20. Dynamic Programming II

Subset Sum Problem, Rucksackproblem, Greedy Algorithmus, Lösungen mit dynamischer Programmierung, FPTAS, Optimaler Suchbaum [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

Aufgabe

Hannes und Niklas sollen eine grosse Menge Geschenke unterschiedlichen monetären Wertes bekommen.

Die Eltern wollen die Geschenke vorher so gerecht aufteilen, dass kein Streit eintritt. Frage: wie geht das?

Antwort: wer Kinder hat, der weiss dass diese Aufgabe keine Lösung hat.

Realistischere Aufgabe

Teile obige “Gegenstände” so auf zwei Mengen auf, dass beide Mengen den gleichen Wert haben.

Eine Lösung:

Subset Sum Problem

Seien $n \in \mathbb{N}$ Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ gegeben.

Ziel: Entscheide, ob eine Auswahl $I \subseteq \{1, \ldots, n\}$ existiert mit

$$\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,n\} \setminus I} a_i.$$

Naiver Algorithmus

Prüfe für jeden Bitvektor $b = (b_1, \ldots, b_n) \in \{0,1\}^n$, ob

$$\sum_{i=1}^{n} b_i a_i \stackrel{?}{=} \sum_{i=1}^{n} (1 - b_i) a_i$$

Schlechtester Fall: n Schritte für jeden der 2^n Bitvektoren b. Anzahl Schritte: $\mathcal{O}(n \cdot 2^n)$.

Algorithmus mit Aufteilung

- Zerlege Eingabe in zwei gleich grosse Teile: $a_1, \ldots, a_{n/2}$ und $a_{n/2+1}, \ldots, a_n$.
- Iteriere über alle Teilmengen der beiden Teile und berechne Teilsummen $S_1^k, \ldots, S_{2^{n/2}}^k$ $(k = 1, 2)$.
- Sortiere die Teilsummen: $S_1^k \leq S_2^k \leq \cdots \leq S_{2^{n/2}}^k$.
- Prüfe ob es Teilsummen gibt, so dass $S_i^1 + S_j^2 = \frac{1}{2} \sum_{i=1}^{n} a_i =: h$
  - Beginne mit $i = 1, j = 2^{n/2}$.
  - Gilt $S_i^1 + S_j^2 = h$ dann fertig
  - Gilt $S_i^1 + S_j^2 > h$ dann $j \leftarrow j - 1$
  - Gilt $S_i^1 + S_j^2 < h$ dann $i \leftarrow i + 1$

Beispiel

Menge $\{1, 6, 2, 3, 4\}$ mit Wertesumme 16 hat 32 Teilmengen.

Aufteilung in $\{1, 6\}$, $\{2, 3, 4\}$ ergibt folgende 12 Teilmengen mit Wertesummen:

$\{1,6\}$				$\{2,3,4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1,6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2,3\}$	$\{2,4\}$	$\{3,4\}$	$\{2,3,4\}$
0	1	6	7	0	2	3	4	5	6	7	9

$\Leftrightarrow$ Eine Lösung: $\{1, 3, 4\}$

Analyse

- Teilsummegenerierung in jedem Teil: $\mathcal{O}(2^{n/2} \cdot n)$.
- Sortieren jeweils: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$.
- Zusammenführen: $\mathcal{O}(2^{n/2})$

Gesamtlaufzeit

$$\mathcal{O}\left(n \cdot 2^{n/2}\right) = \mathcal{O}\left(n\left(\sqrt{2}\right)^n\right).$$

Wesentliche Verbesserung gegenüber ganz naivem Verfahren – aber immer noch exponentiell!

Dynamische Programmierung

Aufgabe: sei $z = \frac{1}{2}\sum_{i=1}^{n} a_i$. Suche Auswahl $I \subset \{1, \ldots, n\}$, so dass $\sum_{i \in I} a_i = z$.

DP-Tabelle: $[0, \ldots, n] \times [0, \ldots, z]$-Tabelle T mit Wahrheitseinträgen. $T[k, s]$ gibt an, ob es eine Auswahl $I_k \subset \{1, \ldots, k\}$ gibt, so dass $\sum_{i \in I_k} a_i = s$.

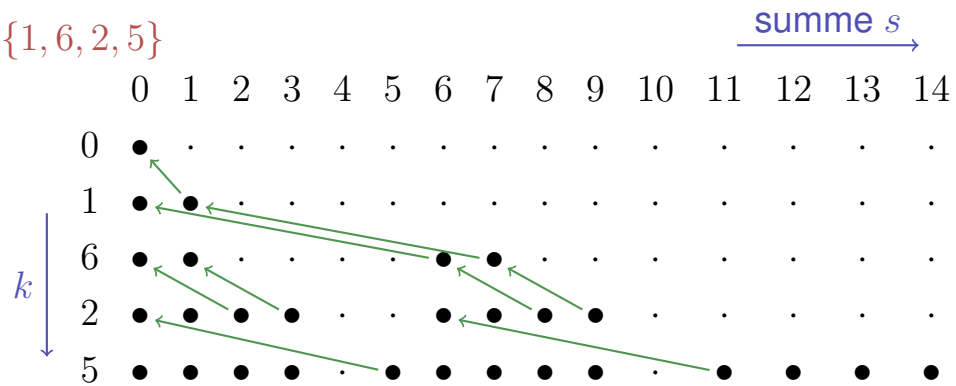
Initialisierung: $T[0, 0] =$ true. $T[0, s] =$ false für $s > 1$.

