Vollständige Klammerung

Def. Ein Produkt von Matrizen heißt vollständig geklammert, falls es aus einer einzelnen Matrix besteht, oder aus zwei vollständig geklammerten Produkten.

Bsp.

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

MATRIX-MULTIPLY(A, B)

if **columns**$[A] \neq rows[B]$
  then error "incompatible dimensions"
    else for $i \leftarrow 1$ **to** **rows**$[A]$
      do for $j \leftarrow 1$ **to** **columns**$[B]$
        do $C[i, j] \leftarrow 0$
          for $k \leftarrow 1$ **to** **columns**$[A]$
            do $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
    return C

Analyse

Kosten werden durch die Anzahl der Multiplikationen in Schritt 7 dominiert.

Sei A eine $p \times q$ Matrix und B eine $q \times r$ Matrix, dann wird Zeile 7

$$p * q * r$$

mal ausgeführt.

Optimierungspotential

Seien folgende Matrizen gegeben:

$$A_1 \ : \ 10 \times 100$$

$$A_2 \ : \ 100 \times 5$$

$$A_3 \ : \ 5 \times 50$$

Dann ergeben sich folgende Kosten:

$$\begin{array}{rcl} cost((A_1A_2)A_3) & = & 10 * 100 * 5 + 10 * 5 * 50 \qquad = \quad 7500 \\ cost(A_1(A_2A_3)) & = & 100 * 5 * 50 + 10 * 100 * 50 \ = \ 75000 \end{array}$$

Optimierungspotential bei NUR drei Matrizen:

- FAKTOR **10**.

Problemdefinition

Gegeben

- Sequenz $\langle A_1, \ldots, A_n \rangle$ von Matrizen

Gesucht

- vollst. geklammertes Produkt der Matrizen

  $$A_1, \ldots, A_n$$

  mit minimalen Kosten (minimaler Anzahl von Multiplikationen).

Anzahl der Klammerungen

Wir überzeugen uns, daß alle Klammerungen zu betrachten sehr ineffizient ist.

Bezeichne $P(n)$ die Anzahl der Klammerungen von n Matrizen.

In der obersten Klammerungsebene (letztes Produkt):

- splittet $\langle A_1, \ldots, A_n \rangle$ an einer Stelle k in zwei Teilsequenzen $\langle A_1, \ldots, A_k \rangle$ und $\langle A_{k+1}, \ldots, A_n \rangle$ $(1 \leq k < n)$.

Anzahl der Klammerungen

Es ergibt sich die Rekurrenz

$$P(n) = \begin{cases} 1 & \text{falls } n = 1 \\ \Sigma_{k=1}^{n-1} P(k)P(n-k) & \text{falls } n > 1 \end{cases}$$

Es gilt:

$$P(n) = C(n-1)$$

mit $C(n)$ gleich den catalanschen Zahlen:

$$\begin{aligned} C(n) &= \frac{1}{n+1}\binom{2n}{n} \\ &= \Omega(4^n/n^{3/2}) \end{aligned}$$

Also ist $P(n)$ exponentiell in n.

Bestimmung der Struktur des Optimums

Sei $A_{i\ldots j}$ das Produkt von $A_i, \ldots, A_j$.

Das Optimum splittet für ein k zwischen A_k und A_{k+1} $(1 \leq k < n)$:

$$A_{1\ldots k}A_{k+1\ldots n}$$

Die Kosten ergeben sich zu

$$cost(A_1 \ldots A_k) + cost(A_{k+1} \ldots A_n) + cost(A_{1\ldots k}A_{k+1\ldots n})$$

wobei die vollständigen Klammerungen von $A_1 \ldots A_k$ und $A_{k+1} \ldots A_n$ optimal sein müssen!

Anm.: Diese Bedingung heißt auch Optimalitätsprinzip.

Rek. Def. (der Kosten) des Optimums

Sei $m[i,j]$ die minimale Anzahl von Multiplikationen, die notwendig sind, um $A_{i\ldots j}$ zu berechnen.

Die Billigste Lösung des Problems hat dann die Kosten $m[1,n]$.

Sei A_i eine Matrix der Dimension $p_{i-1} \times p_i$.

Für $m[i,j]$ gilt:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Um nicht nur die minimalen Auswertungskosten zu berechnen, sondern auch entsprechend optimal Klammern zu können, merken wir uns die optimalen Splits k in einer Matrix $s[i,j]$.

Berechnung der optimalen Kosten

Wir bestimmen zunächst die Anzahl $\#TP$ der Teilprobleme $A_{i\dots j}$, für die eine optimale Klammerung gefunden werden muß.

Es muß gelten:

$$1 \le i \le j \le n$$

Also gilt für $\#TP$:

$$\#TP = \binom{n}{2} + n = \Theta(n^2)$$

Megatrick des dyn. Programmierens:

- Anstelle Lösung rekursiv zu berechnen (top-down)
- werden alle Teillösungen **bottom-up** bestimmt.

Annahmen für Algorithmus

- Matrizen: A_i für $1 \leq i \leq n$
- Dimension von A_i: $p_{i-1} \times p_i$ für $1 \leq i \leq n$
- Eingabe: Sequenz $\langle p_0, \ldots, p_n \rangle$ mit $\mathbf{length}[p] = n + 1$
- Hilfsstrukturen:
  - m[1. . . n, 1 . . . n] für minimalen Kosten $A_{i \ldots j}$
  - s[1. . . n, 1 . . . n] für optimale Splits von $A_{i \ldots j}$

