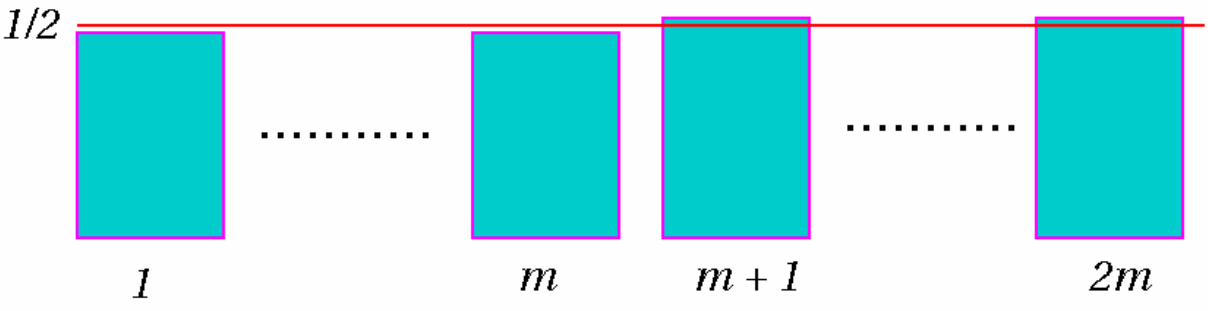
Es gibt verschiedene Klassen von Binpacking Algorithmen:

- Online Bin Packing: Jedes (ankommende) Objekt mus verpackt sein, bevor das nächste Objekt betrachtet wird. Ein Objekt verbleibt in derjenigen Kiste, in die es zuerst gepackt wird

- Offline Bin Packing: Zunächst wird die Anzahl n und alle n Objekte vorgegeben. Dann beginnt die Verpackung.

Beobachtungen:

- Bin Packing ist beweisbar schwer → Offline Version ist NP-schwer. Entscheidungsproblem ist NP-Vollständig)
- Kein Online Bin Packing verfahren kann stets eine optimale Lösung liefern.



- Online Verfahren wissen nie, wann die Eingabe zu Ende ist.

Online Bin Packing Verfahren

Satz (für Online verfahren): Es gibt Eingabefolgen, die jeden Online Bin Packing Algorithmus zwingen, mindestens 4/3 OPT viele Kisten zu verwenden, wenn OPT die minimal mögliche Anzahl an Kisten für die Eingabefolge ist.

Beweisidee: Gegenbeweis führt zum Widerspruch ! (genauen Beweis haben wir weggelassen)

Die verschieden Online Bin Packing Verfahren

Next Fit

- Verpacke das nächste Objekt in dieselbe Kiste, wie das vorherige, wenn es dort noch hineinpasst, ansonsten öffne eine neue Kiste und verpacke es dort
    - Satz:
        - Für alle Eingabefolgen I gilt $NF(I) \leq 2OPT(I)$ → Beweis über eine geschickt gewählt Eingabefolge

First Fit

- Packe das nächste Objekt in die erste Kiste in die es noch hineinpasst, falls es eine solche gibt, ansonsten öffne eine neue Kiste und verpacke es dort.
- Beobachtung: Zu jedem Zeitpunkt kann es höchstens eine Kiste geben, die weniger als Halbvoll ist (Gäbe es zwei könnte man diese ja ineinander schütten.)
- → $FF(I) \leq 2\ OPT(I)$ für alle Eingabefolgen I

Satz:

a. Für alle Eingabefolgen I gilt: $FF(I) \leq 17/10\ OPT(I)$
b. Es gibt Eingabefolgen I mit: $FF(I) \geq 17/10\ (OPT(I)-1) = 1.7$ (schwer zu beweisen)
c. Es gibt Eingabefolgen I mit: $FF(I) \geq 10/6\ OPT(I) = 1.6$ (leicht zu beweisen)

Best Fit

- Verpacke das Objekt in diejenige Kiste, in die es am besten passt (d.h. den geringsten Platz ungenutzt lässt.)
- Verhalten von BF ähnlich zu FF

Laufzeit für Inputfolgen der Länge n

- NF O(n)
- FF $O(n^2)$ → O(n log n)
- BF $O(n^2)$ → O(n log n)

Online Algorithmen Allgemein

Jeder Online Algorithmus benötigt im schlimmsten Fall mindestens 1.54-mal so viele Kisten wie nötig.

Bisheriger Champion: < 1.59

Offline Bin Packing Verfahren

n und $s_1,\ldots,s_n$ sind gegeben, bevor die Verpackung beginnt !

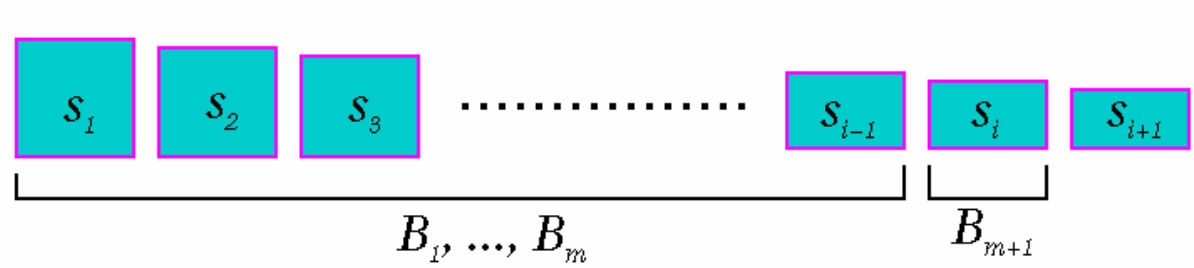
Optimale Packung kann durch erschöpfende Suche bestimmt werden (Laufzeit aber natürlich mies)

Idee für Offline Approximationsalgorithmus:
Sortiere die Objekte zunächst nach abnehmender Größe und verpacke größere Objekte zuerst.

Zwei Typen von Algorithmen

- First Fit Decreasing (FFD), bzw. FFNI
- Best Fit Decreasing (BFD)

First Fit Decreasing



Lemma 1
Sei I eine Folge von n Objekten der Größe

$s_1 \geq s_2 \geq s_3 \geq \ldots \geq s_n$

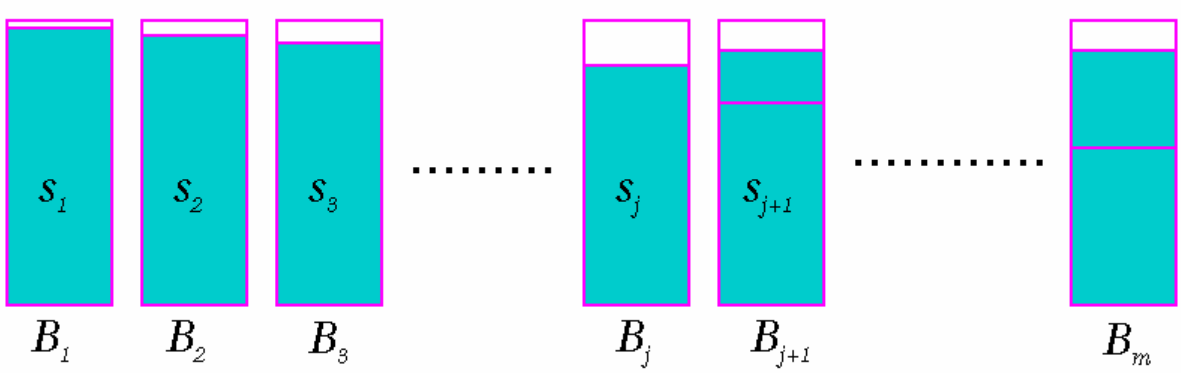
und sei m = OPT(I)

Dann haben alle von FFD in den Kisten

$B_{m+1}, B_{m+2}, \ldots, B_{FFD(I)}$

Verpackten Objekte eine Größe von höchstens 1/3

Pseudoerklärung: Am Anfang gibt es Kisten, in die nur ein Objekt hineinpasst, dann folgen Kisten mit jeweils zwei Objekten. Wenn Kisten folgen mit 3 Objekten, dann ist jedes dieser Objekte kleiner als 1/3 (wegen der Sortierung).



Lemma 2

Sei I eine Folge von n Objekten der Größe

$s_1 \geq s_2 \geq s_3 \geq \ldots \geq s_n$

und sei m = OPT(I)

Dann ist die Anzahl der Objekte, die FFD in den Kisten

$B_{m+1}, B_{m+2}, \ldots, B_{FFD(I)}$

verpackt, höchstens m-1.

Satz

Für alle Eingabefolgen I gilt:

$$FFD(I) \leq 4/3\ OPT(I) + 1 \leq OPT(I) + \lceil (OPT(I)-1)/3 \rceil$$

$OPT(I) + \lceil (OPT(I)-1)/3 \rceil = m + \lceil (m-1)/3 \rceil$

(Die /3 stammen aus Lemma 1, die m-1 aus Lemma 2)

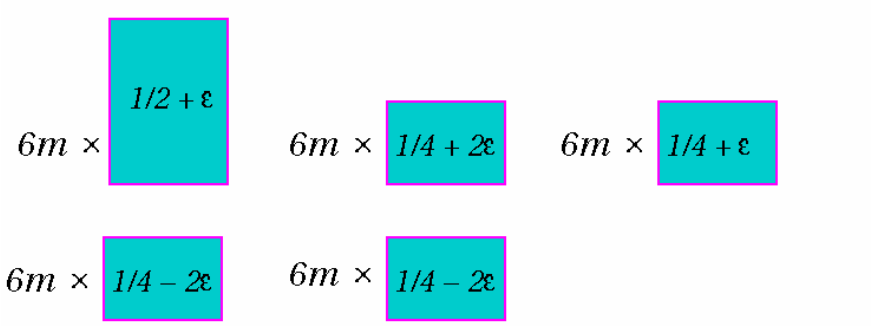
Der Beweis für den Satz ergibt sich also aus den beiden Lemmas

Satz

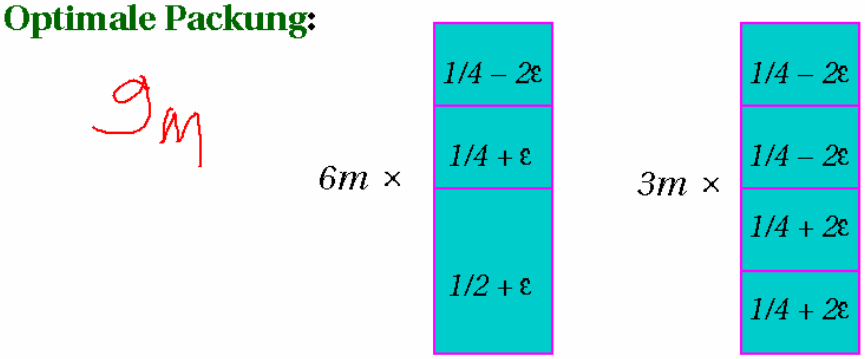
a. Für alle Eingabefolgen I gilt
   i. $FFD(I) \leq 11/9\ OPT(I) + 4$
b. Es gibt Eingabefolgen I mit
   i. $FFD(I) \geq 11/9\ OPT(I)$

$11/9\ OPT(I) + 4 = 1.222\ OPT(I) + 4$ (Obere Schranke für FFD) Bewiesen haben wir 1.333

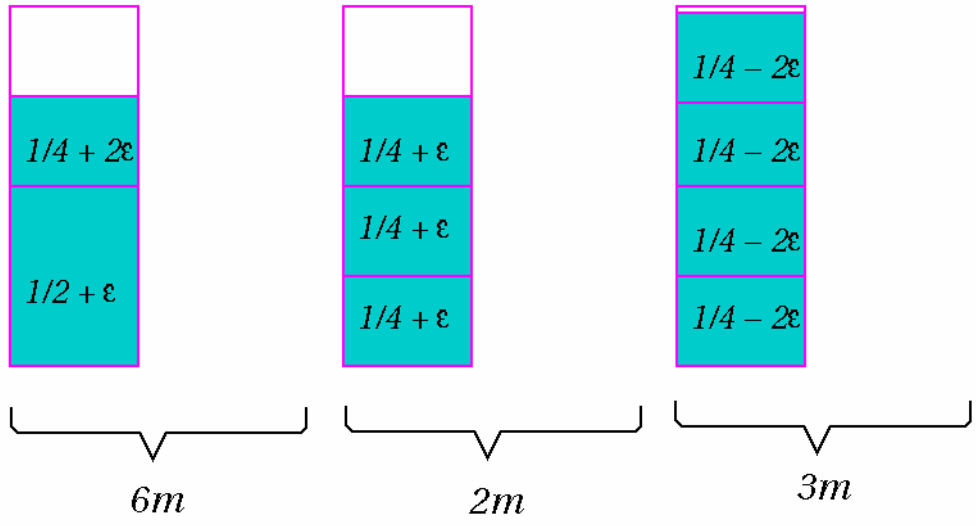
Beweis von b. Betrachte Eingabefolge I der Länge 30m



Optimale Packung:



First Fit Decreasing liefert:



$$OPT(I) = 9m$$

$$FFD(I) = 11m$$

Vorlesung 15 – Online Algorithmen

Online Algorithmen und Kompetitivität

Online Algorithmen sind Verfahren für Probleme, über die nicht alle Informationen a priori vorliegen.

In der Regel versucht man die Lösung des besten Offline Verfahrens zu approximieren.

Modell:
Eingabefolgen $\sigma = \sigma_1\sigma_2\cdots$

Online Algorithmus A:
verarbeitet σ_i bevor σ_{i+1} erscheint

$C_A(\sigma)$: Kosten von A zur Verarbeitung der Eingabefolge σ
$C_{OPT}(\sigma)$: Kosten des optimalen offline Algorithmus OPT

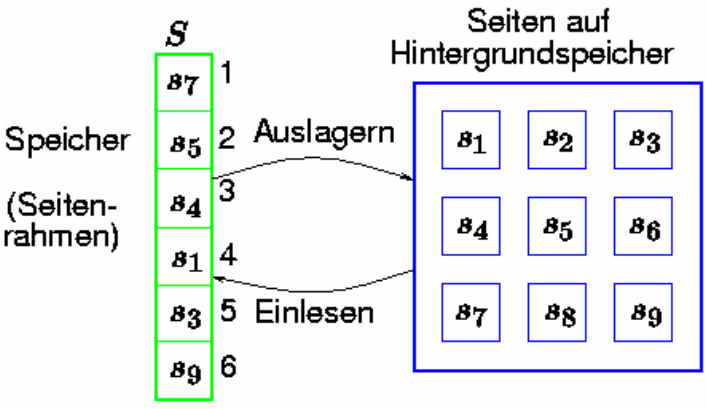
Definition
Ein online Algorithmus A ist c-kompetitiv, falls für alle Eingabefolgen σ gilt:

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a$$

für ein Konstante a.

Seitenaustauschstrategien

Zweistufiges Speichersystem
Einteilung von Speicher in Seiten gleicher Größe (Seiten des Speichers nennt man Seitenrahmen)



Ablauf

Programm P benötigt Daten aus Seite s_i
→ Es gibt zwei Situationen

1. $s_i \in S$: auf Daten kann zugegriffen werden
2. $s_i \notin S$: Seitenfehler
   a. Es gibt noch einen freien Seitenrahmen in S: s_i wird in den freien Rahmen geladen
   b. Der Speicher ist vollständig belegt:

i. Eine Seite s_k wird aus S entfernt
ii. s_i wird in den freigewordenen Seitenrahmen geladen

Seitenfehler: Zugriff auf Hintergrundspeicher
Festplatte: Faktor 10^5 langsamer als interner Speicher

⇨ Verlangsam der Programmausführung

Eingabefolge: $\sigma=\sigma_1\sigma_2\sigma_3\ldots$
σ_t Seitenreferenz, t=1,2,…

$C_A(\sigma)$ = Anzahl der Seitenfehler von A bei Seitenreferenzfolge σ

Online Algorithmus:
⇨ Entscheidet bei einem Seitenfehler zum Zeitpunkt t allein aufgrund von $\sigma_1\ldots\sigma_t$, welche Seite entfernt werden soll.

Mögliche Strategien

- Last-In-First-Out (LIFO)
- First-In-Frst-Out (FIFO)
- Least-Recently-Used (LRU)
- Least-Frequently-Used
- Flush-When-Ready (FWF)
- …

Least Recently Used

Ersetze diejenige Seite, die am längsten nicht benötigt wurde

Speicherzustand S_t: Sortierte Folge $(s_1,\ldots,s_k)$

$\sigma_t = s_i \in S_t$
⇨ $S_{t+1} = (s_i,s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_k)$
$\sigma_t = s_0 \notin S_t$
⇨ $S_{t+1} = (s_0,s_1,\ldots,s_{k-1})$

Beispiel :

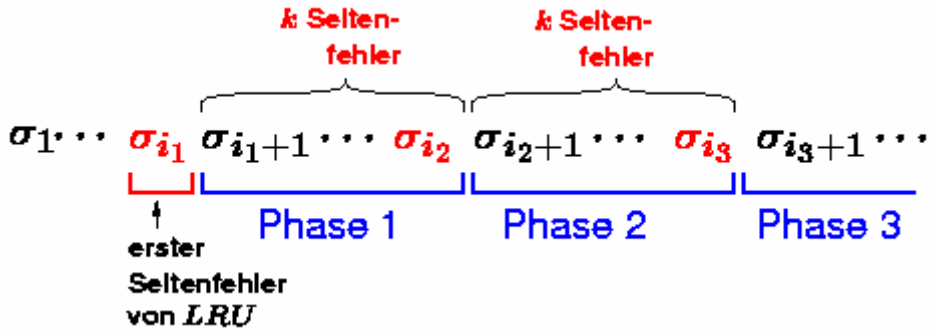
σ		4	1	2	3	2
S_{t+1}	1	4	1	2	3	2
	2	1	4	1	2	3
	3	2	2	4	1	1

(Speicher Spaltenweise)

Satz: LRU ist k-kompetitiv → d.h.

$$C_{LRU}(\sigma) \leq kC_{OPT}(\sigma)$$

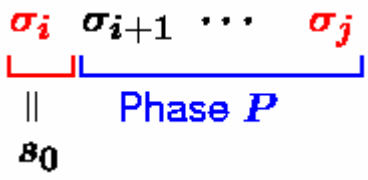
Beweis:
Phase 0 endet it dem ersten Seitenfehler von LRU
Phase i beginnt nach Phase i-1 und endet mit dem k-ten Seitenfehler von LRU in Phase i.



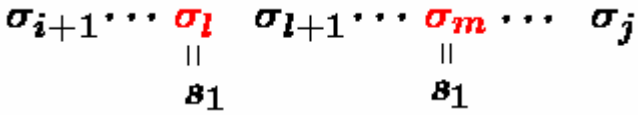
Behauptung: OPT produziert mindestens eine Seitenfehler in jeder Phase P

→ Es gibt 3 Fälle:

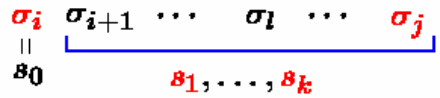
- Fall 1: Es gibt einen Seitenfehler in P bei Zugriff auf s_0



  - → OPT Produziert mindestens eine Seitenfehler in P
- Fall 2: Es gibt eine Seite s_1 in P, die zweimal einen Seitenfehler erzeugt



  -
- Fall 3: Alle Seiten in P sind paarweise verschieden und verschieden von s_0



$$S_{i+1}^{opt} = (s_0, s_1' \ldots, s_{k-1}')$$

  - → OPT Produziert mindestens eine Seitenfehler in P

Optimale offline Strategie MIN

Ersetze diejenige Seite, die am längsten nicht benötigt wird. (Greedy Prinzip)

$$\Phi(t, s) = \min\{t' > t | \sigma_{t'} = s\}$$

Entferne Seite s für die Φ(t,s) maximal ist.
σ muss vollständig im Voraus bekannt sein.

Unter Schranken

Satz: Sei A ein online Seitenaustauschalgorithmus, dann gibt es eine beliebig lange Seitenreferenzfolge σ, so dass:

$$C_A(\sigma) \geq kC_{MIN}(\sigma).$$

Randomisierung und Gegenspieler

Folge σ für untere Schranke:

σ_1 = s_{k+1}
σ_{t+1} = diejenige Seite s_{it}, die A zum Zeitpunkt t ausgelagert hat

Ein Gegenspieler wählt die Anfragen σ_t

Zufällige Bestimmung der entfernten Seite
- ⇨ innerer Zustand bleibt verborgen
- ⇨ besseres kompetitives Verhältnis

Es gibt drei Sorten von Gegenspielern:

Unwissender Gegenspieler G: σ wird zu anfang gewählt

$$E[C_A(\sigma)] \leq c \cdot C_{MIN}(\sigma) + a$$

Adaptiver online Gegenspieler G: σ_t hängt von der entfernten Seite ab, σ_t muß on-line von G beantwortet werden

$$E[C_A(\sigma)] \leq c \cdot E[C_G(\sigma)] + a$$

Adaptiver offline Gegenspieler G: σ_t hängt von der entfernten Seite ab

$$E[C_A(\sigma)] \leq c \cdot E[C_{MIN}(\sigma)] + a$$

Algorithmus Marking (fast gleich wie FWF nur randomisiert)

Algorithmus *Marking*
Input: Eine Seitenreferenz σ_i
Output: Eine ausgelagerte Seite
1 **if** $\sigma_i \notin S$ **then**
2 **if** S ist nicht voll
3 **then** lade σ_i in einen freien Seitenrahmen
4 **else if** alle Seiten sind markiert
5 **then** entferne alle Markierungen
6 wähle eine zufällige unmarkierte Seite s_j (gleichverteilt)
7 entferne s_j und lade σ_i
8 markiere σ_i

Kompetitivität von Marking

Satz: Marking ist $2H_k$-kompetitiv gegen jeden unwissenden Gegenspieler

H_k ist die k-te harmonische Zahl

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$$

$$(H_k \leq 1 + \ln k)$$

Analyse von Marking
Zu Zeigen: Für alle Seitenreferenzfolgen σ

$$E[C_M(\sigma)] \leq 2H_k C_{MIN}(\sigma)$$

(Sehr umfangreicher Beweis von ca. 15 Seiten fehlt)

Satz: Sei A ein ***randomisierter*** online Seitenaustauschalgorithmus, dann gibt es eine beliebig lange Seitenreferenzfolge σ, so dass:

$$C_A(\sigma) \geq H_k C_{MIN}(\sigma).$$

Andere Bereiche für Online Algorithmen

- Bin Packing
- Load Balancing (z.B. Maschinenauslastung Balancieren)
- K-Server Problem (Verallgemeinerung des Seitenaustauschproblems)
- Geometrische Online Suche(z.B. Autonome Roboter die sich in einem unbekannten Raum orientieren müssen)
- …

Vorlesung 16 – Dynamische Programmierung

Was ist dynamische Programmierung ?

Rekursiver Ansatz:

- ⇨ Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt.

Phänomen:

- ⇨ Mehrfachberechnungen von Lösungen

Methode:

- ⇨ Speichern einmal berechneter Lösungen in einer **Tabelle** für spätere Zugriffe

Beispiel: Fibonacci-Zahlen

f(0) = 0
f(1) = 1
f(n) = f(n-1)+f(n-2), falls n ≥ 2

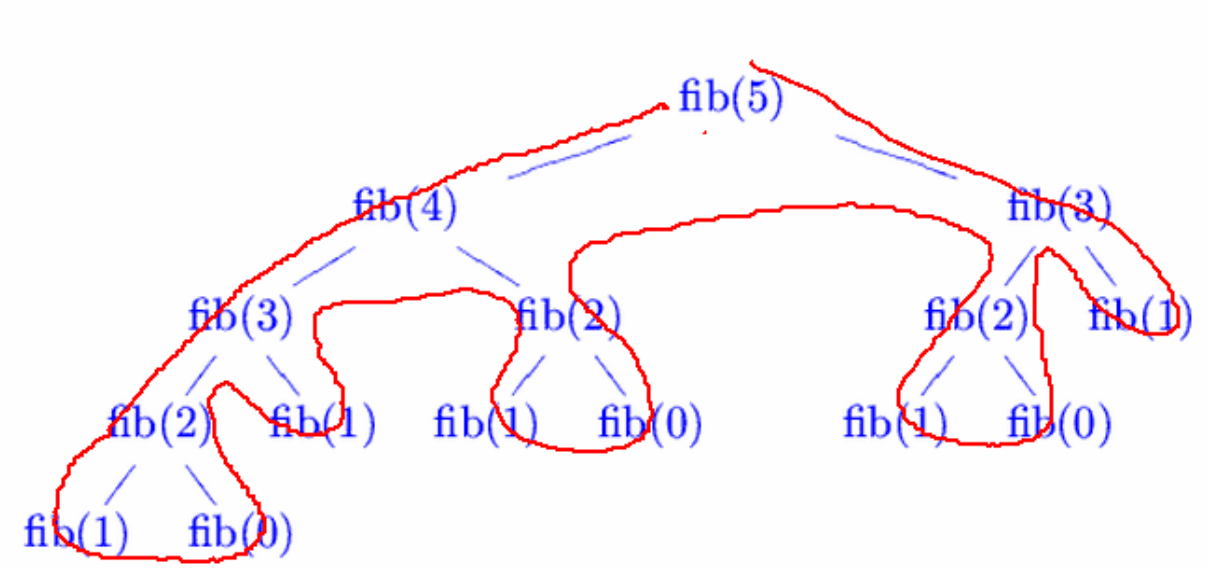
Es gilt: $f(n) = \left[\frac{1}{\sqrt{5}}(1.618\ldots)^n\right]$ (die 1.618… entspricht dem goldenen Schnitt)

Direkte Implementierung der Fibonaccizahlen

(ohne Verbesserungen durch dynamisches Programmieren)

```
procedure fib (n : integer) : integer
if (n == 0) or (n == 1)
   then return n
   else  return fib(n − 1) + fib(n − 2)
```

der Algorithmus spannt den folgenden Aufrufbaum auf



Die rote Linie kennzeichnet die Abarbeitungsreihenfolge (von links nach recht)

Wie man sieht gibt es einige Mehfachberechnungen (z.b. fib(3), fib(2))

Sei T(n) die Anzahl der „fib“-Aufrufe, also die Anzahl der Knoten im Aufrufbaum, bei einem kompletten Durchlauf des Algorithmus, dann gilt

T(n)=T(n-2)+T(n-1)+1 bzw.

$$T(n) \approx \left[\left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{\sqrt{5}+1}{2}\right)^n - 1\right] \approx [1.447 \times 1.618^n - 1]$$

Vorgehensweise beim dynamischen Programmieren:

1. Rekursive Beschreibung des Problems P
2. Bestimmung einer Menge T, die alle Teilprobleme von P enthält, auf die bei der Lösung von P ,auch in tieferen Rekursionsstufen, zurückgegriffen wird.
3. Bestimmung einer Reihenfolge $T_0,\ldots,T_k$ der Probleme in T, so dass bei der Lösung von T_i nur auf Probleme T_j mit $j < i$ zurückgegriffen wird.
4. Sukzessive Berechnung und Speicherung von Lösungen für $T_0,\ldots,T_k$.

Beispiel: Umsetzung bei den Fibonaccizahlen

1. Rekursive Definition der Fibonacci-Zahlen nach gegebener Gleichung
2. T = {f(0),…,f(n-1)}
3. T_i = f(i), i=0,…,n-1
4. Berechnung von fib(i) benötigt von den früheren Problemen nur die zwei letzten Teillösungen fib(i-1) und fib(i-2) für $i \geq 2$

Implementierung mittels dynamischer Programmierung

```
procedure fib (n : integer) : integer
1  f_0 := 0; f_1 := 1
2  for k := 2 to n do
3      f_k := f_{k-1} + f_{k-2}
4  return f_n
```

Laufzeit: O(n)
Platzbedarf: O(n)

```
procedure fib (n : integer) : integer
1  f_vorletzte := 0; f_letzte := 1
2  for k := 2 to n do
3      f_aktuell := f_letzte + f_vorletzte
4      f_vorletzte := f_letzte
5      f_letzte := f_aktuell
6  return (n ≤ 1) ? n : f_aktuell
```

Laufzeit: O(n)
Platzbedarf: O(1)

Memoisierte Version der Berechnung der Fibonacci Zahlen

Berechne jeden Wert genau einmal und speichere ihn in einem Array F[1…n]:

Die folgende Prozedur initialisiert das Array und ruft die Prozedur lookupfib(n) auf

```
procedure fib (n : integer) : integer
1  F[0] := 0; F[1] := 1;
2  for i := 2 to n do
3      F[i] := ∞;
4  return lookupfib(n)
```

lookupfib(n) sieht folgendermaßen aus:

```
procedure lookupfib(k : integer) : integer
1  if F[k] < ∞
2     then return F[k]
3     else  F[k] := lookupfib(k−1)+lookupfib(k−2);
4           return F[k]
```

Platzbedarf: O(n)
(Anmerkung: Im Matheprogramm Maple lässt sich die Memoisierung an bzw. ausschalten.)

Das Optimalitätsprinzip

Typische Anwendung für dynamisches Programmieren: Optimierungsprobleme

Eine optimale Lösung für das Ausgangsproblem setzt sich aus optimalen Lösungen für kleinere Probleme zusammen.

