Dynamische Programmierung II

Malte Matthias

Inhalt

1 Definition: Dynamische Programmierung

- Begriff wurde in den 1950ern vom Richard Bellman eingeführt
- Designtechnik für Algorithmen
- Problem wird in viele gleichartige Teilprobleme zerlegt
- Jedes Teilproblem wird nur einmal gelöst, Ergebnis wird in einer Tabelle gespeichert
- größere Teilprobleme werden mit Lösungen der kleineren Subprobleme gelöst

  -> Rekursive Algorithmen
- Unterschied zu divide&conquer:

  - Subprobleme überlappen sich

  - Dynamische Programmierung arbeitet bottom-up

  - divide&conquer benutzt meistens mehrere Algorithmen (für eigentliches Problem + Zusammenfügen)

2 Optimale binäre Suchbäume (BST)

Motivation

- BST besitzen bei der Suche effiziente Laufzeit O(log n)
- Laufzeitverhalten lässt sich durch geschickte Anordnung der Elemente weiter verbessern

Voraussetzung:

- BST wird nur noch selten verändert (löschen/einfügen)
- Für jede mögliche Suchanfrage ist die Wahrscheinlichkeit bekannt

2 Optimale binäre Suchbäume

- Mittlere Anzahl der Vergleiche bei einer Suche:
- $C = \sum_{i=1}^{n} \left(p_i \cdot \left(Tiefe(i) + 1 \right) \right) + \sum_{i=0}^{n} \left(q_i \cdot Tiefe(i) \right)$
- p_i = Suche nach dem i-ten Element im Baum
- q_i = Suche nach einem Element, das nicht im Baum vorkommt und zwischen a_i und a_{i+1} liegt

Ziel : Die Kostenfunktion soll minimal sein!

Beispiel:

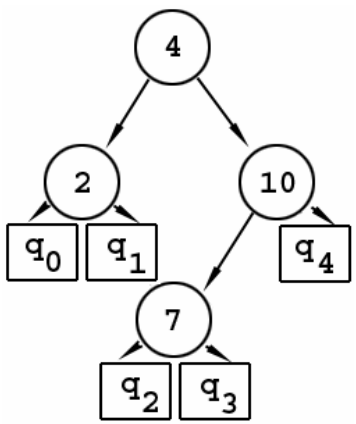
Natürliche Zahlen speichern, alle Zahlen sind gleich wahrscheinlich:

U = {1,2,...,10} (= alle möglichen Suchanfragen)

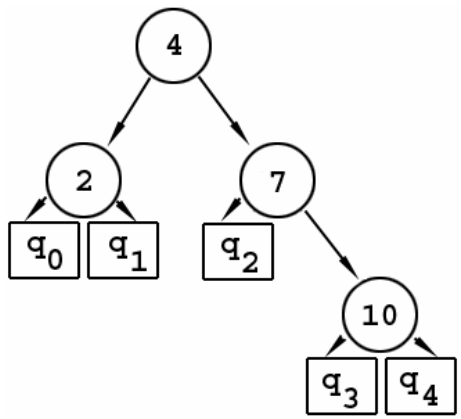
S = {2,4,7,10} (= im BST gespeicherte Elemente)

$p_1 = p_2 = p_3 = p_4 = 0{,}1$

$q_0 = 0{,}1$; $q_1 = 0{,}1$; $q_2 = 0{,}2$; $q_3 = 0{,}2$; $q_4 = 0$



C = (0,1+2*(0,1+0,1)+3*0,1)+
(2*(0,1+0,1+0)+3*(0,2+0,2)) = 2,4



C = (0,1+2*(0,1+0,1)+3*0,1)+
(2*(0,1+0,1+0,2)+3*(0,2+0)) = 2,2

Wie optimalen BST finden?

1. Ansatz: brute force:

   Für alle möglichen Bäume die Kosten berechnen:

   $O(4^n)$

2. Ansatz: Dynamische Programmierung:

   Problem in kleinere Subprobleme zerlegen:

   Optimale Teilbäume berechnen, daraus optimalen Gesamtbaum berechnen:

   $O(n^3)$

2 Optimale binäre Suchbäume

- Kostenfunktion

$$C = \sum_{i=1}^{n} \left(p_i \cdot \left(Tiefe\ \ (i) + 1 \right) \right) + \sum_{i=0}^{n} \left(q_i \cdot Tiefe\ \ (i) \right)$$

- Vereinbarung:

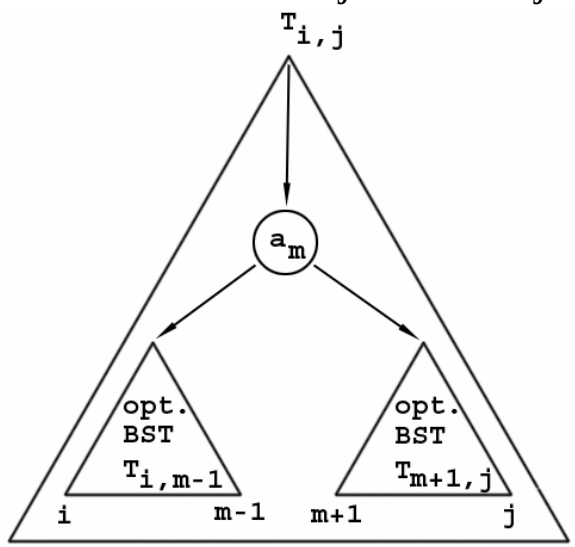
  $C_{i,j}$ = Kosten für den Teilbaum $T_{i,j}$, der a_i bis a_j enthält

  $w_{i,j}$ = Wahrscheinlichkeit, dass ein gesuchtes Element sich im Teilbaum $T_{i,j}$ befindet, also $p_i+...+p_j+q_{i-1}+q_j$

- Rekursiver Ansatz :

  $C_{i,j} = w_{i,j} + C_{i,m-1} + C_{m+1,j}$

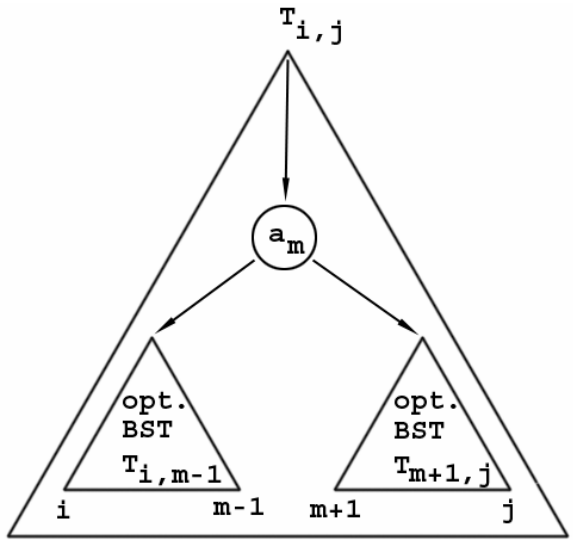
  $w_{i,j} = p_m + w_{i,m-1} + w_{m+1,j}$



- Wie findet man so den optimalen Baum?

$$C_{i,j} = w_{i,j} + \min_{i \le m \le j} \{C_{i,m-1} + C_{m+1,j}\}$$

$$w_{i,j} = w_{i,m-1} + p_m + w_{m+1,j}$$



- Gestartet wird bottom-up
- Je 2 kleinere Teilbäume werden zu einem größeren zusammengefasst
- Teilergebnisse werden in einer Tabelle gespeichert

2 Optimale binäre Suchbäume

Beispiel: S = $\{a_1,a_2,a_3,a_4\}$. Gesucht wird optimaler BST.

p_1=4/16 ; p_2=2/16 ; p_3= p_4=1/16 ; q_0=2/16 ; q_1=3/16 ; q_2= q_3= q_4=1/16

$i\downarrow$

$j \rightarrow$	0	1	2	3	4
1	$W_{1,0} = q_0 = 2$ $C_{1,0} = 0$ $R_{1,0} = 0$	$W_{1,1} = 9$ $C_{1,1} = 9$ $R_{1,1} = 1$	$W_{1,2} = 12$ $C_{1,2} = 18$ $R_{1,2} = 1$	$W_{1,3} = 14$ $C_{1,3} = 25$ $R_{1,3} = 1$	$W_{1,4} = 16$ $C_{1,4} = 33$ $R_{1,4} = 2$
2		$W_{2,1} = q_1 = 3$ $C_{2,1} = 0$ $R_{2,1} = 0$	$W_{2,2} = 6$ $C_{2,2} = 6$ $R_{2,2} = 2$	$W_{2,3} = 8$ $C_{2,3} = 11$ $R_{2,3} = 2$	$W_{2,4} = 10$ $C_{2,4} = 18$ $R_{2,4} = 2$
3			$W_{3,2} = q_2 = 1$ $C_{3,2} = 0$ $R_{3,2} = 0$	$W_{3,3} = 3$ $C_{3,3} = 3$ $R_{3,3} = 3$	$W_{3,4} = 5$ $C_{3,4} = 8$ $R_{3,4} = 4$
4				$W_{4,3} = q_3 = 1$ $C_{4,3} = 0$ $R_{4,3} = 0$	$W_{4,4} = 3$ $C_{4,4} = 3$ $R_{4,4} = 4$
5					$W_{5,4} = q_4 = 1$ $C_{5,4} = 0$ $R_{5,4} = 0$