MATRIX-CHAIN-ORDER(p)

```
n ← length[p] − 1
for i ← 1 to n
    do m[i, j] ← 0
for l ← 2 to n
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

Anmerkungen

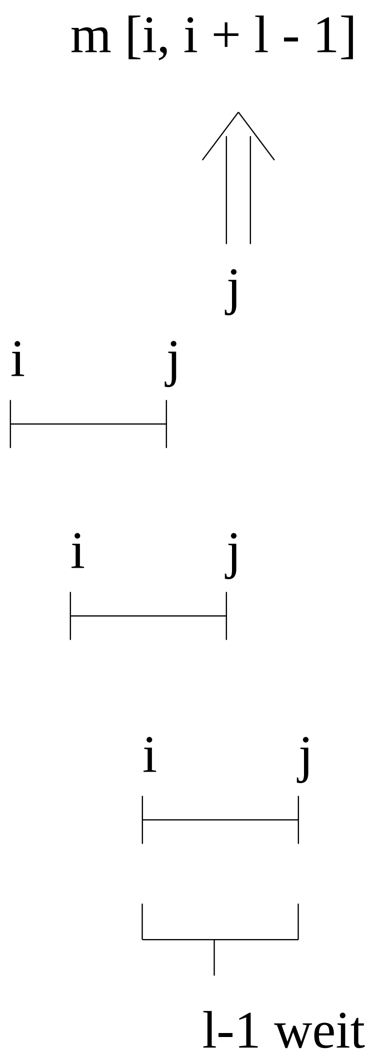
Z1-3: Kosten $m[i, i]$ sind 0.

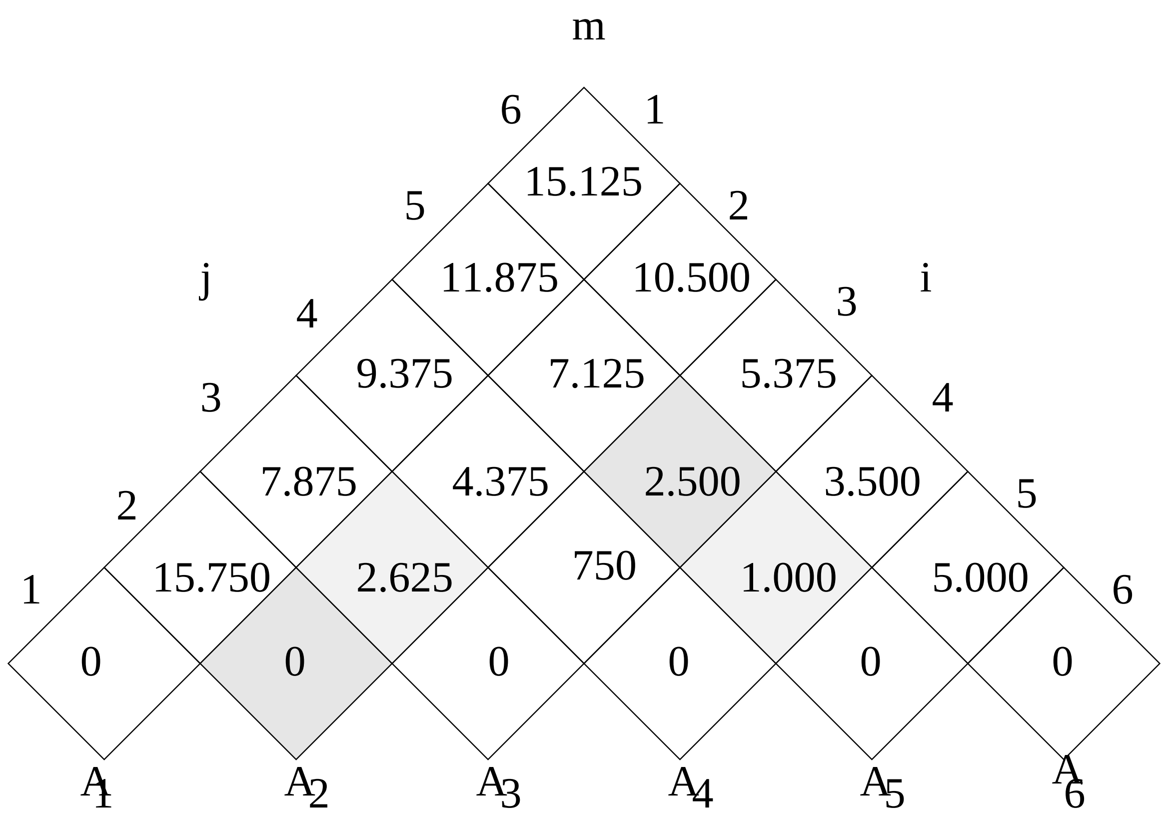
Z4-12: Bestimme Kosten $m[i, i + l - 1]$:

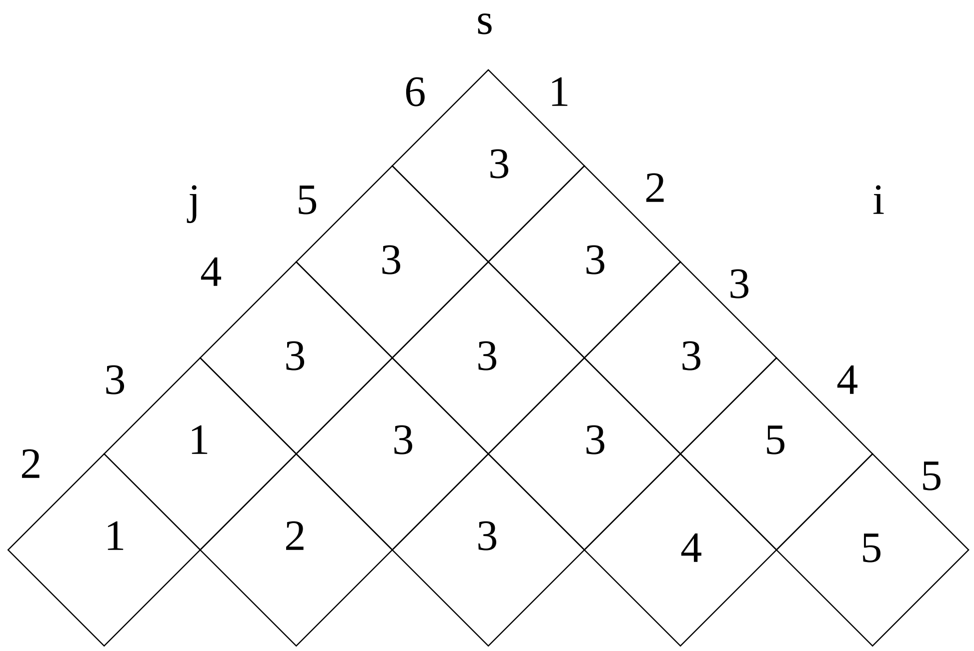
1. $m[i, i + 1]$ zuerst (Länge l der Sequenz $= 2$)
2. $m[i, i + 2]$ dann, usw.

Z9-12: Bestimmung des besten Splits (k)

Anschaulich


m [i, i + l - 1]
j
i
j
i
j
i
j
l-1 weit


m
j
i
15.125
11.875
10.500
9.375
7.125
5.375
7.875
4.375
2.500
3.500
15.750
2.625
750
1.000
5.000
0
0
0
0
0
0
A1
A2
A3
A4
A5
A6


s
j
i
3
3
3
3
3
3
1
3
3
5
1
2
3
4
5

Laufzeit

Laufzeit:

- $O(n^3)$

(viel besser als alle Klammerungen zu bestimmen)

Konstruktion des Optimums

Input:

- Matrizen A $= \langle A_1, \ldots, A_n \rangle$
- Splits $s[i, j]$
- Indizes i, j, deren Bereich zu multiplizieren ist.

Output:

- $A_{i \ldots j}$

Konstruktion des Optimums

$s[i, j]$ beinhaltet das optimale k zum splitten.

letzte Multiplikation um $A_{1...n}$ zu berechen ist:

$$A_{1...s[1,n]}A_{s[1,n]+1...n}$$

letzte M. um $A_{1...s[1,n]}$ z.ber. ist:

$$A_{1...s[1,s[1,n]]}A_{s[1,s[1,n]]+1,s[1,n]}$$

letzte M. um $A_{s[1,n]...n}$ z.ber. ist:

$$A_{s[1,n]+1...s[s[1,n]+1,n]}A + s[s[1, n] + 1, n] + 1, n$$

entsprechend arbeitet der folgende Algorithmus rekursiv:

MATRIX-CHAIN-MULT(A, s, i, j)