Kettenprodukt von Matrizen

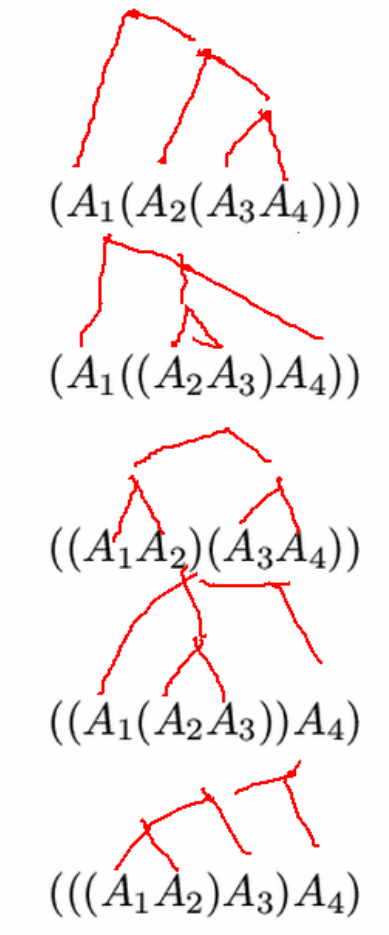
Das Kettenprodukt von Matrizen benötigt man um eine optimale Abarbeitunsreihenfolge der Multipliaktionen zu finden um möglichst wenige skalare Multiplikationen zu haben.

Gegeben: Folge (Kette) $<A_1,A_2,\ldots,A_n>$ von Matrizen
Gesucht: Produkt $(A_1*A_2\ldots..A_n)$

Problem: Organisiere die Multiplikation so, dass möglichst wenige skalare Multiplikationen ausgeführt werden.

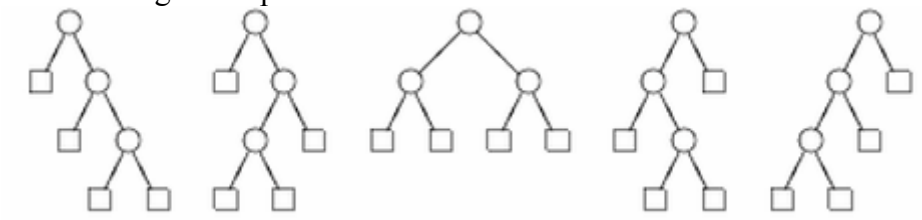
Definition: Ein Matrizenprodukt heißt vollständig geklammert, wenn es entweder eine einzelne Matrix oder das geklammerte Produkt zweier vollständig geklammerter Matrizenprodukte ist.

Alle vollständig geklammerten Matrizenprodukte der Kette $<A_1,A_2,A_3,A_4>$ sind:



Anzahl der veschiedenen Klammerungen

Klammerungen entsprechen strukturell verschiedenen Bäumen:



P(n) sei die # der verschiedenen Klammerungen von:

$(A_1 \ldots A_k)|(A_{k+1} \ldots A_n)$
es gibt n-1 Möglichkeiten den „Schnittpunkt“ k zu setzen um A_1 bis A_n in zwei Teile zu splitten.

$$
\begin{aligned}
P(1) &= 1 \\
P(n) &= \sum_{k=1}^{n-1} P(k)\,P(n-k) \qquad \text{für } n \geq 2 \\
P(n+1) &= \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + \mathrm{O}\left(\frac{4^n}{\sqrt{n^5}}\right) \\
P(n+1) &= C_n = n\text{-te Catalansche Zahl}
\end{aligned}
$$

Bemerkung: finden der optimalen Klammerung durch Ausprobieren ist sinnlos, da es $O(4^n)$ Bäume gibt !

Multiplikation zweier Matrizen

$$A = (a_{ij})_{p\times q},\ B = (b_{ij})_{q\times r},\ A \cdot B = C = (c_{ij})_{p\times r},$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}.$$

(naives Verfahren)

Algorithmus *Matrix-Mult*
Input: Eine $(p \times q)$ Matrix A und eine $(q \times r)$ Matrix B
Output: Die $(p \times r)$ Matrix $C = A \cdot B$

```
1  for i := 1 to p do
2      for j := 1 to r do
3          C[i,j] := 0
4          for k := 1 to q do
5              C[i,j] := C[i,j] + A[i,k] · B[k,j]
```

Anzahl der Multiplikationen und Additionen: p*q*r
Bemerkung: Für zwei n X n Matrizen werden hier n^3 Multiplikationen benötigt. Es geht auch mit $O(n^{2.376})$ Multipliaktionen.
Beispiel: Berechnung des Produktes von A_1, A_2, A_3 mit

A_1: 10 x 100 Matrix
A_2: 100 x 5 Matrix
A_3 : 5 x 50 Matrix

a) Klammerung $((A_1A_2)A_3)$ erfordert

A' = (A_1A_2) :	10 x 100 x 5 =	5000
A'A_3 :	10 x 5 x 50 =	2500

Summer:		7500

b) Klammerung $(A_1(A_2A_3))$ erfordert

$A'' = (A_2A_3)$: 100 x 5 x 50 = 25000
A_1A'' : 10 x 100 x 50 = 50000

Summer: 75000

Man sieht es kommt also enorm auf die Klammerung an

Struktur der optimalen Klammerung

$(A_{i...j}) = ((A_{i...k})|(A_{k+1...j}))$ $i \leq k < j$

Jede optimale Lösung des Matrixkettenproduktes enthält optimale Lösungen von Teilproblemen.

Rekursive Bestimmung des Wertes einer optimalen Lösung:

M[i,j] sei minimale # von Operationen zur Berechnung des Teilproduktes $A_{i...j}$:

$$
\begin{aligned}
m[i,j] &= 0 \quad \text{falls } i = j \\
m[i,j] &= \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} \quad \text{sonst} \\
s[i,k] &= \text{optimaler Splitwert für ein } k, \text{ für das das} \\
&\quad \text{Minimum angenommen wird}
\end{aligned}
$$

Implementierung Rekursives Matrixkettenprodukt

Algorithmus *rek-mat-ket*(p, i, j)
Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$, $p_{i-1} \times p_i$
Dimensionen der Matrix A_i
Invariante: rek-mat-ket(p, i, j) liefert $m[i, j]$

```
1  if i = j then return 0
2  m[i,j] := ∞
3  for k := i to j − 1 do
4      m[i,j] := min(m[i,j], p_{i−1} p_k p_j +
                     rek-mat-ket(p, i, k) +
                     rek-mat-ket(p, k + 1, j) )
5  return m[i,j]
```

Aufruf: rek-mat-ket$(p, 1, n)$

Sei T(n) die Anzahl der Schritte zur Berechnung von rek-mat-ket(p,1,n)

$$
\begin{aligned}
T(1) &\geq 1 \\
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\
&\geq n + 2 \sum_{i=1}^{n-1} T(i) \\
\Rightarrow T(n) &\geq 3^{n-1} \qquad \text{(vollst. Induktion)}
\end{aligned}
$$

⇨ wir haben also exponentielle Laufzeit !!!

Implementierung des Matrixkettenproduktes mit dynamischer Programmierung

Bottom-Up Ansatz (Top-Dow folgt weiter unten)

Algorithmus *dyn-mat-ket*
Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$, $p_{i-1} \times p_i$ Dimensionen der Matrix A_i
Output: $m[1,n]$

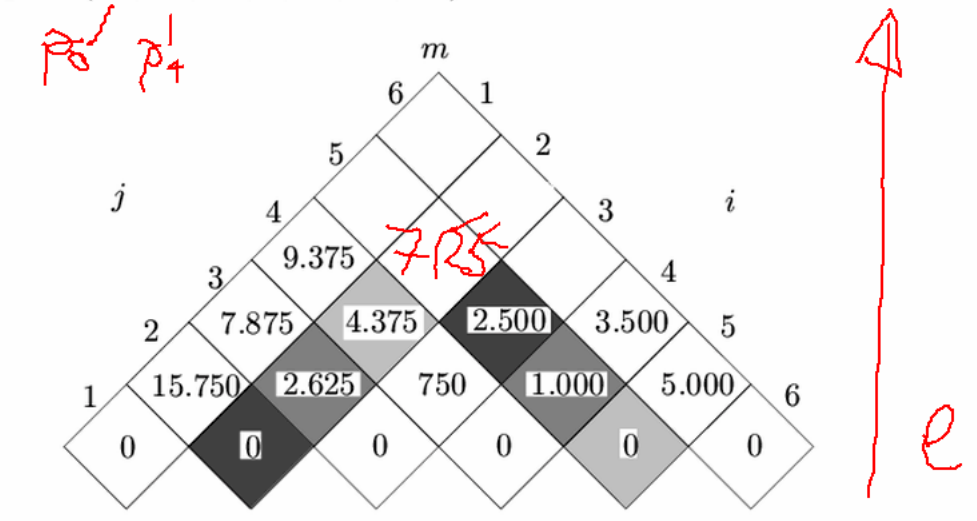
```
1  n := length(p)
2  for i := 1 to n do m[i,i] := 0
3  for l := 2 to n do
       /* l = Länge des Teilproblems */
4      for i := 1 to n − l + 1 do
           /* i ist der linke Index */
5          j := i + l − 1
           /* j ist der rechte Index */
6          m[i,j] := ∞
7          for k := i to j − 1 do
               m[i,j] := min(m[i,j], p_{i−1} p_k p_j +
                             m[i,k] + m[k+1,j])
8  return m[1,n]
```

Zeitbedarf: $O(n^3)$
Platzbedarf: $O(n^2)$

Berechnungsbeispiel:

A_1	30×35	A_4	5×10
A_2	35×15	A_5	10×20
A_3	15×5	A_6	20×25

$p = (30, 35, 15, 5, 10, 20, 25)$



$$m[2,5] =$$

$$\min_{2 \le k < 5} (m[2,k] + m[k+1,5] + p_1 p_k p_5)$$

$$\min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \end{cases}$$

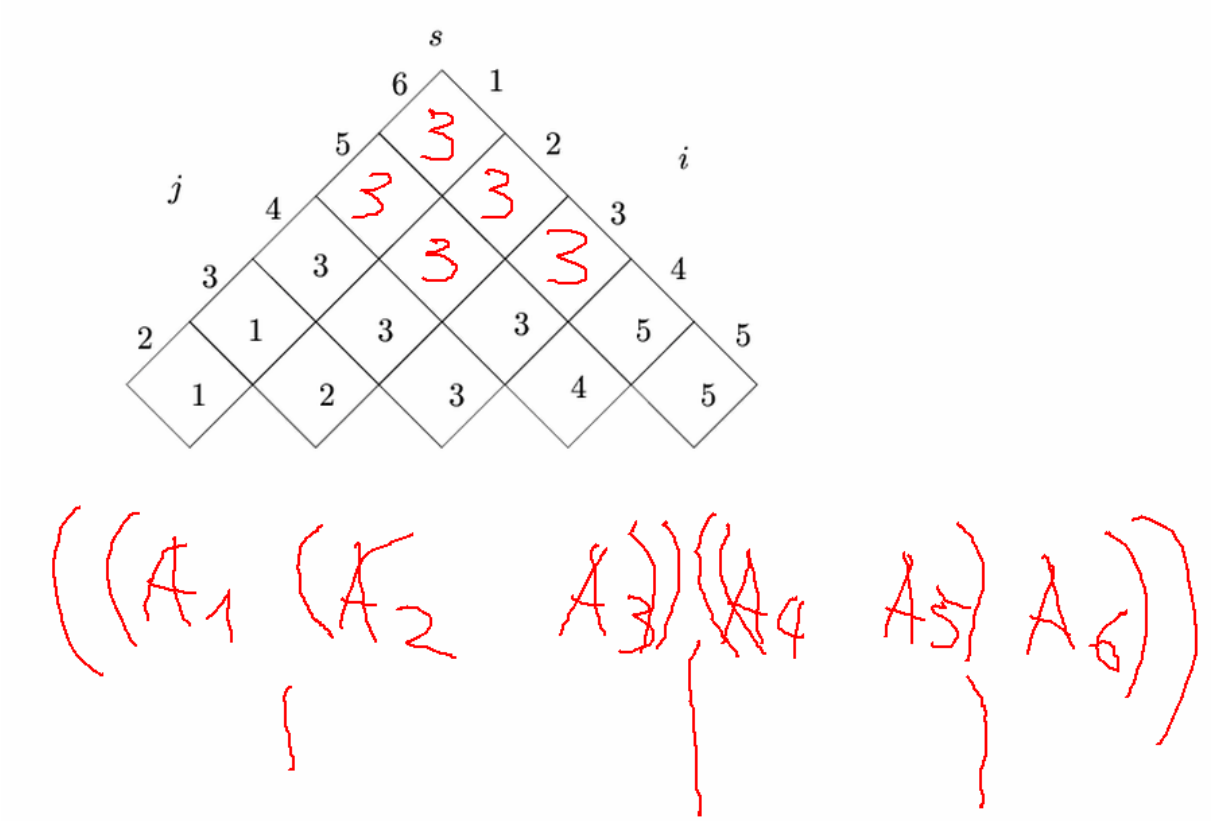
$$\min \begin{cases} 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 \\ 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 \\ 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$

$$= 7125$$

Matrixkettenprodukt und optimale Splitwerte mit dynamischer Programmierung

Algorithmus *dyn-mat-ket*(p)
Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$, $p_{i-1} \times p_i$ Dimensionen der Matrix A_i
Output: $m[1,n]$ und eine Matrix $s[i,j]$ von opt. Splitwerten

```
1  n := length(p)
2  for i := 1 to n do m[i,i] := 0
3  for l := 2 to n do
4      for i := 1 to n − l + 1 do
5          j := i + l − 1
6          m[i,j] := ∞
7          for k := i to j − 1 do
8              q := m[i,j]
9              m[i,j] := min(m[i,j], p_{i−1} p_k p_j +
                                 m[i,k] + m[k+1,j] )
10             if m[i,j] < q then s[i,j] := k
11     return (m[1,n], s)
```



s[1,6] splittet $A_1 \ldots A_6$ an der Stelle 3 → $(A_1A_2A_3)|(A_4A_5A_6)$
s[1,3] splittet $A_1A_2A_3$ an der Stelle 1 → $(A_1(A_2A_3))$
s[4,6] splittet $A_4A_5A_6$ an der Stelle 5 → $((A_4A_5)A_6)$

Implementierung zur Berechnung/Ausgabe der optimalen Klammerung

Algorithmus *Opt-Klam*
Input: Die Sequenz A der Matrizen, die Matrix s der optimalen Splitwerte, zwei Indizes i und j
Output: Eine optimale Klammerung von $A_{i..j}$

```
1  if i < j
2     then X := Opt-Klam(A, s, i, s[i, j])
3          Y := Opt-Klam(A, s, s[i, j] + 1, j)
4          return (X · Y)
5     else return A_i
```

Aufruf: $Opt - Klam(A, s, 1, n)$

Matrixkettenprodukt mir dynamischer Programmierung (Top-Down Ansatz)

Notizblockmethode zur Beschleunigung einer rekursiven Problemlösung:

- ⇨ Ein Teilproblem wird nur beim ersten Auftreten gelöst, die Lösung wir in einer Tabelle gespeichert und bei jedem späteren Auftreten desselben Teilproblems wird die Lösung (ohne erneute Rechnung!) in der Lösungstabelle nachgesehen.

Implementierung der Notizblockmethode zum Matrixkettenprodukt

(Memoisiertes Matrixkettenprodukt)

Algorithmus $mem\text{-}mat\text{-}ket(p, i, j)$
Invariante: mem-mat-ket(p, i, j) liefert $m[i, j]$ und
$m[i, j]$ hat den korrekten Wert, falls
$m[i, j] < \infty$

```
1  if i = j then return 0
2  if m[i,j] < ∞ then return m[i,j]
3  for k := i to j − 1 do
4      m[i,j] := min(m[i,j], p_{i−1} p_k p_j +
                     mem-mat-ket(p, i, k) +
                     mem-mat-ket(p, k + 1, j) )
5  return m[i,j]
```

Aufruf:

```
1  n := length(p) − 1
2  for i := 1 to n do
3      for j := 1 to n do
4          m[i,j] := ∞
5  mem-mat-ket(p, 1, n)
```

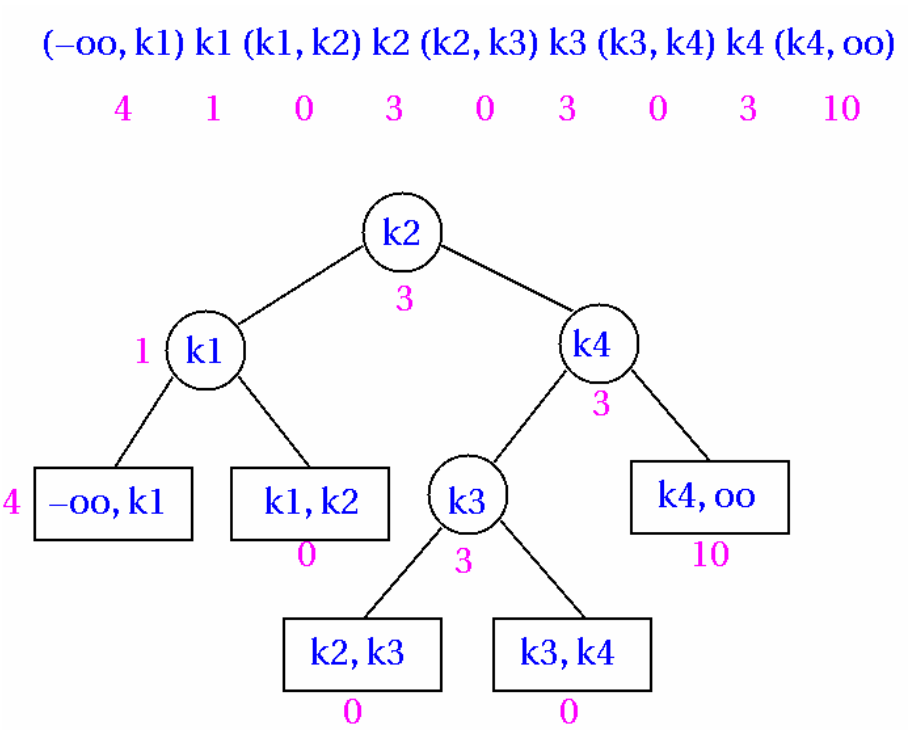
Zur Berechnung aller Einträge $m[i, j]$ mit Hilfe von mem-mat-ket genügen insgesamt $O(n^3)$ Schritte.

$O(n^2)$ Einträge
jeder Eintrag $m[i, j]$ wird einmal eingetragen
jeder Eintrag $m[i, j]$ wird zur Berechnung eines Eintrages $m[i', j']$ betrachtet, falls $i' = i$ und $j' > j$ oder $j' = j$ und $i' < i$
$\Rightarrow \leq 2n$ Einträge benötigen $m[i, j]$

Bemerkungen zum Matrixkettenprodukt:

- Es gibt einen Algorithmus mit linearer Laufzeit O(n), der eine Klammerung findet mit Multiplikationsaufwand ≤ 1.155 M_{opt}
- Es gibt eine Algorithmus mit Laiufzeit O(n log n), der die optimale Klammerung findet.

Konstruktion optimaler Suchbäume

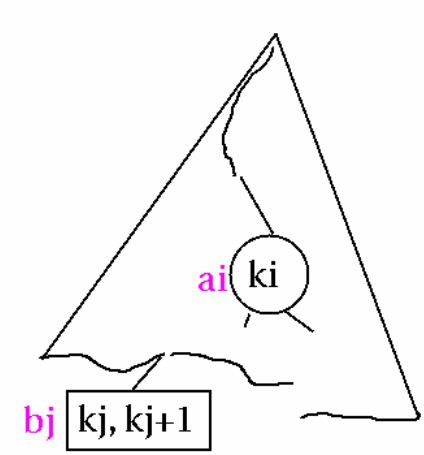


Gesamte Pfadlänge des obigen Baumes:
1*3+2*(1+3)+3*3+2*(4+0)+3*(0+0)+2*10

Schlüsselmenge S= {k1, k2,…, kn},k1<k2<…<kn

- ai =(absolute) Häufigkeit, mit der nach ki gesucht wird
- bj =(absolute) Häufigkeit, mit der nach einem Schlüssel x in (kj,kj+1) gesucht wird

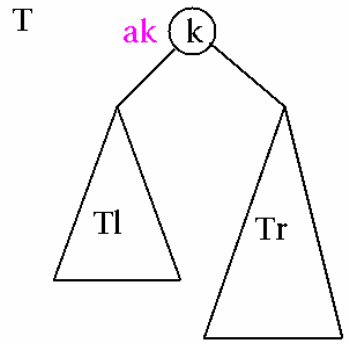
gewichtete Pfadlänge:



$$P = \sum_{i=1}^{n} (\text{Tiefe}(ki) + 1)\, ai \; + \sum_{j=0}^{n} \text{Tiefe}((kj, kj+1))\, bj$$

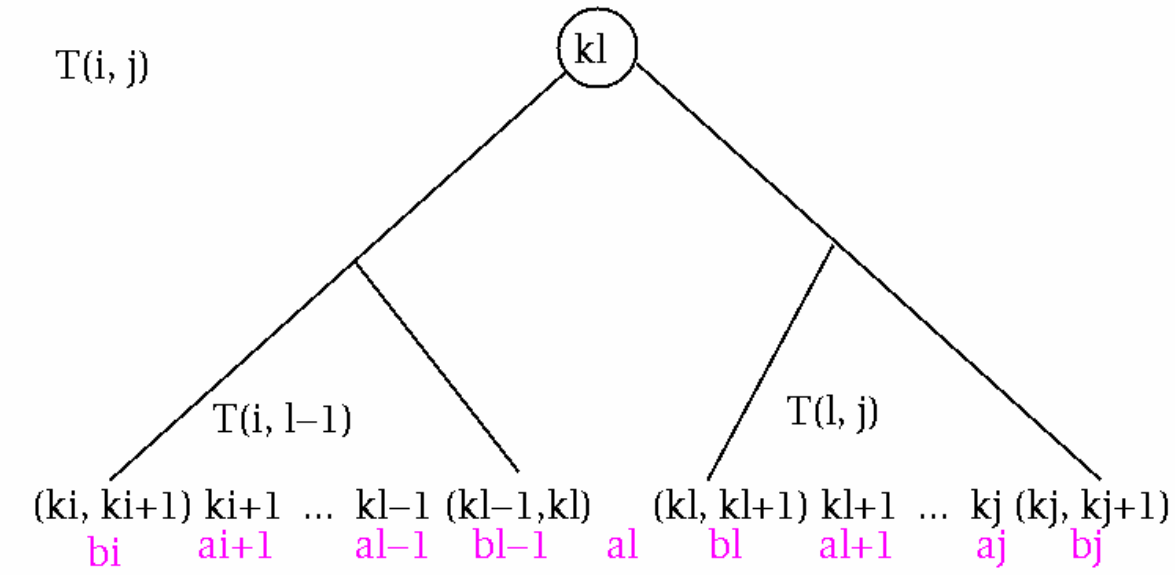
Ein Suchbaum mit minimal möglicher gewichteter Pfadlänge ist ein optimaler Suchbaum

Rekursive Bestimmung der gewichteten Pfadlänge



$$P(T) = P(Tl) + Gewicht(Tl) + P(Tr) + Gewicht(Tr) + ak$$
$$= P(Tl) + P(Tr) + Gewicht(T)$$

Ist T ein Baum mit minimaler gewichteter Pfadlänge, so auch Tl und Tr (Optimalitätsprinzip)



$$W(i, i) = bi \quad , \text{ für } 0 <= i <= n$$
$$W(i, j) = W(i, j-1) + aj + bj \; , \text{ für } 0 <= i < j <= n$$

$$P(i, i) = 0 \quad , \text{ für } 0 <= i <= n$$
$$P(i, j) = W(i, j) + \min_{i<l<=j} \{ P(i, l-1) + P(l, j) \} \; , \text{ für } 0 <= i < j <= n$$

W(i,j) ist das Gesamtgewicht des Baumes T(i,j)
P(i,j) ist die Gesamtfadlänge des Baumes T(i,j)

Fall 1: $h = j - i = 0$

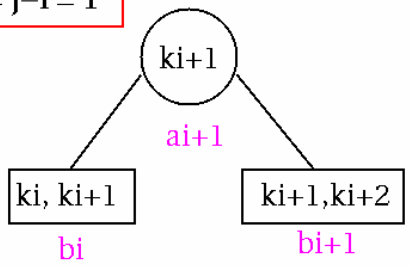
$T(i, i) =$ ki, ki+1

$W(i, i) = bi$

$P(i, i) = 0$, r(i,i) undefiniert

Fall 2: $h = j - i = 1$

T(i, i+1)



$W(i, i+1) = bi + ai+1 + bi+1$

$P(i, i+1) = W(i, i+1)$

$r(i, i+1) = i+1$

Berechnung von P(0,n) nach dem Prinzip der dynamischen Programmierung:

```
for h = 2 to n do
    for i = 0 to (n – h) do
        {j = i + h;
         finde das (größte) l, i < l <= j, für das
         P(i, l–1) + P(l, j) minimal wird;
         P(i, j) = P(i, l–1) + P(l,j) + W(i,j);
         r(i, j) = l;
        }
```

Satz: Ein optimaler Suchbaum für n Schlüssel und n+1 Intervalle mit gegebenen Zugriffshäufigkeiten kann in Zeit $O(n^3)$ und Platz (n^2) konstruiert werden.

Vorlesung 17 – Dynamische Programmierung – Editierdistanz

Vorteile und Nachteile für die beiden Ansätze der dynamischen Programmierung:

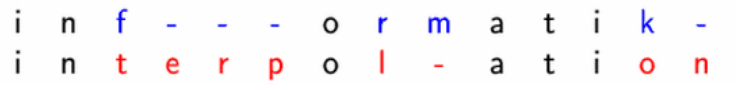
Bottom-Up Ansatz

+ kontrollierte effiziente Tabellenverwaltung, spart Zeit
+ spezielle optimierte Berechnungsreihenfolge, spart Platz
- weitgehende Umcodierung des Originalprogramms erforderlich

- möglicherweise Berechnung nicht benötigter Werte
Top-Down Ansatz (Memoisierung, Notizblockmethode):
+ Originalprogramm wird nur gering oder nicht verändert
+ Nur tatsächlich benötigte Werte werden berechnet
- separate Tabellenverwaltung benötigt Zeit
- Tabellengröße oft nicht optimal

Probleme der Ähnlichkeit von Zeichenketten

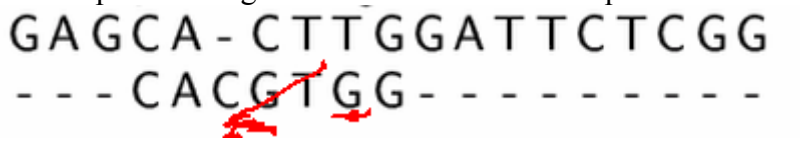
Editierdistanz
Berechne für zwei gegebene Zeichenfolgen A und B möglichst effizient die Editier-Distanz D(A,B) und eine minimale Folge von Editieroperationen, die A in B überführt.