2 Optimale binäre Suchbäume

- Pseudocode:

for i=1 **to** n+1 **do**
  $C_{i,i-1} = 0$
  $W_{i,i-1} = q_{i-1}$
  $R_{i,i-1}$ = null
end for
for d = 0 **to** n-1 **do** //Diagonale in Tabelle
  for i = 1 **to** n-d **do**
    j = i+d
    m mit $i \leq m \leq j$ so bestimmen, dass $C_{i,m-1} + C_{m+1,j}$ minimal ist
    $C_{i,j} = W_{i,j} + C_{i,m-1} + C_{m+1,j}$
    $W_{i,j} = p_m + W_{i,m-1} + W_{m+1,j}$
    $R_{i,j} = m$
  end for
end for

- Laufzeitabschätzung: $O(n^3)$

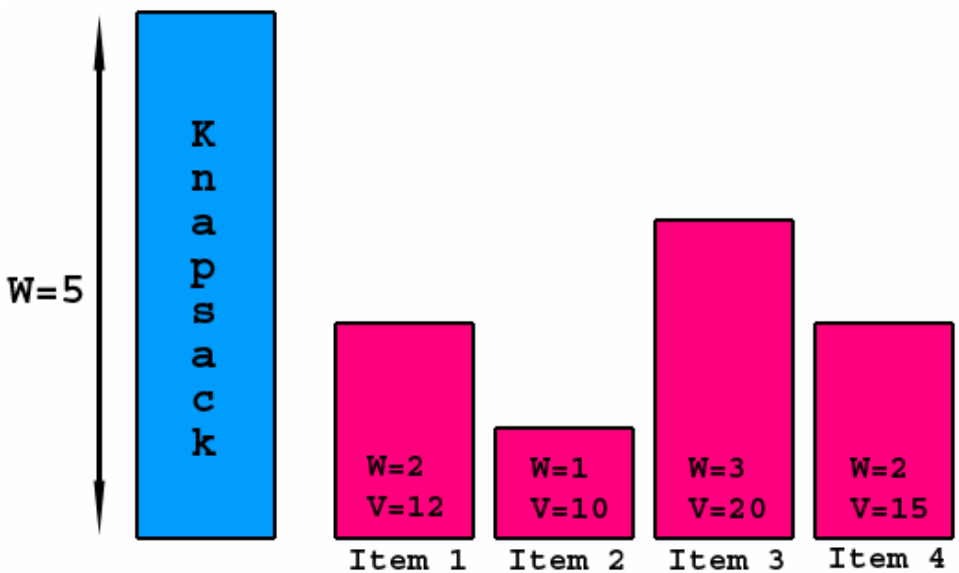
  Mit Optimierung: $O(n^2)$
  $\left(R_{i,j-1} \leq R_{i,j} \leq R_{i+1,j}\right)$

- Speicherplatz: $O(n^2)$

- Baum aufbauen: $O(n)$

3 Knapsack mit memory functions

- Rucksackproblem
- Rucksack kann nur bestimmtes Gewicht aufnehmen
- Gegenstände mit unterschiedlichem Gewicht und Wert so einpacken, dass Rucksack größtmöglichen Wert enthält



3 Knapsack mit memory functions

- Rucksack-Problem ist NP-vollständig
  - Nicht in polynomieller Zeit lösbar
- Vereinfachung:
  - Gewichte und Kapazität nur ganzzahlig
    - Lösung mittels dyn. Programmierung in polynomieller Zeit möglich

- Problem muss in Subprobleme zerlegt werden
  - Kapazität des Rucksacks von 0 schrittweise erhöhen
  - Gegenstände nacheinander hineinlegen
- Optimale Lösungen der kleineren Instanzen zur Lösung von größeren Instanzen benutzten
  - ➢ Tabellen benutzen

3 Knapsack mit memory functions

- Rekursiver Ansatz wird benötigt:

  j: Kapazität des Rucksacks

  i: Gegenstände 1 bis i wurden schon getestet

  V[i,j]: Optimale Lösung für i und j

  w_i > j : Gegenstand passt nicht hinein: **V[i,j] = V[i-1,j]**

  w_i <= j : **1) Optimale Lösung mit i-tem Gegenstand:**

  Optimale Lösung vom Rucksack ohne das Element und mit der Kapazität j-w_i plus den Wert des Gegenstands oder

  2) Optimale Lösung ohne i-ten Gegenstand:
  Optimale Lösung vom Rucksack der gleichen Kapazität ohne das Element