```
if j > i
   then X ← MATRIX-CHAIN-MULT(A, s, i, s[i, j])
        Y ← MATRIX-CHAIN-MULT(A, s, s[i, j] + 1, j)
        return MATRIX-MULT(X, Y)
   else return A_i
```

Zutaten für DP

Wann ist DP anwendbar?

Wenn zwei Voraussetzungen gegeben sind:

1. Optimalitätsprinzip
2. gemeinsame Teilprobleme
   häufiges Vorkommen derselben
   weniger Teilprobleme als potentielle Lösungen

Optimalitätsprinzip

Langversion:

> Eine optimale Folge von Entscheidungen besitzt die Eigenschaft, daß unabhängig vom Anfangszustand und von der Anfangsentscheidung die übrigen Entscheidungen eine optimale Entscheidungsreihenfolge bilden müssen, unter Berücksichtigung des aus der ersten Entscheidung resultierenden Zustands.

Kurzversion:

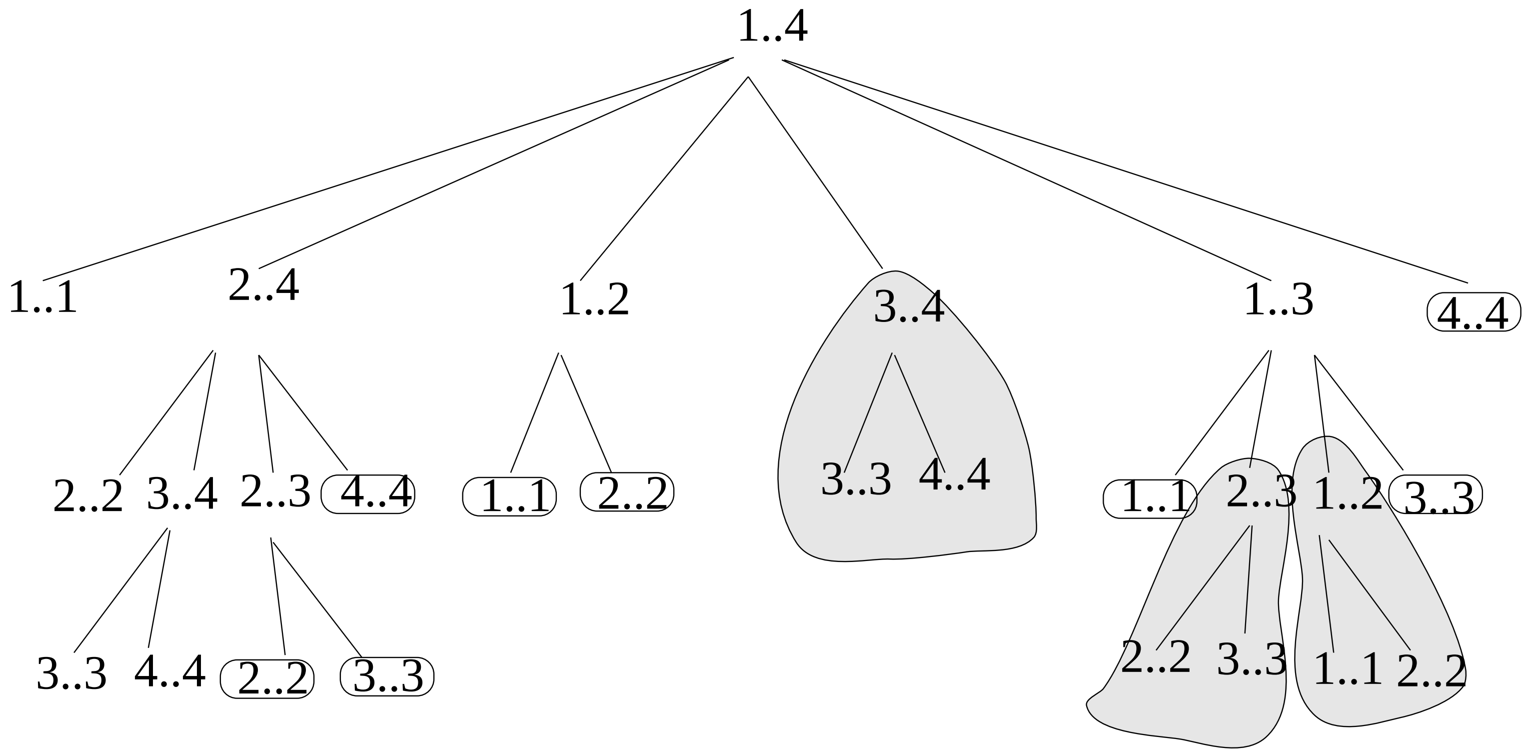
> Die in einer optimalen Lösung vorkommenden Teillösungen sind wiederum optimal.

Gemeinsame Teilprobleme

liegen vor, falls eine rekursive Lösung eines Problems die gleichen Teilprobleme mehr als einmal generiert.

Bsp

REC-MATRIX-CHAIN(p, i, j)
if $i = j$
  then return 0
$m[i, j] \leftarrow \infty$
for $k \leftarrow i$ **to** $j - 1$
    do $q \leftarrow$ REC-MATRIX-CHAIN(p, i, k)
      $+$ REC-MAT-CHAIN$(p, k + 1, j)$
      $+\ p_{i-1} p_k p_j$
      if $q < m[i, j]$
        then $m[i, j] \leftarrow q$
return $m[i, j]$



1..4
1..1
2..4
1..2
3..4
1..3
4..4
2..2
3..4
2..3
4..4
1..1
2..2
3..3
4..4
1..1
2..3
1..2
3..3
3..3
4..4
2..2
3..3
2..2
3..3
1..1
2..2

Analyse

Behauptung: Die Laufzeit $T(n)$ des Algorithmus RECURSIVE-MATRIX-CHAIN ist zumindest exponentiell.

Die Schritte $1-2$ und $6-7$ benötigen mindestens eine Zeiteinheit.

Es ergibt sich die Rekurrenz:

$$\begin{aligned} T(1) &\geq 1 \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\ &= 2 \sum_{i=1}^{n-1} T(i) + n \end{aligned}$$

Wir zeigen $T(n) = \Omega(2^n)$ durch die Substitutionsmethode.

Wir zeigen: $T(n) \geq 2^{n-1}$

I.A.: $T(1) \geq 1 = 2^0$

I.S.:

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2\left(2^{n-1} - 1\right) + n \\ &= \left(2^{n} - 2\right) + n \\ &\geq 2^{n-1} \end{aligned}$$

Memoization

Problem bei rek. Variante:

- Teilprobleme werden mehrfach gelöst

Idee:

- errechnete Lösungen abspeichern
- nachschauen vor lösen

```
MEMOIZED-MATRIX-CHAIN(p)
    n ← length[p] − 1
    for i ← 1 to n
      do for j ← i to n
           do m[i, j] ← ∞
    return LOOKUP-CHAIN(p, 1, n)
```

```
LOOKUP-CHAIN(p, i, j)
    if m[i, j] < ∞
      then return m[i, j]
    if i = j
      then m[i, j] ← 0
      else for k ← i to j − 1
             do q ← LOOKUP-CHAIN(p, i, k)
                    + LOOKUP-CHAIN(p, k + 1, j)
                    + p_{i−1} p_k p_j
                if q < m[i, j]
                  then m[i, j] ← q
    return m[i, j]
```

Laufzeit

$O(n^3)$

Wann DP, wann MEM?

- Wenn alle Teilprobleme berechnet werden müssen, so ist DP schneller.

Längste gemeinsame Teilsequenz

Def. Gegeben seien zwei Sequenzen

$$X = \langle x_1, \ldots, x_m \rangle$$

und

$$Z = \langle z_1, \ldots, z_k \rangle.$$

Z heißt **Untersequenz** von X, falls es eine Sequenz

$$\langle i_1, \ldots, i_k \rangle$$

von Indizes von X gibt, so daß für alle $1 \leq j \leq k$ gilt:

$$x_{i_j} = z_j.$$

Bsp

$$Z = \langle B, C, D, B \rangle$$

ist eine Untersequenz von

$$X = \langle A, B, C, B, D, A, B \rangle$$

Längste gemeinsame Teilsequenz

Def. Gegeben zwei Sequenzen X und Y. Eine Sequenz Z heißt **gemeinsame Untersequenz** von X und Y, falls Z eine Untersequenz von X und Y ist.

Bsp. Für

$$\begin{aligned} X &= \langle A, B, C, B, D, A, B \rangle \\ Y &= \langle B, D, C, A, B, A \rangle \end{aligned}$$

ist

$$\langle B, C, A \rangle$$

eine gemeinsame Untersequenz von X und Y.

Sie ist aber nicht die **längste gemeinsame Untersequenz** (LCS) LCS's von X und Y sind $\langle B, C, B, A \rangle$ und $\langle B, D, A, B \rangle$.

Längste gemeinsame Teilsequenz

Problemdefinition

Gegeben

- Sequenzen
  - $X = \langle x_1, \ldots, x_m \rangle$
  - $Y = \langle y_1, \ldots, y_n \rangle$

Gesucht

- längste gemeinsame Teilsequenz Z von X und Y

Brute Force Solution

1. bestimme alle Teilsequenzen von X
2. test jede, ob sie auch Teilsequenz von Y ist
3. nimm längste solche

Brute Force Solution (Aufwand)

- jede Teilsequenz von X korrespondiert zu einer Menge $\{1, \ldots, m\}$ von Indizes von X
- es gibt 2^m solche Teilsequenzen
- Gesamtaufwand also mindestens exponentiell

Optimalitätsprinzip

basiert auf Präfixen.

Def. Sei $X = \langle x_1, \ldots, x_m \rangle$ eine Sequenz. Für $i \leq m$ ist $X_i = \langle x_1, \ldots, x_i \rangle$ ein Präfix von X.

Bsp.

- $X = \langle A, B, C, B, D, A, B \rangle$
- $X_4 = \langle A, B, C, B \rangle$

Optimalitätsprinzip

Satz Seien $X = \langle x_1, \ldots, x_m \rangle$ und $Y = \langle y_1, \ldots, y_n \rangle$ Sequenzen und sei $Z = \langle z_1, \ldots, z_k \rangle$ eine LCS. Dann gilt:

1. $x_m = y_n \Longrightarrow$
   - $z_k = x_m = y_n$ und
   - Z_{k-1} ist eine LCS von X_{m-1} und Y_{n-1}
2. $x_m \neq y_n$ und $z_k \neq x_m \Longrightarrow$
   - Z ist eine LCS von X_{m-1} und Y
3. $x_m \neq y_n$ und $z_k \neq y_n \Longrightarrow$
   - Z ist eine LCS von X und Y_{n-1}

Anm. Jede LCS enthält eine LCS von zwei Präfixen.

Beweis

(1) Falls $z_k \neq x_m$, so könnten wir $x_m = y_n$ an Z hängen und hätten eine längere LCS von X und Y. (Widerspruch).

Also muß gelten $z_k = x_m = y_n$.

Offensichtlich: Z_{k-1} ist eine $(k-1)$-lange CS von X_{m-1} und Y_{n-1}.

z.z.: Z_{k-1} ist LCS.

Annahme: $\exists$ W CS von X_{m-1} und Y_{n-1}, W länger als $k-1$.

Dann erhalten wir durch anhängen von $x_m = y_n$ eine CS länger als k.

(Widerspruch).

(2) Da $z_k \neq x_m$ ist Z eine CS von X_{m-1} und Y.

Annahme: $\exists$ W CS von X_{m-1} und Y und $|W| > k$.

Dann ist W auch CS von X und Y.

(Widerspruch).

(3) Analog.

Rekursive Lösung

Nach Satz: Zur Lösung des Problems sind entweder eine oder zwei Alternativen zu untersuchen:

- Falls $x_m = y_n$, müssen wir eine LCS von X_{m-1} und Y_{n-1} finden und $x_m = y_n$ anhängen.
- Falls $x_m \neq y_n$, so müssen wir die LCS von

  1. X und Y_{n-1}
  2. X_{m-1} und Y

  suchen und die Längere zurückgeben.

  Beide dieser Probleme haben das Unterproblem

  – finde LCS von X_{m-1} und Y_{n-1}.

Rekursive Lösung

Wie beim Matrizensequenzenmultiplikationsproblem erarbeiten wir eine Rekurrenz zur Beschreibung der Kosten der optimalen Lösung.

Sei $c[i,j]$ die Länge einer LCS der Sequenzen X_i und Y_j.

Falls entweder $i = 0$ oder $j = 0$, so ist $c[i,j] = 0$.

Rekurrenz:

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ oder } j = 0 \\ c[i-1][j-1] + 1 & i,j > 0 \text{ und } x_i = y_j \\ max(c[i,j-1], c[i-1,j]) & i,j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Berechnung der Länge einer LCS

Mit dieser Rekurrenz könnten wir schnell einen rekursiven Algorithmus schreiben. Dieser hätte aber exponentiellen Aufwand.

Da die Anzahl der Teilprobleme aber nur $\Theta(mn)$ ist, benutzen wir dyn. Programmieren (also einen bottom-up Ansatz).

Die Prozedure LCS-LENGTH nimmt als Argumente zwei Sequenzen $X = \langle x_1, \ldots, x_m \rangle$ und $Y = \langle y_1, \ldots, y_n \rangle$ und berechnet die $c[i,j]$ in einem Feld $c[0 \ldots m, 0 \ldots n]$.

(Berechnung der $c[i,j]$ erfolgt von zeilenweise links nach rechts und dann von oben nach unten.)

Um die LCS zu bestimmen, verwalten wir noch $b[i,j]$. Dies enthält Verweise auf die optimalen Teillösungen, die für $c[i,j]$ gewählt wurden.

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
   do c[i,0] ← 0
for j ← 0 to n
   do c[0,j] ← 0
for i ← 1 to m
   do for j ← 1 to n
      do if x_i = y_j
         then c[i, j] ← c[i −1, j − 1] + 1
            b[i, j] ← "↖"
         else if c[i − 1, j] ≥ c[i, j − 1]
               then c[i, j] ← c[i − 1, j]
                  b[i, j] ← "↑"
               else c[i, j] ← c[i, j − 1]
                  b[i, j] ← "←"
return c and b
```

