Kapitel IV

Allgemeine Entwurfs- und Optimierungsmethoden

Inhalt Kapitel IV

Zum Inhalt

- Greedy-Algorithmen: die Lösung wird Schritt für Schritt verbessert, zu jedem Zeitpunkt wird die lokal optimale Verbesserung durchgeführt. **16**
- Dynamische Programmierung: die optimale Lösung wird aus rekursiv gewonnenen optimalen Lösungen für Teilprobleme berechnet. Der Zeitaufwand bei der Rekursion wird dadurch verbessert, dass Ergebnisse für die Teillösungen tabelliert werden und ggf. aus der Tabelle genommen werden, statt neu berechnet zu werden. **15**

Zum Inhalt

- Amortisierte Analyse: Erlaubt es, den Aufwand für eine ganze Folge von Operationen besser abzuschätzen, als durch einfaches Aufsummieren. Sinnvoll, wenn es teure Operationen gibt, die sich durch eine "Verbesserung" der Datenstruktur "amortisieren". **17** Anwendung: *union find* Datenstruktur. **21**
- Backtracking: Man exploriert systematisch (Tiefensuche) alle Möglichkeiten, verwirft aber einen Teilbaum, sofern festgestellt werden kann, dass dieser keine (optimale) Lösung enthalten kann.

Greedy-Algorithmen

Entwurfsprinzip für Algorithmen, speziell für Optimierungsprobleme.

- Grundsätzlich anwendbar, wenn Lösung eines Problems sich als Folge von Einzelentscheidungen / partiellen Lösungen ergibt.
  Beispiel: Finden einer Fahrtroute: Einzelentscheidungen = jeweils nächstes Ziel.
- Lösung L wird als Folge von partiellen Lösungen $L_0, L_1, L_2, \ldots, L_n = L$ konstruiert. Die partielle Lösung L_{i+1} ergibt sich aus L_i, indem unter allen Ein-Schritt-Erweiterungen von L_i diejenige gewählt wird, die den Nutzen oder Gewinn maximiert. Als Anfang L_0 nimmt man meist die leere partielle Lösung.
- Ist die optimale Lösung nur auf dem Umweg über schlechte partielle Lösungen erreichbar, so verfehlt diese Heuristik das Ziel

Beispiel: ein Auswahlproblem

Gegeben: Menge $A = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ mit $s_i \leq e_i \in \mathbb{R}^+$ für alle i.

Ziel: Finde $B \subseteq \{1, \ldots, n\}$ maximaler Größe mit $s_i \geq e_j$ oder $s_j \geq e_i$
für alle $i \neq j \in B$.

Annahme: A nach den e_i sortiert: $e_1 \leq e_2 \leq \cdots \leq e_n$.

Beispiel: $\{(2,3),(1,4),(3,4),(2,5),(3,6),(1,6),(4,7),(5,8)\}$

Lösung mit Greedy-Algorithmus

Greedy Algorithmus

$B \leftarrow \emptyset$
$m \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  if $s_i \geq m$
    then $B \leftarrow B \cup \{i\}$
      $m \leftarrow e_i$
return B

Beispiel:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
e_i	4	5	6	7	8	9	10	11	12	13	14

Lösung: $\{(1,4),(5,7),(8,11),(12,14)\}$

Korrektheitsbeweis

Dass solch ein Greedy Algorithmus korrekt ist, kann man (wenn es überhaupt der Fall ist!) mit der folgenden Strategie zu beweisen versuchen.

- Versuche als Invariante zu zeigen: L_i lässt sich zu einer optimalen Lösung erweitern. Dies gilt sicher am Anfang wo noch keine Entscheidungen getroffen worden sind. Wichtig ist also, dies beim Übergang von L_i auf L_{i+1} zu erhalten. Dabei hilft das folgende Austauschprinzip.
- Sei L_i eine partielle Lösung und L eine sie erweiternde optimale Lösung. Sei L_{i+1} diejenige partielle Teillösung, die vom Greedy Algorithmus ausgewählt wird. Dann lässt sich L zu einer optimalen Lösung L' umbauen, die dann auch L_{i+1} erweitert.

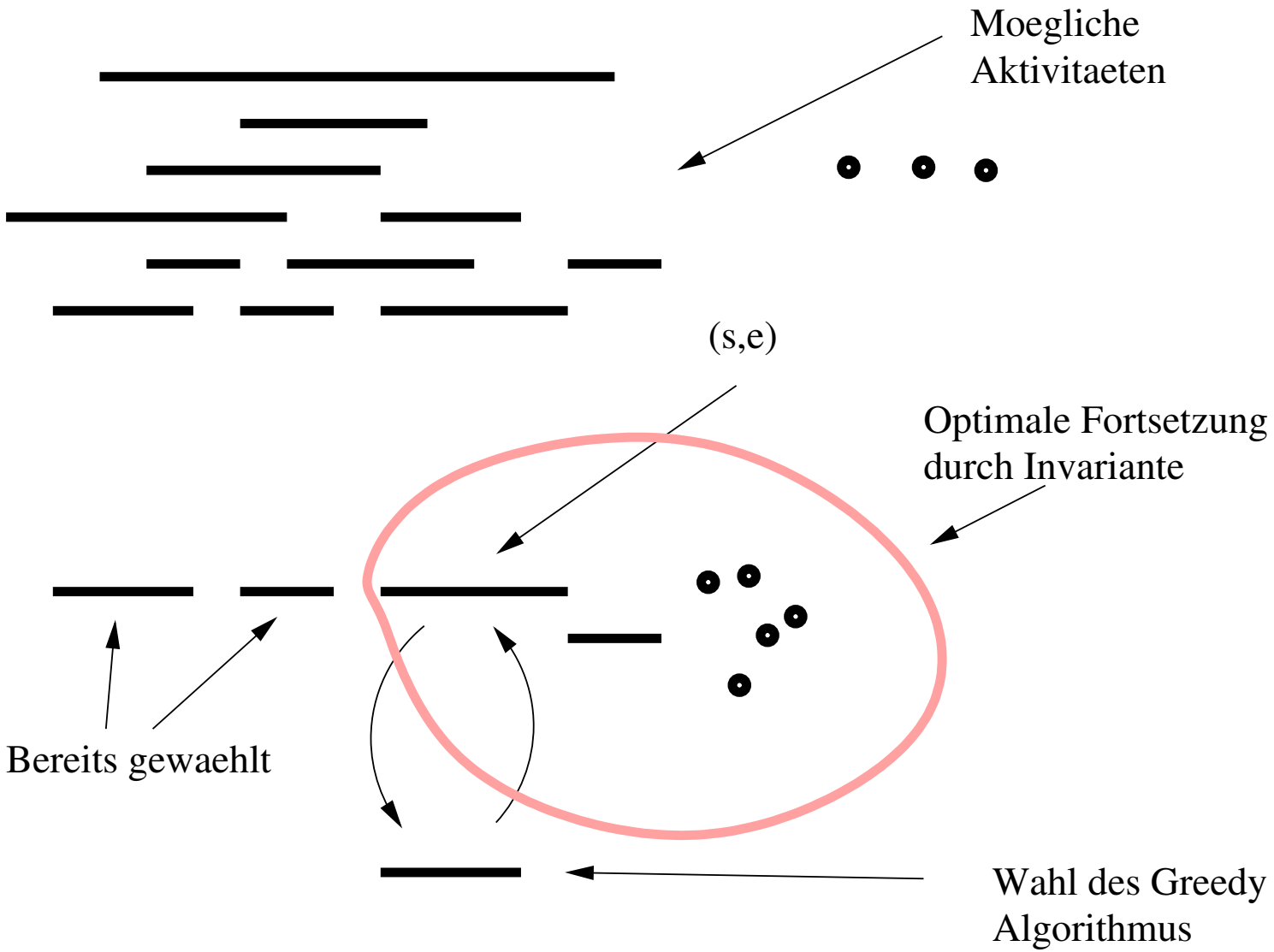
Anwendung im Beispiel

Satz

Der Greedy-Algorithmus findet im Beispiel eine optimale Lösung.