  V[i,j] = max {v_i+V[i-1,j-w_i], V[i-1,j]}

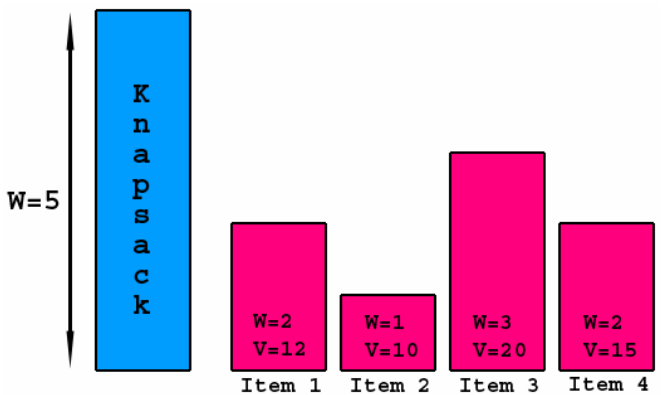
3 Knapsack mit memory functions

$$V[i,j] = \begin{cases} \max\{v_i + V[i-1, j-w_i],\ V[i-1,j]\} & \text{für } w_i <= j \\ V[i-1,j] & \text{für } w_i > j \end{cases}$$

Beispiel:

Tabelle erstellen, Spalte 0 und Zeile 0 mit Nullen füllen

Vorgehensweise bottom-up



$j \rightarrow$

Ohne memory function	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=2, V_1=12	1	0	0	12	12	12	12
W_2=1, V_2=10	2	0	10	12	22	22	22
W_3=3, V_3=20	3	0	10	12	22	30	32
W_4=2, V_4=15	4	0	10	15	25	30	37

3 Knapsack mit memory functions

- Wie berechnet man aus der Tabelle die optimale Menge an Gegenständen?

Ohne memory function

$W_1=2, V_1=12$

$W_2=1, V_2=10$

$W_3=3, V_3=20$

$W_4=2, V_4=15$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Treppenfunktion: Starten bei V[n,W]

Ist V[i-1,j] ≠ V[i,j] dann ist i in Lösung enthalten: Springen zu V[i-1,j-w_i]
Ist V[i-1,j] = V[i,j] dann ist i nicht in der Lösung: Springen zu V[i-1,j]

Hier also : item4, item2, item1 in der Lösung

3 Knapsack mit memory functions

- Nachteil durch bottom-up Vorgehensweise:
  Es werden auch Teillösungen berechnet, die nicht gebraucht werden
- Abhilfe: top-down vorgehen:
  Eine Funktion wird benötigt, die bestimmt, ob eine Teillösung benötigt wird

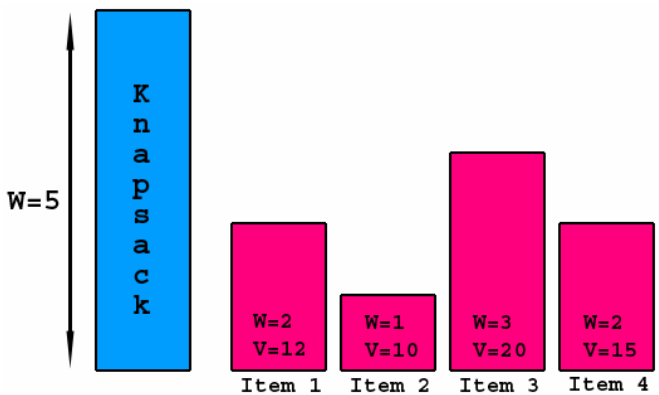
- Es wird eine Funktion benötigt, die das realisiert:
  Die **Memory Function**

- Vorgehensweise:
  - Tabelle mit null-Zeichen füllen
  - Starten im Feld V[n,W]
  - Wird eine Teillösung benötigt und der Tabelleneintrag ist null diese berechnen

3 Knapsack mit memory functions

- **Beispiel zu Knapsack mit memory function**:

$$V[i,j] = \begin{cases} \max\{v_i + V[i-1, j-w_i],\ V[i-1,j]\} & \text{für } w_i <= j \\ V[i-1,j] & \text{für } w_i > j \end{cases}$$



Mit memory function

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
4	0	-	-	-	-	37

$W_1=2, V_1=12$

$W_2=1, V_1=10$

$W_3=3, V_1=20$

$W_4=2, V_1=15$

3 Knapsack mit memory functions

- Laufzeitabschätzung ohne memory function:
  O(n*W) : *(pseudo-)*polynomiell

- Laufzeitabschätzung mit memory function:
  O(n*W), Berechnung wird also nur um konstanten Faktor erhöht

- Speicherplatz:
  O(n*W)

- Gegenstände der optimalen Lösung finden:
  O(n+W)

ENDE