Laufzeit: $O(mn)$.

Zeiger: Merken sich, wo man größte Sublösung gefunden hat:

$\nwarrow$ in $b[i,j]$: $x_i = y_j$ ist in LCS

$\uparrow$ in $b[i,j]$: LCS enthält nicht $x_i = y_j$, schaue bei $b[i-1,j]$

$\leftarrow$ in $b[i,j]$: LCS enthält nicht $x_i = y_j$, schaue bei $b[i,j-1]$

Z1-6 Initialisierung

Z7/8 Schleife zum füllen von $c[i,j]$ und $b[i,j]$

Z9-16 Fallunterscheidung gemäß Rekurrenz

Beispiel

Konstruktion der LCS

PRINT-LCS(b, X, i, j)
if $i = 0$ or $j = 0$
  then return
if $b[i, j] =$ "$\nwarrow$"
  then PRINT-LCS($b, X, i - 1, j - 1$)
    print x_i
elseif $b[i, j] =$ "$\uparrow$"
  then PRINT-LCS($b, X, i - 1, j$)
else PRINT-LCS($b, X, i, j - 1$)

Initialer Aufruf: LCS-LENGTH(b,X,length[X],length[Y]).

Laufzeit: $O(n + m)$.

Achtung: Besuchsreihenfolge ist umgekehrter Ausgabereihenfolge. (rechtsrekursiv)

Codeverbesserungen

$b[i,j]$ ist nicht notwendig, da wir anhand von $c[i-1,j-1]$, $c[i,j-1]$ und $c[i-1,j]$ die Entscheidung bei $c[i,j]$ rekonstruieren können.

Details: Übung.

Einleitung

Optimierungsalgorithmen durchlaufen üblicherweise eine Anzahl von Schritten, bei denen dann eine

Wahl

getroffen werden muß.

Ein **gieriger Algorithmus** (greedy algorithm) nimmt die Wahl, die im Moment am günstigsten aussieht:

Nimm immer das größte Stück Kuchen.

Die Hoffnung hier ist dann, daß die global optimale Lösung sich aus lokalen optimalen Entscheidungen zusammensetzt.

Dies ist aber nicht immer garantiert!!!

Anwendungen

Viele Algorithmen, die wir noch kennenlernen werden, sind gierige Algorithmen:

- Huffman-Kompression
- minimale spannende Bäume
- kürzeste Wege

Aktivitätsauswahlproblem

Wir wollen Aktivitäten für eine Resource einplanen.

Gegeben

1. eine Resource R, die zu einem Zeitpunkt nur von einer Aktivität genutzt werden kann.
   Bsp: R ist ein Hörsaal.
2. n Aktivitäten $S = \{1, 2, \ldots, n\}$
3. jede Aktivität i hat
   - eine Startzeit s_i und
   - eine Endzeit f_i.

   mit $s_i \leq f_i$.

Zwei Aktivitäten i und j heißen **kompatibel**, falls die Intervalle $[s_i, f_i[$ und $[s_j, f_j[$ nicht überlappen.

Gesucht

- maximale Teilmenge kompatibler Aktivitäten von S

Der gierige Algorithmus

Annahme

- die Aktivitäten sind sortiert nach aufsteigenden Endzeiten:
$$f_1 \leq f_2 \leq \ldots \leq f_n$$
- Repräsentation der s und f als arrays.
- nimm immer diejenige Aktivität, die nicht mit den vorherigen Aktivitäten kollidiert und die kleinste Endzeit hat
- somit wird die Zeit, die noch für die Resource zur Verfügung steht maximiert

GREEDY-ACTIVITY-SELECTOR(s, f)

$n \leftarrow$ **length**[s]
$A \leftarrow \{1\}$
$j \leftarrow 1$
for $i \leftarrow 2$ **to** n
    do if $s_i \geq_j$
        then $A \leftarrow A \cup \{i\}$
            $j \leftarrow i$
return A

Anmerkungen

1. $f_j = max\{f_k | k \in A\}$
2. Z2-3: initialisiere A und j.
3. Z4-7: untersuche jede Aktivität i (Z4) auf Kompatibilität mit den Aktivitäten in A (Z5). Falls i kompatibel zu A, so kann A um i erweitert werden (Z6) und j muß angepaßt werden.
4. Z8: gib Ergebnis A zurück
5. Laufzeit: $\Theta(n)$, falls die Aktivitäten schon nach f_i sortiert. Zusätzliches Sortieren kostet höchstens $O(n \lg n)$.

Optimalität

Satz Der Algorithmus GREEDY-ACTIVITY-SELECTOR findet für ein Aktivitätsauswahlproblem eine optimale Lösungen.

Beweis Sei $S = \{1, 2, \ldots, n\}$ die Menge der Aktivitäten.

Sei $f_1 \leq f_2 \leq \ldots \leq f_n$.

Wir zeigen, daß es eine optimale Lösung gibt, die Aktivität 1 enthält.

Sei $A \subseteq S$ eine optimale Lösung.

Seien die Aktivitäten in A gemäß Endzeiten sortiert.

Sei k die erste Aktivität in A.

Falls $k = 1$, so sind wir fertig.

Falls $k \neq 1$, so zeigen wir, daß $B = A \setminus \{k\} \cup \{1\}$ eine optimale Lösung ist (die 1 enthält).

Da $f_1 \leq f_k$, sind die Aktivitäten in B kompatibel.