Beweis mit Invariante:
Die triviale Lösung $B = \emptyset$ lässt sich zu einer optimalen Lösung erweitern (nur kennen wir sie nicht...).
Sei B_i zu optimaler Lösung B erweiterbar. Nehmen wir an, diese optimale Lösung enthalte nicht das vom Greedy Algorithmus gewählte Intervall (s_j, e_j). Sei (s, e) das Intervall in $B \setminus B_i$ mit dem kleinsten Endzeitpunkt e. Der Greedy-Algorithmus wählt (s', e') wobei (s', e') nicht im Konflikt mit B_i steht und außerdem $e' \leq e$. Daraus folgt, dass $B \setminus \{(s, e)\} \cup \{(s', e')\}$ auch eine Lösung ist (keinen Konflikt enthält) und, da von derselben Kardinalität, wie B, auch eine optimale.

Illustration



Klassische Anwendung: Steinitzscher Austauschsatz

Satz

Sei $v_1, \ldots, v_k$ eine Menge linear unabhängiger Vektoren und B eine Basis, die $\{v_1, \ldots, v_k\}$ erweitert. Sei $\{v_1, \ldots, v_k, u\}$ linear unabhängig. Dann existiert ein Vektor $u' \in B \setminus \{v_1, \ldots, v_k\}$ sodass $B \setminus \{u'\} \cup \{u\}$ auch eine Basis ist.

Beweis: Sei $B = \{v_1, \ldots, v_k, w_1, \ldots, w_l\}$. Repräsentiere u in der Basis B als $u = \lambda_1 v_1 + \cdots + \lambda_k v_k + \mu_1 w_1 + \cdots + \mu_l w_l$. Die μ_i können nicht alle Null sein wegen der Annahme an u. O.B.d.A. sei $\mu_1 \neq 0$. Dann ist $\{v_1, \ldots, v_k, u, w_2, \ldots, w_l\}$ auch eine Basis. Daher findet folgender Greedyalgorithmus immer eine Basis: Beginne mit beliebigem Vektor $v_1 \neq 0$. Sind $v_1, \ldots, v_k$ schon gefunden, so wähle für v_{k+1} irgendeinen Vektor, der von $v_1, \ldots, v_k$ linear unabhängig ist. Gibt es keinen solchen mehr, so liegt Basis vor.

Huffman-Codes

Definition

Für Wörter $v, w \in \{0,1\}^*$ heißt v ein Präfix von w, falls $w = vu$ für ein $u \in \{0,1\}^*$.
Ein Präfixcode ist eine Menge von Wörtern $C \subseteq \{0,1\}^*$ mit:
für alle $v, w \in C$ ist v nicht Präfix von w.

Das Huffman-Code-Problem

Gegeben: Alphabet $A = \{a_1, \ldots, a_n\}$ mit Häufigkeiten $h(a_i) \in \mathbb{R}_+$

Ziel: Finde Präfixcode $C = \{c_1, \ldots, c_n\}$ derart dass $\sum_{i=1}^{n} h(a_i) \cdot |c_i|$ minimal wird.

Ein solches C heißt optimaler Präfixcode, oder Huffman-Code; diese werden bei der Datenkompression verwendet.

Beispiel

Symbol	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	5
Codierung 1 (C_1)	000	001	010	011	100	101
Codierung 2 (C_2)	0	101	100	111	1101	1100

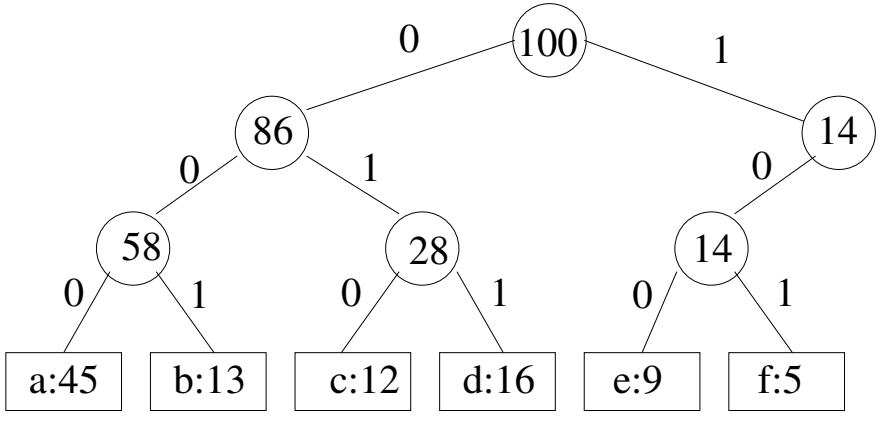
Sowohl C_1 als auch C_2 sind Präfixcodes. Man kann also Codierungen ohne Trenner hintereinanderschreiben und doch eindeutig decodieren. Z.B. bei C_2:

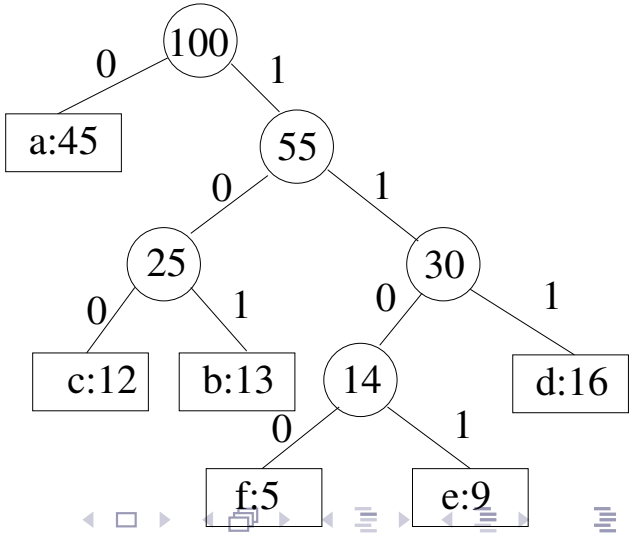
$100101010110101101010 1 \hat{=} 100\ 101\ 0\ 101\ 101\ 0\ 1101\ 0\ 101 \hat{=} cbabbaeab$

Code C_1 hat Kosten 300, während C_2 Kosten 224 hat.
Die mittlere Länge ist also 3, bzw. 2,24, da sich die Häufigkeiten zu 100 addieren.

Der Huffman-Algorithmus

- Erzeuge Knoten $z_1, \ldots, z_n$ mit Feldern $B[z_i] \leftarrow a_i$ und $H[z_i] \leftarrow h(a_i)$.
- Speichere diese in einer Priority Queue Q mit Operationen INSERT und EXTRACT-MIN (z.B. realisiert als Heap).
- Baue sukzessive einen binären Baum auf: Blätter sind die Knoten z_i, innere Knoten x haben nur ein Feld $H[x]$.
- Codewort c_i entspricht Pfad im Baum zu z_i, wobei $\mathit{left} \mathrel{\hat{=}} 0$ und $\mathit{right} \mathrel{\hat{=}} 1$.





Der Algorithmus

Aufbau des Baumes nach dem folgenden Algorithmus:

Huffman Algorithmus

for $i \leftarrow 1$ **to** $n-1$ **do**
  erzeuge neuen Knoten x
  $left[x] \leftarrow$ EXTRACT-MIN(Q)
  $right[x] \leftarrow$ EXTRACT-MIN(Q)
  $H[x] \leftarrow H[left[x]] + H[right[x]]$
  INSERT(Q, x)
return EXTRACT-MIN(Q)

Man fasst also immer die beiden Bäume mit der niedrigsten Wurzelhäufigkeit zu einem neuen Baum zusammen (dessen Wurzelbeschriftung) dann die Summe der beiden ist.

Laufzeit: $O(n) + 3(n-1) \cdot O(\log n) = O(n \log n)$.

Korrektheit des Huffman-Algorithmus

Wir identifizieren Huffman-Codes mit Code-Bäumen, wie sie der Algorithmus liefert. Also binäre Bäume, deren Knoten k mit Zahlen $H[k]$ beschriftet sind, so dass gilt:

- k innerer Knoten mit Kindern k_1 und k_2: $H[k] = H[k_1] + H[k_2]$
- k Blatt: $H[k] = h(B[k])$.