→ D(A,B) = 8

Approximative Zeichenkettensuche
Finde für einen gegebenen Text T, ein Muster P und eine Distanz d alle Teilketten P' in T mit D(P,P')≤d

Sequence Alignment
Finde optimale Alignments zwischen DNA-Sequenzen



Editier-Distanz

Gegeben: Zwei Zeichenketten $A = a_1a_2\ldots a_m$ und $B=b_1b_2\ldots b_n$
Gesucht: Minimale Kosten D(A,B) für eine Folge von Editieroperationen, um A in B zu überführen

Editieroperationen:

1. Ersetzen eines Zeichens von A durch ein Zeichen von B
2. Löschen eines Zeichens von A
3. Einfügen eines Zeichens von B

Einheitskostenmodell:

$$c(a,b) = \begin{cases} 1 & \text{falls } a \neq b \\ 0 & \text{falls } a = b \end{cases}$$

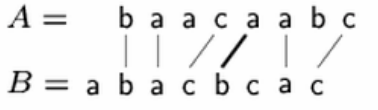
a = ε, b = ε möglich

Dreiecksungleichung soll für c im allgemeinen gelten:

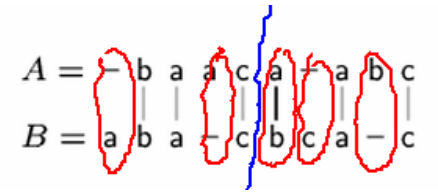
$$C(a,c) \leq c(a,b) + c(b,c)$$

⇨ Ein Buchstabe wird höchstens einmal verändert

Spur als Repräsentation von Editiersequenzen



oder mit Indels



Editier-Distanz (Kosten): 5

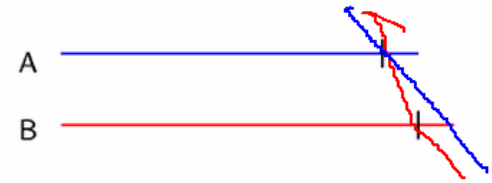
Aufteilung einer optimalen Spur ergibt zwei optimale Teilspuren

⇨ Dynamische Programmierung anwendbar

Berechnung der Editier-Distanz

Sei $A_i = a_1 \ldots a_i$ und $B_j = b_1 \ldots b_j$

$D_{i,j} = D(A_i,B_j)$



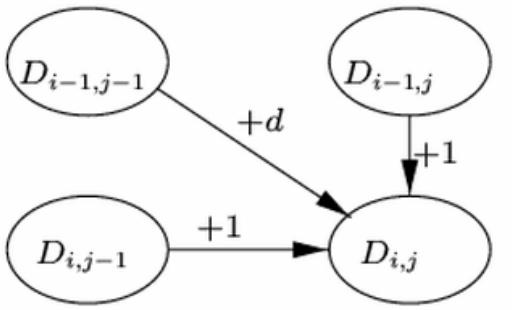
Drei Möglichkeiten eine Spur zu beenden:

1. a_m wird durch b_n ersetzt: $D_{m,n} = D_{m-1,n-1} + c(a_m,b_n)$
2. a_m wird gelöscht: $D_{m,n} = D_{m-1,n}+1$
3. b_n wird eingefügt: $D_{m,n} = D_{m,n-1}+1$

Rekursionsgleichung falls $m,n \geq 1$

$$D_{m,n} = \min \left\{ \begin{array}{lcl} D_{m-1,n-1} & + & c(a_m, b_n), \\ D_{m-1,n} & + & 1, \\ D_{m,n-1} & + & 1 \end{array} \right\}$$

⇨ Berechnung aller $D_{i,j}$ erforderlich, $0 \leq i \leq m$, $0 \leq j \leq n$



Rekursionsgleichung für die Editier-Distanz

Anfangsbedingungen:

$D_{0,0} = D(\varepsilon,\varepsilon) = 0$

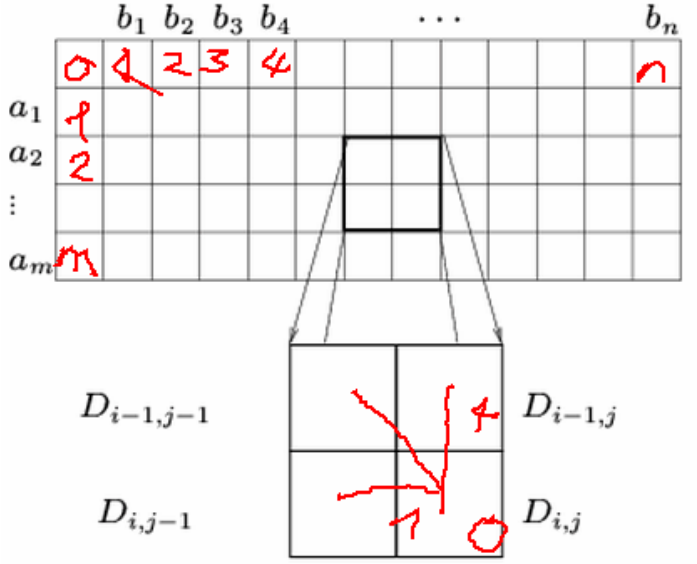
$D_{0,j} = D(\varepsilon,B_j) = j$

$D_{i,0} = D(A_i, \varepsilon) = i$

Rekursionsgleichung:

$$D_{i,j} = \min \left\{ \begin{array}{lcl} D_{i-1,j-1} & + & c(a_i, b_j), \\ D_{i-1,j} & + & 1, \\ D_{i,j-1} & + & 1 \end{array} \right\}$$

Berechnungsreihenfolge für die Editier-Distanz



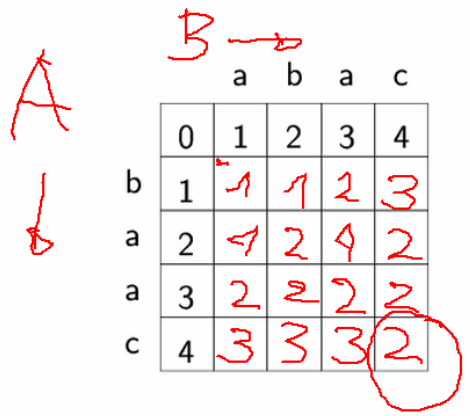
Algorithmus für die Editier-Distanz

Algorithmus Editierdistanz
Input: Zwei Zeichenketten $A = a_1 \cdots a_m$ und $B = b_1 \cdots b_n$
Output: Die Matrix $D = (D_{ij})$

```
1  D[0,0] := 0
2  for i := 1 to m do D[i,0] = i
3  for j := 1 to n do D[0,j] = j
4  for i := 1 to m do
5     for j := 1 to n do
6        D[i,j] := min(D[i-1,j] + 1,
7                      D[i,j-1] + 1,
8                      D[i-1,j-1] + c(a_i, b_j))
```

Laufzeit: O(m*n)
Platzbedarf: O(m*n)

Beispiel für Editier-Distanz



		a	b	a	c
	0	1	2	3	4
b	1	1	1	2	3
a	2	1	2	1	2
a	3	2	2	2	2
c	4	3	3	3	2

A b a a c
B a b a c

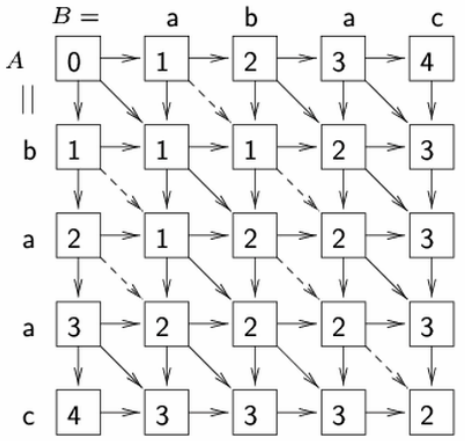
Implementierung: Berechnung der Editieroperationen

Algorithmus Editieroperationen(i, j)
Input: Berechnete Matrix D
1 **if** $i = 0$ **and** $j = 0$ **then return**
2 **if** $i \neq 0$ **and** $D[i, j] = D[i - 1, j] + 1$
3 **then** "lösche $a[i]$"
4 Editieroperationen$(i - 1, j)$
5 **else if** $j \neq 0$ **and** $D[i, j] = D[i, j - 1] + 1$
6 **then** "füge $b[j]$ ein"
7 Editieroperationen$(i, j - 1)$
8 **else**
$/* \; D[i, j] = D[i - 1, j - 1] + c(a[i], b[j]) \; */$
9 "ersetze $a[i]$ durch $b[j]$"
10 Editieroperationen$(i - 1, j - 1)$

Aufruf: Editieroperationen(m,n)

Spurgraph der Editieroperationen

Beispiel Editieroperationen:



Spurgraph:
Übersicht über alle möglichen Spuren zu Transformation von A in B, gerichtete Kante von Knoten (i,j) zu (i+1,j), (i,j+1) und (i+1,j+1).
Gewichtung der Kanten entprechend Editierkosten.

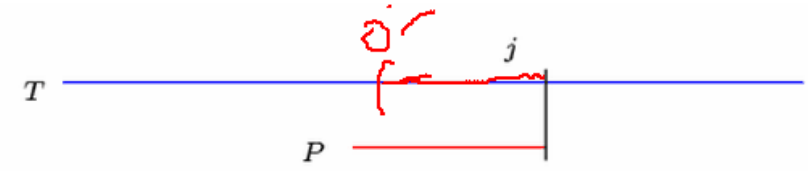
Kosten nehmen entlang eines optimalen Weges monoton zu.

Jeder Weg mit monoton wachsenden Kosten von der linken oberen Ecke zur rechten unteren Ecke entspricht einer optimalen Spur.

Approximative Zeichenkettensuche

Gegeben: zwei Zeichenketten $P = p_1p_2\ldots p_m$ (Muster) und $T = t_1t_2\ldots t_n$ (Text)
Gesucht: Ein Intervall $[j',j]$, $1 \leq j' \leq j \leq n$, so dass das Teilwort $T_{j',j} = t_{j'}\ldots t_j$ das dem Muster P ähnlichste Teilwort von T ist, d.h. für alle anderen Intervalle $[k',k]$, $1 \leq k' \leq k \leq n$, gilt:

$$D(P,T_{j',j}) \leq D(P,T_{k',k})$$

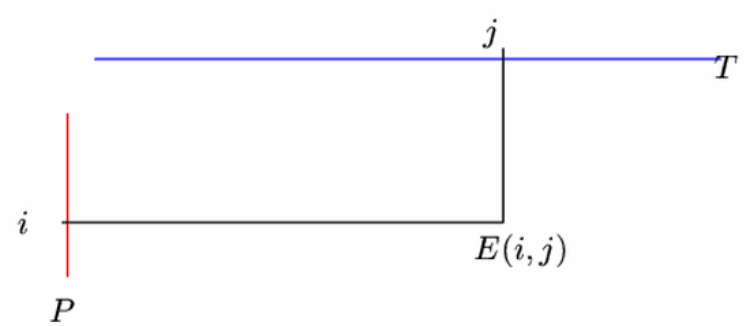


Naives Verfahren:

for all $1 \leq j' \leq j \leq n$ **do**
  Berechne $D(P, T_{j',j})$
wähle Minimum

Laufzeit: $O(n^2 * n * m)$

Betrachte verwandtes Problem:



Für jede Textstelle j und jede Musterstelle i berechne die Editierdistanz des zu P_i ähnlichsten, bei j endenden Teilstücks $T_{j',j}$ von T.

Methode:

for all $1 \leq j \leq n$ **do**
Berechne j', so daß $D(P, T_{j',j})$ minimal ist

Für $0 \leq i \leq m$ und $0 \leq j \leq n$ sei:

$$E_{i,j} = \min_{1 \leq j' \leq j+1} D(P_i, T_{j',j})$$

Optimale Spur:

$$\begin{array}{rl} P_i = & \mathsf{b\ a\ a\ c\ a\ a\ b\ c} \\ & \mid\ \mid\ \diagup\ \diagup\ \mid\ \diagup \\ T_{j',j} = & \mathsf{b\ a\ c\ b\ c\ a\ c} \end{array}$$

Rekursionsgleichung:

$$E_{i,j} = \min \left\{ \begin{array}{l} E_{i-1,j-1} + c(p_i, t_j), \\ E_{i-1,j} + 1, \\ E_{i,j-1} + 1 \end{array} \right\}$$

Bemerkung:
J' kann für $E_{i-1,j-1}$, $E_{i-1,j}$ und $E_{i,j-1}$ ganz verschieden sein. Teilspur einer optimalen Spur ist eine optimale Teilspur (wieder Optimalitätsprinzip)

Anfangsbedingungen:

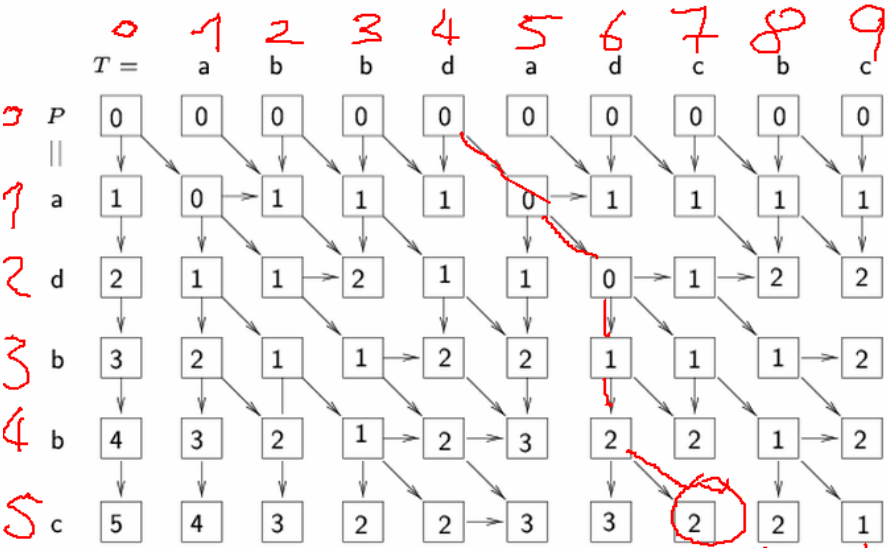
$$E_{0,0} = E(\epsilon, \epsilon) = 0$$
$$E_{i,0} = E(P_i, \epsilon) = i$$

aber

$$E_{0,j} = E(\epsilon, T_j) = 0$$

Beobachtung:
Die optimale Editiersequenz von P nach $T_{j',j}$ beginnt nicht mit einer Einfügung von $t_{j'}$

Abhängigkeitsgraph



$D(P_5,T_{57})=2$

Satz: Gibt es im Abhängigkeitsgraphen einen Weg von $E_{0,j'-1}$ nach $E_{i,j}$, so ist $T_{j',j}$ ein zu P_i ähnlichstes, bei j endendes Teilstück von T mit

$D(P_i,T_{j',j}) = E_{i,j}$

Ähnlichkeit von Zeichenketten

Sequence Alignment
Finde für zwei DNA-Sequenzen Einfügestellen von Leerzeichen, so, dass die Sequenzen möglichst ähnlich sind.

Ähnlichkeitsmaß für Zeichenpaare

Bsp.	Situation	allgemein
$+1$	für Match	$s(a,b)$
-1	für Mismatch	
-2	für Leerzeichen	$-c$

Für die obige Zeichenfolge ergibt sich daraus ein Wert von 6 (-2+(-1)+9)

Ähnlichkeitsmaß für Sequenzen:

$$S(A,B) = \sum_{\text{alle Zeichenpaare}} \text{Ähnlichkeit des Zeichenpaars}$$

ZIEL: Finde Alignment mit optimaler Ähnlichkeit

Operationen:

1. Ersetzen eines Zeichens a durch ein Zeichen b: Gewinn s(a,b)
2. Löschen eines Zeichens von A, Einfügen eines Zeichens von B: Verlust –c

Aufgabe: Finde eine Folge von Operationen zur Umwandlung von A in B, so dass die Summe der Gewinne maximiert wird

$$S_{i,j} = S(A_i, B_j),\ 0 \le i \le m,\ 0 \le j \le n$$

Rekursionsgleichung:

$$\begin{aligned} S_{m,n} &= \max\ (S_{m-1,n-1} + s(a_m, b_n), \\ &\qquad S_{m-1,n} - c, S_{m,n-1} - c) \end{aligned}$$

Anfangsbedingungen:

$$\begin{aligned} S_{0,0} &= S(\epsilon,\epsilon) = 0 \\ S_{0,j} &= S(\epsilon, B_j) = -jc \\ S_{i,0} &= S(A_i, \epsilon) = -ic \end{aligned}$$

Ähnlichste Teilzeichenketten

Gegeben: zwei Zeichenketten A = $a_1a_2\ldots a_m$ und B= $b_1b_2\ldots b_n$
Gesucht: Ein zwei Intervalle [i',i] ⊆ [1,m] und [j',j] ⊆ [1,n], so dass:

$$S(A_{i',i}, B_{j',j}) \ge S(A_{k',k}, B_{l',l}),$$

für alle $[k',k] \subseteq [1,m]$ und $[l',l] \subseteq [1,n]$.

Naives Verfahren:

for all $[i', i] \subseteq [1, m]$ **and** $[j', j] \subseteq [1, n]$ **do**
Berechne $S(A_{i',i}, B_{j',j})$

Laufzeit: $O(m^2n^2mn)$ als kubische Laufzeit !!!

Methode:

for all $1 \leq i \leq m$, $1 \leq j \leq n$ **do**
Berechne i' und j', so daß $S(A_{i',i}, B_{j',j})$ maximal ist

Für $0 \leq i \leq m$ und $0 \leq j \leq n$ sei:

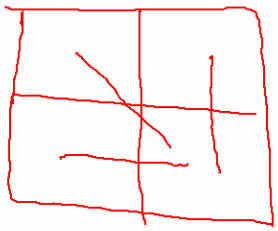
$$H_{i,j} = \max_{\substack{1 \leq i' \leq i+1, \\ 1 \leq j' \leq j+1}} S(A_{i',i}, B_{j',j})$$

Optimale Spur:

$$\begin{array}{lcccccccccc} A_{i',i} = & \mathsf{b} & \mathsf{a} & \mathsf{a} & \mathsf{c} & \mathsf{a} & - & \mathsf{a} & \mathsf{b} & \mathsf{c} \\ & | & | & & | & | & & | & & | \\ B_{j',j} = & \mathsf{b} & \mathsf{a} & - & \mathsf{c} & \mathsf{b} & \mathsf{c} & \mathsf{a} & - & \mathsf{c} \end{array}$$

Rekursionsgleichung:

$$H_{i,j} = \max \left\{ \begin{array}{l} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - c, \\ H_{i,j-1} - c, \\ 0 \end{array} \right\}$$



Anfangsbedingungen:

$$\begin{aligned} H_{0,0} &= H(\epsilon, \epsilon) = 0 \\ H_{i,0} &= H(A_i, \epsilon) = 0 \\ H_{0,j} &= H(\epsilon, B_j) = 0 \end{aligned}$$

Vorlesung 18 – Suche in Texten (KMP/BM)

Verschiedene Szenarios

Dynamische Texte:

- Texteditoren
- Symbolmanipulatoren

Statische Texte

- Literaturdatenbank
- Bibliotheksysteme

- Gen-Datenbanken
- WWW-Verzeichnisse

Datentyp *string*:
- array of character
- file of character
- list of character

Operationen: (seien T,P vom Typ *string*)

Länge:	length ()	
i-tes Zeichen:	T[i]	
Verkettung:	cat (T,P)	T.P

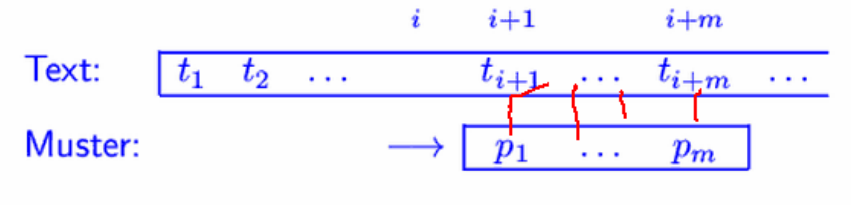
Problemdefinition

Gegeben:

Text	$t_1\, t_2\, \ldots\, t_n$	$\in \Sigma^n$
Muster	$p_1\, p_2\, \ldots\, p_m$	$\in \Sigma^m$

Gesucht:
Ein oder alle Vorkommen des Musters im Text, d.h. Verschiebungen i mit 0≤i≤n-m und

$$\begin{aligned} p_1 &= t_{i+1} \\ p_2 &= t_{i+2} \\ &\vdots \\ p_m &= t_{i+m} \end{aligned}$$



Aufwandsabschätzung (Zeit):
1. # mögl. Verschiebungen: n-m+1 #Musterstellen: m

→ O(n*m)

2. mind. 1 Vergleich pro m aufeinanderfolgende Textstellen

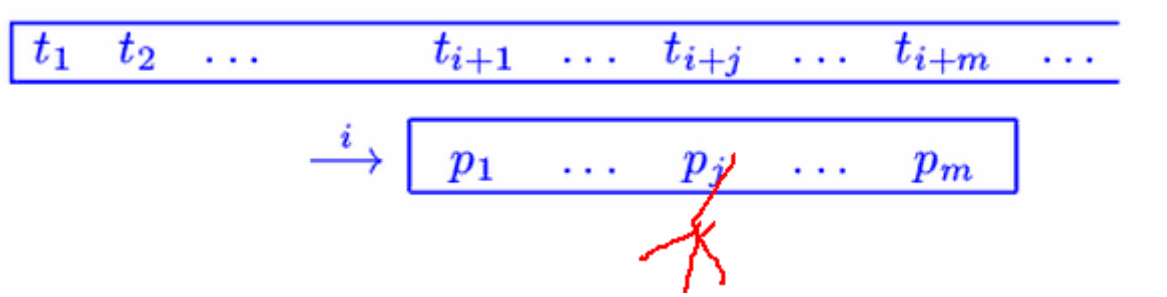
→ Ω(m+⌊ n/m ⌋)

Naives Verfahren

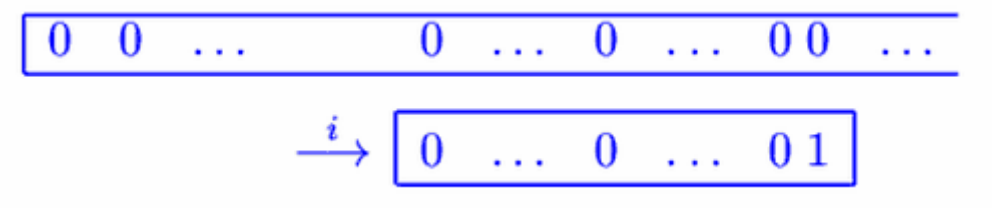
Für jede mögliche Verschiebung 0≤i≤n-m prüfe maximal m Zeichenpaare.

Bei Mismatch beginne mit neuer Verschiebung

```
textsearchbf := proc (T::string, P::string)
# Input:  Text T und Muster P
# Output: Liste L mit Verschiebungen i
#         an denen P in T vorkommt
   n := length (T); m := length (P);
   L := [];
   for i from 0 to n-m do
      j := 1;
      while j <= m and T[i+j] = P[j]
         do j := j+1 od;
      if j = m+1 then L := [L[],i] fi;
   od;
   RETURN (L)
end;
```



Aufwandsabschätzung (Naives Verfahren)

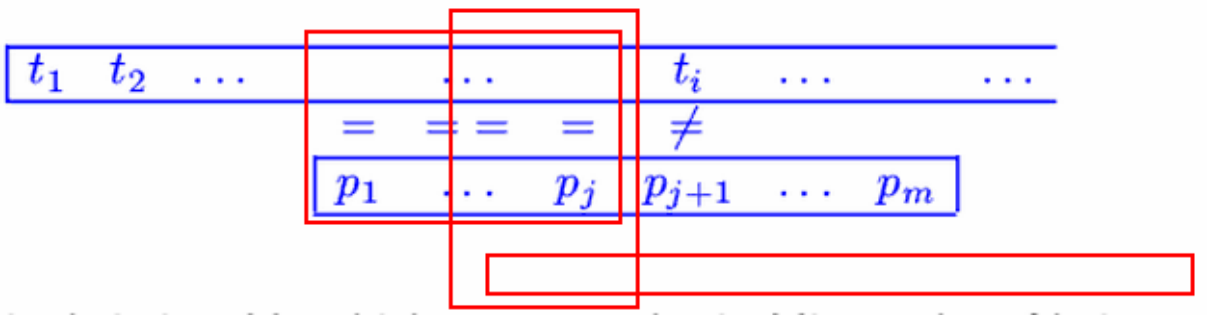


Worst Case: $\Omega(m\ n)$

In der Praxis oft: Mismatch tritt sehr früh auf
→ Laufzeit ~ cn (c ist die Stelle an der im Durchschnitt der Mismatch auftaucht)

Verfahren von Knuth-Morris-Pratt

Seien t_i und p_{j+t} die zu vergleichenden Zeichen



Tritt bei einer Verschiebung erstmals ein Mismatch auf bei t_i und p_{j+1} dann gilt:

- Die zuletzt verglichenen j Zeichen in T stimmen mit den ersten j Zeichen in P überein
- $t_i \neq p_{j+1}$

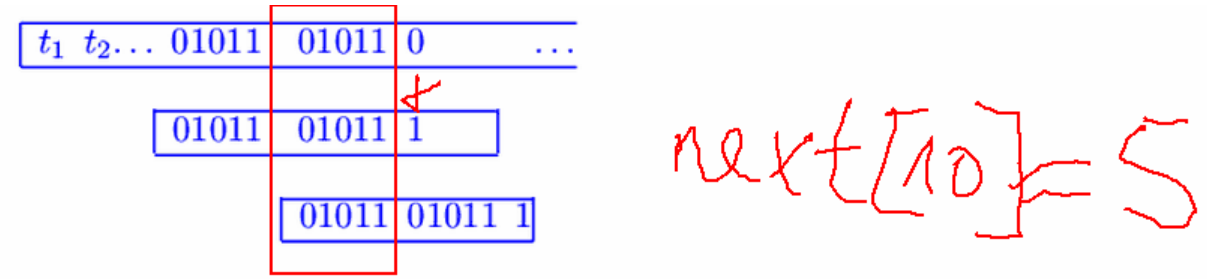
Idee:

Bestimme j' = next[j] < j, so dass t_i anschließend mit $p_{j'+1}$ verglichen werden kann.

Bestimme j' < j, so dass $P_{1\ldots j'} = P_{j-j'+1\ldots j}$

Bestimme das längste Präfix von P, das echtes Suffix von $P_{1\ldots j}$ ist.

Beispiel für die Bestimmung von next[j]:



next[j] = Länge des längsten Präfixes von P, das echtes Suffix von $P_{1\ldots j}$ ist.

⇨ für P = 0101101011 ist next = [0,0,1,2,0,1,2,3,4,5]:

1	2	3	4	5	6	7	8	9	10
0	1	0	1	1	0	1	0	1	1
		0							
		0	1						
					0				
					0	1			
					0	1	0		
					0	1	0	1	
					0	1	0	1	1

```
KMP := proc (T::string, P::string)
# Input:  Text T und Muster P
# Output: Liste L mit Verschiebungen i
#         an denen P in T vorkommt
   n := length (T); m := length (P);
   L := [];
   next := KMPnext (P);
   j := 0;
   for i from 1 to n do
      while j>0 and T[i] <> P[j+1]
         do j := next [j] od;
      if T[i] = P[j+1] then j := j+1 fi;
      if j = m then L := [L[],i-m];
                    j := next [j]
      fi;
   od;
   RETURN (L);
end;
```

i ist Zeiger auf Zeichen in T
j ist Zeiger vor Zeichen in P

Muster: abrakadabra, next $= [0, 0, 0, 1, 0, 1, 0, 1, 2, 3, 4]$

```
a b r a k a d a b r a b r a b a b r a k ···
| | | | | | | | | | |
a b r a k a d a b r a
```
$next[11] = 4$

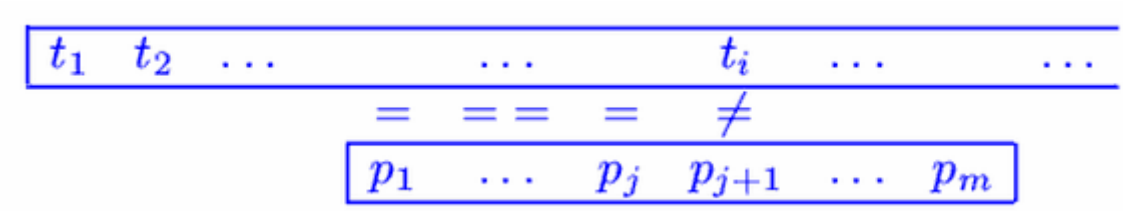
```
a b r a k a d a b r a b r a b a b r a k ···
            - - - - ✗
            a b r a k
```
$next[4] = 1$

```
a b r a k a d a b r a b r a b a b r a k ···
                - | | | ✗
                a b r a k
```
$next[4] = 1$

```
a b r a k a d a b r a b r a b a b r a k ···
                      - | ✗
                      a b r a k
```
$next[2] = 0$

```
a b r a k a d a b r a b r a b a b r a k ···
                          | | | | |
                          a b r a k
```

Korrektheit:



Situation am Beginn der for-Schleife:
$P_{1...j}$ $T_{i-j...i-1}$ und $j \neq m$

falls j=0: man steht auf erstem Zeichen vom P
falls j≠0: P kann verschoben werdeb, solange $j > 0$ und $t_i \neq p_{j+1}$

Ist dann T[i] = P[j+1], können j und i (am Schleifenende] erhöht werden.

Wurde ganz P verglichen (j=m), ist eine Stelle gefunden, und es kann verschoben werden.

Laufzeit:

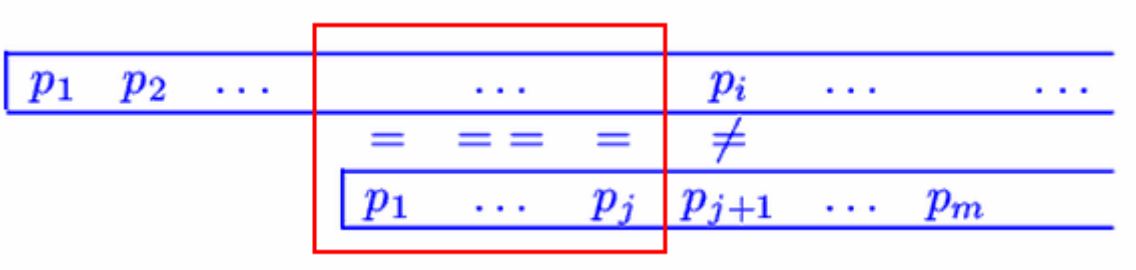
- Textzeiger i wird nie zurückgesetzt
- Textzeiger i und Musterzeiger j werden stets gemeinsam inkrementiert
- Es ist next[j] < j; j kann nur so oft herabgesetzt werden wie es heraufgesetzt wurde.

Der KMP-Algorithmus kann in Zeit O(n) ausgeführt werden, wenn das next-Array bekannt ist.