Da $|B| = |A|$ ist B auch eine optimale Lösung.

Also gibt es immer eine optimale Lösung, die 1 enthält.
Einfache Induktion ergibt die Behauptung:

Optimalitätsprinzip

> Falls A eine optimale Lösung (für S) ist, die 1 enthält, so ist $A' = A \backslash \{1\}$ eine optimale Lösung für $S' = \{i \in S | s_i \geq f_1\}$

Da

> Falls es eine Lösung B' von S' gibt mit $|B'| > |A'|$, so wäre $B = B' \cup \{1\}$ eine Lösung von S mit $|B| > |A|$. (Widerspruch.)

Daher verkleinert sich das Problem nach einer gierigen Auswahl.

Induktion über die Anzahl der Auswahlen vervollständigt den Beweis.

Zutaten für gierige Algorithmen

1. Gierige-Auswahl-Eigenschaft
2. Optimalitätsprinzip

Gierige-Auswahl-Eigenschaft

Eine global optimale Lösung kann durch eine Folge von lokal optimalen (gierige) Auswahlen konstruiert werden.

Zur Erinnerung: Beim DP wurde hing die Auswahl von der optimalen Lösung von Teilproblemen ab. Dies darf hier nicht sein.

Die Auswahl bei gierigen Algorithmen kann von vorherigen Auswahlen abhängig sein, aber niemals von noch zu machenden.

Gierige Algorithmen arbeiten top-down: ein Problem wird durch eine Wahl in ein kleineres transformiert.

Optimalität ist aber immer zu beweisen. Typische Vorgehensweise wie oben: Man nehme eine optimale Lösung und zeige, daß man sie in eine optimale Lösung transformieren kann, die von gierigen Algorithmus erzeugt wird. (per Induktion)

Optimalitätsprinzip

wie bei DP, war oben bei Induktionsschluss notwendig.

Wann DP wann GA?

Zwei Fehler:

1. Man könnte DP verwenden, wenn GA ausreicht.
2. Man könnte GA verwenden, wenn DP notwendig ist.

Zur Illustration zwei Beispielprobleme:

1. 0-1-Rucksack-Problem
2. fraktale Rucksack-Problem

0-1-Rucksackproblem

Raub eines Ladens. Jeder Gegenstand i kostet v_i SF und wiegt w_i Kilogramm.

Der Dieb will natürlich den Wert seines Raubs maximieren. Leider kann er aber maximal W Kilogramm tragen.

Heißt 0-1, da der Dieb einen Gegenstand entweder ganz oder gar nicht und nicht mehrfach mitnehmen kann.

Fraktales Rucksackproblem

Gleiches Szenario, aber der Dieb kann auch Teile von Gegenständen mitnehmen.

(Juwelierladen, Brillianten rausbrechen, Brilliantsplitter, Goldbarren, halbe Goldbarren, Goldstaub)

Eigenschaften

Beide Probleme gehorchen Optimalitätsprinzip

0-1:

> Man nehme einen optimal gefüllten Rucksack mit Gewicht W.
>
> Entfernt man einen Gegenstand i, so muß der Rucksack eine optimale Lösung für das Gewicht $W - w_i$ sein.

Frak:

> Nimm optimalen Rucksack mit Gewicht $\leq W$. Falls wir das (einen Teil eines Gegenstandes mit) Gewicht w entfernen, so muß das Ergebnis eine optimale Lösung für $W - w$ sein.
>
> Dabei kommen alle Gegenstände außer j in Frage plus der Teil von j mit Gewicht $w_j - w$.

Eigenschaften

Obwohl die Probleme ähnlich sind gilt:

- Frak. Rucksack kann mit gierigem Algorithmus gelöst werden.
- 0-1 Rucksack kann nicht mit gierigem Algorithmus gelöst werden, erfordert also DP.

Frak. Rucksack

gieriger Algorithmus:

1. sortiere die Gegenstände nach abfallenden v_i/w_i Koeffizienten.
   (also nach Wert pro gewicht).
2. nimm immer soviel wie möglich von dem Gegenstand mit höchstem v_i/w_i Koeffizienten.
   Grenzen: W, Gegenstand ganz geschluckt.

Beweis, daß die gierige Auswahleigenschaft gilt: Übung.

Aufwand: $O(n \lg n)$

0-1 Rucksack

gieriger Algorithmus suboptimal (Bsp.,$W = 50$):

Gegenstand	Wert	Gewicht	w/g
1	60	10	6
2	100	20	5
3	120	30	4

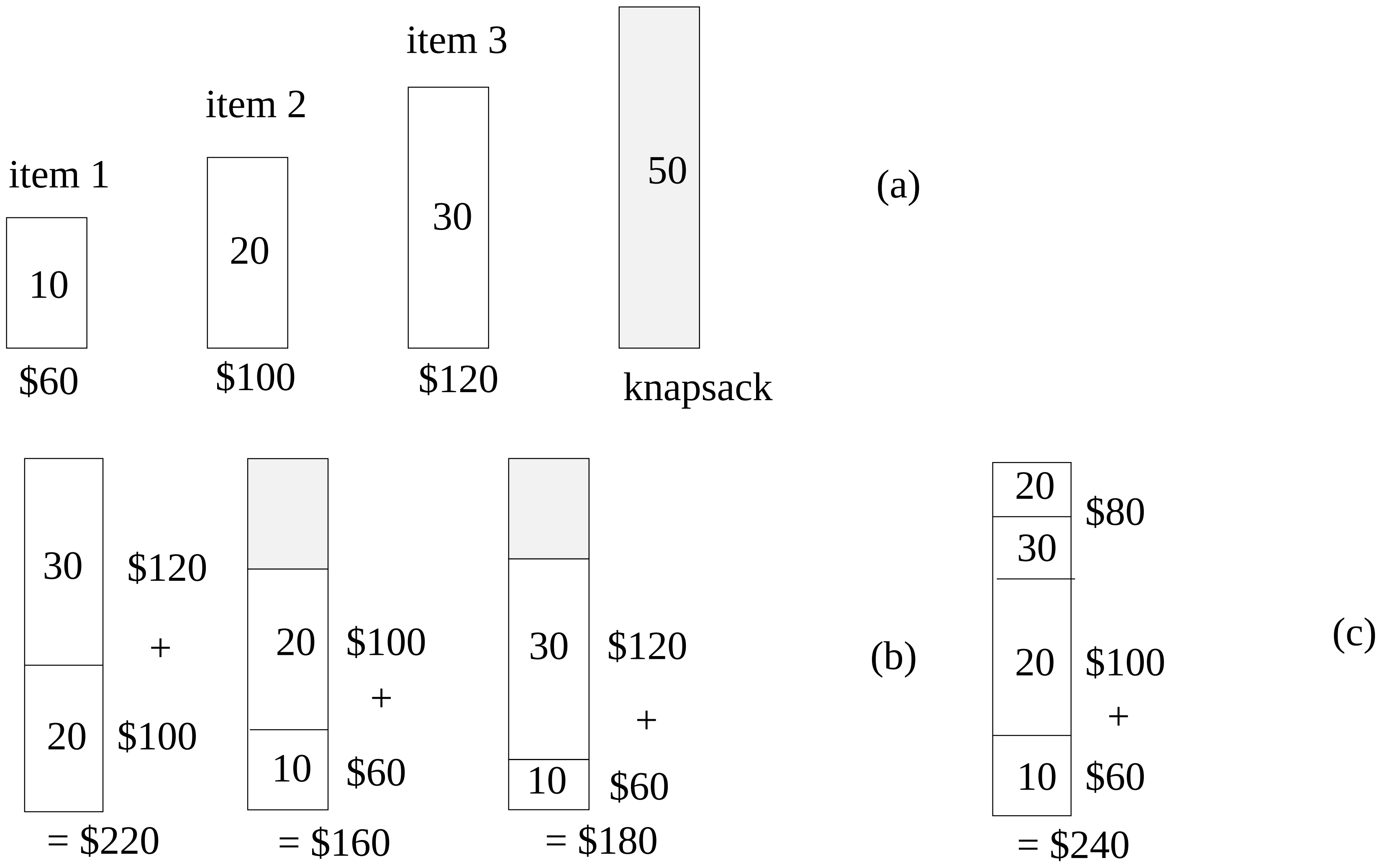
Ergebnis:

Wahl	Gesamtgewicht	Gesamtwert
1	10	60
2	30	160

Besser:

Wahl	Gesamtgewicht	Gesamtwert
2	20	100
3	50	220

Jede Lösung mit 1 ist suboptimal.



item 1
item 2
item 3
10
20
30
50
$60
$100
$120
knapsack
(a)
30
$120
+
20
$100
= $220
20
$100
+
10
$60
= $160
30
$120
+
10
$60
= $180
(b)
20
$80
30
20
$100
+
10
$60
= $240
(c)

Huffman Kompression

Aufgabe:

Gegegeben eine Datei. Kann man die enthaltenen Daten so komprimieren, so daß die komprimierte Datei weniger Plattenplatz weg nimmt, und die Originaldatei wieder rekonstruiert werden kann?