Die Kosten solch eines Baumes sind

$$K(T) := \sum_{x \in A} \text{„Tiefe von } x \text{ in } T\text{“} \cdot h(x) \quad = \quad \sum_{k \text{ innerer Knoten von } T} H[k]$$

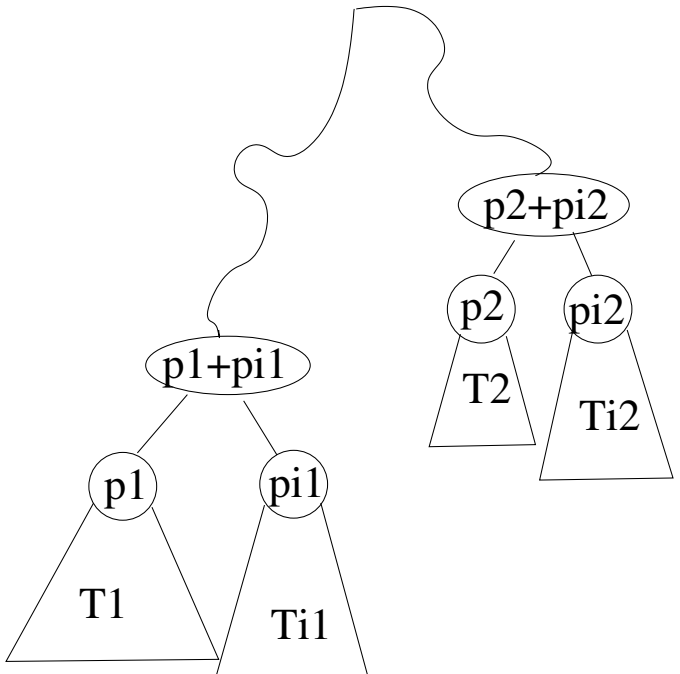
Dies deshalb, weil im rechtsstehenden Ausdruck jede relative Häufigkeit $h(x)$ gerade so oft summiert wird, wie der entsprechende Buchstabe tief steht.

Korrektheit II

Nehmen wir an, dass zu einem gewissen Zeitpunkt die Queue die Teilbäume $T_1, \ldots, T_m$ enthalte mit Wurzelbeschriftungen $p_1, \ldots, p_m$, wobei $p_1 \leq p_2 \leq p_j$ für $j > 2$. Es existiere (Invariante!) ein optimaler Codebaum T, der die T_i als Teilbäume besitzt.
Sind in T die Wurzeln von T_1 und T_2 Geschwister, so erweitert T auch die Queue im nächsten Schritt zu optimaler Lösung.

Korrektheit III

Anderenfalls seien k_1 und k_2 die Eltern von T_1, T_2 und T_{i_1}, T_{i_2} die entsprechenden Geschwister. Es gilt $H[k_1] = p_1 + p_{i_1}, H[k_2] = p_2 + p_{i_2}$. Sei d_1 die Tiefe von k_1 in T und d_2 die Tiefe von k_2 in T. O.B.d.A. sei $d_1 \geq d_2$. Sei T' der Baum, den man erhält, indem man T_{i_1} und T_2 vertauscht, sodass nunmehr k_1 die Bäume T_1, T_2 als Kinder hat und k_2 die Kinder T_{i_1} und T_{i_2} hat.



Es gilt $K[T] = d_1(p_1 + p_{i_1}) + d_2(p_2 + p_{i_2}) + c, K[T'] = d_1(p_1 + p_2) + d_2(p_{i_2} + p_{i_2}) + c$ für eine Konstante c. Es folgt $K[T'] \leq K[T]$, somit ist T' auch optimal und setzt die vom Huffman Algorithmus getroffene Entscheidung zu Optimallösung fort.

Dynamische Programmierung

Entwurfsprinzip, hauptsächlich für Algorithmen für Optimierungsprobleme:

- Zerlegung des Problems in kleinere, gleichartige Teilprobleme.
- Lösung ist zusammengesetzt aus den Lösungen der Teilprobleme.
- Aber: Teilprobleme sind nicht unabhängig, sondern haben ihrerseits gemeinsame Teilprobleme.

$\rightsquigarrow$ rekursives Divide-and-Conquer-Verfahren würde die gleichen Teilprobleme immer wieder lösen.
Lösung: Rekursion mit Memoisierung, oder Dynamische Programmierung: *Bottom-Up*-Berechnung der Lösungen größerer Probleme aus denen kleinerer, die dabei in einer Tabelle gespeichert werden.

Beispiel: Matrizen-Kettenmultiplikation

Problem:

n Matrizen $M_1, \ldots, M_n$ sollen multipliziert werden, wobei M_i eine $n_i \times n_{i+1}$-Matrix ist. Aufwand hängt von der Klammerung ab.

Bezeichne $M_{i..j}$ das Produkt $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$, und $m[i,j]$ die minimale Zahl von Multiplikationen zum Berechnen von $M_{i..j}$.

- Optimale Klammerung von $M_{1..n}$ ist von der Form $M_{1..k} \cdot M_{k+1..n}$, für optimale Klammerungen von $M_{1..k}$ und $M_{k+1..n}$.
- Deshalb gilt:

$$m[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1} & \text{sonst.} \end{cases} \quad (*)$$

Lösung mit Dynamischer Programmierung

- Rekursive Auswertung von (*) wäre enorm ineffizient.
- Beobachte, dass nur $O(n^2)$ Aufrufe der Form $m[i, j]$ möglich sind.
- ENTWEDER (Memoisierung): Merke bei der Rekursion Ergebnisse bereits berechneter $m[i, j]$. Löse keine rekursiven Aufrufe aus, falls das Ergebnis schon vorliegt.
- ODER (dynamische Programmierung): Fülle eine Tabelle, in der unten die $m[i, i] = 0$ stehen, und der h-ten Ebene die Werte $m[i, j]$ mit $j - i = h$. An der Spitze steht schließlich $m[1, n]$.
- Speichere in jedem Schritt auch einen Wert $s[i, j]$, nämlich dasjenige k, für das in $(*)$ das Minimum angenommen wird.

Beispiel

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

m	1	2	3	4	5	
6	15125	10500	5375	3500	5000	0
5	11875	7125	2500	1000	0	
4	9375	4375	750	0		
3	7875	2625	0			
2	15750	0				
1	0					

Es gilt z.B.:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

Die s-Tabelle ist nicht dargestellt.

Fibonaccizahlen

$$F(0) = 1$$
$$F(1) = 1$$
$$F(h+2) = F(h+1) + F(h)$$

Naive rekursive Auswertung von $F(h)$ macht Aufwand $\Omega(\phi^h)$, wobei $\phi = 1,618\ldots$.
Verwendet man eine Tabelle der Größe $h+1$ zur Speicherung der Werte $F(0)\ldots F(h)$, so wird der Aufwand linear.
Die Tabelle kann man entweder zur Verbesserung der Rekursion verwenden (Memoisierung) oder iterativ auffüllen (Dynamische Programmierung).

h	0	1	2	3	4	5	6	7	8
$F(h)$	1	1	2	3	5	8			

Längste gemeinsame Teilfolge (lgT)

Definition

$Z = \langle z_1, z_2, \ldots, z_m \rangle$ ist Teilfolge von $X = \langle x_1, x_2, \ldots, x_n \rangle$, falls es Indizes $i_1 < i_2 < \ldots < i_m$ gibt mit $z_j = x_{i_j}$ für $j \leq m$.

Problem

Gegeben $X = \langle x_1, x_2, \ldots, x_m \rangle$ und $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Finde $Z = \langle z_1, z_2, \ldots, z_k \rangle$ maximaler Länge k, das Teilfolge von X und Y ist.