Berechnung des next-Arrays

next[i] = Länge des längsten Prefixes von P, das echtes Suffix von $P_{1...i}$ ist

- next[1]=0
- Sei next[i-1]=j;



Betrachte 2 Fälle:

1) $p_i = p_{j+1} \Rightarrow \texttt{next}[i] = j + 1$
2) $p_i \neq p_{j+1} \Rightarrow$ ersetze j durch $\texttt{next}[j]$, bis $p_i = p_{j+1}$ oder $j = 0$.
   Falls $p_i = p_{j+1}$ ist, kann $\texttt{next}[i] = j + 1$ gesetzt werden, sonst ist $\texttt{next}[i] = 0$

Algorithmus – Berechnung des next-Arrays

```
KMPnext := proc (P::string)
# Input:  Muster P
# Output: next-Array fuer P
   m := length (P);
   next := array (1..m);
   next [1] := 0;
   j := 0;
   for i from 2 to m do
      while j > 0 and P[i] <> P[j+1]
         do j := next [j] od;
      if P[i] = P[j+1] then j := j+1 fi;
      next [i] := j
   od;
   RETURN (next);
end;
```

Zeitaufwand: O(m) (nur für das next-Arrays)

Zeitaufwand Gesamt für KMP Algorithmus:

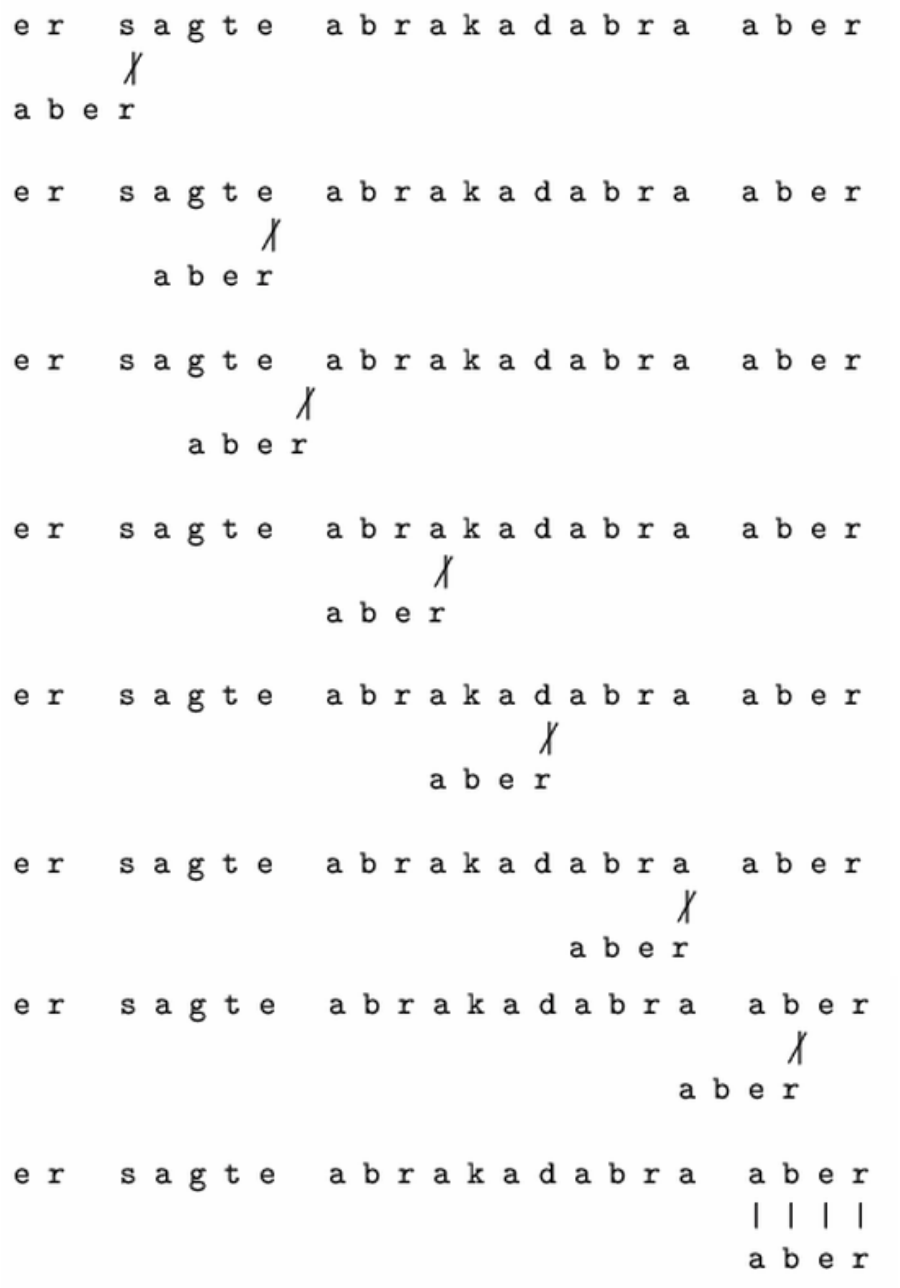
O(n+m)

Kann die Textsuch noch schneller sein ? → n+m bedeutet doch eigentlich, jedes Zeichen einmal zu betrachten, aber es geht noch schneller !!

Verfahren von Boyer-Moore

Idee:
Das Muster von links nach rechts anlegen, aber zeichenweise von rechts nach links verglichen.

Beispiel:



Große Sprünge: Wenig vergleiche
Erhoffte Laufzeit: $O(m+\lfloor n/m \rfloor)$

Die Vorkommensheuristik

Für c ∈ Σ und das Muster P sei (Σ ist das Alphabet !)

$$\begin{aligned}\delta(c) &:= \text{Index des von rechts her ersten Vorkommens von } c \text{ in } P \\ &= \max\{j \mid p_j = c\} \\ &= \begin{cases} 0 & \text{falls } c \notin P \\ j & \text{falls } c = p_j \text{ und } c \neq p_k \text{ für } j < k \leq m \end{cases}\end{aligned}$$

Wie teuer ist die Berechnung aller δ-Werte?

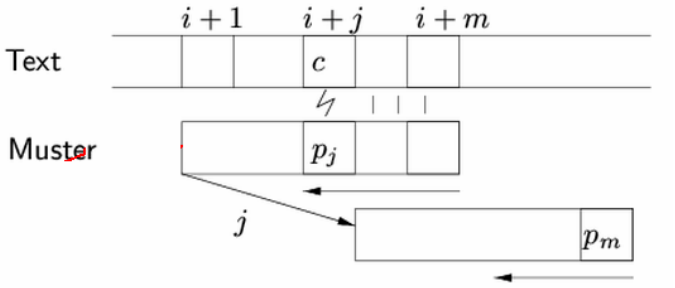
Sei |Σ| = l:

O(l+m)

Seien:
c = das den Mismatch verursachende Zeichen
j = Index des aktuelllen Zeichens im Muster ($c \neq p_j$)

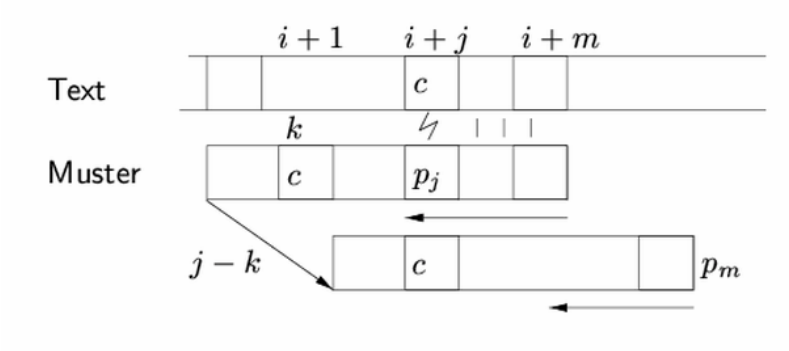
Berechnung der Musterverschiebung

Fall 1 c kommt nicht im Muster P vor. (δ(c)=0)
Verschiebe das Muster um j Positionen nach rechts

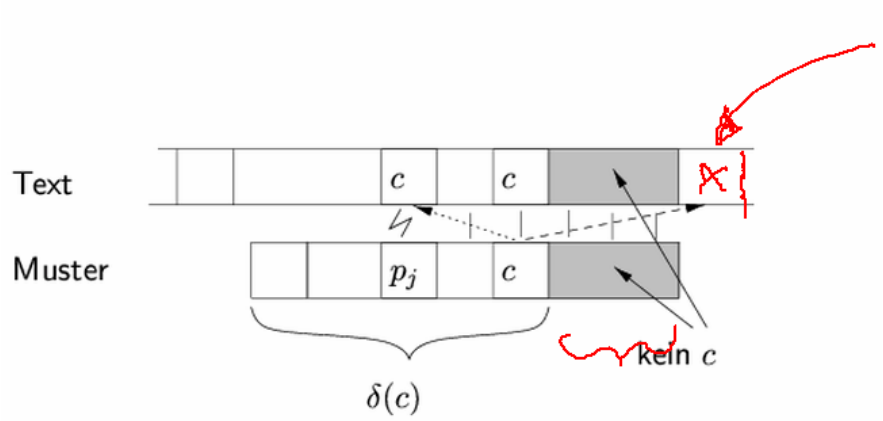


$\Delta(i) = j$

Fall 2 c kommt im Muster vor. (δ(c)≠0)
Verschiebe das Muster soweit nach rechts, dass das rechteste c im Muster über einem potentiellen c im Text liegt



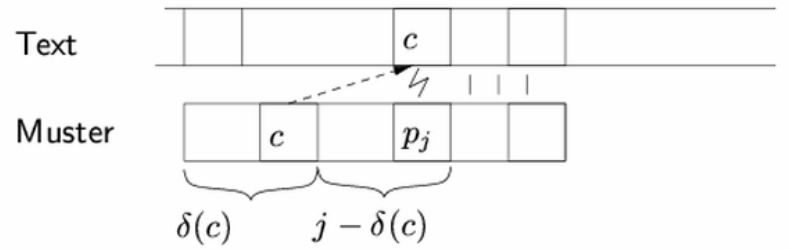
Fall 2a δ(c)>j



Verschiebung des rechtesten c im Muster auf ein potentielles c im Text

⇨ Verschiebung um $\Delta(i) = m - \delta(c) + 1$

Fall 2b δ(c)<j



Verschiebung des rechtesten c im Muster auf c im Text:
Verschiebung um $\Delta(i) = j - \delta(c)$

Algorithmus zum Verfahren von Boyer-Moore (1. Version)

Algorithmus BM-search1
Input: Text T und Pattern P
Output: Verschiebungen für alle Vorkommen von P in T

```
1  n := length(T); m := length(P)
2  berechne δ
3  i := 0
4  while i ≤ n − m do
5      j := m
6      while j > 0 and P[j] = T[i + j] do
7          j := j − 1
       end while;
8      if j = 0
9        then gebe Verschiebung i aus
10             i := i + 1
11     else if δ(T[i + j]) > j
12       then i := i + m + 1 − δ[T[i + j]]
13       else i := i + j − δ[T[i + j]]
   end while;
```

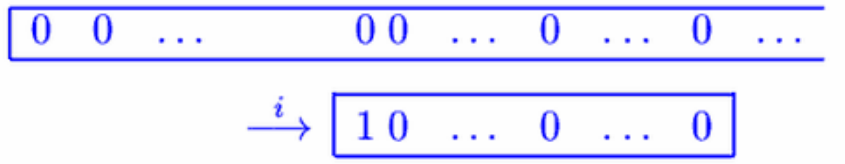
Verschiebungen

δ ≥ 0

Laufzeitanalyse (1. Version):

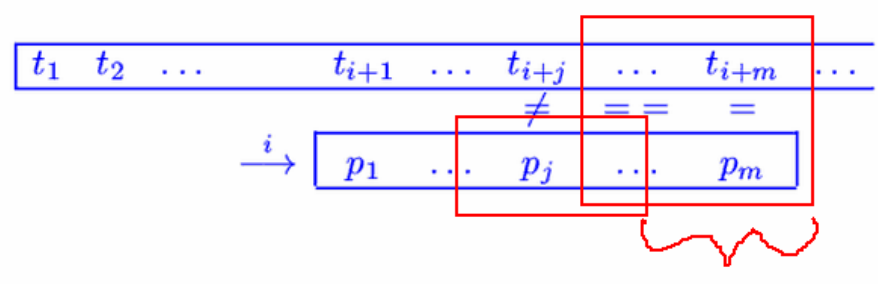
- ⇨ gewünschte Laufzeit: c(m+n/m)
- ⇨ worst-case Laufzeit: Ω(n m)

Es gibt folgende schlechte Eingabefolge die den worst-case erzeugt:



Match-Heuristik

Nutze die bis zum Auftreten eines Mismatches $p_j \neq t_{i+j}$ gesammelte Information



wrw[j] = Position, an der das von rechts her nächste Vorkommen des Suffixes $P_{j+1...m}$ endet, dem nicht das Zeichen P_j vorangeht.

Mögliche Verschiebung: $\gamma[j] = m - wrw[j]$

Beispiel für die wrw-Berechnung

wrw[j] = Position, an der das von rechts her nächste Vorkommen des Suffix $P_{j+1...m}$ endet, dem nicht das Zeichen P_j vorangeht.

Muster: banana

$wrw[j]$	betracht. Suf.	verb. Zeich.	weit. Auft.	Pos.
$wrw[5]$	a	n	banana	2
$wrw[4]$	na	a	***banana	0
$wrw[3]$	ana	n	banana	4
$wrw[2]$	nana	a	banana	0
$wrw[1]$	anana	b	banana	0
$wrw[0]$	banana	ϵ	banana	0

$\Rightarrow$ wrw (banana) $= [0, 0, 0, 4, 0, 2]$

```
a b a a b a b a n a n a n a n a
          ≠ = = =
      b a n a n a
        b a n a n a
```

Algorithmus zum Verfahren von Boyer-Moore (2. Version)

Algorithmus BM-search2
Input: Text T und Pattern P
Output: Verschiebungen für alle Vorkommen von P in T

```
1  n := length(T); m := length(P)
2  berechne δ und γ
3  i := 0
4  while i ≤ n − m do
5      j := m
6      while j > 0 and P[j] = T[i + j] do
7          j := j − 1
       end while;
8      if j = 0
9          then gebe Verschiebung i aus
10             i := i + γ[0]
11         else  i := i + max(γ[j], j − δ[T[i + j]])
   end while;
```

Vorlesung 19 – Suche in Texten - „Suffix Bäume“

Suchindex
Zu einem Text σ für Suche nach verschiedenen Mustern α

Eigenschaften:

1. **Teilwortsuche** in Zeit O(|α|)
2. **Anfragen an σ selbst**, z.B.:
   a. Längstes Teilwort von σ, das an mind. 2 Stellen auftritt
3. **Präfix-Suche**: Alle Stellen in σ mit Präfix α
4. **Bereichs-Suche**: alle Stellen in σ im Intervall [α,β] mit $\alpha \leq_{lex} \beta$, z.B.
   a. abrakadabra, acacia ∈ [abc,acc],
   b. abacus ∉ [abc, acc]
5. **Lineare Komplexität**: Speicherplatzbedarf und Konstruktionszeit ∈ O(|σ|)

Tries

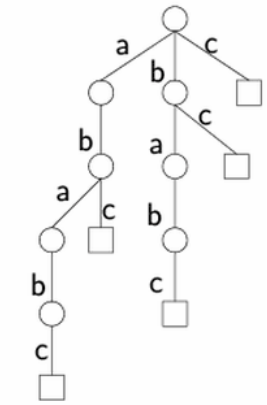
Trie: Baum zur Repräsentation von Schlüsseln.

Alphabet Σ, Menge S von Schlüsseln, $S \subset \Sigma^*$

Schlüssel entspricht Zeichenkette aus Σ^*
Kante eines Tries T: Beschriftung mit einzelnen Zeichen aus Σ
Benachbarte Kanten: verschiedene Zeichen
Blatt repräsentiert Schlüssel:
Entspricht Beschriftungen der Kanten des Weges von der Wurzel zu Blatt

! Schlüssel werden nicht in Knoten gespeichert !

Beispiel:

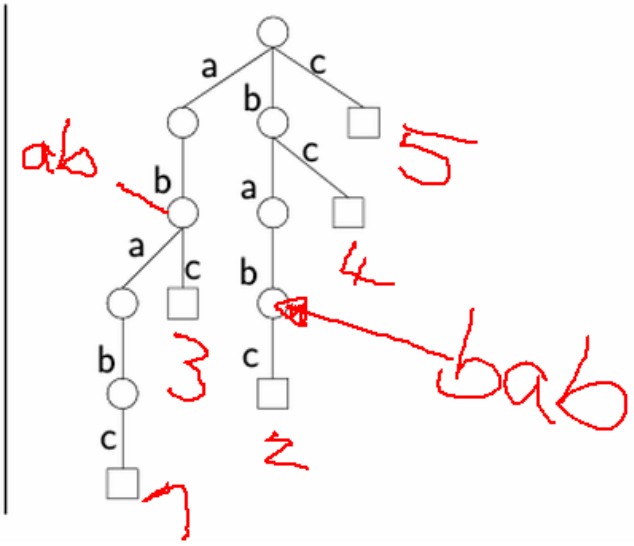


Suffix-Tries

Trie für alle Suffixe eines Wortes

Beispiel: $\sigma =$ ababc

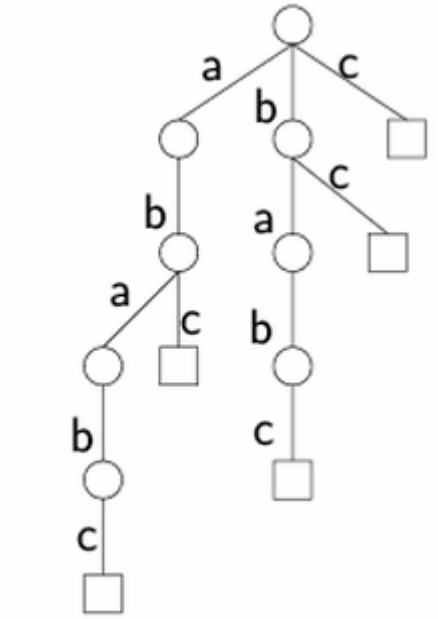
Suffixe:
ababc = suf_1
babc = suf_2
abc = suf_3
bc = suf_4
c = suf_5



Innerer Knoten eines Suffix-Tries = Teilwort von σ.
Beispiel:
Sei $\sigma = a^n b^n$: $\exists n^2 + 2n + 1$ versch. Teilwörter = innere Knoten
$a_i b_j$ mit $i,j \in \{0 \dots n\}$

⇨ Speicherplatzbedarf $\in O(n^2)$

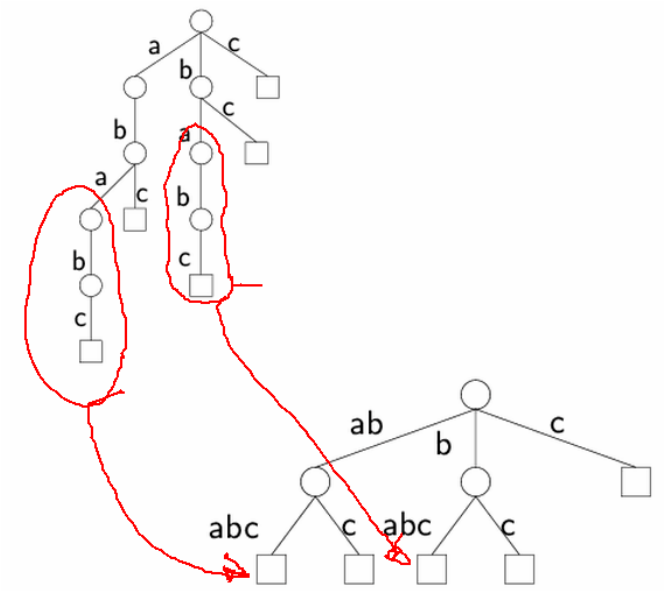
Ein Suffix-Trie T erfüllt bereits einige der geforderten Eigenschaften:



1. Zeichenkettensuche nach α: Folge dem Weg mit Kantenbeschriftung α in T in Zeit O(|α|). Blätter des Teilbaumes $_{\text{entspricht}}$ Vorkommen von α.
2. Längstes, doppelt auftretendes Wort: Innerer Knoten mit größter Tiefe, der mindestens zwei Söhne hat.
3. Präfix-Suche: alle Vorkommen von Zeichenketten mit Präfix α finden sich in dem Teilbaum unterhalb des inneren Knotens von α in T.

Suffix-Bäume

Suffix-Baum entsteht durch Kontraktion von unären Knoten aus Suffix-Trie:



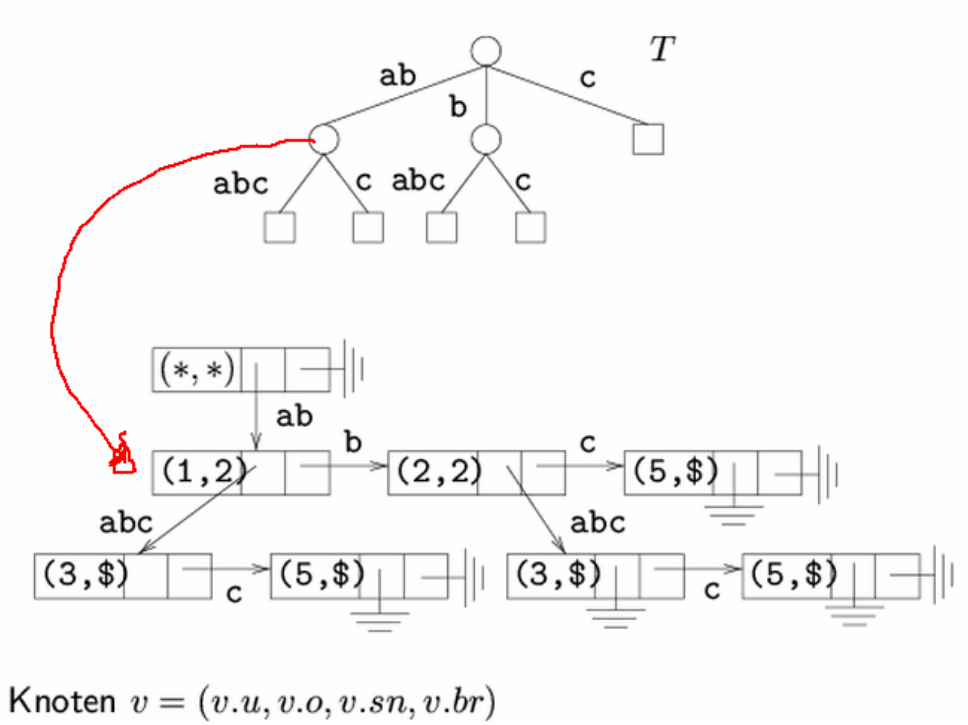
Suffix-Baum = kontraktierter Suffix-Trie

Interne Repräsentation von Suffix-Bäumen

Sohn/Bruder-Repräsentation

Teilworte: Zahlenpaare (i,j)

Beispiel: σ = ababc



Knoten $v = (v.u, v.o, v.sn, v.br)$

Weitere Zeiger (Suffix-Zeiger) kommen später hinzu.

Eigenschaften von Suffix-Bäumen

$(S1)$ Kein Suffix ist Präfix eines anderen Suffixes; gilt, falls (letztes Zeichen von σ) $= \$ \notin \Sigma$

Suche:

$(T1)$ Kante $\widehat{=}$ nichtleeres Teilwort von σ.

$(T2)$ Benachbarte Kanten: zugeordnete Teilworte beginnen mit verschiedenen Zeichen.

Größe

$(T3)$ Innerer Knoten ($\neq$ Wurzel): mind. zwei Söhne.

$(T4)$ Blatt: $\widehat{=}$ (nicht-leeres) Suffix von σ.

Sei $n = |\sigma| \neq 1$

$\overset{(T4)}{\Longrightarrow}$ Anzahl der Blätter: n

$\overset{(T3)}{\Longrightarrow}$ Anzahl der inneren Knoten: $\leq n - 1$

⇨ Speicherplatzbedarf: O(n) = O(|σ|)

Konstruktion von Suffix-Bäumen

Definitionen:

Partieller Weg: Weg von der Wurzel zu einem Knoten von T

Weg: Ein partieller Weg, der bei einem Blatt endet.

Ort einer Zeichenkette α: Knoten am Ende des mit α beschrifteten partiellen Weges (falls er existiert).

Erweiterung einer Zeichenkette α: Zeichenkette mit Präfix α

Erweiterter Ort einer Zeichenkette α: Ort der kürzesten Erweiterung von α, deren Ort definiert ist.

Kontraktierter Ort einer Zeichenkette α: Ort des längsten Präfixes von α, dessen Ort definiert ist.

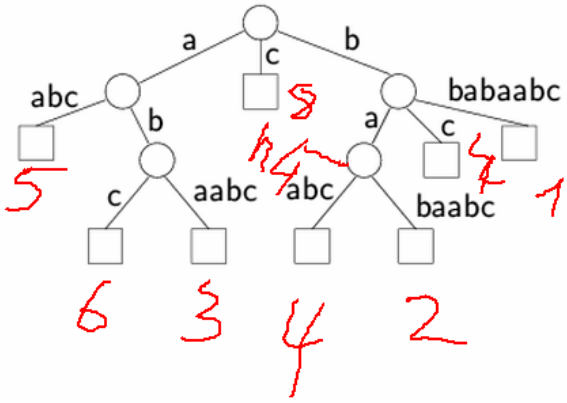
Definitionen:

suf_i: an Position i beginnendes Suffix von σ, also z.B.
$suf_1 = \sigma$, $suf_n = \$$.

$head_i$: längstes Präfix von suf_i, das auch Präfix von suf_j für ein $j < i$ ist.

Beispiel:

$\sigma =$ bbabaabc $\alpha =$ baa (hat keinen Ort)
$suf_4 =$ baabc
$head_4 =$ ba



Naive Suffix-Baum-Konstruktion

Beginne mit dem leeren Baum T_0
Der Baum T_{i+1} entsteht aus T_i durch Einfügen des Suffixes suf_{i+1}.

Algorithmus Suffix-Baum

Input: Eine Zeichenkette σ
Output: Der Suffix-Baum T von σ

```
1   n := |σ|; T_0 := ∅;
2   for i := 0 to n − 1 do
3       füge suf_{i+1} in T_i ein, dies sei T_{i+1};
4   end for
```

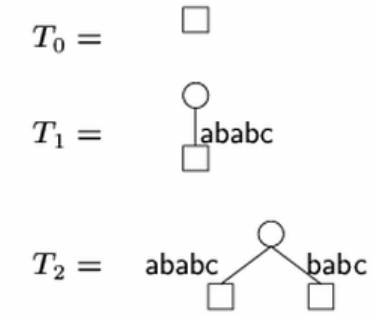
In T_i haben alle Suffixe suf_j, $j<i$ bereits einen Ort.
⇨ $head_i$ = längstes Präfix von suf_i, dessen erweiterter Ort in T_{i-1} existiert.

Definition:
$Tail_i := suf_i - head_i$, d.h. also $suf_i = head_i tail_i$.

$\overset{(S1)}{\Longrightarrow} tail_i \neq \varepsilon.$

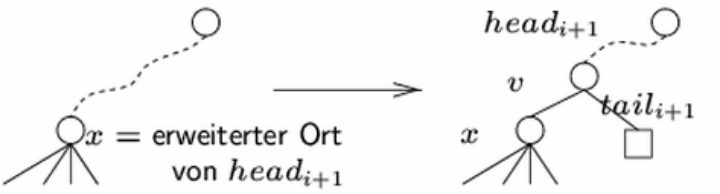
Beispiel: $\sigma =$ ababc

$suf_3 = abc$
$head_3 = ab$
$tail_3 = c$

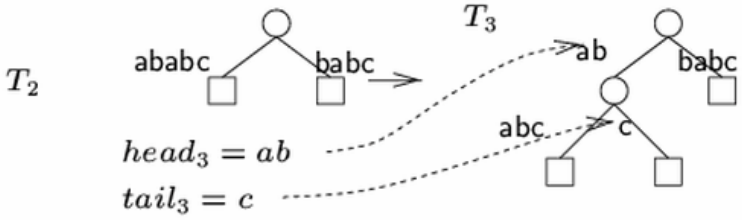


T_{i+1} kann aus T_i wie folgt konstruiert warden:

1. Man bestimme den erweiterten Ort von $head_{i+1}$ in T_i und teile die letzte zu diesem Ort führende Kante in zwei neue Kanten auf durch Einfügen eines neuen Knotens.
2. Man schaffe ein neues Blatt als Ort für suf_{i+1}.



Beispiel: $\sigma =$ ababc



Algorithmus Suffix-Einfügen

Input: Der Baum T_i und der Suffix suf_{i+1}
Output: Der Baum T_{i+1}

```
1   v := Wurzel von T_i
2   j := i
3   repeat
4     finde Sohn w von v mit σ_{w.u} = σ_{j+1}
5     k := w.u − 1;
6     while k < w.o and σ_{k+1} = σ_{j+1} do
7       k := k + 1; j := j + 1
      end while
8     if k = w.o then v := w
9   until k < w.o or w = nil ;
10  /* v ist kontraktierter Ort von head_{i+1} */
11  füge den Ort von head_{i+1} und tail_{i+1} in T_i unter v ein
```

Laufzeit für Suffix-Einfügen:O(n-i)
Gesamtlaufzeit für naive Suffix-Baum-Konstruktion: $O(n^2)$
Es gibt Beispiele die $\Omega(n^2)$ benötigen.

Der Algorithmus M (McCreight, 1976)

Falls erweiterter Ort von $head_{i+1}$ in T_i gefunden: Erzeugen eines neuen Knotens und Aufspalten einer Kante $\in O(1)$ Zeit.

Ziel: Erweiterten Ort von $head_{i+1}$ in konstanter amortisierter Zeit in T_i bestimmen. (Zusatzinformation erforderlich!)

Lemma 1. Wenn $head_i = a\gamma$ für ein Symbol a und eine (eventuell leere) Zeichenkette γ ist, dann ist γ ein Präfix von $head_{i+1}$.

Beweis: Sei $head_i = a\gamma$, dann existiert ein $j<i$, so dass $a\gamma$ Präfix von suf_i und suf_j ist nach der Definition von $head_i$. Also ist γ ein Präfix sowohl von suf_{i+1} als auch von suf_{j+1}.

Suffix-Zeiger:
Von jedem inneren Knoten, der der Ort eines Wortes $a\gamma$ ist, gibt es einen Zeiger auf den Ort des Wortes γ.