Dies ist oft mit Huffman Kompression möglich. Kompressionsraten: 20%-80%.

Grundlage:

Kodierung aller Zeichen einer Daten mittels unterschiedlich langer Bitmuster.

Die Kodierung heißt dann Huffman Code.

Huffman Code

Gegeben: Datei mit 100.000 Zeichen, aber nur 6 verschiedenen (a-f).

Normaler Speicheraufwand: 100KB.

Kodieren:

	a	b	c	d	e	f
Häufigkeit (in K)	45	13	12	16	9	5
Code fester Länge	000	001	010	011	100	101
Code variabler L.	0	101	100	111	1101	1100

Aufwand:

- feste L.: 37.5KB
- $(45*1+13*3+12*3+16*3+9*4+5*4)/8 = 28.0KB$

Präfix-Kodierungen

Damit wir auch wieder dekodieren können, ist es sinnvoll nur Präfixkodierungen zu betrachten.

Def. Eine **Präfixkodierung** ist eine Kodierung bei der keine zwei Codeworte den gleichen Präfix haben.

Satz Für jede Zeichenkodierung, die zu einer optimalen Kompression führt, gibt es eine nicht schlechtere Präfixkodierung.

Vorteile Präfixkodierung

einfache Codierung:

$$abc \rightsquigarrow 0 \cdot 101 \cdot 100 = 0101100$$

einfache Dekodierung:

$$0101100 \rightsquigarrow a101100 \rightsquigarrow ab100 \rightsquigarrow abc$$

1. erster Buchstabe kann nur a sein.
2. dann kann zweiter Buchstabe nur b sein und
3. dritter nur c.

Repräsentation von Codierungen

Repräsentation als Binärbäume:

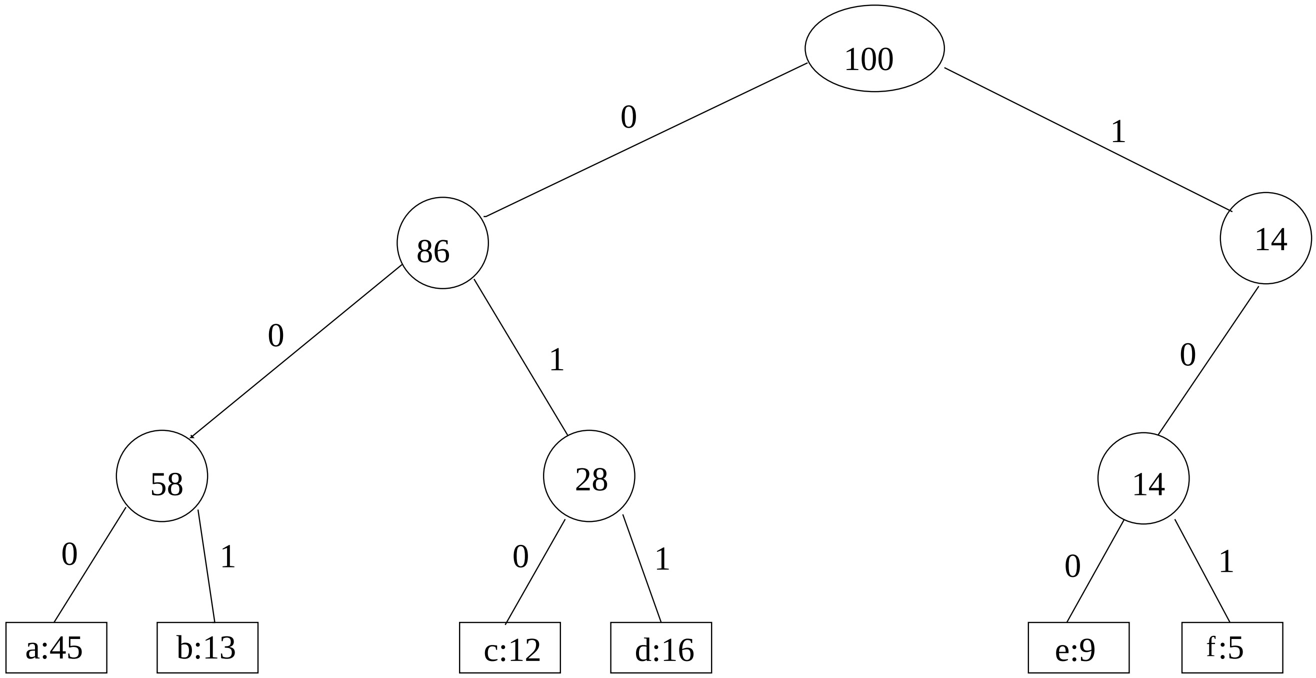
Linker Sohn für 0 rechter Sohn für 1.

Diese sind zum Dekodieren sehr nützlich.

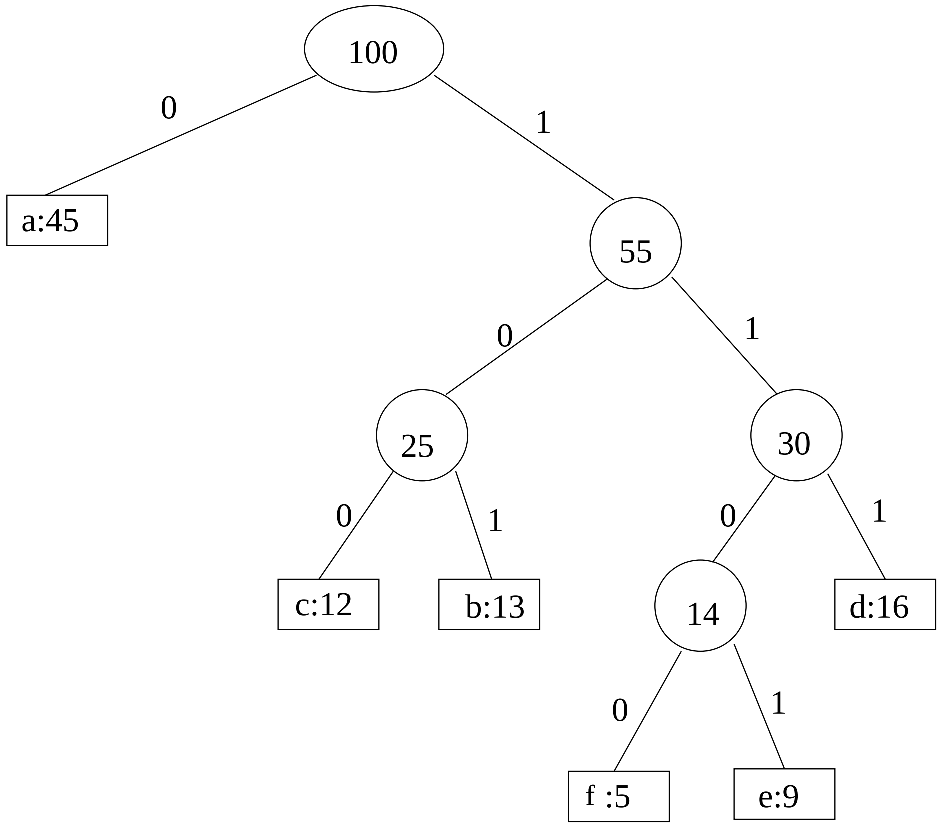
Optimale Kodierungen sind immer vollständige Binärbäume, das heißt, jeder nicht-Blattknoten hat zwei Söhne.

Wir werden nur solche betrachten.

Falls C das Alphabet ist (unsere Zeichenmenge), dann gibt es genau $|C|$ Blätter, eines für jedes Zeichen, und genau $|C| - 1$ innere Knoten.



(a)



(b)

Anzahl der benötigten Bits

Gegeben sei

- zu einem Präfixcode ein Baum T
- für jedes Zeichen c die Häufigkeit $f(c)$ in der zu kodierenden Datei
- und $d_T(c)$ die Tiefe des Blattes von c, also die Anzahl der Bits, die notwendig sind um c zu kodieren.