Rekurrenz

Sei $c[i,j]$ die Länge einer lgT von X_i und Y_j. Es gilt:

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

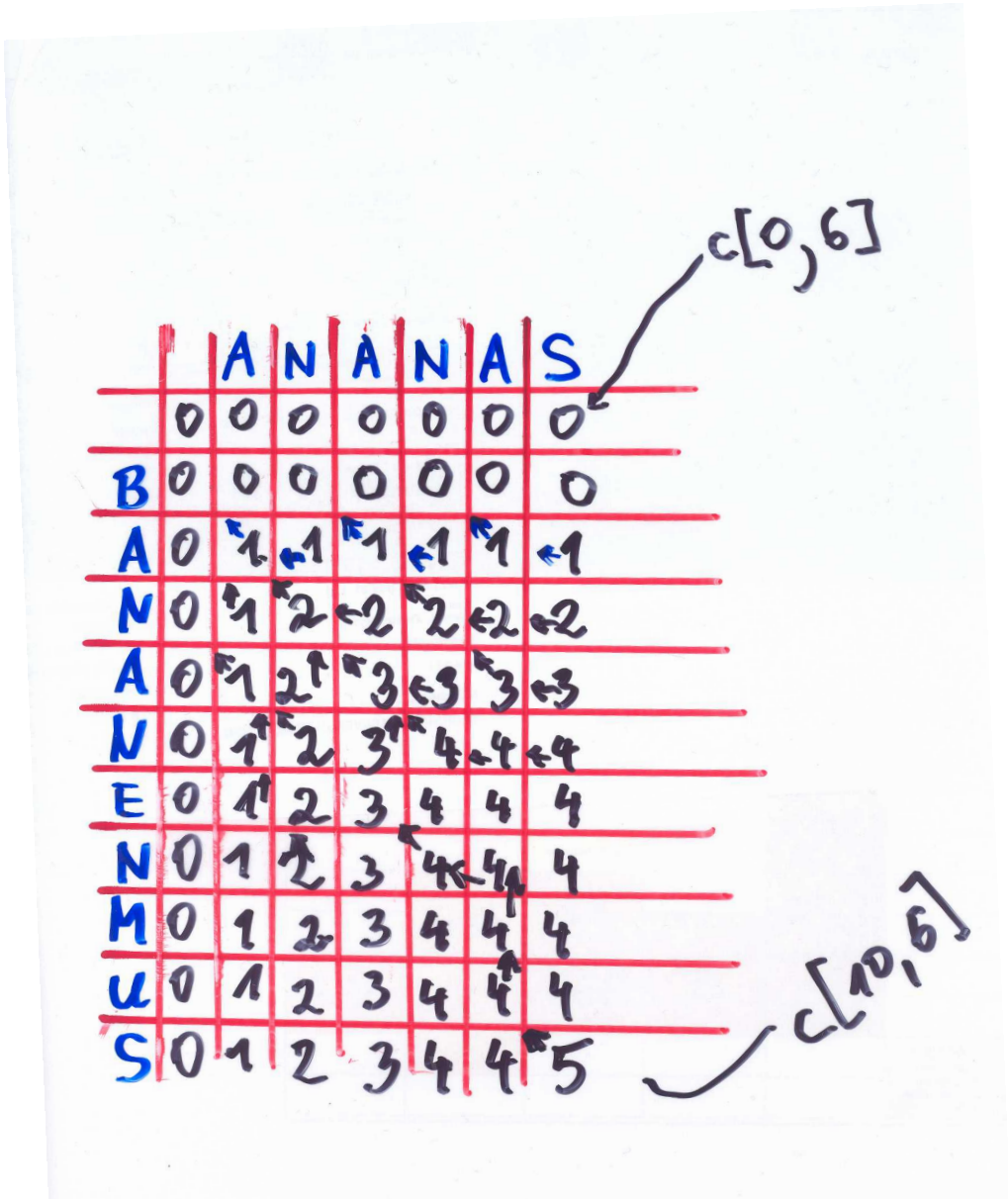
Berechnung mit dynamischer Programmierung

- Fülle die Tabelle der $c[i,j]$ zeilenweise, jede Zeile von links nach rechts.
  Am Ende erhält man $c[m,n]$, die Länge einer lgT von X und Y.
- Für $i,j \geq 1$, speichere einen Wert $b[i,j] \in \{\uparrow, \nwarrow, \leftarrow\}$, der anzeigt,
  wie $c[i,j]$ berechnet wurde:

$$\begin{aligned} \uparrow &: c[i,j] = c[i-1,j] \\ \nwarrow &: c[i,j] = c[i-1,j-1]+1 \\ \leftarrow &: c[i,j] = c[i,j-1] \end{aligned}$$

Damit läßt sich eine lgT aus der Tabelle ablesen.

Beispiel


c[0,6]
A N A N A S
0 0 0 0 0 0 0
B 0 0 0 0 0 0 0
A 0 1 1 1 1 1 1
N 0 1 2 2 2 2 2
A 0 1 2 3 3 3 3
N 0 1 2 3 4 4 4
E 0 1 2 3 4 4 4
N 0 1 2 3 4 4 4
M 0 1 2 3 4 4 4
U 0 1 2 3 4 4 4
S 0 1 2 3 4 4 5
c[10,6]

Zusammenfassung Dynamische Programmierung

- Ausgangspunkt ist immer eine rekursive Lösung des gegebenen Problems.
- Dynamische Programmierung bietet sich an, wenn die Zahl der rekursiven Aufrufe bei naiver Implementierung größer ist, als die Zahl der überhaupt möglichen voneinander verschiedenen Aufrufe der rekurive definierten Funktion.
- Man studiert dann die Reihenfolge, in der die Ergebnisse der Aufrufe benötigt werden und berechnet diese iterativ von unten her.

Amortisierungs-Analyse

Prinzip zur Analyse von Operationen auf Datenstrukturen.

- Man interessiert sich für die Gesamtkosten einer Folge von Operationen auf einer Datenstruktur jeweils beginnend mit der initialen Struktur (leerer Baum/Keller/Menge, etc.)
- Diese Gesamtkosten können niedriger sein, als die Summe der worst-case Kosten der Einzeloperationen, da die Konstellationen, die zum worst-case führen, jeweils andere sind.
- Um diese Gesamtkosten bequem berechnen zu können, weist man den Einzeloperationen zu einem gewissen Grad willkürliche amortisierte Kosten zu, die so zu wählen sind, dass die Summe der amortisierten Kosten einer Folge von Operationen (beginnend bei der leeren Datenstruktur) eine obere Schranke an die tatsächlichen Kosten der Folge bildet.
- teure Operationen können sich durch nachfolgende oder voraufgegangene billigere amortisieren.

Drei Methoden

- Drei Methoden, um amortisierte Kosten festzulegen:
  1. Aggregat-Methode:
     Berechne worst-case Kosten $T(n)$ einer Folge von n Operationen direkt. Amortisierte Kosten: $T(n)/n$.
  2. Konto-Methode:
     Elemente der Datenstruktur haben ein Konto, auf dem zuviel berechnete Kosten gutgeschrieben werden. Zu niedrig veranschlagte Operationen werden aus dem Konto bezahlt. Amortisierte Kosten: Tatsächliche Kosten - Kontosaldo
  3. Potenzial-Methode:
     Verallgemeinert die Konto-Methode: Potenzial ("Potenzielle Energie") wird nach willkrlicher Definition der gesamten Datenstruktur zugeordnet. Kosten können aus der Potenzialdifferenz (Nachher - Vorher) bezahlt werden.

Einführendes Beispiel: Binärzähler

Datenstruktur: Array A von $length[A] = k$ Bits.

Increment

Increment(A)

```
1  i ← 0
2  while  i < length[A]  und  A[i] = 1
3        do  A[i] ← 0
4            i ← i + 1
5  if  i < length[A]
6     then A[i] ← 1
```

Worst-case-Kosten eines Increment: k Wertzuweisungen.

Aggregat-Methode

Betrachte Folge von n INCREMENT-Operationen (vereinfachende Annahme: $k \geq \lceil \log_2(n+1) \rceil$, also kein Überlauf).
Berechne Kosten $T(n)$ der ganzen Folge von n INCREMENT-Operationen:

- $A[0]$ *wird bei jedem* INCREMENT *verändert,*
  $A[1]$ *nur bei jedem zweiten,*
  $A[2]$ *nur bei jedem vierten,*
- Gesamtzahl der Wertzuweisungen bei n INCREMENT:

$$T(n) \leq \sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n$$

- Also: $T(n) = O(n)$, somit amortisierte Kosten von INCREMENT:

$$O(n)/n = O(1)$$

Konto-Methode

- Kosten einer Wertzuweisung seien 1€.
- Jede Array-Position $j \leq k$ hat ein Konto.
- Setze amortisierte Kosten eines INCREMENT zu 2€fest.
- Jedes INCREMENT setzt nur ein Bit $A[i] \leftarrow 1$ (Zeile 6):
  Zahle dafür 1€, und schreibe 1€ auf dessen Konto gut.