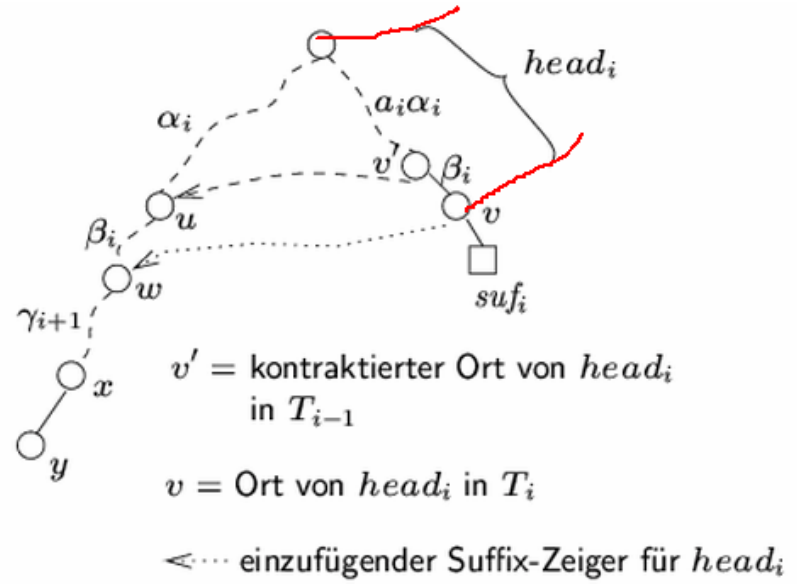
Bemerkung 1. Der Ort von γ liegt niemals im Teilbaum T mit Wurzel beim Ort von $a\gamma$, da T nur Erweiterungen von $a\gamma$ enthält.

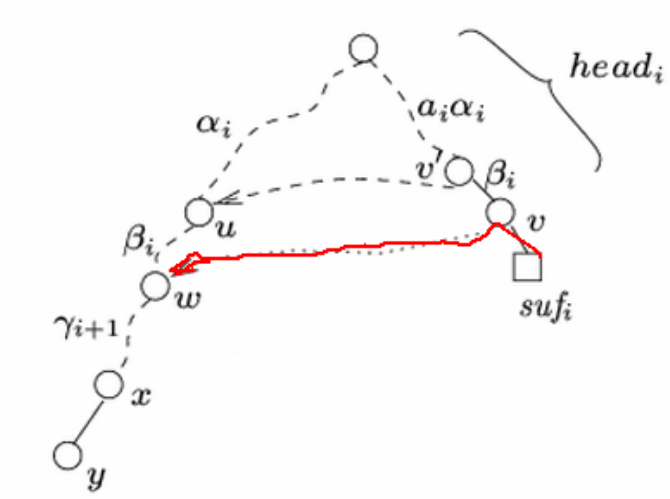
Die Invarianten des Algorithmus M

Nach Konstruktion von T_i gilt:
(Inv1) Alle inneren Knoten von T_{i-1} haben einen korrekten Suffix-Zeiger in T_i
(Inv2) Bei der Konstruktion von T_i wird der kontrakierte Ort von $head_i$ in T_{i-1} besucht.

i=1:
i>1: Konstruktion von T_{i+1} aus T_i kann beim kontraktierten Ort von $head_i$ in T_{i-1} starten

Definition: Ist $head_i \neq \varepsilon$, so sei α_i die Konkatenation der Kantenbeschriftungen des Weges zum kontraktierten Ort von $head_i$ ohne den ersten Buchstaben a_i . Es sei $\beta_i = head_i - a_i\alpha_i$, d.h. $head_i = a_i\alpha_i\beta_i$. Für $head_i \neq \varepsilon$, gilt in T_i:





Wegen Lemma 1 ist

$$head_{i+1} = \alpha_i\beta_i\gamma_{i+1}$$

(Inv2) => Konstruktion von T_{i+1} kann beim kontraktierten Ort v' von $head_i$ in T_{i-1} beginnen.

(Inv1) => von v' gibt es bereits einen korrekten Suffix-Zeiger in T_i zu einem Knoten u.

Folge v'.suffix-zeiger zu u im Gegensatz zum naiven Verfahren, das bei der Wurzel beginnt.

Der Algorithmus M als Code

Schritt 1: Einfügen des Ortes von $head_{i+1}$

1 $v' =$ kontraktierter Ort von $head_i$ in T_{i-1}
$u = v'.suffix\text{-}zeiger$

2 **if** $\beta_i \neq \varepsilon$
then rescan β_i in T_i beginnend bei u

3 **if** Ort w von $\alpha_i\beta_i$ in T_i existiert
then Scan γ_{i+1} ausgehend von w
$(x, y) =$ letzte Kante
else $/*\ head_{i+1} = \alpha_i\beta_i$ (s.u.) $*/$
$x =$ kontraktierter Ort von $\alpha_i\beta_i$
$y =$ erweiterter Ort von $\alpha_i\beta_i$

4 Schaffe bei (x, y) einen inneren Knoten z für den Ort von $head_{i+1}$ und ein Blatt für den Ort von suf_{i+1}

rescan β_i in T_i : Folge einem Weg in T_i ausgehend von u, dessen Kantenbeschriftungen β_i ergeben.

Scan γ_{i+1} ausgehend von ω: Folge einem Weg in T_i von ω aus, dessen Kantenbeschriftungen mit suf_{i+1} übereinstimmen, bis man aus dem Baum bei der Kante (x,y) herausfällt.

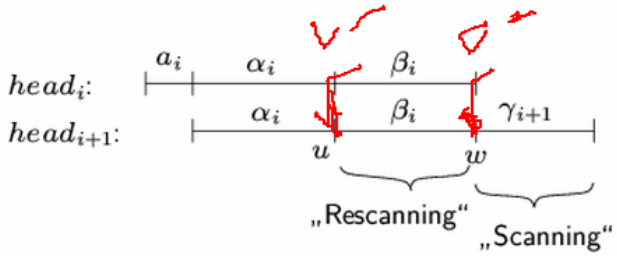
Schritt 2: Einfügen des Suffix-Zeigers für den Ort v von $head_i$

1 $u = v'.suffix\text{-}zeiger$

2 **if** $\beta_i \neq \varepsilon$
then rescan β_i in T_i bis zum Ort w von $\alpha_i\beta_i$
else $w = u$

3 $v =$ Ort von $head_i$
$v.suffix\text{-}zeiger = w$

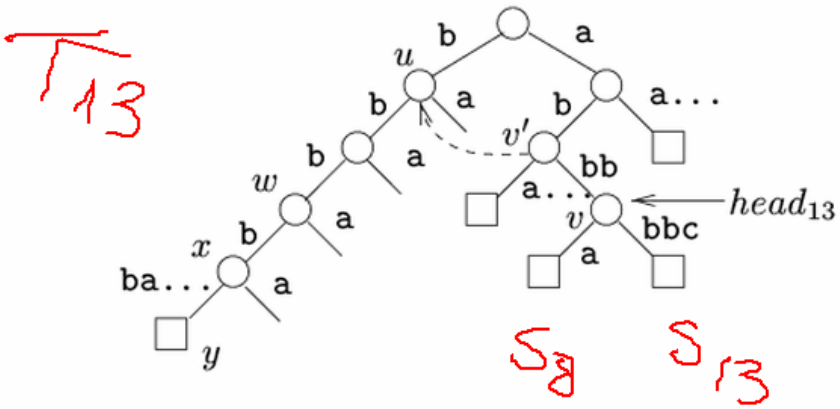
Implementation: Verflechtung von Schritt 1 und 2



Lemma 2: Falls der Ort von $\alpha_i\beta_i$ in T_i nicht existiert, dann ist $head_{i+1} = \alpha_i\beta_i$ d.h. $\gamma_{i+1} = \varepsilon$
⇨ OHNE BEWEIS !

Beispiel für das Suffix-Einfügen

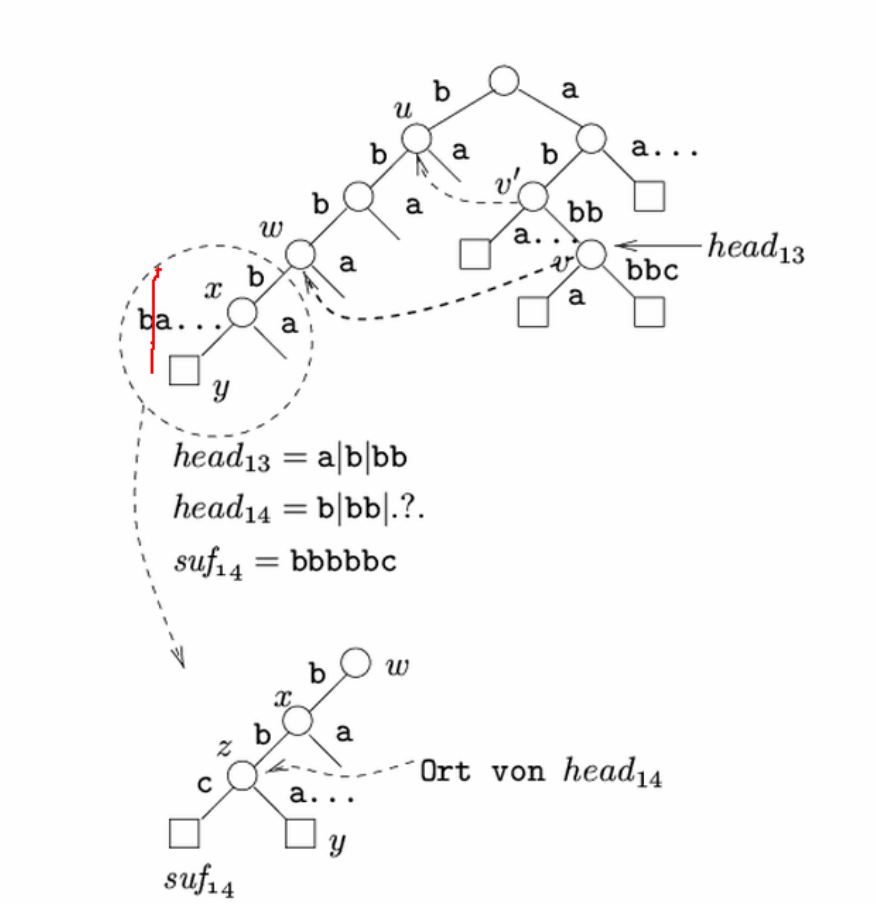
Beispiel: $\sigma = b^5abab^3a^2b^5c$. Konstruktion von T_{14} aus T_{13} durch Einfügen von $suf_{14} = bbbbbc$ in T_{13}.



$suf_{13} =$ a|b|bb|bbc

$suf_{14} =$ b|bb|bbc

finde w durch rescannen von bb


u
b
a
b
a
b
a...
v'
b
a
bb
w
a...
$head_{13}$
a
x
b
v
bbc
ba...
a
a
y
$head_{13} = \mathtt{a}|\mathtt{b}|\mathtt{bb}$
$head_{14} = \mathtt{b}|\mathtt{bb}|.?.$
$suf_{14} = \mathtt{bbbbbc}$
b
w
x
z
b
a
c
Ort von $head_{14}$
a...
y
suf_{14}

Analyse des Rescannens

$\alpha_i\beta_i$ ist Präfix von $head_{i+1}$: an einer Kante muß nur festgestellt werden, um wieviele Zeichen in β_i vorgerückt werden muss $\Rightarrow$ konstante Zeit pro besuchte Kante $\Rightarrow$
#(Schritte beim Rescannen) $= O(\#(\text{besuchte Kanten}))$

Sei

$$res_{i+1} := suf_{i+1} - \alpha_i = \beta_i\gamma_{i+1}tail_{i+1}.$$

($res_{i+1} \widehat{=}$ Zeichen, die ab Schritt $i+1$ überhaupt noch regescannt werden können).

Bei jeder Kante e, um die während des Rescannens von β_i vorgerückt wird, wird α_{i+1} um die Beschriftung δ_e der Kante e länger, d.h. δ_e ist in res_{i+1}, aber nicht in res_{i+2}.

(β_{i+1} ist der Teil von $head_{i+1}$ zwischen dem kontraktierten und dem erweiterten Ort von $head_{i+1}$)

$$|\delta_e| \geq 1 \Rightarrow |res_{i+2}| \leq |res_{i+1}| - k_{i+1}$$

mit $k_i = \#(\text{regesannte Kanten in Schritt } i)$. Somit

$$\sum_{i=1}^{n} k_i \leq \sum_{i=1}^{n} (|res_i| - |res_{i+1}|) = |res_1| - |res_{n+1}| \leq n$$

$\Rightarrow$ #(regesannte Kanten) $\leq n$.

Analyse des Scannens

#(gescannte Zeichen in Schritt $i+1$) $= |\gamma_{i+1}|$, mit

$$\begin{aligned} |\gamma_{i+1}| &= |head_{i+1}| - |\alpha_i\beta_i| \\ &= |head_{i+1}| - (|head_i| - 1)\,. \end{aligned}$$

wegen $head_i = a_i\alpha_i\beta_i$.

Also ist #(gescannte Zeichen) $=$

$$\begin{aligned} \sum_{i=1}^{n} |\gamma_i| &= \sum_{i=1}^{n} (|head_i| - |head_{i-1}| + 1) \\ &= n + |head_n| - |head_0| = n. \end{aligned}$$

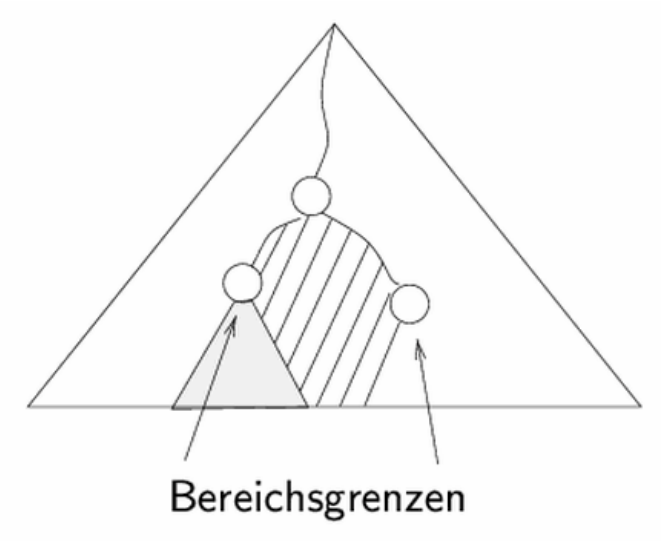
Analyse des Algorithmus M

Theorem 1: Algorithmus M liefert in Zeit O(|σ|) einen Suffix-Baum für σ mit |σ| Blättern in höchstens |σ|-1 inneren Knoten.

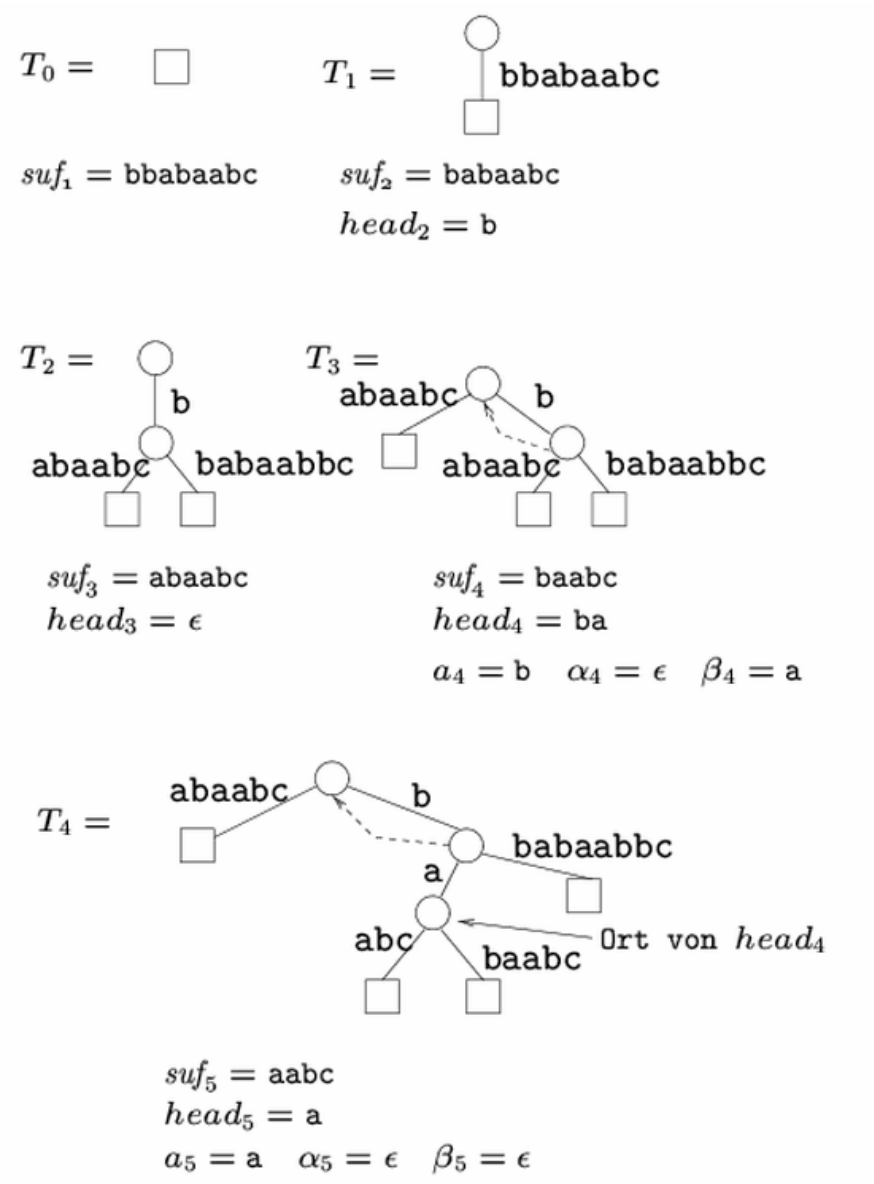
Suffix Baum Anwendung

Verwendung von Suffix-Baum T:

1. Suche nach Zeichenkette α: Folge dem Weg mit Kantenbeschriftung α in T in Zeit O(|α|). → Blätter des Teilbaumes $_{\text{entsprechen}}$ Vorkommen von α
2. Suche längstes, Doppelt auftretendes Wort: Finde Ort eines Wortes mit größter gewichteter Tiefe, der innerer Knoten ist.
3. Suche nach Präfix: Alle Vorkommen von Zeichenketten mit Präfix α finden sich in dem Teilbaum unterhalb des „Ortes“ von α in T.
4. Bereichssuche nach [α,β]:



Suffix-Baum-Beispiel



Eigenschaften von Suffix-Bäumen

Suffix-Bäume sind ein Suchindex für einen Text $\sigma \in \Sigma^n$ für eine Suche nach einem Muster $\alpha \in \Sigma^p$

Eigenschaften:
Teilwortsuche, Anfragen an σ selbst, Präfix-Suche, Bereichs-Suche, Kostengünstige Konstruktion.

Kosten:

1. Konstruktion in O(n log |Σ|) Zeit und O(n) Platz, Suche in O(p log |Σ|) Zeit
2. Konstruktion in O(n) Zeit und O(n|Σ|) Platz, Suche in O(p) Zeit

Nachteile:

- Komplexe Datenstruktur => viele Bytes/Zeichen

- Komplexe Algorithmen

Vorlesung 20 – Suche in Texten - „Suffix Arrays“

Suffix-Array:

- Eine sortierte Liste aller Suffixe eines Textes,
- Plus Zusatzinformationen über längste gemeinsame Prefixe (lcps = longest common prefixes) zur Beschleunigung der Suche.

Eigenschaften:

- Teilwortsuche in O(p + log n) Zeit, genauer: $\leq p + \lceil \log_2(n-1) \rceil$ Vergleiche zwischen Zeichen
- Zusätzlicher Speicherbedarf: 2n Ganzzahlen (Integers)
- Konstruktionszeit: O(n log n) (worst case) bzw. O(n) (average case)

Definitionen: Sei

$$
\begin{array}{lcll}
A & = & a_0\, a_1 \ldots a_{n-1} & \in \Sigma^n \quad \text{Text} \\
A_i & = & a_i \ldots a_{n-1} & \in \Sigma^{n-i} \quad i\text{-tes Suffix } (i \leq n-1) \\
A_{i,j} & = & a_{\max(0,i)} \ldots a_{\min(j,n-1)} & \text{Teilwort} \\
pos & = & [i_0, \ldots, i_{n-1}] & \text{mit } pos[k] \in \{0, \ldots, n-1\} \text{ und} \\
A_{pos[k]} & = & \text{das } k\text{-kleinste Suffix von } A &
\end{array}
$$

Dann gilt: $A_{pos[0]} < A_{pos[1]} < \cdots < A_{pos[n-1]}$

Weiter sei:

$$A <_p B \;:\Longleftrightarrow\; A_{0,p-1} < B_{0,p-1}$$

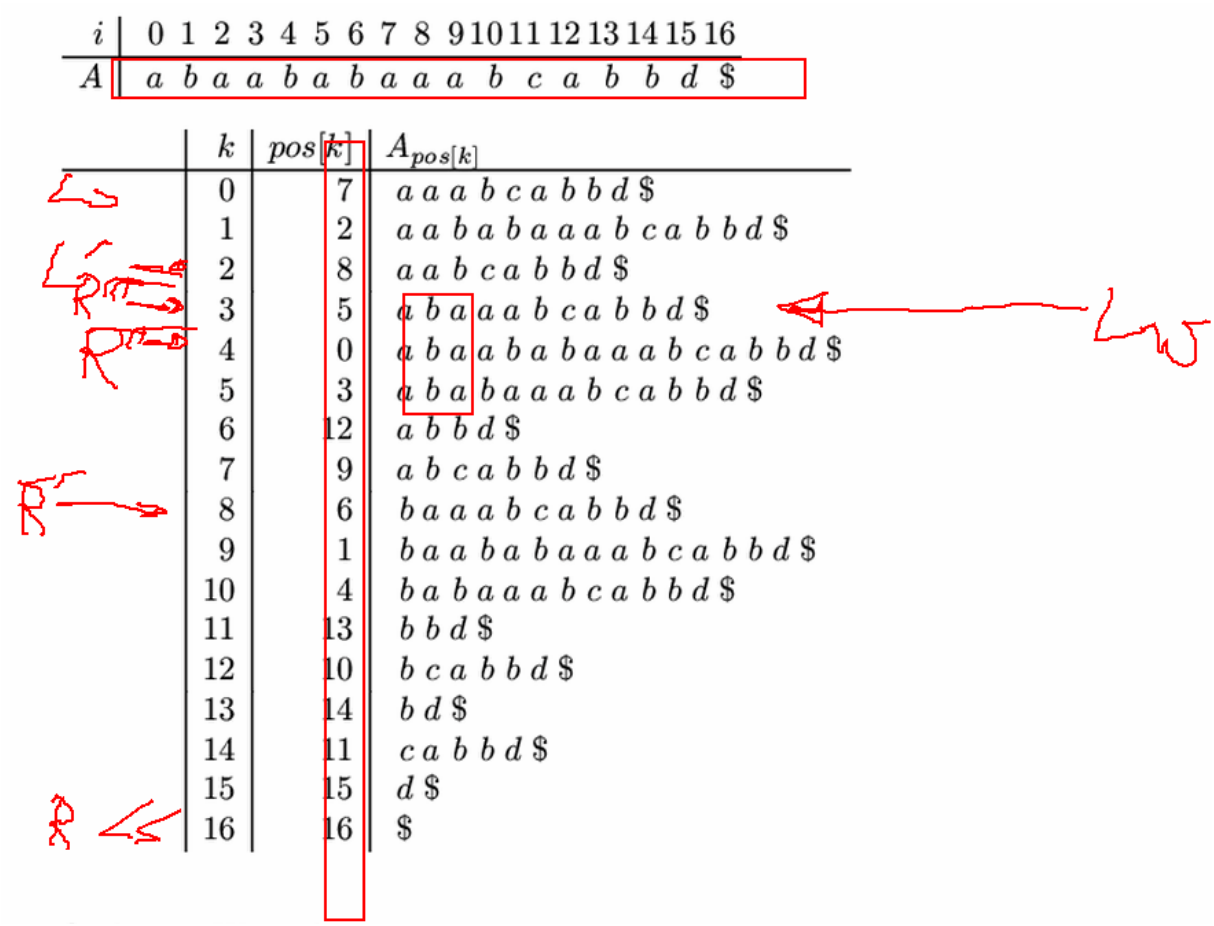
analog: $=_p, \leq_p, >_p, \geq_p, \neq_p$

Fakt: pos ist auch bzgl. $\leq_p$ sortiert:

$$A_{pos[0]} \leq_p A_{pos[1]} \leq_p \cdots \leq_p A_{pos[n-1]}$$

Alle Prefixe zu einem gesuchten Anfang W mit Länge p liegen im pos-Array hintereinander.

Beispiel zum Suffix-Array



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	a	b	a	a	b	a	b	a	a	a	b	c	a	b	b	d	$

k	$pos[k]$	$A_{pos[k]}$
0	7	a a a b c a b b d $
1	2	a a b a b a a a b c a b b d $
2	8	a a b c a b b d $
3	5	a b a a a b c a b b d $
4	0	a b a a b a b a a a b c a b b d $
5	3	a b a b a a a b c a b b d $
6	12	a b b d $
7	9	a b c a b b d $
8	6	b a a a b c a b b d $
9	1	b a a b a b a a a b c a b b d $
10	4	b a b a a a b c a b b d $
11	13	b b d $
12	10	b c a b b d $
13	14	b d $
14	11	c a b b d $
15	15	d $
16	16	$

Suchwort: W = aba

Prefix-Suche im Suffix-Array

Alle Prefixe zu einem gesuchten Anfang W mit Länge p liegen im pos-Array hintereinander.

Suche die untere Grenze L_W für das Vorkommen von Prefix W mit |W| = p im pos-Array:

$$L_W := \min\{k \mid W \leq_p A_{pos[k]} \vee k = n\}$$

Binäre Suche nach L_W im pos-Array:
Zeitbedarf: O(p log n) da immer p Buchstaben in jedem Schritt der binären Suche verglichen werden müssen.

Es geht noch besser, und zwar in O(p + log n) was wir im folgenden noch besprechen werden.

Algorithmus L_W-Search

Input: Wort $W = w_0 \dots w_{p-1}$ mit $p \leq n$
Output: $L_W = \min\{k \mid W \leq_p A_{pos[k]} \vee k = n\}$

```
1    p := length(W)
2    if W ≤_p A_pos[0] then L_W := 0
3    else if W >_p A_pos[n-1]
4       then L_W := n
5       else (L, R) := (0, n − 1)
6            while L − R > 1 do
7               M := (L + R)/2
8               if W ≤_p A_pos[M] then R := M
9                  else                L := M
11           end while
12           L_W := R
14   return L_W
```

1. Verbesserung von L_W-Search

Idee:
Spare Vergleiche zwischen Zeichen, deren Ergebnis bekannt ist.

Für $v, w \in \Sigma^*$ sei:

$$lcp(v, w) := \max\{p \mid v =_p w\}$$

$lcp(v, w)$: Länge des längsten gemeinsamen Prefixes von v, w

In L_W-search sei

$\ell = lcp(A_{pos[L]}, W)$ zuerst: $\ell = lcp(A_{pos[0]}, W)$
$r = lcp(A_{pos[R]}, W)$ zuerst: $r = lcp(A_{pos[n-1]}, W)$
$h = \min\{\ell, r\}$

Zeile 2
Zeile 3

Dann gilt: $A_{pos[L]} =_h W =_h A_{pos[R]}$

⇨ Die h ersten Zeichen brauchen nicht mehr verglichen zu werden.

Das worst case-Verhalten wird nicht geändert.

2. Verbesserung von L_W-Search

Idee:
Berechne und nutze Zusatzinformationen über

$lcp(A_{pos[M]}, A_{pos[L]})$: $Llcp$
$lcp(A_{pos[M]}, A_{pos[R]})$: $Rlcp$

statische Arrays

$k = M$	L	$Llcp$	$pos[k]$	$A_{pos[k]}$
0			7	$a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
1	0	2	2	$a\,a\,b\,a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
2	0	2	8	$a\,a\,b\,c\,a\,b\,b\,d\,\$$
3	2	1	5	$a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
4	0	1	0	$a\,b\,a\,a\,b\,a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
5	4	3	3	$a\,b\,a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
6	4	2	12	$a\,b\,b\,d\,\$$
7	6	2	9	$a\,b\,c\,a\,b\,b\,d\,\$$
8	0	0	6	$b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
9	8	3	1	$b\,a\,a\,b\,a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
10	8	2	4	$b\,a\,b\,a\,a\,a\,b\,c\,a\,b\,b\,d\,\$$
11	10	1	13	$b\,b\,d\,\$$
12	8	1	10	$b\,c\,a\,b\,b\,d\,\$$
13	12	1	14	$b\,d\,\$$
14	12	0	11	$c\,a\,b\,b\,d\,\$$
15	14	0	15	$d\,\$$
16			16	$\$$

Berechne Llcp und Rlcp bei der Sortierung des pos-Arrays.

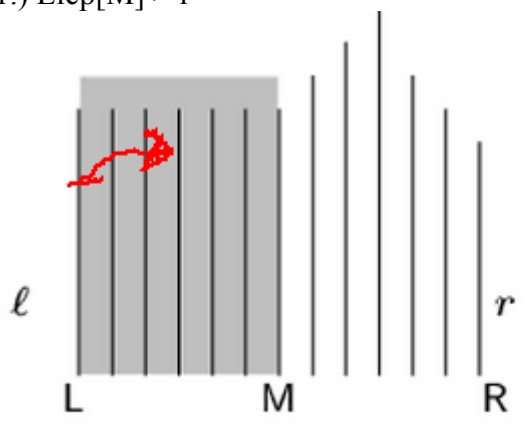
Nutzen von Llcp und Rlcp in L_W-Search

Sei
H = max{l,r}
ΔH = Differenz zwischen H am Anfang und Ende einer Iteration

Im Folgenden sei o.B.d.A.: r ≤ l = H: Llcp wird benutzt.