Dann ist die Anzahl der Bits, die notwendig sind um eine Daten zu kodieren:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Dies seien die **Kosten** des Codes.

Konstruktion von Huffmankodierungen

Huffman erfand einen gierigen Algorithmus, der den optimalen Präfixcode konstruiert.

Der Algorithmus baut einen entsprechenden Baum T auf.

Er startet mit $|C|$ Blättern und verbindet dann je zwei zu einem Baum solange bis nur noch ein Baum übrigbleibt.

Im Algorithmus:

- $|C| = n$
- $f[c]$ gleich $f(c)$
- Q ist eine Priority-Queue mit f als Schlüssel

Das f für einen verbundenen Baum ist die Summe der zwei Teil-f.

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do z ← ALLOCATE-NODE()
        x ← left[z] ← EXTRACT-MIN(Q)
        y ← right[z] ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT (Q, z)
return EXTRACT-MIN(Q)
```

Z2: initialisiert Q

Z3-8: holt die beiden Knoten x und y mit der geringsten Häufigkeit aus der Queue Q und fügt den neuen Knoten z der x und y als Söhne hat wieder ein.

Korrektheit

Um zu zeigen, daß HUFFMAN immer den optimalen Präfixcode liefert, zeigen wir, daß das entsprechende Problem die gierige Auwahleigenschaft hat und das Optimalitätsprinzip gilt.

Lemma 17.2 Sei C ein Alphabet und seien $f[c]$ die Häufigkeit von $c \in C$. Sind x und y die beiden Zeichen mit geringster Häufigkeit, dann gibt es einen optimalen Präfixcode für C, in dem die Codes für x und y die gleiche Länge haben und der nur um ein Bit differiert.

Beweis Sei T ein Baum, der einen optimalen Präfixcode repräsentiert.

Wir modifizieren T solange, bis die Folgerung des Satzes gilt, also x und y Brüder sind und beide die maximale Tiefe im neuen Baum haben.

Dabei achten wir darauf, daß die Kosten von T gleich bleiben.

Seien b und c zwei Zeichen, die Brüder sind und die größte Tiefe in T haben.

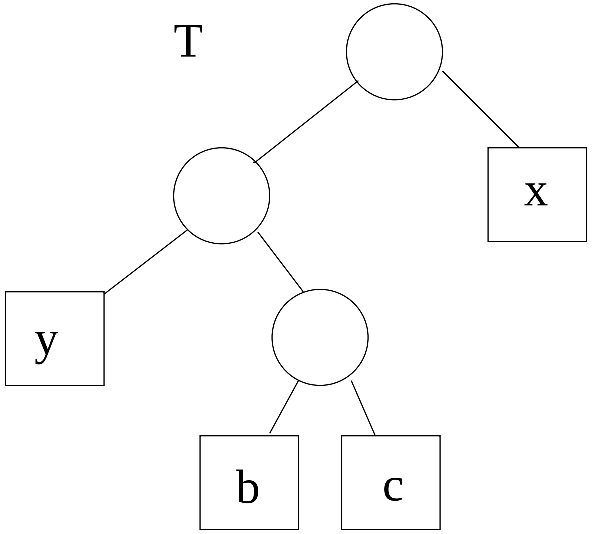
ObdA: $f[b] \leq f[c]$ und $f[x] \leq f[y]$.

Da x und y die geringste Häufigkeit haben, gilt $f[x] \leq f[b]$ und $f[y] \leq f[c]$.

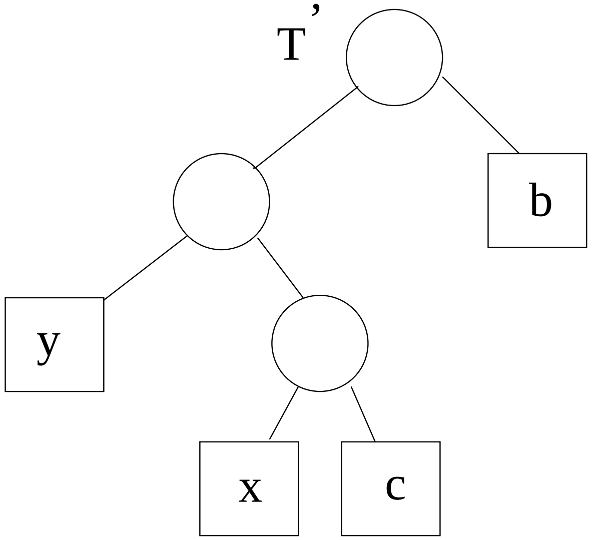
Wie in der folgenden Abbildung gezeigt, modifizieren wir

- T indem wir x und b vertauschen um T' zu erhalten und modifizieren T' weiter
- indem wir y und c vertauschen. Dies ergibt T''.

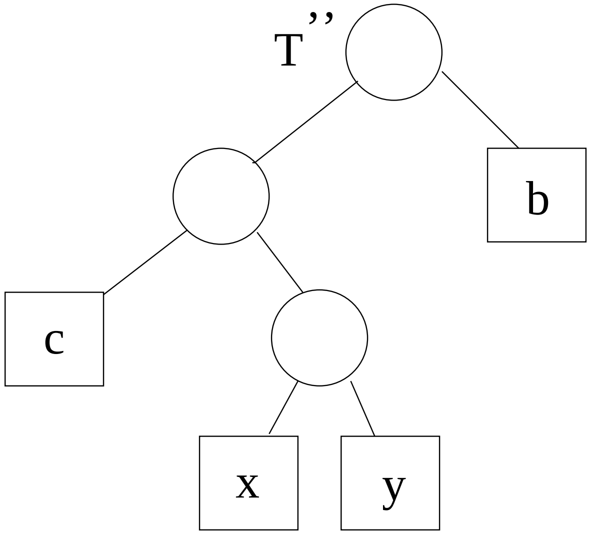
T



T'



T''



Wir erhalten $B(T) - B(T')$

$$\begin{aligned}
&= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) f[b]d_{T'}(b) \\
&= f[x]d_T(x) + f[b]d_T(b) - f[x]d_T(b) f[b]d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geq 0
\end{aligned}$$

da $f[b] - f[x] \geq 0$ und $d_T[b] - d_T[x] \geq 0$

Analog: $B(T') - B(T'') \geq 0$.

Also: $B(T'') \leq B(T)$.

Da T optimal gilt $B(T'') = B(T)$.

$\square$

Korrektheit

Lemma 17.3 Sei T ein vollständiger Binärbaum (jeder nicht-Blattknoten hat zwei Nachfolger), der einen optimalen Präfixcode eines Alphabets C mit Häufigkeiten $f[c]$ für $c \in C$ darstellt.

Seien x und y zwei Brüder in T und z der Vater.

Falls wir dann z als einen (neuen) Buchstaben mit Häufigkeit $f[z] := f[x]+f[y]$ betrachten, so ist $T' = T \setminus \{x, y\}$ ein optimaler Präfixbaum für $C' = C \setminus \{x, y\} \cup \{z\}$.

Beweis Es gilt:

1. $\forall c \in C \setminus \{x, y\}\ d_T(c) = d_{T'}(c)$ und damit
2. $f[c]d_T[c] = f[c]d_{T'}(c)$. Weiter gilt
3. $d_T(x) = d_T(y) = d_{T'}(z) + 1$.

Also:

$$\begin{aligned} f[x]d_t(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

Es folgt:

$$B(T) = B(T') + f[x] + f[y]$$

Falls T' keinen optimalen Präfixcode für C' repräsentiert, so gibt es einen optimalen T'' mit $B(T'') < T(T)$, indem z als Blatt vorkommt.

Machen wir x und y zu Söhnen von z, so erhalten wir einen Präfixcode von C mit Kosten

$$B(T'') + f[x] + f[y] < B(T)$$

Widerspruch zur Optimalität von T.

Korrektheit

Aus den beiden vorherigen Lemmata folgt direkt:

Theorem Die Funktion HUFFMANN erzeugt einen optimalen Präfixcode.

Mengensysteme

Def Ein **Mengensystem** ist ein Paar (E, S), wobei E eine Menge und S eine unter Inklusion abgeschlossene Teilmenge der Potenzmenge von E ist. $(S \subseteq \mathcal{P}(E))$.

Die Elemente aus S heißen unabhängige Mengen.

Zu (S, E) gibt es ein zugehöriges Optimierungsproblem: Für eine gegebene Gewichtsfunktion $w : E \to \mathcal{R}_I^+$ ist eine unabhängige Menge zu bestimmen, deren Gewicht

$$w(A) = \sum_{e \in A} w(e)$$

maximal ist.

Die Beschränkung auf nicht-negative Gewichte sicher zu, daß eine unabängige Menge maximalen Gewichts o.B.d.A. auch eine maximale unabhängige Menge ist.

Matroid

Def. Ein Mengensystem (E, S) heißt Matroid, falls die folgende Bedingunge erfüllt ist:

- Für $J, K \in S$ mit $|J| = |K| + 1$ gibt es stets ein $a \in J \setminus K$ mit $K \cup \{a\} \in S$.

Diese Eigenschaft heißt auch Austauscheigenschaft.

Sie ist das Analogon des Steinitzschen Austauschsatzes der linearen Algebra.

Bsp. Sei $G = (V, E)$ ein Graph. Sei S die Menge der Teilmengen von E, die Bäume bilden. Dann ist (E, S) ein Matroid.

Bsp. Sei E eine endliche Teilmenge eines Vektorraums V und S die Menge aller linear unabhängigen Teilmengen von E. Dann ist (E, S) ein Matroid.

Greedy Algorithm

Gegeben sei ein Mengensystem $M = (E, S)$

GREEDY(M, w)