  $\rightsquigarrow$ Jedes j mit $A[j] = 1$ hat ein Guthaben von 1€.
  Zahle damit die Wertzuweisungen $A[i] \leftarrow 0$ in Zeile 3.

Potenzial-Methode

Sei D_i die Datenstruktur nach der i-ten Operation.

- Definiere eine Potenzialfunktion Φ, die jedem D_i ein $\Phi(D_i) \in \mathbb{R}$ zuordnet.
- Amortisierte Kosten $\hat{c}_i$ definiert durch $\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$.
- Gesamte amortisierte Kosten:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Ist $\Phi(D_n) \geq \Phi(D_0)$, so sind tatsächliche Gesamtkosten höchstens die amortisierten Gesamtkosten.
- Meist ist n variabel, und man hat $\Phi(D_0) = 0$, und $\Phi(D_i) \geq 0$ für $i > 0$.
- Manchmal (vgl. Kontomethode) hängt das Potenzial auch noch vom Zeitpunkt ab: Φ_i statt nur Φ.

Im Beispiel:

- Definiere $\Phi(D_i) = b_i :=$ Anzahl der 1 im Zähler nach i INCREMENT-Operationen .
- Damit ist $\Phi(D_0) = 0$, und $\Phi(D_i) > 0$ für alle $i > 0$.
- Sei t_i die Zahl der Durchläufe der **while** -Schleife beim i-ten INCREMENT.
- Damit ist $c_i = t_i + 1$: Es werden t_i bits auf 0 gesetzt, und eines auf 1.
- Außerdem gilt: $b_i = b_{i-1} - t_i + 1$.
- Daher ist die Potenzialdifferenz:

  $$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i ,$$

  also sind die amortisierten Kosten:

  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 + (1 - t_i) = 2$$

Queue als zwei stacks

Eine Queue Q kann mittels zweier Keller S_{in} und S_{out} realisiert werden:

PUT, GET

PUT(Q, x)

    PUSH(S_{in}, x)

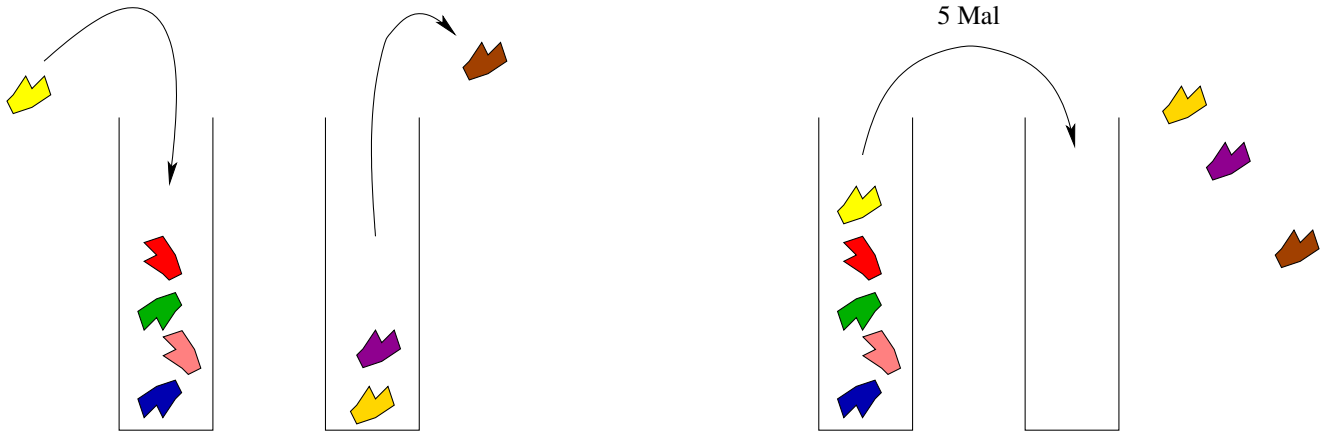
GET(Q)

    if EMPTY(S_{out})
      then while nicht EMPTY(S_{in}) **do**
                PUSH$(S_{\text{out}}$, POP$(S_{\text{in}}))$
    return POP(S_{out})

Aufwand von PUT und GET

Wieviele Kelleroperationen benötigen die Queue-Operationen im worst-case?

PUT(Q, x): 1
GET(Q): $2n + 1$



Amortisierungsanalyse der Queue

Definiere Potenzial $\Phi(Q) := 2 \cdot |S_{\text{in}}|$.
Sei Q' die Queue nach Anwendung der Operation auf Q.

$\textsc{Put}(Q, x)$: es ist $\Phi(Q') = \Phi(Q) + 2$.

$$c_{\textsc{Put}} = 1 \qquad \hat{c}_{\textsc{Put}} = 1 + \Phi(Q') - \Phi(Q) = 1 + 2 = 3$$

$\textsc{Get}(Q)$, Fall 1: S_{out} nicht leer. Dann ist $\Phi(Q') = \Phi(Q)$.

$$c_{\textsc{Get}} = 1 \qquad \hat{c}_{\textsc{Get}} = 1 + \Phi(Q') - \Phi(Q) = 1$$

$\textsc{Get}(Q)$, Fall 2: S_{out} leer. Dann ist $\Phi(Q') = 0$ und $\Phi(Q) = 2 \cdot |S_{\text{in}}| = 2n$.

$$c_{\textsc{Get}} = 2n + 1 \qquad \hat{c}_{\textsc{Get}} = 2n + 1 + \Phi(Q') - \Phi(Q) = 2n + 1 + 0 - 2n = 1$$

Union-Find: Datenstruktur für disjunkte Mengen

Es soll eine Familie von disjunkten dynamischen Mengen

$$M_1, \ldots, M_k \quad \text{mit} \quad M_i \cap M_j = \emptyset \text{ für } i \neq j$$

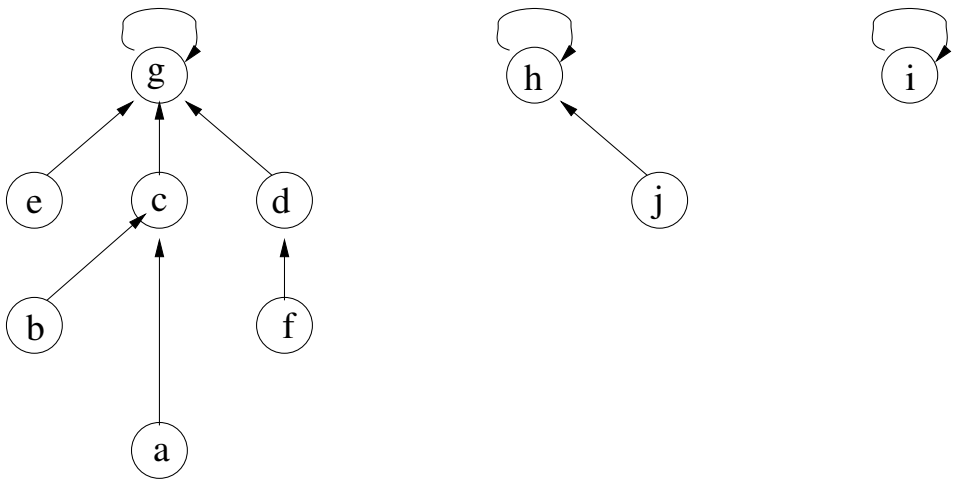
verwaltet werden.
Dabei sollen folgende Operationen zur Verfügung stehen:

- $\textsc{Make-Set}(x)$ fügt die Menge $\{x\}$ zur Familie hinzu. Es ist Aufgabe des Benutzers, sicherzustellen, dass x vorher in keiner der M_i vorkommt.
- $\textsc{Find}(x)$ liefert ein kanonisches Element der Menge M_i mit $x \in M_i$, also $\textsc{Find}(x) = \textsc{Find}(y)$, falls x und y in der gleichen Menge M_i liegen.
- $\textsc{Union}(x, y)$ ersetzt die Mengen M_i und M_j mit $x \in M_i$, $y \in M_j$ durch ihre Vereinigung $M_i \cup M_j$.