Unterscheide 3 Fälle:

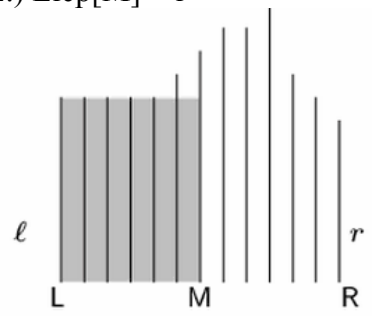
1.) Llcp[M] > l



$A_{pos[M]} =_{\ell+1} A_{pos[L]} \neq_{\ell+1} W$

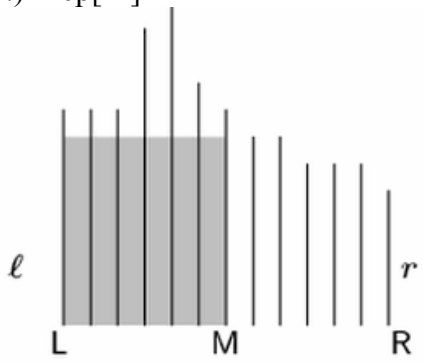
L_W gehört in rechte Hälfte, l nicht geändert

In diesem Fall sind somit keine Zeichenvergleiche nötig.
2.) Llcp[M] = l



$A_{pos[M]} =_\ell W$
Vergleiche Zeichen $\ell + 1 \cdots \ell + j$ bis
$A_{pos[M]} \neq_{\ell+j} W$.
L_W liegt links oder rechts, setze entsprechend
$r = \ell + j - 1$ bzw. $\ell = \ell + j - 1$.
Zeichenvergleiche: $\Delta H + 1$

3.) Llcp[M] < l



$W =_\ell A_{pos[L]}$ und
$W =_{\ell'} A_{pos[M]}$ mit $\ell' < \ell$
L_W gehört in linke Hälfte, $r := Llcp[M]$

⇨ Durch Nutzung von Llcp und Rlcp werden pro Iteration ΔH+1 Vergleiche benötigt.

Mit $\Sigma\Delta H \leq p$ folgt:

Satz:
Die Anzahl der Symbolvergleiche beträgt höchstens $p + \lceil \log_2(n-1) \rceil$.

Vorlesung 21 – Textcodierungen

Kompressionsverfahren für Texte

Ziel:
Finde umkehrbare Codierung, so, dass der Text wieder (verlustfrei) rekonstruiert werden kann

Beispiele:
- ⇨ Huffman-Code
- ⇨ Lauflängen Codierung
- ⇨ Arithmetische Codierung
- ⇨ Lempel-Ziv Codierung

Gegensatz sind die verlustbehafteten Kompressionen, z.B. für Bilder, Audio oder Video
- ⇨ JPeg
- ⇨ MPeg
- ⇨ MP3

Arithmetische Codierung

Ziel bei der verlustfreien Kompression ist es, die Anzahl der Bits der codierten Nachricht umkehrt proportional zur Wahrscheinlichkeit des Auftretens von Zeichenketten zu codieren.

d.h.:
- ⇨ Wahrscheinlichkeit für Zeichenkette groß → Anzahl der Bits soll klein sein
- ⇨ Wahrscheinlichkeit für Zeichenkette klein → Anzahl der Bits groß

Eine Zeichenkette kann auch ein einzelnes Zeichen sein.

Beispiel Huffman-Code:

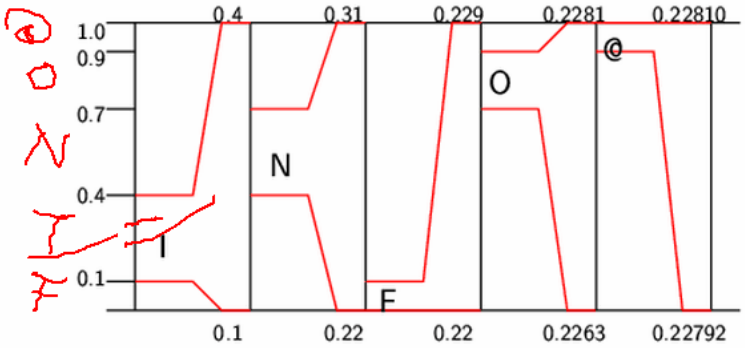
Symbol	p(Symbol)	Code
A	0.01	1
B	0.99	0

$$AAAA \rightarrow 1111$$
$$BBBB \rightarrow 0000$$

Idee:
Stelle Symbole und Symbolfolgen als Intervall [l,r) bzw. als Element daraus dar.

Beispiel:

Symbol	p(Symbol)	Intervall
F	0.1	$[0.0, 0.1)$
I	0.3	$[0.1, 0.4)$
N	0.3	$[0.4, 0.7)$
O	0.2	$[0.7, 0.9)$
@	0.1	$[0.9, 1.0)$



$$[x, y) \xrightarrow{[u,o)} [x + (y - x)\, u, x + (y - x)\, o)$$

Gestartet wird mit dem Intervall: [0,1)

Decodierung durch Umkehrung:

$$[v, w) \xrightarrow{[u,o)} [(v - u)/(o - u), (w - u)/(o - u))$$

Beispiel:

$$\begin{aligned}
& [0.22792, 0.2281) \\
[0.1,0.4):I \longrightarrow\ & [0.4264, 0.427) \\
[0.4,0.7):N \longrightarrow\ & [0.088, 0.09) \\
[0.0,0.1):F \longrightarrow\ & [0.88, 0.9) \\
[0.7,0.9):O \longrightarrow\ & [0.9, 1.0) \\
[0.9,1.0):@ \longrightarrow\ & [0.0, 1.0)
\end{aligned}$$

Es reicht natürlich ein wert aus dem Intervall, z.B. der untere.

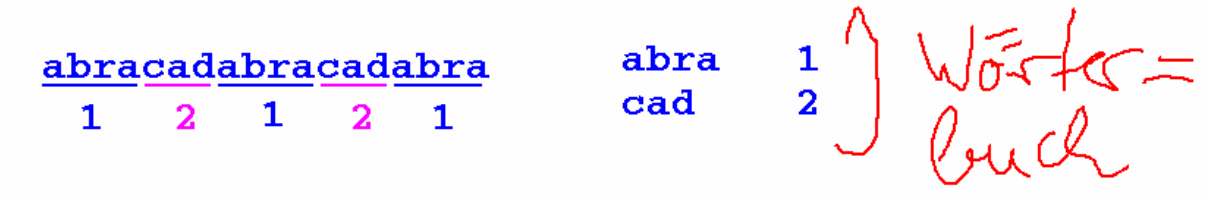
Die Gesamtnachricht wird zur Codierung in Teilzeichenfolgen zerlegt, die durch Stoppzeichen beendet werden.

Nachteil:

- ⇨ Statisches Verfahren, passt sich nicht wechselnden Wahrscheinlichkeiten an.
- ⇨ Aufwendige Berechnungen

Lempel-Ziv Codierung

Einfaches verlustfreies Kompressionsverfahren
Ersetze häufig auftretendes Muster durch kurzes Codewort, verwende Wörterbuch für die Codeworte



Wörterbuchbasiertes Kompressionsverfahren:
- ⇨ Statisch: Wörterbuch wird vor Codierung festgelegt und bleibt unverändert
- ⇨ Dynamisch: Wörterbuch passt sich dem zu komprimierenden Text dynamisch an.

Lempel-Ziv: zip, Tiff
Lempel-Ziv-Welsh: Compress in Unix

Lempel-Ziv Idee: Baue das Wörterbuch simultan mit der Kodierung des Textes auf; anfangs seien für jeden Buchstaben des Alphabets Codenummern im Wörterbuch.

Kodierung von: a b a b c b a b a b ... $\Sigma = \{a, b, c\}$

Wörterbuch Eintrag	#
a	1
b	2
c	3
ab	4
ba	5
abc	6
cb	7
bab	8
:	

Ausgabe
1
2
4
3
5
8
:

Decodierung von: 1 2 4 3 5 8

Wörterbuch Eintrag	#
a	1
b	2
c	3
ab	4
ba	5
abc	6

Ausgabe
a
b
ab
c
:

Implementierung des Algorithmus „Lempel-Ziv“

Input: Ein Text T;
Output: Die Lempel-Ziv Kodierung von T

```
1   initialisiere Wörterbuch D
2   bestimme aktuellen Match m von D in T
3   gebe #(m) aus
4   T := T − m; m_last := m;
5   while T ≠ ϵ do
6       bestimme aktuellen Match m = am'
7       gebe #(m) aus
8       füge m_last a zu D hinzu
9       T := T − m; m_last := m;
    end while;
```

Weiteres Beispiel für Lempel-Ziv Codierung:

$T =$ COCOA AND BANANAS

Wörterbuch D (anfangs)

m	$\#(m)$	m	$\#(m)$
A	000 001	O	001 111
B	000 010	⋮	⋮
C	000 011	S	010 011
D	000 100	⋮	⋮
⋮	⋮	Z	011 010
N	001 110		011 011

$T =$ COCOA AND BANANAS

m	$\#(m)$	add to D		T
C	000 011	—		OCOA AND BANANAS
O	001 111	CO	011 100	COA AND BANANAS
CO	011 100	OC	011 101	A AND BANANAS
A	000 001	COA	011 110	AND BANANAS
	011 011	A	011 111	AND BANANAS
A	000 001	A	100 000	ND BANANAS
N	001 110	AN	100 001	D BANANAS
D	000 100	ND	100 010	BANANAS
	011 011	D	100 011	BANANAS
B	000 010	B	100 100	ANANAS
AN	100 001	BA	100 101	ANAS
AN	100 001	ANA	100 110	AS
A	000 001	ANA	100 110	S
S	010 011	AS	100 111	

$\#(T) = 000\,011|001\,111|011\,100|000\,001|011\,011|\ldots$

Lempel-Ziv-Welsh Eigenschaften

- ⇨ Wörterbuch passt sich dynamisch an die zu komprimierende Zeichenkette an, d.h. es enthält schließlich die am häufigsten vorkommenden Zeichenketten.
- ⇨ Wörterbuch (Code Tabelle) muss nicht übertragen werden. Bekannt sein muss nur die Anfangstabelle, alles weitere wird beim Decodieren dynamisch erzeugt.
- ⇨ Codieren und Decodieren ist in linearer Zeit möglich
- ⇨ Lempel-Ziv führt i.a. zu höheren Kompressionsraten als der Huffman-Code

Heuristische Optimierungsverfahren

Vorlesung 22 – Genetische Algorithmen

Beispiel: 0/1 Mehrfach-Rucksack-Problem

→ Kombinatorisches Optimierungsproblem (KOP)

Problem Instanz:

Rucksäcke		1	2	…	m		
Kapazitäten		c_1	c_2	…	c_m	;	$c_j>0$
Objekte		1	2	…	n		
Nutzen	p_1	p_2	…	p_n	;	$p_i>0$	
Gewichte		w_{1j}	w_{2j}	…	w_{nj}	;	$w_{ij}\geq 0,\ 1 \leq j \leq m$

Zulässige Lösung: $x= (x_1,\ldots,x_n) \in \{0,1\}^n$ mit

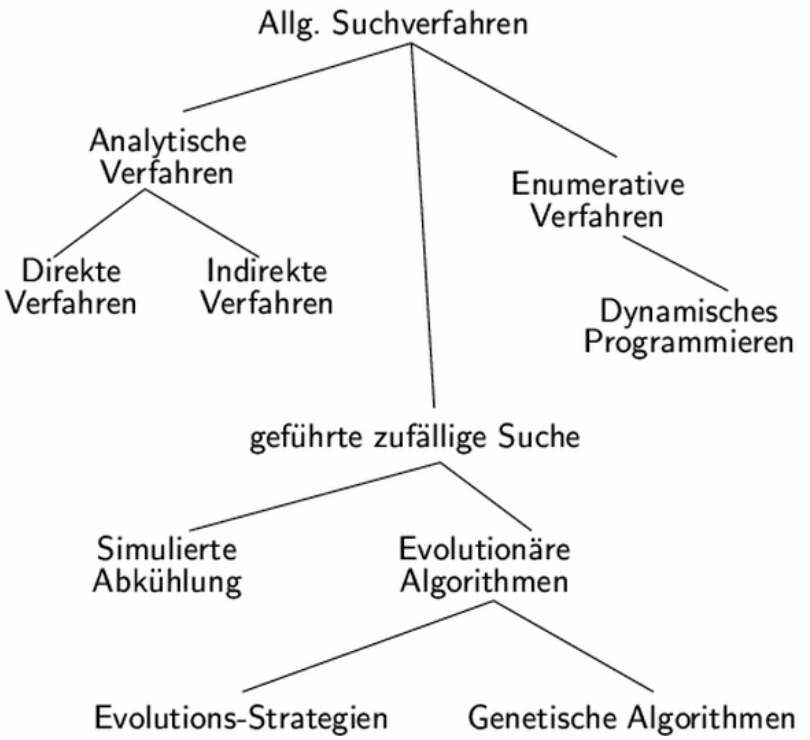
$$\sum_{i=1}^{n} w_{ij}\, x_i \ \leq\ c_j \qquad (\forall j \in \{1,\ldots,m\})$$

Zielfunktion: $P(\mathbf{x}) = \sum_{i=1}^{n} p_i\, x_i$

Optimale Lösung: Zulässige Lösung x mit maximalem Nutzen P(x)

Größe des Lösungsraumes: 2^n

Suchverfahren für kombinatorische Optimierungsprobleme (KOP)



Der Simple genetische Algorithmus (SGA)

```
SGA :=
      proc ()
 (1)    t := 1;   // Generationszähler
 (2)    G_t := initiale Population mit |G_t| = μ gerade;
 (3)    while true do
 (4)      ∀x ∈ G_t do berechne ν(x) od;   // Bewertung
 (5)      ∀x ∈ G_t do berechne f(x) aus ν(x) od ; // Fitness
 (6)      if fertig (G_t) then break fi;
 (7)      G_{t+1/2} := Selektion (G_t);   // Zwischengeneration
 (8)      t := t + 1;
 (9)      G_t := leere Population;
(10)      while |G_{t-1/2}| ≠ 0 do
(11)        entferne zufällig x, y aus G_{t-1/2};
(12)        mit W'keit p_c do   // p_c = Kreuzungsrate
(13)          ersetze (x, y) durch Kreuzungen (x, y) od;
(14)        füge x, y zu G_t
(15)      od;
(16)      ∀x ∈ G_t, ∀b ∈ x do
(17)        mit W'keit p_m do   // p_m = Mutationsrate
(18)          b := Mutation (b) od
(19)      od
(20)    od;
(21)    gib x ∈ G_t mit maximalem f(x) aus;
      end;
```

Legende zum SGA:

Individuum $x \in G_t$:

- ⇨ beschreibt zulässige oder unzulässige Problemlösung, die als String über Alphabet Σ codiert wird, Chromosom; oft gilt: Σ = {0,1}

Generation/Population:

- ⇨ Liste von μ Individuen; etwa: μ = 50

Bewertung ν(x):

- ⇨ Anwendungsabhängige, zu optimierende oder minimierende Zielfunktion

Fitness f(x):

- ⇨ nicht-negativ, abgeleitet von ν(x), korreliert positiv mit # Reproduktionsmöglichkeiten φ(x) von x

Selektion:

- ⇨

bestimme $\varphi(\mathbf{x})$ aus $f(\mathbf{x})$, so dass $\sum_{\mathbf{x}\in G_t}\varphi(\mathbf{x}) = \mu$;
$\lfloor\varphi(\mathbf{x})\rfloor$: sichere # Kopien von x in $G_{t+1/2}$,
$\varphi(\mathbf{x}) - \lfloor\varphi(\mathbf{x})\rfloor$: W'keit für eine weitere Kopie; etwa
$\varphi(\mathbf{x}) := f(\mathbf{x})/\bar{f}$ mit $\bar{f} := 1/\mu \sum_{\mathbf{x}\in G_t} f(\mathbf{x})$. Bsp.:

i	1	2	3	4	
$\nu(\mathbf{x}^i)$	1	0	−1	0	min
$f(\mathbf{x}^i)$	1	2	3	2	$2-\nu(\mathbf{x}^i)$
$\varphi(\mathbf{x}^i)$	0.5	1	1.5	1	$f(\mathbf{x})/\bar{f}$

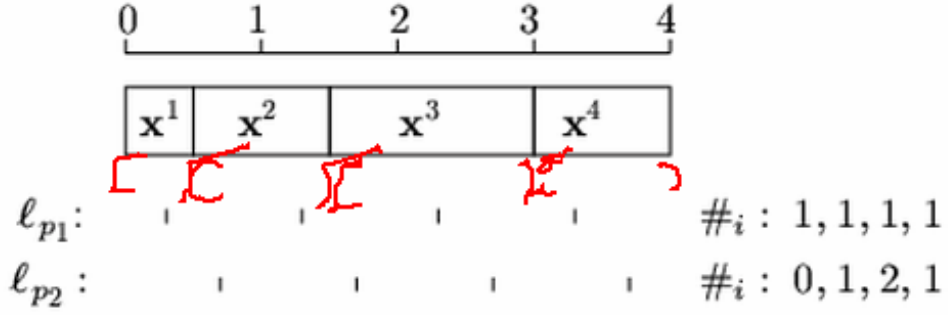
$\bar{f}$ =2

Ermittlung der # Kopien #$_i$ von x^i:

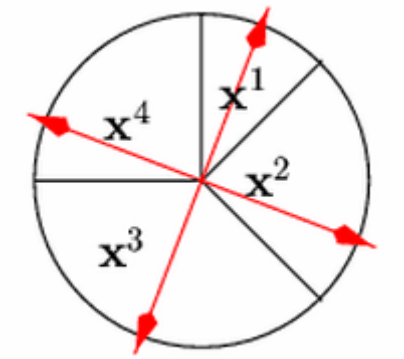
⇨ Wähle p ∈ [0,1) zufällig, sei $l_p := \{p,p+1,\ldots,p+\mu-1\}$, dann sei

$$\#_i = \left|\ell_p \cap \left[\sum_{j=1}^{i-1}\varphi(\mathbf{x}^j), \sum_{j=1}^{i}\varphi(\mathbf{x}^j)\right)\right|$$

Beispiel:



„gewichtetes Roulettrad“



Kreuzung zweier Individuen

⇨ $\mathbf{x} = (x_1,\ldots,x_n)$ $\mathbf{y} = (y_1,\ldots,y_n)$

in Teilsequenzen und Rekombination zu x' und y', etwa

- 1-Punkt Kreuzung:
  - wähle l ∈ {1,…,n-1} zufällig, und

$$\mathbf{x}' = (x_1, \ldots, x_\ell, y_{\ell+1}, \ldots, y_n)$$
$$\mathbf{y}' = (y_1, \ldots, y_\ell, x_{\ell+1}, \ldots, x_n)$$

- 2-Punkt Kreuzung:
  - wähle l,r ∈ {1,…,n}, l,r zufällig, und

$$\mathbf{x}' = (x_1, \ldots, x_\ell, y_{\ell+1}, \ldots, y_r, x_{r+1}, \ldots, x_n)$$
$$\mathbf{y}' = (y_1, \ldots, y_\ell, x_{\ell+1}, \ldots, x_r, y_{r+1}, \ldots, y_n)$$

- **gleichmäßige Kreuzung:**
  - **erzeuge zufällige Bit-Maske b = (b_1,…,b_n) und**

$$\mathbf{x}' = \mathbf{b} \,\&\, \mathbf{x} \mid {\sim}\mathbf{b} \,\&\, \mathbf{y}$$
$$\mathbf{y}' = \mathbf{b} \,\&\, \mathbf{y} \mid {\sim}\mathbf{b} \,\&\, \mathbf{x}$$

Mutation

⇨ Jede Komponente b eines Strings (Bits) wird mit Wahrscheinlichkeit p_m verändert (gekippt); z.B. $p_m = 0.1\%$ oder $p_m = 1/n$

Vorgehensweise bei der Anwendungserstellung

Gegeben: KOP
Gesucht: Lösung für KOP

Vorgehensweise:

- Wähle Codierung für potentiell Lösungen
- Wähle Bewertungs-, Fitness-Funktion
- Wähle Parameter μ, p_c, p_m, Abbruchbedingung,
- Implementiere Verfahren, z.B. mit Hilfe von Tools/Bibliotheken

Anwendungsbeispiel 1: Subset Sum-Problem

Problem Instanz:

$$\mathbf{w} = (w_1, \ldots, w_n) \subset I\!N^+, C \in I\!N^+$$

Zulässige Lösung:

$$\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n \quad \text{mit}$$
$$\sum_{i=1}^{n} w_i x_i \leq C$$

Zielfunktion:

$$P(\mathbf{x}) = \sum_{i=1}^{n} w_i \, x_i$$

Optimale Lösung: Zulässige Lösung x mit maximalem P(x)

Codierung gegeben, aber wie mit unzulässigen Lösungen umgehen?

Idee: unzulässige Lösungen zulassen für einen Preis (Strafe), Strafterm

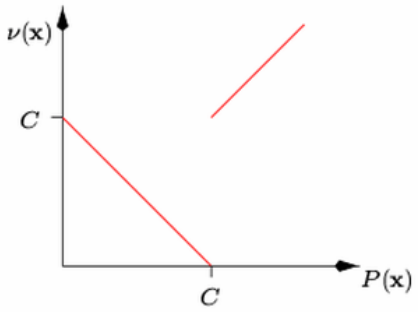
- Strafterm ist eine Funktion des Abstandes von zulässiger Region
- Der beste unzulässige Vektor ist nie besser als der schlechteste zulässige

⇨ Kurze Überlebenszeit der unzulässigen Vektoren

Formuliere Subset Sum als Minimierungsproblem:

$$\nu(\mathbf{x}) = s(\mathbf{x})\,(C - P(\mathbf{x})) + (1 - s(\mathbf{x}))\,P(\mathbf{x}) \qquad \text{mit}$$

$$s(\mathbf{x}) = \begin{cases} 1 & \text{falls } \mathbf{x} \text{ zulässig} \\ 0 & \text{sonst} \end{cases}$$



Definiere Fitness:

$$f(\mathbf{x}) = \sum_{i=1}^{n} w_i - \nu(\mathbf{x})$$

Tests mit n = 1000 zeigen: Nach 20 Generationen (also 20000 Bewertungen) wird in 95% der Fälle das Optimum gefunden.

Anwendungsbeispiel 2: Maximum Cut-Problem

Problem Instanz:

$G = (V, E)$	gewichteter Graph
$V = \{1, \ldots, n\}$	Knoten
w_{ij}	Gewicht der Kante $(i, j) \in E$ mit $w_{ij} = w_{ji}$; $w_{ii} = 0$

Zulässige Lösung:

$$V_0, V_1 \subseteq V \text{ mit } V_0 \cap V_1 = \emptyset, V_0 \cup V_1 = V$$

$$C := \{(v, v') \in E \mid v \in V_0, v' \in V_1\}$$

Zielfunktion

$$W(C) = \sum_{(i,j)\in C} w_{ij}$$

Optimale Lösung: Zulässig Lösung C mit W(C)
Codierung von V_0, V_1 durch x = $\{x_1,\ldots,x_n\}$, so dass

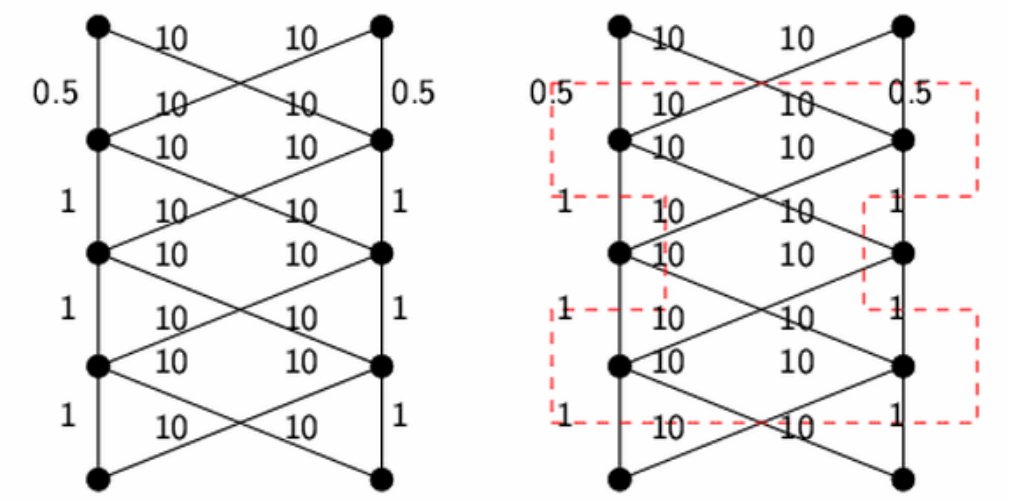
$$x_i = 1 \Leftrightarrow i \in V_1$$

Formuliere Maximum Cut als Maximimierungsproblem:

$$\nu(\mathbf{x}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij} \left[x_i (1 - x_j) + x_j (1 - x_i)\right]$$

Anmerkung: Da ν(x) immer positiv ist, kann man f(x) = ν(x) setzen

Test Beispiel (n=10)



f_{opt}= 21 + 11(n-4)

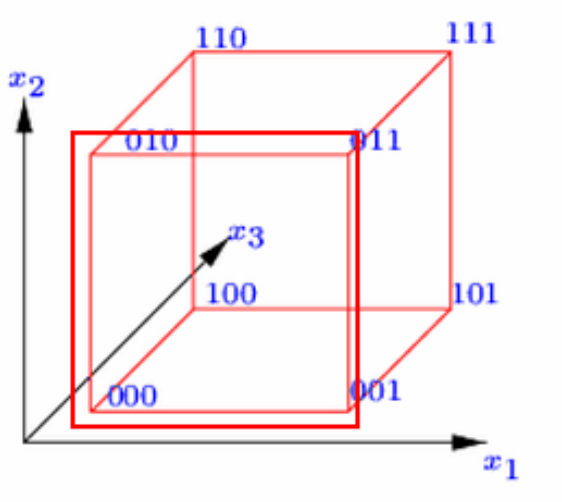
Tests mit n=100 zeigen: Nach 1000 Generationen (also 50 000 Bewertungen)
Ist die gefundene Lösung im Schnitt nur 5% vom Optimum entfernt.

Nur $(5*10^4)/2^{100} \approx 4 *10^{-24}$ % des Suchraumes musste exploriert werden!

Selektion + Rekombination = Innovation

Iterierte Abtastung von Hyperebenen-Partitionen

Bsp.: $x \in \{0,1\}^3$:



Vordere Fläche (des Hypercubes): 0**

Schema

⇨ String aus {0,1,*}, mind. Ein nicht-*-Zeichen

Ordnung o(s) eines Schemas s

⇨ # der nicht-*-Zeichen in s

Bsp: o(1**1*****0**) = 3

Bit-String x passt zu Schema s,x ∈ s, wenn s durch Verändern von *-Zeichen in x verwandelt werden kann

⇨ alle x, die zu s passen, liegen in Hyperebene

Jedes $\mathbf{x} \in \{0,1\}^n$ gehört zu $\sum_{k=0}^{n-1} \binom{n}{k} = 2^n - 1$ Hyperebenen.

Es gibt insgesamt $3^n - 1$ verschiedene Hyperebenen

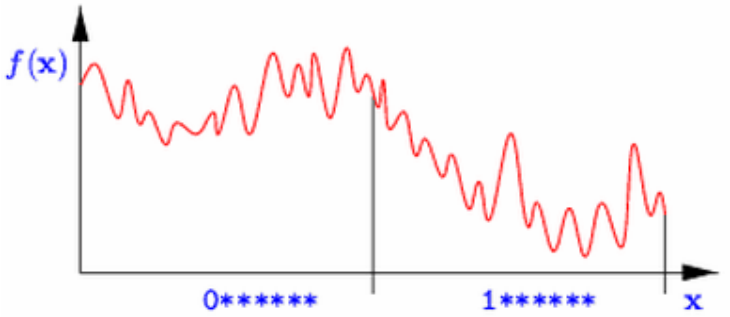
Zufällig gewählte Population der Größe μ enthält oft viele Exemplare, die zu Schema s mit kleinem o(s) passen, im Schnitt:

$$\frac{\mu}{2^{o(s)}}$$

Impliziter Parallelismus: Evaluation von x liefert Informationen über 2^n-1 Schemata

Konkurrenz der Schemata: GA tastet Schemata ab, # Repräsentationen guter Schemata in Folgegeneration wird erhöht.

Veranschaulichung, Beispiel:



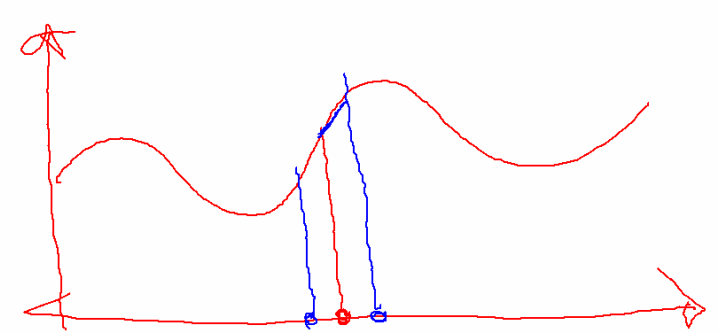
Individuen x mit $x_1 = 0$ sind im Schnitt besser als die mit $x_1 = 1$ und daher in Folgegenerationen häufiger vertreten.