sort $E = \{e_1, \ldots, e_n\}$ fallend
$w(e_1) \geq \ldots \geq w(e_n)$
T $\leftarrow \emptyset$
for $i \leftarrow 1$ **to** n
  do if $T \cup \{e_i\} \in S$
    then $T \leftarrow T \cup \{e_i\}$
return T

Matroid&Greedy

Satz Sei $M = (E, S)$ ein Mengensystem. Dann sind folgende Aussagen äquivalent:

- GREEDY(M, w) produziert für all w das optimale Ergebnis
- M ist ein Matroid
- Für jede Teilmenge A von E haben alle maximal unabhängigen Teilmengen von A dieselbe Mächtigkeit

Anm Bedingung (3) ist das Analogon für die Gleichmächtigkeit der Basen eines Unterraums.

Beweis

(1) $\Longrightarrow$ (2): Angenommen, GREEDY(M, w) produziert für all w das optimale Ergebnis und M ist kein Matroid $\Longrightarrow$

$\exists J, K \in S \;\; |J| = |K| + 1 \wedge (\forall a \in J \setminus K \;\; K \cup \{a\} \notin S$

Für $k := |K|$:

$$w(e) := \begin{cases} k+2 & e \in K \\ k+1 & e \in J \setminus K \\ 0 & \text{sonst} \end{cases}$$

K keine Lösung des Optimierungsproblems, da

$$w(K) = k(k+2) < (k+1)^2 = w(J)$$

GREEDY wählt aber zuerst alle Elemente aus K, da diese das größte Gewicht haben. Danach wird aber das Gewicht nicht weiter vergrößert, da für alle e entweder $w(e) = 0$ oder $e \in J \setminus K$ (kann also nicht zu K dazugenommen werden).

Also erzeugt GREEDY suboptimales Ergebnis. (Widerspruch.)

(2) $\Longrightarrow$ (3):

Sei $A \subseteq E$, beliebig.

Seien $J, K \subseteq A$ zwei maximale unabhängige Teilmengen.

(d.h.: keine J oder K echt enthaltende Teilmenge liegt in S)

Annahme: $|K| < |J|$. Da S unter Inklusion abgeschlossen ist:

$$\exists J' \subseteq J \quad |K| = |J'| + 1$$

Nach (2)

$$\exists a \in J' \setminus K \quad K \cup \{a\} \in S$$

(Widerspruch.)

(3) $\Longrightarrow$ (1):

Annahme: (3) und GREEDY suboptimal

Dann existiert w so daß GREEDY

$$K = \{e_1, \ldots, e_k\}$$

konstruiert, obwohl es

$$J = \{e'_1, \ldots, e'_h\}$$

gibt mit $w(J) > w(K)$.

obdA: K, J nach absteigendem Gewicht geordnet und J maximale unabh. Teilmenge von E.

Nach Konstruktion ist auch K maximal.

mit (3) für $A = E$ gilt also $h = k$.

Wir zeigen durch Induktion:

$$w(e_i) \geq e(e'_i)$$

(Widerspruch zu $w(K) < w(J)$)

I.A.: Nach Definition von GREEDY hat e_1 maximales Gewicht. $\surd$

I.S.: Annahme gelte für $i \leq n$.

Annahme: $w(e_{n+1}) < w(e'_{n+1})$.

Sei $A = \{e \in E | w(e) \geq w(e'_{m+1}\}$.

Dann ist $\{e_1, \ldots, e_m\}$ eine maximale unabh. Teilmenge von A, denn wenn es ein e geben würde mit $\{e_1, \ldots, e_m, e\} \in S$, so $w(e) \leq w(e_{m+1}) < w(e'_{m+1})$, also $e \notin A$.

Da aber auch $\{e'_1, \ldots, e'_{m+1}\}$ eine unabh. Teilmenge von A ist, erhalten wir einen Widerspruch zu (3).

Matroide

Die maximalen unab. Teilmengen eines Matroids $M = (E, S)$ werden **Basen** genannt.

Damit berechnet GREEDY eine Basis von M.

Der **Rang** $\rho(A)$ einer Teilmenge A von E ist die Mächtigkeit einer maximalen unabh. Teilmenge von A.

Jede nicht in S enthaltene Menge heißt abhängig.

Eine minimale abhängige Menge heißt Zyklus.

Dualität

Satz Sei $M = (E, S)$ ein Matroid. Dann ist auch $M^* = (E, S^*)$ mit

$$S^* = \{J \subseteq E | \exists \text{Basis } B \text{ von } M \text{ mit } J \subseteq E \setminus B\}$$

ein Matroid, dessen Rangfunktion ρ^* durch

$$\rho^*(A) = |A| + \rho(E \setminus A) - \rho(A)$$

gegeben ist.

Def M^* heißt dualer Matroid von M.

Dieser Dualitätsbegriff ist ein anderer als in der linearen Algebra. Es gilt also insbesondere nicht, daß der duale Matroid eines endlichen Vektorraums der Matroid des Dualraums ist.

Ein Zyklus in M^* heißt Cozyklus von M.

Gegeben sei ein Matroid $M = (E, S)$ und eine Gewichtsfunktion w.

DUAL-GREEDY(M, w)
sort $E = \{e_1, \ldots, e_n\}$ steigend
$w(e_1) \leq \ldots \leq w(e_n)$
T $\leftarrow$ E
for $i \leftarrow 1$ **to** n
    do if $(E \setminus T) \cup \{e_i\}$ enthält keinen Cozyklus
        then $T \leftarrow T \setminus \{e_i\}$
return T

Satz Der Algorithmus DUAL-GREEDY berechnet eine Basis B von $M = (E, S)$, die maximales Gewicht hat.

Motivation

Graphen kommen immer und immer wieder vor.

Wir betrachten einige elementare Algorithmen auf Graphen, die Probleme lösen, die man immer wieder zu lösen hat.

1. Repräsentation von Graphen
2. Graphtraversierungen
3. topologische Sortierung
4. Zerlegung in strenge Zusammenhangskomponenten
5. minimale Spannbäume
6. kürzeste Wege
7. Flußprobleme

Vereinbarungen

Die Komplexität von Algorithmen, die auf einem Graphen $G = (V, E)$ arbeiten, hängt sowohl von $|V|$ als auch $|E|$ ab. Wir haben daher zwei Parameter.

Innerhalb asymptotischer Notationen kürzen wir $|V|$ und $|E|$ als V und E ab.

Also ist $O(VE)$ die Abkürzung von $O(|V||E|)$.

In Algorithmen bezeichen wir für einen Graphen G die Menge der Knoten als $V[G]$ und die Menge der Kanten als $E[G]$.

Wir betrachten also die Kantenmenge und Knotenmenge eines Graphen als dessen Attribute.

Repräsentationen von Graphen

Zwei gebräuchliche Möglichkeiten:

1. Adjazenzlisten
2. Adjazenzmatrizen

Adjazenzlisten werden normalerweise bevorzugt, insbesondere bei spärlichen Graphen (solche, bei denen $|E|$ viel kleiner als $|V|^2$ ist).

Bei dichten Graphen bevorzugt man auch öfter Adjazenzmatrizen.