Typische Anwendungen: Äquivalenzklassen einer Äquivalenzrelation, Zusammenhangskomponenten eines Graphen , ...

Union-Find: Realisierung als Wälder

Jede Menge wird durch einen Baum dargestellt. Knoten x haben Zeiger $p[x]$ auf ihren Vater. Kanonische Elemente sind an der Wurzel und haben $p[x] = x$.



Die disjunkten Mengen $\{a, b, c, d, e, f, g\}, \{h, j\}, \{i\}$ als union-find Wald.

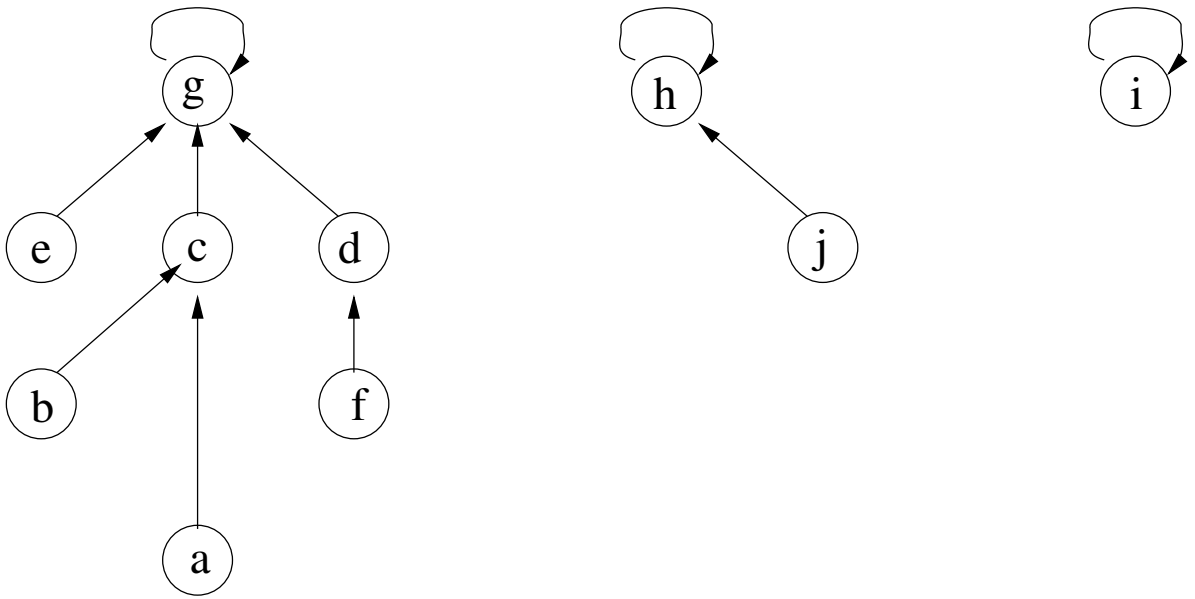
Hier gilt z.B.: $\textsc{Find}(d) = \textsc{Find}(f) = \textsc{Find}(g) = g$ und $\textsc{Find}(j) = h$.

NB; Argument x von $\textsc{Make-Set}$(x) muss ein $p[]$-Feld haben. (bzw. entsprechende getter und setter).

Entstehung des Beispiels

Die im Beispiel gezeigte Situation entsteht z.B. durch die Befehle:

MAKE-SET(x) für $x = a, b, c, \ldots, i, j$
UNION(h, j)
UNION(c, b)
UNION(c, a)
UNION(g, e)
UNION(d, f)
UNION(e, a)
UNION(e, f)



NB: UNION(x, y) hängt Wurzel von y unterhalb Wurzel von x.

Pseudocode der Operationen

Find, Make-Set, Union

```
FIND(x)
  while x ≠ p[x] do
    x ← p[x]
  return x

MAKE-SET(x)
  p[x] ← x

LINK(x, y) ▷ x, y müssen Wurzeln sein
  p[y] ← x

UNION(x, y)
  LINK(FIND(x), FIND(y))
```

Optimierung Union by Rank

Jedes Element hat Rang-Feld *rank*[], welches die Höhe des entsprechenden Baumes angibt.

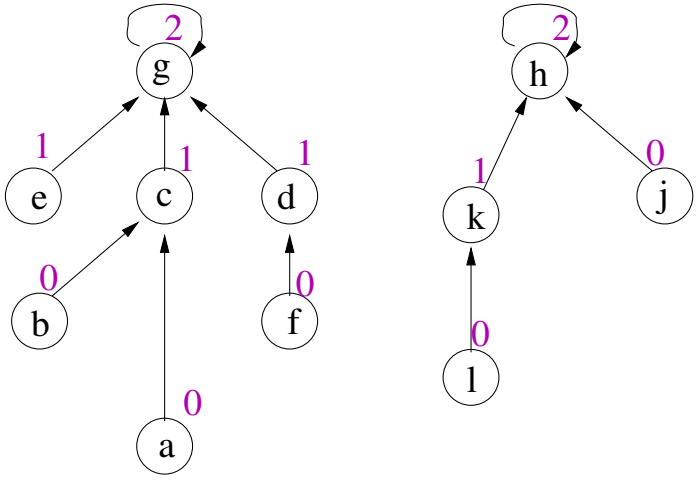
MAKE-SET und LINK mit Rang

MAKE-SET(x)
  $p[x] \leftarrow x$; $rank[x] = 0$

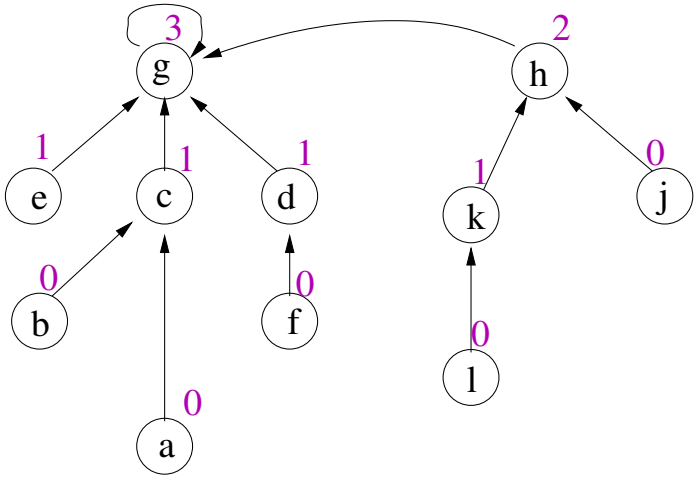
LINK(x, y)
  if $rank[x] > rank[y]$
    then $p[y] \leftarrow x$
  else if $rank[x] = rank[y]$
    $p[y] \leftarrow x$; $rank[x] \leftarrow rank[x] + 1$
  else $p[x] \leftarrow y \triangleright$ $rank[x] < rank[y]$

FIND und UNION bleiben unverändert.

Beispiel



$$\text{—}\textsc{Union}(g, h) \longrightarrow$$



Analyse von *union-by-rank*

Sei $size(x)$ die Größe des Teilbaumes unter x und $height(x)$ seine Höhe (längster Pfad).

Lemma

Für jede Wurzel x gilt:

- $size(x) \geq 2^{rank[x]}$.
- $rank[x] \geq height(x)$

Analyse von *union-by-rank*

Satz

Jede Folge von m MAKE-SET-, UNION- und FIND-Operationen, von denen n MAKE-SET sind, hat für Wälder mit Rangfunktion eine Laufzeit von $O(m \log n)$.

Die einzelnen Operationen haben also amortisierte Laufzeit $O(\log n)$ (Aggregatmethode).