⇨ Abtastrate in Bereichen mit $f(x) > \bar{f}$ wird erhöht.
(Konvergenz zum Optimum hin ist nicht garantiert)

Selektion + Mutation = stetige Verbesserung

Mutation: kleine Änderungen an Individuen, kann neue Information generieren

Ausschließliche Verwendung von Selektion und Mutation funktioniert ähnlich wie Hillclimbing-Verfahren.



Vorlesung 23 – Genetische Algorithmen

Das Schema Theorem

Untere Abschätzung der erwarteten Änderung der Abtastrate für ein Schema s beim Übergang von G_t nach G_{t+1} (Holland 1975):

$$M(\mathbf{s},t) := |\{\mathbf{x} \in G_t \mid \mathbf{x} \in \mathbf{s}\}|$$

$$M(\mathbf{s},t+1/2) = M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}} \quad \text{mit}$$

$$f(\mathbf{s},t) := \frac{1}{M(\mathbf{s},t)} \sum_{\mathbf{x}\in G_t,\mathbf{x}\in\mathbf{s}} f(\mathbf{x})$$

Definierende Länge Δ(s) eines Schemas s ist Differenz zwischen max. und min. Index eines nicht-*-Zeichens

Bsp: Δ(****1**0**10**) = 12 - 5 = 7

Wahrscheinlichkeit dafür, dass bei 1-Punkt-Kreuzung Schnittpunkt in s liegt:

$$\Delta(\mathbf{s})/(n-1)$$

$$M(\mathbf{s},t+1) = (1-p_c)\,M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}} + p_c\left[M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}(1-\mathcal{V})+\mathcal{G}\right]$$

Konservative Annahmen:

- Jede Kreuzung innerhalb Δ(s) wirkt zerstörend: ν
- Gewinne sind vernachlässigbar: G = 0

$$\begin{aligned} M(\mathbf{s},t+1) &\geq (1-p_c)\,M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}} \\ &\quad + p_c\left[M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}(1-d)\right] \\ &= M(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}\,[1-p_c\,d] \qquad (*) \end{aligned}$$

Bemerkung: Werden x,y ∈ s rekombiniert, findet kein Verlust statt.

P(s,t) := M(s,t) / μ

Ist Wahrscheinlichkeit dafür, das für zufällig gezogenes $x \in G_t$ gilt: $x \in s$

$$d := \frac{\Delta(\mathbf{s})}{n-1}\,(1-P(\mathbf{s},t))$$

(Einsetzen in (*) und durch μ teilen =>)

Erste Version des Schema-Theorems:

$$P(\mathbf{s},t+1) \geq P(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}\left[1-p_c\,\frac{\Delta(\mathbf{s})}{n-1}\,(1-P(\mathbf{s},t))\right]$$

Unter Berücksichtigung, dass anderer Partner von $x \in s$ bei Kreuzung mit Verlust aus $G_{t+1/2}$ stammt, folgt:

$$\tilde{d} := \frac{\Delta(\mathbf{s})}{n-1}\left(1-P(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}\right)$$

⇨ zweite Version des Schema-Theorems:

$$P(\mathbf{s},t+1) \geq P(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}\left[1-p_c\,\frac{\Delta(\mathbf{s})}{n-1}\left(1-P(\mathbf{s},t)\,\frac{f(\mathbf{s},t)}{\bar{f}}\right)\right]$$

Unter Berücksichtigung der zerstörerischen Wirkung der Mutation:

$$P(\mathbf{s},t+1) \geq P(\mathbf{s},t)\frac{f(\mathbf{s},t)}{\bar{f}}\left[1-p_c\frac{\Delta(\mathbf{s})}{n-1}\left(1-P(\mathbf{s},t)\frac{f(\mathbf{s},t)}{\bar{f}}\right)\right] \cdot (1-p_m)^{o(\mathbf{s})}$$

Parameterlose GA

Ziel: Codieren des Problems & Knopf drücken => Lösung (d.h. keine explizite Einstellung der Parameter)

- Verwendung von geschachtelten GA
  - Meta-GA sucht nach optimalen Werten für μ, p_m, p_c, …
  - Innere GA lösen eigentliches Problem

⇨ aufwendig!

- Parameterlose GA,
  - Idee: Automatische Bestimmung optimaler Paramter

Verfahren nach Harik-Lobo:

- Wähle f und p_c für jedes t, so dass für optimales s gilt:

$$2 \stackrel{!}{=} \frac{P(\mathbf{s},t+1)}{P(\mathbf{s},t)} \approx \frac{f(\mathbf{s},t)}{\bar{f}}[1-p_c]$$

- Statt einen GA mit $|G_t| = \mu$ führe Wettbewerb mehrerer GA_i mit $\mu_i = \mu_0 2^i$ aus.
  - GA_0 - μ_0 → *2 → GA_1 - μ_1

Algorithmus zum Parameterlosen Genetischen Algorithmus (PGA)

```
PGA :=
    proc ()
(1)   z = z_m ... z_2 z_1 z_0 (4) := 0 ;   // Zähler zur Basis 4
(2)   μ_0 := initiale Populationsgröße, gerade ;
(3)   while not fertig () do
(4)     z̃ := z + 1 ;
(5)     ∀i ∈ {j ∈ {0,...,m} | z_j ≠ z̃_j} do
(6)       berechne neue Generation in GA_i mit μ_i = μ_0 2^i
(7)     od ;
(8)     z := z̃ ;
(9)     if (∃ i,j) mit j > i und f̄_j > f̄_i then
(10)      schließe GA_i von weiteren Berechnungen aus fi
(11)  od
(12)  gib x mit maximalem f(x) aus
    end;
```

Betrachte Indices i der ausgeführten GA_i:

0000100001000010000120000100 . . .

- 3 Mal GA_0 ausführen
- GA_0 und GA_1 ausführen
- 3 Mal GA_0 ausführen

- GA_0 und GA_1 ausführen
- 3 Mal GA_0 ausführen
- GA_0 und GA_1 ausführen
- 3 Mal GA_0 ausführen
- GA_0, GA_1 und GA_2 ausführen
- 3 Mal GA_0 ausführen
- usw…

ε_i := #Evaluation in GA_i bis zum Abbruch, dann ist

$\varepsilon_i \geq 2\ \varepsilon_{i+1}$

=> für O_i := (#Evaluationen in GA_i mit j > i) gilt:

$$\mathcal{O}_i \leq \sum_{j=1}^{\infty} \left(\frac{1}{2}\right)^j \mathcal{E}_i = \mathcal{E}_i$$

Systeme zur Unterstützung von GA-Applikationen

Anwendungsorientierte Systeme „black boxes“, verstecken GA-Details, erlauben Programmierung von Anwendungen für

- Finanzbereich, OMEGA
- Telekommunikation, PC/BEAGLE
- Planung

Algorithmen-orientierte Systeme unterstützen spezielle GA

- Algorithmen-spezifische, GENESIS, GAGA
- Algorithmen-Bibliotheken mit vielen Algorithmen, Operatoren, OOGA, GALIB

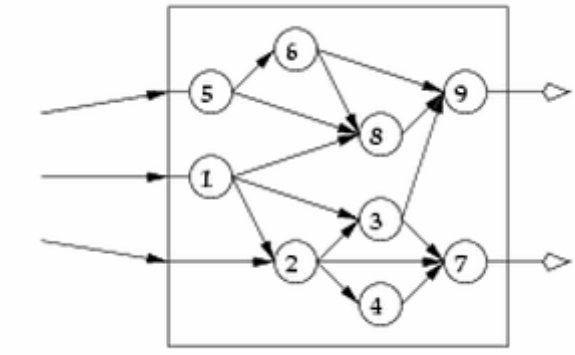
Tool Kits enthalten viele Programmierhilfsmittel für viele Bereiche

- Unterrichtssysteme zur Einführung in GA, GA WORKBENCH
- Umfangreiche allgemeine Systeme für komplexe Anwendungen, SPLICER, EN*GENE*ER, MICROGA, PEGASUS

`www.faqs.org/faqs/by-newsgroup/comp/comp.ai.genetic.html`

Vorlesung 24 – Künstliche Neuronale Netze (KNN)

Idee; Ahme Prinzip der biologischen neuronalen Netze durch KNN nach



Mögliche Vorteile:

- Übertragbarkeit von Lösungsansätzen aus der Natur
- Parallelisierbarkeit ➔ hohe Rechenleistung
- Redundanz ➔ Fehlertoleranz
- Lernfähigkeit ➔ geringer Programmieraufwand

Was ist für ein KNN zu spezifizieren?

- Verhalten der kpnstlichen Neuronen
  - Berechnungsreihenfolge
  - Aktivierungsreihenfolge
- Netzstruktur (Topologie)
  - Rekurrente Netze
  - Feed-Forward Netze
- Einbettung in Umgebung
- Lernverfahren

Ein Hopfield Netz

Neuronen 1 2 … d

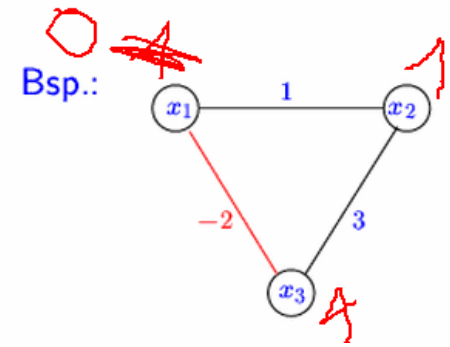
Aktivierungen x_1 x_2 … x_d ; $x_i \in \{0,1\}$

Verbindungen

$w_{ij} \in \mathbb{R} \quad (1 \leq i,j \leq d)$ mit
$w_{ii} = 0,\ w_{ij} = w_{ji} \Rightarrow \mathbf{W} := [w_{ij}]_{d \times d}$
ist symmetrisch mit 0-Diagonale

Update asynchron & stochastisch, gemäß

$$x'_j := \begin{cases} 0 & \text{falls } \sum_{i=1}^{d} x_i w_{ij} < 0 \\ 1 & \text{falls } \sum_{i=1}^{d} x_i w_{ij} > 0 \\ x_j & \text{sonst} \end{cases}$$



$$\mathbf{W} = \begin{bmatrix} 0 & 1 & -2 \\ 1 & 0 & 3 \\ -2 & 3 & 0 \end{bmatrix}$$

x_1 ändert sich nach einem Update zum Wert „0“, alles andere bleibt so. ➔ Stabiler Zustand erreicht.

Mögliche Verwendung:

- als assoziativer Speicher
- zur Berechnung Boolescher Funktionen
- zur kombinatorischen Optimierung

Die Energie eine Hopfield-Netzes

Sei $\mathbf{x} = (x_1, x_2, \ldots, x_d)^{\mathrm{T}}$ dann ist

$$E(\mathbf{x}) := -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\,\mathbf{W}\,\mathbf{x} = -\sum_{i<j} x_i\, w_{ij}\, x_j$$

die Energie eines Hopfield Netzes

Satz: Jeder Update, der das Hopfield Netz ändert, verringert seine Energie.
Beweis: Annahme: Der Update ändere x_k in x'_k. Dann ist

$$\begin{aligned} E(\mathbf{x}) - E(\mathbf{x}') &= -\sum_{i<j} x_i\, w_{ij}\, x_j + \sum_{i<j} x'_i\, w_{ij}\, x'_j \\ &= -\sum_{j\neq k} x_k\, w_{kj}\, x_j + \sum_{j\neq k} x'_k\, w_{kj}\, \underbrace{x'_j}_{=x_j} \\ &= -(x_k + (-x'_k)) \sum_{j\neq k} w_{kj}\, x_j \\ &> 0 \end{aligned}$$

a) $-(0 \quad -1) \sum > 0 \quad > 0$

b) $-(1 \quad 0) \sum < 0 \quad > 0$

Die Differenz ist immer größer „0“ ➔ Die Energie nimmt immer ab → qed

Vorgehensweise bei der Lösung eines KOP mittels Hopfield-Netz

Gegeben: KOP
Gesucht: Lösung für KOP

Vorgehensweise:

- Wähle Hopfield-Netz mit den Parametern des KOP als Gewichten und Lösung im Energie-Minimum
- Starte Netz mit zufällig gewählten Aktivierungen
- Berechne Folge von Updates bis Stabilität erreicht
- Lese Parameter aus Netz ab
- Teste Zulässigkeit und Optimalität der Lösung

Anwendungsbeispiel I – Multi Flop Problem

Problem-Instanz:	$k, n \in \mathbb{N}, \quad k < n$
Zulässige Lösung:	$\tilde{\mathbf{x}} = (x_1, \ldots, x_n) \in \{0,1\}^n$
Zielfunktion:	$P(\tilde{\mathbf{x}}) = \sum_{i=1}^{n} x_i$
Optimale Lösung:	Lösung $\tilde{\mathbf{x}}$ mit $P(\tilde{\mathbf{x}}) = k$

Formuliere Multi Flop als Minimierungsproblem:

Im folgenden Beispiel wird ein Vektor mit n Einträgen gesucht, der k-viele Einsen enthält. Im Beispiel ist n=3 und k=1. Initialisiert man das Netz mit 3 Nullen und beginnt bei x_3, dann springt x_3 auf „1“. Die anderen Knoten können sich nun nicht mehr verändern, da ihre Summen immer kleiner 0 sind und dann per Definition (vom Hopfield Netz) nicht mehr „1“ sein dürfen.

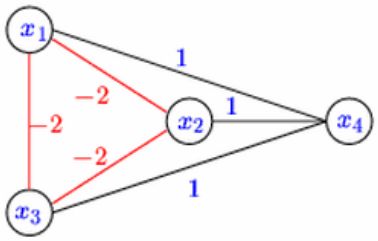
Beispielrechnung

- Wähle zufällig x_1 als Aktivierungsknoten → 0*(-2)+0*(-2)+1*1 = 1 > 0 → wird auf „1“ gesetzt
- Wähle zufällig x_2 als Aktivierungsknoten → 0*(-2)+1*(-2)+1*1 = -1 < 0 → bleibt auf „0“
- Wähle zufällig x_3 als Aktivierungsknoten → 1*(-2)+0*(-2)+1*1 = -1 < 0 → bleibt auf „0“
- Das Netz ist nun stabil ! → Die Lösung ist der Vektor (1,0,0)

Sei $d = n+1$, $x_d = 1$, $\mathbf{x} = (x_1, x_2, \ldots, x_n, x_d)^{\mathrm{T}}$ und

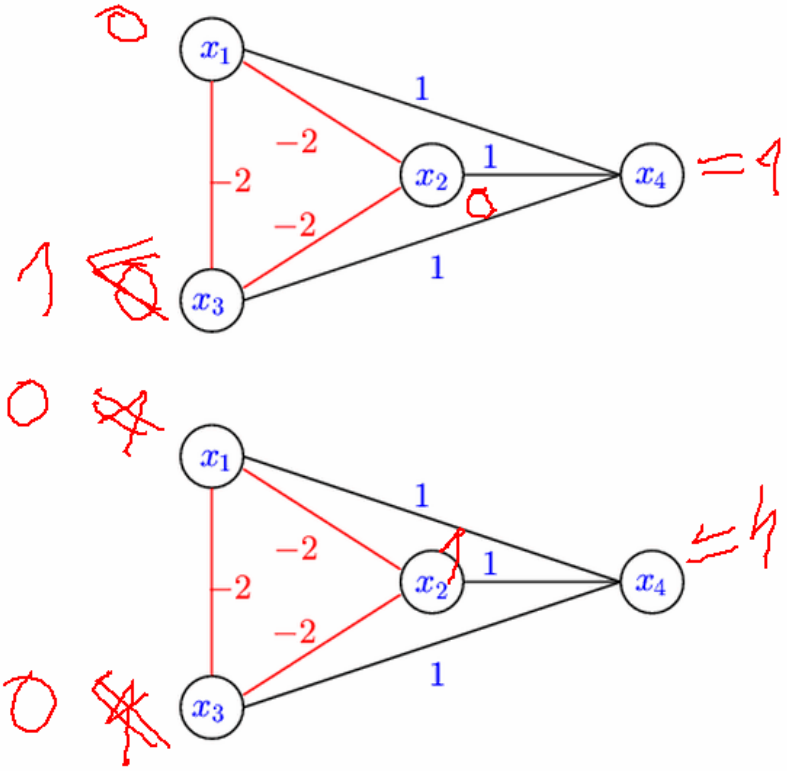
$$
\begin{aligned}
E(\mathbf{x}) &= \Big(\sum_{i=1}^{d} x_i - (k+1)\Big)^2 \\
&= \sum_{i=1}^{d} \underbrace{x_i^2}_{=x_i} + \sum_{i \neq j} x_i x_j - 2(k+1) \sum_{i=1}^{d} x_i + (k+1)^2 \\
&= \sum_{i \neq j} x_i x_j - (2k+1) \sum_{i=1}^{d-1} x_i x_d + k^2 \\
&= -\frac{1}{2} \sum_{i<j} x_i(-4)x_j - \frac{1}{2} \sum_{i<d} x_i (4k+2) x_d + k^2
\end{aligned}
$$

Bsp. $(n = 3, k = 1)$:



$$E(\mathbf{x}) = -\frac{1}{2}\sum_{i<j} x_i(-4)x_j - \frac{1}{2}\sum_{i<d} x_i\,(4k+2)\,x_d + k^2$$

Bsp. $(n = 3, k = 1)$:



Anwendungsbeispiel 2 – Travelling Salesperson Problem (TSP)

Problem Instanz:

Städte: 1 2 … n

Entfernungen: $d_{ij} \in \mathbb{R}^+ \quad (1 \le i, j \le n)$ mit $d_{ii} = 0$

Zulässige Lösung: Permutation π von (1,2,…,n)

Zielfunktion: $P(\pi) = \sum_{i=1}^{n} d_{\pi(i),\pi(i \bmod n+1)}$

Optimale Lösung: Zulässige Lösung π mit minimalem P(π)

Idee: Hopfield-Netz mit $d=n^2 + 1$ Neuronen:

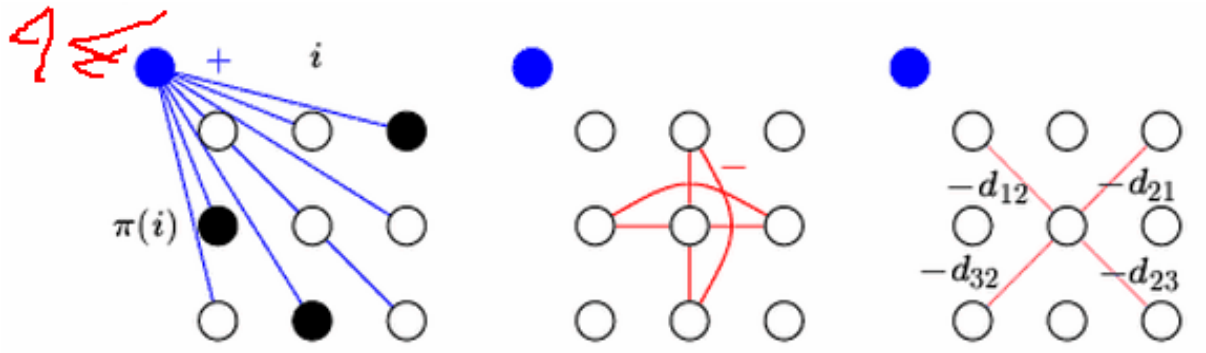
- Die Zeilen der folgenden Matrizen sind die Ablaufreihenfolge
- Die Spalten der folgenden Matrizen sind die Städte des TSP

- Die schwarzen Punkt im Beispiel markieren die gewählte Ablaufreihenfolge, d.h. bei (1.) 2→3→1

- Jeder Punkt der Matrix ist mit dem „1“-Neuron (blaues) positiv verbunden

- Die 2. Bedingung mit den negativen Kanten, verhindert, dass mehr als ein Knoten pro Zeile bzw. Spalte auf „1“ gesetzt wird. Was nicht erlaubt ist.

- Die 3. Bedingung bezieht die Gewichte (Entfernungen der Städte) mit ein um eine (annährend) optimale Lösung zu erhalten. Eingezeichnet sind die Entfernungen der Stadt Zwei zu den anderen Städten.



1. Irgendeine Lösung 2. zulässige Lösung 3. Gute Lösung

Problem:

⇨ „Größe“ der Gewichte (der Bedingung 1. und 2.), um gleichzeitig zulässige und gute (optimale) Lösung zu erzwingen.

Abhilfe:

⇨ Übergang zu stetigem Hopfield-Netz und modifizierten Gewichten → gute Lösung des TSP

Selbstorganisierende Karten (SOM)

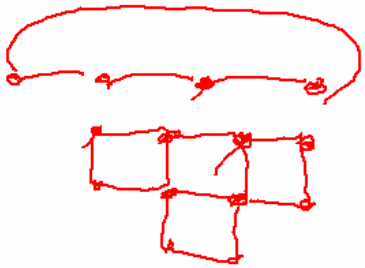
Ein SOM (Kohonen Netz):

Neuronen:

Eingabe: $1, 2, \ldots, d$ für Komponenten x_i der Eingabe

Karte: $1, 2, \ldots, m$; Reguläres (Linear-, Rechteck, Hexagonal-) Gitter $\mathbf{r}$ zur Speicherung der *Mustervektoren* $\boldsymbol{\mu}_i \in \mathbb{R}^d$

Ausgabe: $1, 2, \ldots, d$ für $\boldsymbol{\mu}_c$



Update:

$L \subset \mathbb{R}^d$, Lernmenge

Zur Zeit $t \in \mathbb{N}^+$ wird $\mathbf{x} \in L$ zufällig gewählt, dann $c \in \{1, \ldots, m\}$ bestimmt (*bmu*) mit

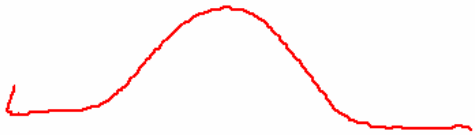
$$\|\mathbf{x} - \boldsymbol{\mu}_c\| \leq \|\mathbf{x} - \boldsymbol{\mu}_i\| \quad (\forall i \in \{1, \ldots, m\}$$

und die Muster angepasst:

$$\boldsymbol{\mu}_i' := \boldsymbol{\mu}_i + h(c, i, t)\,(\mathbf{x} - \boldsymbol{\mu}_i) \quad (\forall i \in \{1, \ldots, m\}$$

mit $h(c, i, t)$ zeitabhängige *Nachbarschaftsfunktion* und $h(c, i, t) \to 0$ für $t \to \infty$, z.B.:

$$h(c, i, t) = \alpha(t) \cdot \exp\left(-\frac{\|\mathbf{r}_c - \mathbf{r}_i\|^2}{2\sigma(t)^2}\right)$$

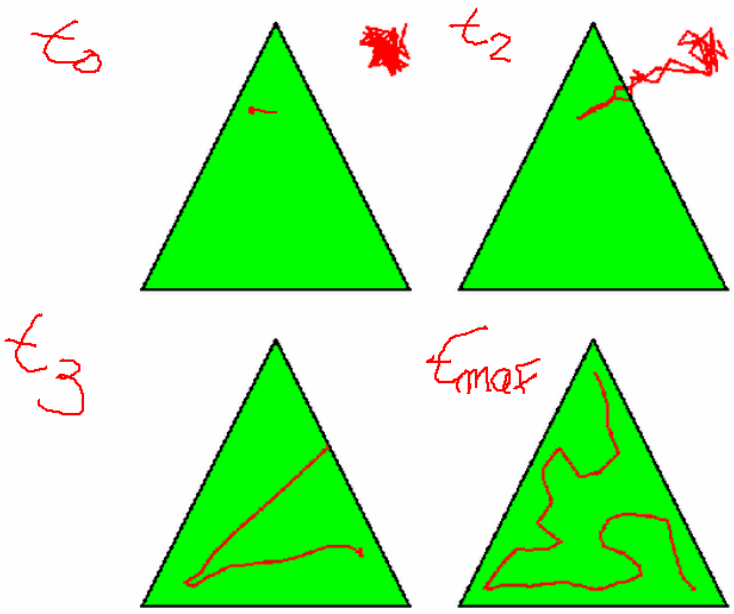


Mögliche Verwendung:

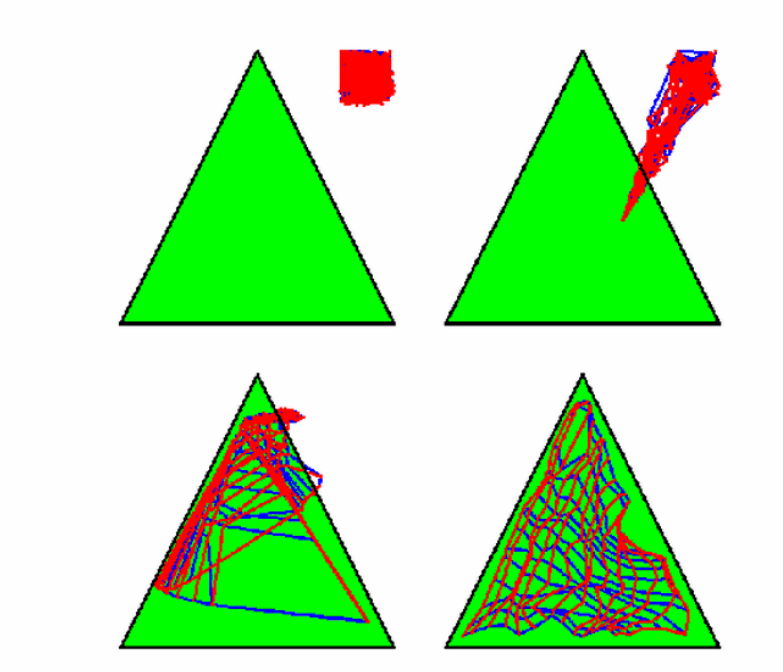
- zur Visualisierung, Interpretation, Dimensionsreduktion, Dichte-Repräsentation, Klusterung, Klassifikation von Daten
- zur kombinatorischen Optimierung

Anwendungsbeispiele für SOM

Eine 50-er Kette passt sich an Punkte aus dem Dreieck an:



Ein 15×15-Gitter passt sich an Punkte aus dem Dreieck an:



Euklidisches Travelling Salesperson Problem (ETSP)

Ähnlich TSP, doch jede Stadt i hat Position

Ähnlich TSP, doch jede Stadt i hat Position $\mathbf{p}_i \in I\!R^2$ und #Städte $\approx n$

$$d(i,j) = \|\mathbf{p}_i - \mathbf{p}_j\|$$

Idee: Verwende *wachsenden* Ring (elastisches Band) von Neuronen

FLEXMAP :=
proc ()
(1) initialisiere Ring mit 3 Neuronen;
(2) **while** $(\exists j)$ j ist bmu für ≥ 2 Städte **do** n
(3) $\forall z \in \{1,\ldots,N\}$ **do** zufällig
(4) passe $c :=$ bmu(i) und endl. Nachbarschaft an; n
(5) fehler(c) += $\|\boldsymbol{\mu}_c - \mathbf{p}(i)\|$
(6) **od**;
(7) finde Nachbarn i,k mit max. fehler(i) + fehler(k); n
(8) füge neues Neuron j zwischen i und k ein;
(9) $\boldsymbol{\mu}_j := (\boldsymbol{\mu}_i + \boldsymbol{\mu}_k)/2$;
(10) fehler(i) *= $2/3$;
(11) fehler(k) *= $2/3$;
(12) fehler(j) := (fehler(i) + fehler(j))$/2$
(13) **od**;
(14) gib zum Ring gehörende Tour aus;
end;

$2(n)$

Zeit $\in O(n^2)$

Komplexität

Die While Schleife wird n mal durchlaufen. Innerhalb der While Schleife existieren zwei Abschnitte, die beide O(n) Zeit benötigen als 2 O(n) → Gesamtzeit $O(n^2)$

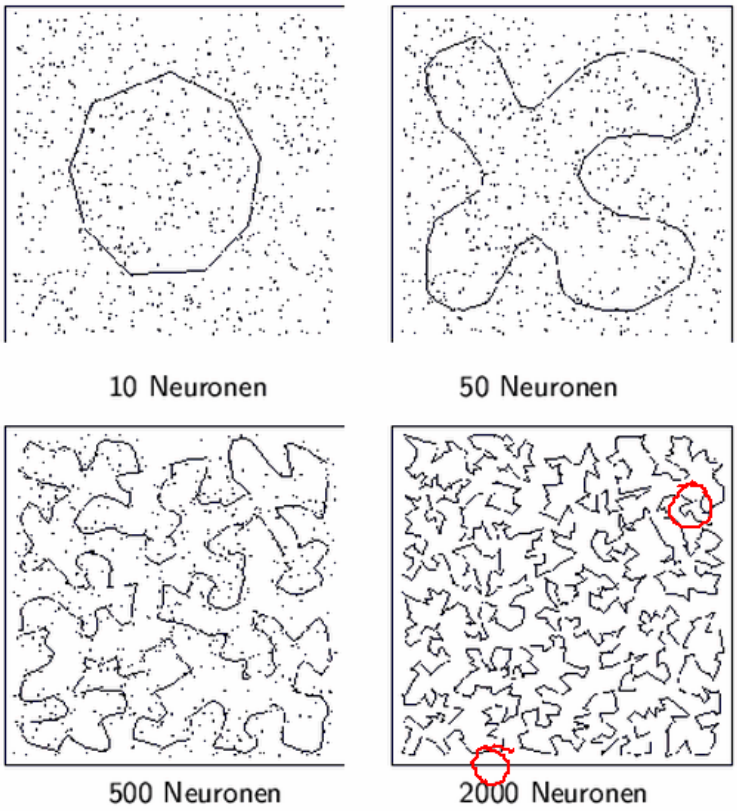
SOM für kombinatorische Optimierung

Heuristische Änderungen zur Linearisierung der Laufzeit von FLEXMAP:

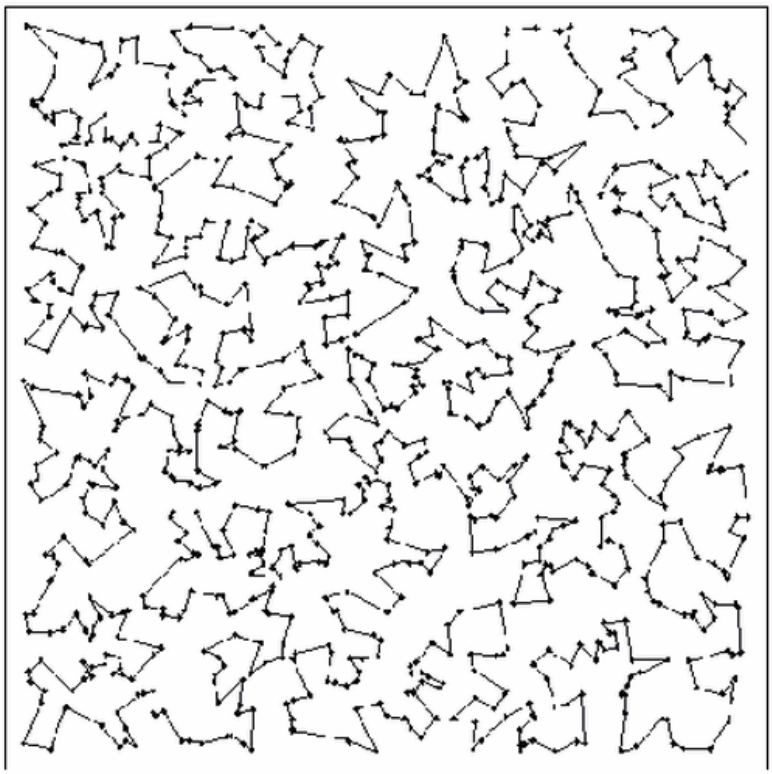
- Speichere für jede Stadt letzte bmu(i) (Best-Matching-Unit). Bei erneuter Auswahl von i bestimme neues bmu(i) unter endl. Nachbarschaft des alten bmu(i)
- Bestimme Kante höchsten Fehlers unter denen, die adjazent zu Neuronen aus Programmschritten (3)-(5) sind
- War ein Neuron M-mal in Folge bmu für dieselbe Stadt i, so wird i fixiert → FLEXMAP bricht ab (Schritt(3)), wenn alle Städte fixiert sind

Tests mit $n \leq 2392$ zeigen, dass die Laufzeit linear skaliert und die gefundenen Lösungen um weniger als 9% vom Optimum abweichen.