Weitere Laufzeitverbesserung durch Pfadkompression

Pfadkompression

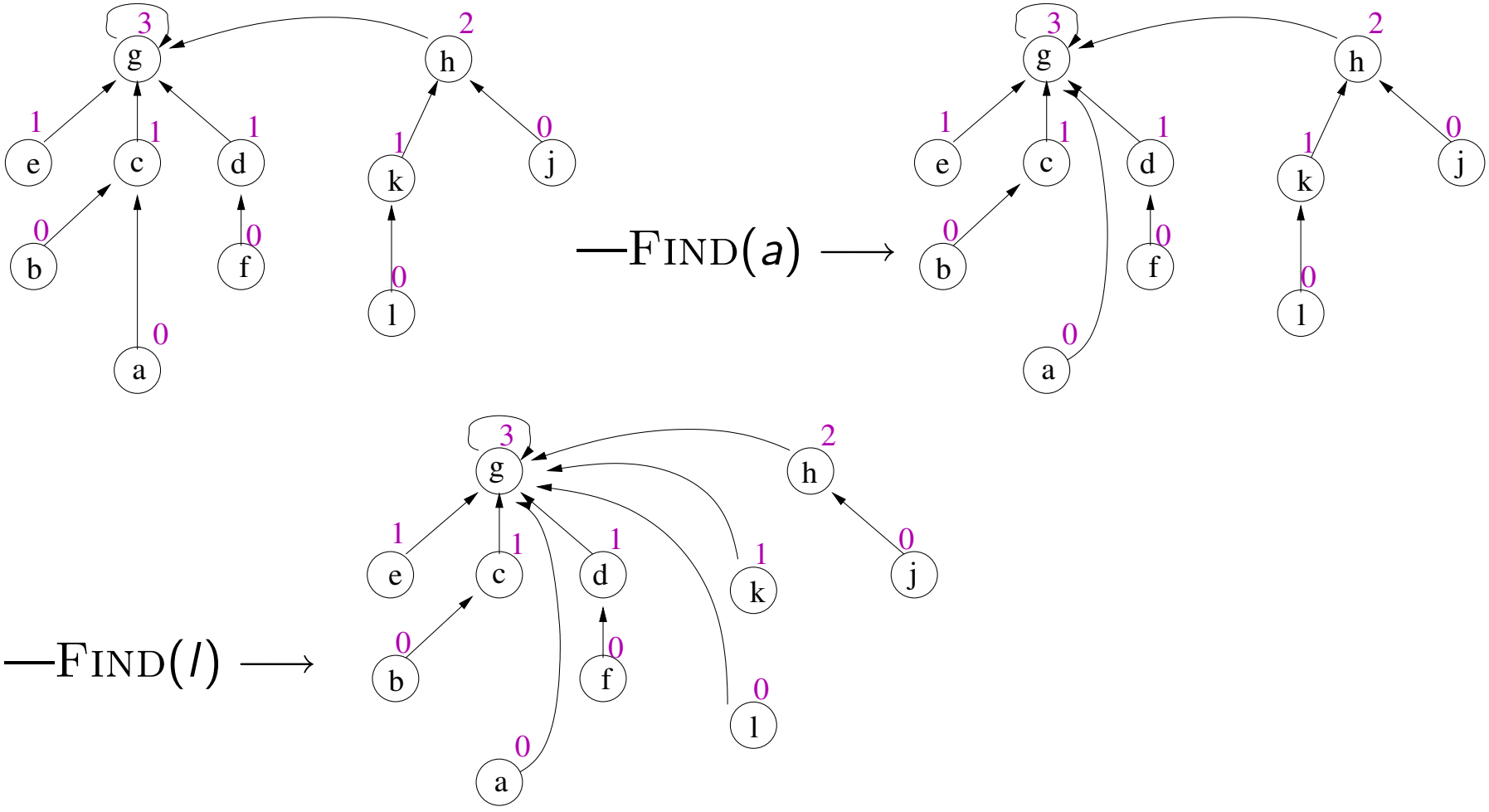
Man hängt jeden im Rahmen von FIND besuchten Knoten direkt unter die entsprechende Wurzel. Spätere Suchvorgänge werden dadurch effizienter.

FIND

```
FIND(x)
if x = p[x] then return x
   else
      root ← FIND(p[x])
      p[x] ← root; return root
```

NB: Bei Pfadkompression muss weiterhin zusätzlich *union-by-rank* in Kraft sein.

Beispiel



NB: Eine Wurzel mit Rank i (z.B. hier $i = 3$) muss nicht notwendigerweise Höhe i (hier 3) haben.

Amortisierte Analyse von Union-Find (mit *union-by-rank*)

Definition

Der iterierte Logarithmus $\log^* n$ ist gegeben durch:

$$\log^* n := \min\{i \mid \underbrace{\log \cdots \log}_{i \text{ Stück}} n \leq 1\}$$

Satz

Jede Folge von m MAKE-SET-, UNION- und FIND-Operationen, von denen n MAKE-SET sind, hat für Wälder mit Rangfunktion und Pfadkompression eine Laufzeit von $O(m \log^* n)$.

Die amortisierte Komplexität der Operationen ist also $O(\log^* n)$.

Bemerkung: Für $n \leq 10^{19.728}$ ist $\log^* n \leq 5$.

Beweis des Satzes

Sei eine Folge von m Operationen gegeben, unter denen n MAKE-SET sind.
Dem Wald, der unmittelbar nach der q-ten Operation vorliegt, ordnen wir ein Potential $\Phi_q := \sum_x \phi_q(x)$ zu, wobei die Summe über die Knoten des Waldes rangiert und die Knotenpotentiale $\phi_q(x)$ wie folgt gegeben sind.

- Ist $\mathit{rank}[x] = 0$, so $\phi_q(x) = 0$,
- Ist x Wurzel, so $\phi_q(x) = (\log^*(n) + 2) \cdot \mathit{rank}[x]$,

In allen anderen Fällen richtet sich das Potential eines Knotens nach dem Abstand seines Ranges zu dem des Elternknotens. Man beachte, dass dieser Abstand stets ≥ 1 ist und dass im Falle, den wir betrachten, auch $\mathit{rank}[x] \geq 1$.

Wir teilen die Knoten in drei Stufen ein und definieren zu jedem Knoten ein Maß, welches sich nach dem Abstand und nach der Stufe richtet.

Stufe 0 Diese liegt vor, wenn $rank[x] + 1 \leq rank[p[x]] < 2 \cdot rank[x]$. Man setzt dann $maß(x) = rank[p[x]] - rank[x]$.

Stufe 1 Diese liegt vor, wenn $2 \cdot rank[x] \leq rank[p[x]] < 2^{rank[x]}$. Man setzt dann $maß(x) = \max\{k \mid 2^k \cdot rank[x] \leq rank[p[x]]\}$.

Stufe 2 Diese liegt vor, wenn $2^{rank[x]} \leq rank[p[x]]$. Man setzt dann $maß(x) = \max\{k \mid 2_k^{rank[x]} \leq rank[p[x]]\}$, wobei $2_0^j = j, 2_{i+1}^j = 2^{2_i^j}$.

In den Stufen 0 und 1 ist $1 \leq maß(x) < rank[x]$. In Stufe 2 hat man $1 \leq maß(x) \leq \log^*(n)$.

Beweis, Forts.

Nunmehr definiert man das Potential des Knotens x zu

$$\phi_q(x) = (2 + \log^*(n) - S) \cdot \mathit{rank}[x] - \mathit{maß}[x]$$

wobei S die Stufe von x ist. Das Potential ist also nichtnegativ. Im Laufe der Zeit vergrößert sich der Abstand zum Elternknoten. Verändert sich dabei das Maß (bei Stufe 0 stets, bei Stufe 1,2 nicht immer), so nimmt das Potential echt ab, auch wenn das Maß abnimmt, weil ja dann die Stufe steigt.

Wir zeigen jetzt, dass alle Operationen amortisierte Kosten $O(\log^* n)$ haben.

Amortisierte Kosten von Make-Set und *Link*

Make-Set: keine Potentialänderung, tatsächliche Kosten $O(1)$.

Link: Tatsächliche Kosten $O(1)$; das Gesamtpotential kann aber um maximal $2 + \log^*(n)$ ansteigen, nämlich dann, wenn sich der Rang einer Wurzel erhöht.

Amortisierte Kosten von FIND

Es sei s die Länge des Pfades ("Bestimmungspfad") von der Wurzel zum gesuchten Knoten k.

Die tatsächlichen Kosten der Operation FIND(k) sind dann $O(s)$.

Durch die Pfadkompression tritt für mindestens $s-5$ Knoten eine Potentialverringerung ein. Sei nämlich x ein innerer Knoten auf dem Bestimmungspfad, auf den weiter oben ein Knoten y der gleichen Stufe folgt und sei $f(n) = n+1$, falls diese Stufe 0 ist, $f(n) = 2n$, falls diese Stufe 1 ist, $f(n) = 2^n$, falls diese Stufe 2 ist. Es gilt dann

$$\begin{aligned} rank[p[x]] &\geq f^{(maß(x))}(rank[x]) \\ rank[p[y]] &\geq f(rank[y]) \\ rank[y] &\geq rank[x] \end{aligned}$$

Also $rank[p[y]] \geq f^{(maß(x)+1)}(rank[x])$. Nach der Pfadkompression nimmt also entweder das Maß von x oder die Stufe um mindestens 1 zu.

Bestmögliche Schranke

Durch Einbeziehung höherer Stufen mit entsprechender Definition erhält man eine Schranke von $O(m \cdot \alpha(n))$ an die Komplexität einer Folge von m Operationen auf n Daten. Hierbei ist $\alpha(n)$ die Inverse der Ackermannfunktion, also das kleinste k sodass $A(k, 1) > n$. Tarjan hat gezeigt, dass diese Schranke für eine große Klasse von Algorithmen bestmöglich ist. Siehe Buch.

Backtracking

Backtracking ist eine Lösungsstrategie für Suchprobleme des folgenden Formats:

- Mögliche Lösungen erscheinen als Blätter eines (immensen) Baumes.
- Teillösungen entsprechen den inneren Knoten.

Der Baum wird gemäß der Tiefensuche durchgearbeitet.
Sieht man einer Teillösung bereits an, dass sie nicht zu einer Lösung vervollständigt werden kann, so wird der entsprechende Teilbaum nicht weiter verfolgt.

Damenproblem

Man soll n Damen auf einem $n \times n$ Schachbrett so platzieren, dass keine Dame eine andere schlägt.
Wir platzieren die Damen zeilenweise von unten her und repräsentieren Teillösungen als Listen.

Beispiel

`[0;1;6]` bedeutet:
Die unterste Zeile enthält eine Dame auf Position 6 (=siebtes Feld von links).
Die zweite Zeile von unten enthält eine Dame auf Position 1.
Die dritte Zeile von unten enthält eine Dame auf Position 0.

Solch eine Teillösung heißt konsistent, wenn keine Dame eine andere schlägt.
Die Prozedur KONSISTENT(i, l) liefere **true**, falls $i :: l$ konsistent ist unter der Voraussetzung, dass l selbst konsistent ist, falls also die neue Dame keine der alten schlägt.

Lösung mit Backtracking

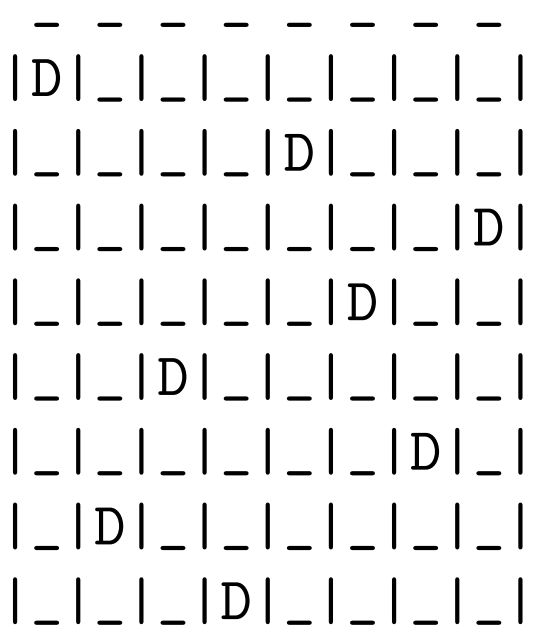
Die folgende Prozedur vervollständigt eine konsistente Teillösung l zu einer Lösung der Größe n, falls eine solche existiert.
Wenn es keine solche Vervollständigung gibt, wird NIL zurückgeliefert.

Finde-Damen

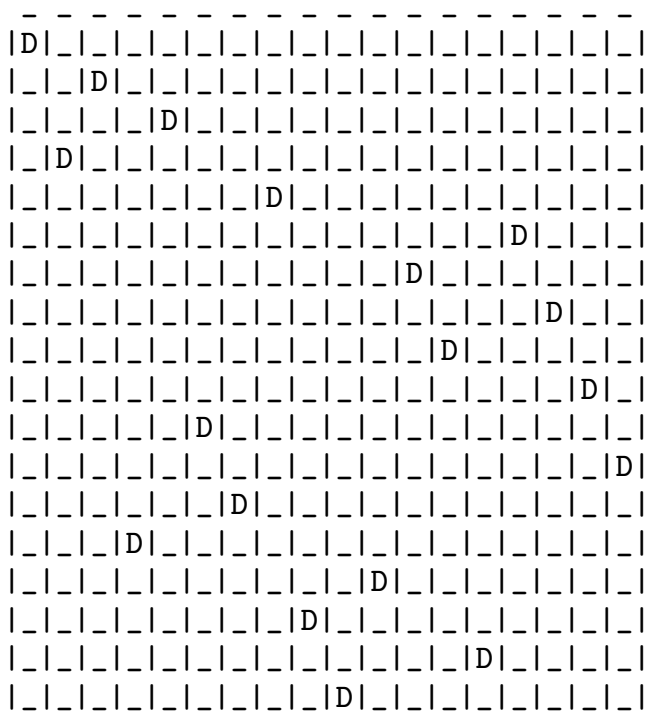
```
Finde-Damen(l, n)
if |l| = n then return l else
  for i = 0 ... n − 1 do
    if Konsistent(i, l) then
    result ← Finde-Damen(i :: l, n)
    if result ≠ NIL then return result
  return NIL
```

Eine Lösung für $n = 8$: `[3; 1; 6; 2; 5; 7; 4; 0]`

Beispiellösungen für $n = 8$ und $n = 18$



[3; 1; 6; 2; 5; 7; 4; 0]



[9; 13; 8; 10; 3; 6; 17; 5;
15; 11; 14; 7; 1; 4; 2; 0]

Iterative Deepening

Oft ist die Problemstellung so, dass eine Lösung mit steigender Tiefe im Lösungsbaum immer unwahrscheinlicher oder weniger brauchbar wird.
Im Prinzip würde es sich dann anbieten, die Tiefensuche durch eine Breitensuche zu ersetzen, was allerdings meist zu aufwendig ist (Speicherplatzbedarf steigt exponentiell mit der Tiefe).
Man kann dann der Reihe nach Tiefensuche bis maximale Tiefe 1, dann bis zur maximalen Tiefe 2, dann 3, dann 4, etc. durchführen.
Zwar macht man dann beim nächsten Tiefenschritt die gesamte bisher geleistete Arbeit nochmals; dennoch erzielt man so eine höhere Effizienz als bei der Breitensuche. Diese Technik heißt *iterative deepening*.

Branch and bound

Oft sind die Lösungen eines Baum-Suchproblems angeordnet gemäß ihrer Güte und man sucht entweder die beste Lösung, oder aber eine, deren Güte eine bestimmte Schranke überschreitet.
Manchmal kann man die Güte von Lösungen schon anhand einer Teillösung abschätzen. Zeigt sich aufgrund dieser Abschätzung, dass keine Lösung in einem Teilbaum besser ist, als eine bereits gefundene, so braucht man diesen Teilbaum nicht weiter zu verfolgen.
Diese Verbesserung von Backtracking bezeichnet man als *Branch-and-Bound*.

Abstraktes Beispiel

Sei $f : \{0,1\}^* \to \mathbb{R}_0^+$ eine Funktion mit der Eigenschaft, dass $|l| \geq |l'| \Rightarrow f(l) \leq f(l')$. Man bestimme $\max\{f(l) \mid |l| = n\}$.

Folgendes Programm kann das tun:

B&B-Optimum

```
B&B-Optimum(l, n, b)
  if f(l) ≤ b return b else
    if |l| = n then return f(l) else
      opt_L ← B&B-Optimum(0 :: l, n, b)
      opt_R ← B&B-Optimum(1 :: l, n, opt_L)
      return opt_R
```

Aufrufsequenz: B&B-Optimum$([], n, 0)$.

In der Praxis verwendet man B&B gern bei der linearen Optimierung mit Ganzzahligkeitsbedingungen (ILP).