Veränderung des FLEXMAP-Rings für ein 1000-Städte-Beispiel:



Von FLEXMAP gefundene Lösung für ein 1000-Städte-Beispiel:



Tour mit 2526 Neuronen

Geschichtete Feed-Forward Netze (MLP)

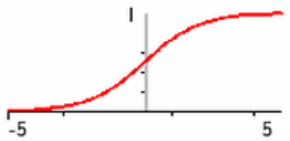
Ein L-schichtiges MLP (Mehr-Lagiges Perzeptron)

Schichten:	$S_0, S_1, \ldots, S_{L-1} S_L$ (Input-, Verdeckte-, Output-Schicht)
Verbindugen:	Von jedem Neuron i in S_l zu j in S_{l+1} mit Gewicht w_{ij}, ausser 1-Neuronen
Update:	Schichtweise synchron gem.

$$x'_j := \varphi\left(\sum_{i \in \mathcal{V}(j)} x_i \, w_{ij}\right)$$

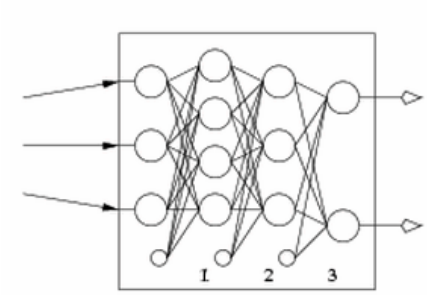
mit φ differenzierbar,

z.B. $\varphi(a) = \sigma(a) = \frac{1}{1+\exp(-a)}$



Mögliche Verwendung:

- Funktionsapproximation
- Klassifikation



$f : \mathbb{R}^3 \to \mathbb{R}^2$

Satz (ohne Beweis):

⇨ Boolesches Funktionen sind durch 2-schichtige MLP berechenbar.

Satz (ohne Beweis):

⇨ Stetige reelle Funktionen und ihre partiellen Ableitungen können gleichzeitig durch 2-schichtige MLP auf einem kompakten Bereich beliebig genau approximiert werden.

Paramterlernen bei MLP

Lernen:

gegeben: $\mathbf{x}^1, \ldots, \mathbf{x}^N \in \mathbb{R}^d$ und $\mathbf{t}^1, \ldots, \mathbf{t}^N \in \mathbb{R}^c$,
MLP mit d Input-, c Output-Neuronen,
$\mathbf{w} = w_1, \ldots, w_M$ enthalte sämtliche Gewichte,
$f(\mathbf{x}, \mathbf{w})$ sei die Netzfunktion.

gesucht: optimales $\mathbf{w}^*$, welches den Fehler

$$E(\mathbf{w}) := \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{c} \left(f_k(\mathbf{x}^n, \mathbf{w}) - t_k^n\right)^2$$

minimiert.

Bemerkung: Da die partiellen Ableitungen von f in Bezug auf die Inputs und die Parameter existieren, können Gradienten-Basierte Optimierungsverfahren angewandt werden (Konjugierte Gradienten, Levenberg-Marquardt,…)

Es ist

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \sum_{n=1}^{N} \sum_{k=1}^{c} \left(f_k(\mathbf{x}^n, \mathbf{w}) - t_k^n\right) \nabla_{\mathbf{w}} f_k(\mathbf{x}^n, \mathbf{w})$$

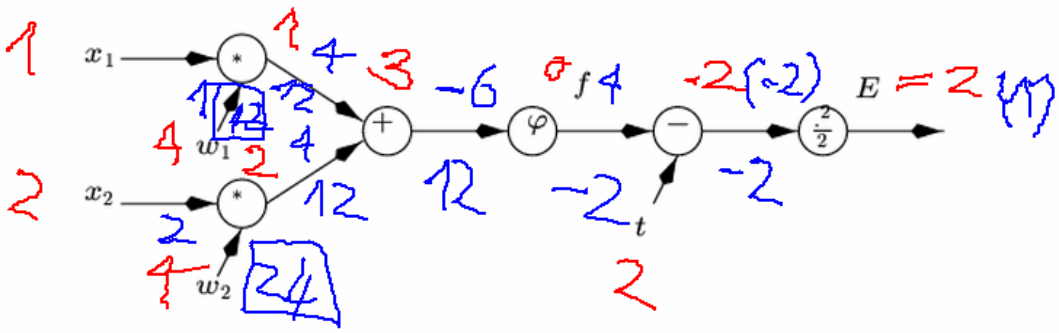
(Dies ist die Ableitung der oberen Funktion)

Gradientenbestimmung mit Backpropagation

Grundlage: Die Kettenregel:

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right)$$

Bsp.: Sei $\varphi(a) := 9 - a^2$, $\mathbf{x} = (1,2)^{\mathrm{T}}$, $\mathbf{w} = (1,1)^{\mathrm{T}}$, $t = 2$:



$$\nabla_{\mathbf{w}} E(\mathbf{w})|_{\mathbf{w}=(1,1)^{\mathrm{T}}} = (12, 24)$$

$$\begin{aligned}
h(x,y) &= x * y \;\Rightarrow\; \partial/\partial x\; h(x,y) = y \\
h(x,y) &= x + y \;\Rightarrow\; \partial/\partial x\; h(x,y) = 1 \\
h(x,y) &= x - y \;\Rightarrow\; \partial/\partial x\; h(x,y) = 1 \\
\varphi(x) &= 9 - x^2 \;\Rightarrow\; \partial/\partial x\; \varphi(x) = -2x \\
h(x) &= x^2/2 \;\Rightarrow\; \partial/\partial x\; h(x) = x
\end{aligned}$$

Satz. *$\nabla_{\mathbf{w}} E(\mathbf{w})$ kann in Zeit $O(N \times M)$ bestimmt werden, wenn die Größe des Netzes in $O(M)$ ist.*

Das Backpropagation-Verfahren:
Wiederhole für alle n ∈ {1,…,N}:

- Berechne Netzfunktion f(x^n, w) und dazugehörigen Fehler E in Vorwärtsrichtung, speichere Zwischenwerte im Netz.
- Berechne partielle Ableitungen von E in Bezug auf alle Zwischenwerte in Rückwärtsrichtung und addiere Beiträge zum Gesamtgradienten

Vorlesung 25 – Ameisen Algorithmen

Einleitung

Achilles (zum Ameisenbär): … Ich verstehe nicht, wie alle Ameisen unintelligent sein können, wenn Tante Colonia (die Ameisenkolonie) Sie viele Stunden lang mit geistreichen Späßen unterhalten kann.

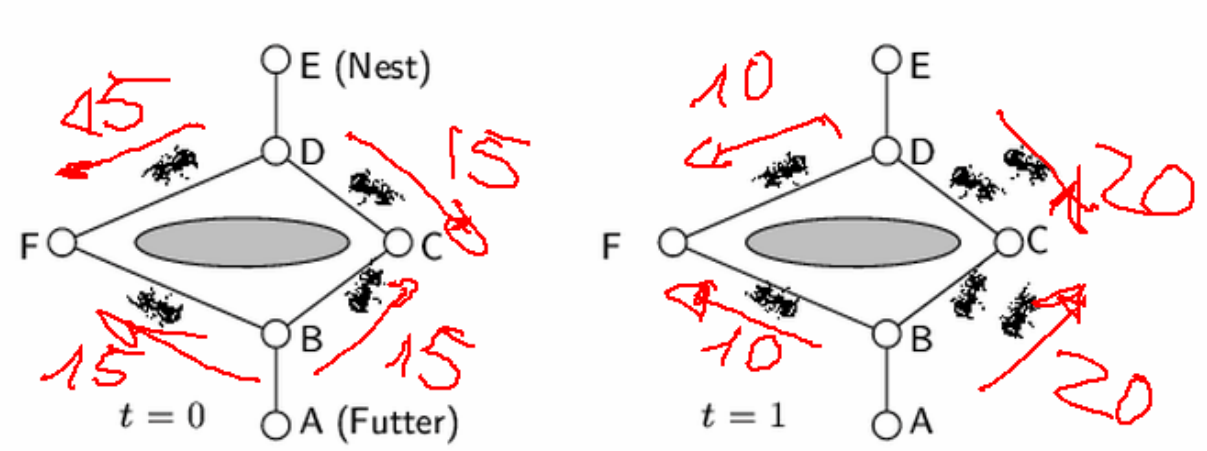
Schildkröte: Ich glaube, dass sich die Dinge ähnlich verhalten, wie bei der Zusammensetzung des menschlichen Gehirns mit Neuronen. Gewiss würde niemand darauf bestehen, dass einzelne Hirnzellen intelligente Lebewesen sein müssen, um die Tatsache zu erklären, dass ein Mensch eine intelligente Unterhaltung führen kann.

.
.
.

Ameisenbär: … obgleich Ameisen als Individuen auf ein Art und Weise herumstreifen, die rein zufällig scheint, gibt es darunter allgemeine, aus dem Chaos auftauchende Trends, die große Mengen von Ameisen betreffen.

(aus D. Hofstadter: Gödel, Escher, Bach – ein Endloses geflochtenes Band, [8, S. 338])

Betrachte Ameisen bei der Futtersuche:



Kommunikation durch Pheromone (Duftstoffe):
Zufällige Entscheidungen → angepasste Entscheidungen

Künstliche Ameisen

- lösen Optimierungsprobleme durch Sequenzen von Entscheidungen
- treffen einzelne Entscheidungen zufällig, aber gesteuert durch Pheromone und weitere Kriterien
- besitzen ein Gedächtnis, können daher zulässige Optionen erkennen
- verteilen Pheromone proportional zur Güte der gefundenen Lösungen

Traveling Salesperson-Problem (TSP)

Problem Instanz:

Städte: 1 2 … n

Entfernungen: $d_{ij} \in \mathbb{R}^+ \quad (1 \leq i, j \leq n) \text{ mit } d_{ii} = 0$

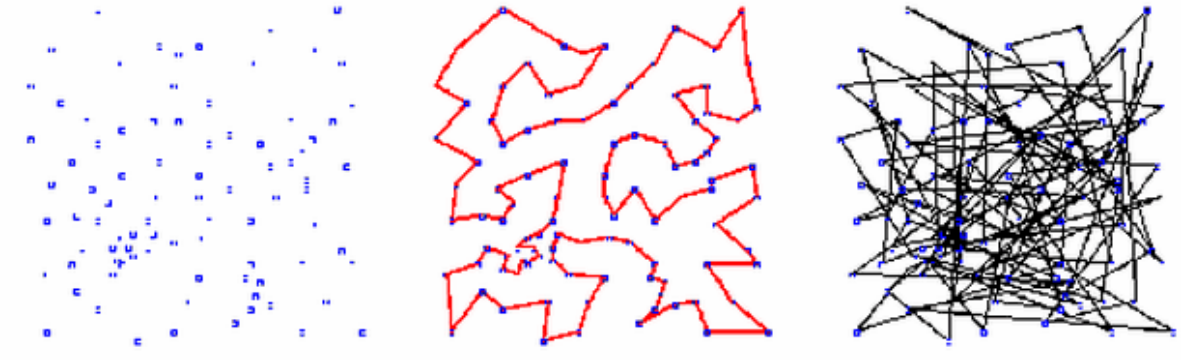
Zulässige Lösung: Permutation π von (1,2,…,n)

Zielfunktion: $P(\pi) = \sum_{i=1}^{n} d_{\pi(i),\pi(i \bmod n+1)}$

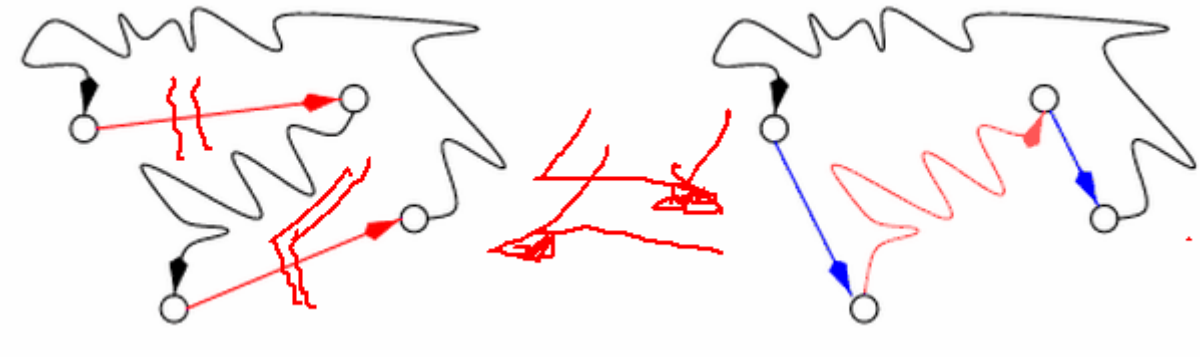
Optimale Lösung: Zulässige Lösung π mit minimalem P(π)

#Potentieller Lösungen: (n-1)!/2; für n = 101: ≈ 4.7 x 10^{157}

Bemerkung: (n-1)!/2 da das TSP hier symmetrisch ist, d.h. dij = dji



Ein einfacher Optimierungsschritt (LIN-2-OPT):



Bemerkung:

- Dieser Optimierungsschritt geht nur dann, wenn die Summe der blauen Weglängen (blaue Pfeile) kleiner als die Summe der roten Weglängen (rote Pfeile) ist.
- Handelt es sich um das unsymmetrische TSP, muss diese Unsymmetrie noch berücksichtigt werden.

#Nachbarlösungen: n * (n-3);

- bei n = 101 also 9898 (unsymmetrisch)
- wenn symmetrisch muss noch durch 2 geteilt werden: 4949

Ameisen-Algorithmus für das TSP

Ein simpler Ameisen-System-Algorithmus (ASA):

```
ASA_TSP :=
    proc ()
(1)   t := 0;    // Zeit
(2)   ∀i, j ∈ {1, ..., n} do    // n: # Städte
(3)     τ_ij := Pheromon (i, j, t) od;
(4)   while not fertig () do
(5)     t += 1;
(6)     for k from 1 to μ do    // μ: # Ameisen
(7)       while not tourKomplett (k) do
(8)         j := auswahlStadt (k) od;
(9)       L_k := tourLaenge (k)
(10)    od;
(11)    ∀i, j ∈ {1, ..., n} do
(12)      τ_ij := Pheromon (i, j, t) od
(13)  od;
(14)  gib kürzeste Tour aus;
    end;
```

Laufzeit: $O(t_{max} * \mu * n * n)$

Bemerkung:

- t_{max}: Maximale Zeit
- μ: Anzahl Ameisen
- n: wegen Zeile 7
- n: wegen Zeile 8

Legende zum ASA_TSP

Auswahl einer Stadt j für Ameise k in Stadt i:
Falls i nicht definiert, wird j initialisiert, sonst nach folgenden Bedingungen bestimmt:

- j darf nicht bereits in der begonnenen Tour vorkommen
- die Wahrscheinlichkeit P_{ij} der Auswahl von j ist proportional zu $1/d_{ij}$ und τ_{ij}

$$P_{ij} := \begin{cases} \dfrac{[\tau_{ij}]^{\alpha}\,[\eta_{ij}]^{\beta}}{\sum_{h\in\Omega(k)}[\tau_{ih}]^{\alpha}\,[\eta_{ih}]^{\beta}} & \text{für } j \in \Omega(k) \\ 0 & \text{sonst} \end{cases}$$

τ_{ij} : Pheromon-Level auf Kante (i, j)
α : Gewichtung von τ_{ij}
η_{ij} : $1/d_{ij}$ Sichtbarkeit
β : Gewichtung von η_{ij}
$\Omega(k)$: Menge der von k unbesuchten Städte

Berechnung der Pheromon-Menge $\tau_{ij(t)}$ für Kante (i,j):

Falls $t = 0$, wird $\tau_{ij}(t)$ initialisiert, sonst:
ein Anteil von $(1 - \rho)$ verduftet und Ameise k verteilt Q/L_k:

$$\begin{aligned} \tau_{ij}(t+1) &:= \rho\,\tau_{ij}(t) + \Delta\tau_{ij} \\ \Delta\tau_{ij} &= \textstyle\sum_{k=1}^{\mu} \Delta\tau_{ij}^{k} \\ \Delta\tau_{ij}^{k} &= \begin{cases} Q/L_k & \text{falls } (i,j) \in \mathsf{Tour}(k) \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Parameter des ASA_TSP

μ: #Ameisen; oft wird μ = n empfohlen, jede Ameise startet dann in einer anderen Stadt

α, β ∈ [0,∞):

- ⇨ Gewichten die relative Bedeutung der Pheromone im Vergleich zur Sichtbarkeit.
  - α zu groß: frühe Stagnation, Exploitation
  - α zu klein: iterierte Heuristik, Exploration
- ⇨ Vorschlag: α = 1, β = 5

ρ ∈ [0,1]:

- ⇨ beeinflusst Gedächtnis des Verfahrens
  - ρ zu groß: frühe Konvergenz
  - ρ zu klein: kein Ausnutzen von Erfahrung
- ⇨ Vorschlag: ρ = !/2

Q:

- ⇨ bestimmt Einfluss neuer Information in Relation zur Initialisierung $\tau_{ij}(0)$, wenig problematisch
- ⇨ Vorschlag: Q = 100

Tests zeigen, dass Qualität und Laufzeit von ASA_TSP für n ≈ 30 vergleichbar ist mit anderen heuristischen Verfahren.

Quadratisches Zuordnungsproblem (QAP)

QAP = Quadratic Assignment Problem)

Problem-Instanz:

$$\begin{array}{ll} \text{Orte:} & 1 \quad 2 \quad \ldots \quad n \\ \text{Aktivitäten:} & 1 \quad 2 \quad \ldots \quad n \\ \text{Entfernungen:} & d_{ih} \in \mathbb{N}^{+} \quad (1 \le i,h \le n) \text{ mit} \\ & d_{ih} = d_{hi},\ d_{ii} = 0,\ \mathbf{D} := [d_{ih}] \\ \text{Flüsse:} & f_{jk} \in \mathbb{N}^{+} \quad (1 \le j,k \le n) \text{ mit} \\ & f_{jk} = f_{kj},\ f_{jj} = 0,\ \mathbf{F} := [f_{jk}] \end{array}$$

Zulässige Lösung: Permutation π von (1,2, … ,n) (Aktivität π(i) wird an Ort i ausgeführt.)

Zielfunktion: $P(\pi) = \sum_{i,h \in \{1,\ldots,n\}} d_{ih} f_{\pi(i)\pi(h)}$

Optimale Lösung: Zulässige Lösung π mit minimalem P(π)

Äquivalente Formulierung: Finde Permutationsmatrix

$$\mathbf{X} \in \{0,1\}^{n \times n} \text{ mit } x_{ij} = 1 \Leftrightarrow \pi(i) = j \text{ und minimalem}$$

$$P(\mathbf{X}) = \sum_{i,j \in \{1,\ldots,n\}} \sum_{h,k \in \{1,\ldots,n\}} d_{ih} f_{jk} x_{ij} x_{hk}$$

Ameisen-Algorithmus für das QAP

Ameise k setzt $x_{ij} = 1$ ($\pi(i) = j$) falls

- bisher keine 1 in Zeile i und Spalte j von X
- die potentielle Güte und die Pheromone die Bindung (i,j) begünstigen

Die potentielle Güte ergibt sich aus dem Distanz-Potential $\mathcal{D} = \mathbf{D}\,[1,\ldots,1]^{\mathrm{T}}$ und dem Fluss-Potential $\mathcal{F} = \mathbf{F}\,[1,\ldots,1]^{\mathrm{T}}$.

Beispiel:

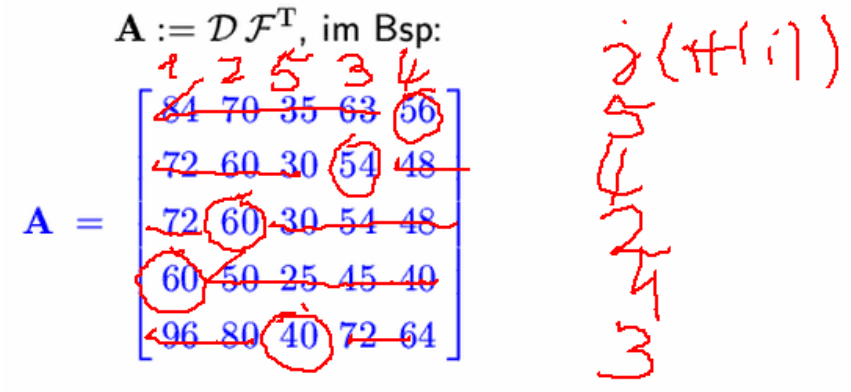
$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 \\ 1 & 0 & 2 & 1 & 2 \\ 1 & 2 & 0 & 1 & 2 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 2 & 2 & 1 & 0 \end{bmatrix} \quad \mathcal{D} = \begin{bmatrix} 7 \\ 6 \\ 6 \\ 5 \\ 8 \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} 0 & 5 & 2 & 4 & 1 \\ 5 & 0 & 3 & 0 & 2 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 5 \\ 1 & 2 & 0 & 5 & 0 \end{bmatrix} \quad \mathcal{F} = \begin{bmatrix} 12 \\ 10 \\ 5 \\ 9 \\ 8 \end{bmatrix}$$

Bemerkung: Die Stadt mit dem D-Wert = 5 (minimalster Wert) liegt im Zentrum, wohingegen die Stadt mit dem D-Wert = 8 (maximalster Wert) liegt am Rand (siehe die 2 markierten Stellen im D-Vektor).

Min-max-Regel:

⇨ Binde Aktivitäten mit hohem Fluss-Potential an Orte mit niedrigem Distanz-Potential und umgekehrt.

Kopplungs-Matrix



Bemerkung: i sind die Zeilen

Ameisen-Algorithmus für das QAP

Heuristik:

Betrachte Aktivitäten j in Reihenfolge absteigender Fluss-Potentiale und ordne ihnen noch einen freien Ort i mit minimalem a_{ij}-Wert zu.

Ameise k $\in \{1, \ldots, \mu\}$ geht nun ähnlich vor:

$$P_{ij} := \begin{cases} \dfrac{[\tau_{ij}]^{\alpha}\,[\eta_{ij}]^{\beta}}{\sum_{h\in\Omega(k)}[\tau_{ih}]^{\alpha}\,[\eta_{ih}]^{\beta}} & \text{für } i \in \Omega(k) \\ 0 & \text{sonst} \end{cases}$$

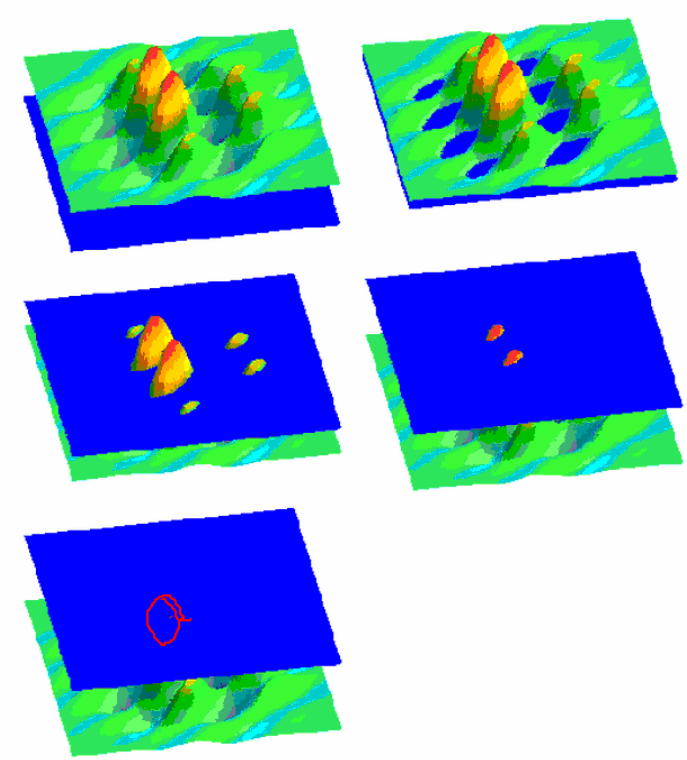
τ_{ij} : Pheromon-Level für Bindung (i,j)
α : Gewichtung von τ_{ij}
η_{ij} : $1/a_{ij}$ Wünschbarkeit
β : Gewichtung von η_{ij}
$\Omega(k)$: Menge der von k nicht zugeordneten Orte

Update von $\tau_{ij}(t+1)$ erfolgt wie beim ASA_TSP mit $P(\pi^k)$ für L_k.

Tests zeigen, dass das Verfahren sehr erfolgreich ist (α=1, β=1, ρ=0.9, Q=1, n≤33).

Sintflut-Algorithmus

Ameisen bewegen sich auf einer Landschaft, die langsam im Wasser versinkt:



Hoffnung: Am Ende befindet sich eine Ameise auf dem Gipfel.

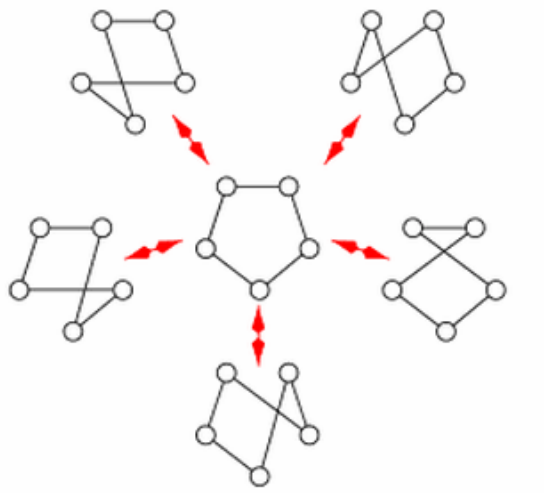
Der Sintflut-Algorithmus (als Code)

```
SINTFLUT :=
    proc ()
(1)   t := 1;    // Zeit
(2)   W := initWasserstand ();
(3)   for i from 1 to μ do    // μ: # Ameisen
(4)     i.position := initPosition ();
(5)     i.aktiv := true
(6)   od;
(7)   while not fertig () do
(8)     for i from 1 to μ do
(9)       if i.aktiv then
(10)        i.position := nachbarPosition (i, W);
(11)        i.aktiv := trocken (i, W)
(12)      fi
(13)    od;
(14)    W += Regen ();
(15)    t += 1
(16)  od;
(17)  gib j.position aus mit
(18)    j.hoehe ≥ {i.hoehe | i ∈ {1, ..., μ}};
    end;
```

Legende zum Sintflut-Algorithmus

Ameise:	hat Position und Höhe
Position:	Lösung eines Problems, es gibt Nachbar-Positionen
Höhe:	Güte der Lösung
Wasserstand W:	ist erst niedrig, steigt mit der Zeit an
Trocken:	Nur trockene Ameisen sind weiter aktiv

Beispiel: LIN-2-OPT-Nachbarn beim TSP mit n = 5:



(TSP-Höhe: -P(π))

Tests bestätigen die Wirksamkeit des Sintflut-Algorithmus für Probleme mit großer Zahl von Nachbarschafts-Beziehungen: Lokale Maxima sind dann selten.