Berechnung:

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{falls } s < a_k \\ T[k-1, s] \vee T[k-1, s-a_k] & \text{falls } s \geq a_k \end{cases}$$

für aufsteigende k und innerhalb k dann s.

Beispiel



Auslesen der Lösung: wenn $T[k, s] = T[k-1, s]$ dann a_k nicht benutzt und bei $T[k-1, s]$ weiterfahren, andernfalls a_k benutzt und bei $T[k-1, s-a_k]$ weiterfahren.

Rätselhaftes

Der Algorithmus benötigt $\mathcal{O}(n \cdot z)$ Elementaroperationen.

Was ist denn jetzt los? Hat der Algorithmus plötzlich polynomielle Laufzeit?

Aufgelöst

Der Algorithmus hat nicht unbedingt eine polynomielle Laufzeit. z ist eine *Zahl* und keine *Anzahl*!

Eingabelänge des Algorithmus $\cong$ Anzahl Bits zur *vernünftigen* Repräsentation der Daten. Bei der Zahl z wäre das $\zeta = \log z$.

Also: Algorithmus benötigt $\mathcal{O}(n \cdot 2^{\zeta})$ Elementaroperationen und hat exponentielle Laufzeit in ζ.

Sollte z allerdings polynomiell sein in n, dann hat der Algorithmus polynomielle Laufzeit in n. Das nennt man *pseudopolynomiell*.

NP

Man weiss, dass der Subset-Sum Algorithmus zur Klasse der *NP*-vollständigen Probleme gehört (und somit *NP-schwer* ist).

P: Menge aller in Polynomialzeit lösbarer Probleme.

NP: Menge aller Nichtdeterministisch in Polynomialzeit lösbarer Probleme.

Implikationen:

- NP enthält P.
- Probleme in Polynomialzeit *verifizierbar*.
- Unter der (noch?) unbewiesenen[27] Annahme, dass NP $\neq$ P, gibt es für das Problem *keinen Algorithmus mit polynomieller Laufzeit*.

[27] Die bedeutenste ungelöste Frage der theoretischen Informatik!

Das Rucksackproblem

Wir packen unseren Koffer und nehmen mit ...

- Zahnbürste
- Hantelset
- Kaffemaschine
- Oh jeh – zu schwer.

- Zahnbürste
- Luftballon
- Taschenmesser
- Ausweis
- Hantelset
- Oh jeh – zu schwer.

- Zahnbürste
- Kaffemaschine
- Taschenmesser
- Ausweis
- Oh jeh – zu schwer.

Wollen möglichst viel mitnehmen. Manche Dinge sind uns aber wichtiger als andere.

Rucksackproblem (engl. Knapsack problem)

Gegeben:

- Menge von $n \in \mathbb{N}$ Gegenständen $\{1, \ldots, n\}$.
- Jeder Gegenstand i hat Nutzwert $v_i \in \mathbb{N}$ und Gewicht $w_i \in \mathbb{N}$.
- Maximalgewicht $W \in \mathbb{N}$.
- Bezeichnen die Eingabe mit $E = (v_i, w_i)_{i=1,\ldots,n}$.

Gesucht:

eine Auswahl $I \subseteq \{1, \ldots, n\}$ die $\sum_{i \in I} v_i$ maximiert unter $\sum_{i \in I} w_i \leq W$.

Gierige (engl. greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht v_i/w_i: Permutation p mit $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Füge Gegenstände in dieser Reihenfolge hinzu ($I \leftarrow I \cup \{p_i\}$), sofern das zulässige Gesamtgewicht dadurch nicht überschritten wird.

Das ist schnell: $\Theta(n \log n)$ für Sortieren und $\Theta(n)$ für die Auswahl. Aber ist es auch gut?

Gegenbeispiel zur greedy strategy

$$v_1 = 1 \qquad w_1 = 1 \quad v_1/w_1 = 1$$

$$v_2 = W - 1 \quad w_2 = W \quad v_2/w_2 = \frac{W-1}{W}$$

Greedy Algorithmus wählt $\{v_1\}$ mit Nutzwert 1.
Beste Auswahl: $\{v_2\}$ mit Nutzwert $W - 1$ und Gewicht W.

Greedy *kann* also beliebig schlecht sein.

Dynamic Programming

Unterteile das Maximalgewicht.

Dreidimensionale Tabelle $m[i, w, v]$ (“machbar”) aus Wahrheitswerten.

$m[i, w, v] =$ true genau dann wenn

- Auswahl der ersten i Teile existiert ($0 \leq i \leq n$)
- deren Gesamtgewicht höchstens w ($0 \leq w \leq W$) und
- Nutzen mindestens v ($0 \leq v \leq \sum_{i=1}^{n} v_i$) ist.

Berechnung der DP Tabelle

Initial

- $m[i, w, 0] \leftarrow$ true für alle $i \geq 0$ und alle $w \geq 0$.
- $m[0, w, v] \leftarrow$ false für alle $w \geq 0$ und alle $v > 0$.

Berechnung

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{falls } w \geq w_i \text{ und } v \geq v_i \\ m[i-1, w, v] & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach w und für festes i und w aufsteigend nach v.

Lösung: Grösstes v, so dass $m[i, w, v] =$ true für ein i und w.

Beobachtung

Nach der Definition des Problems gilt offensichtlich, dass

- für $m[i, w, v] =$ true gilt:
  $m[i', w, v] =$ true $\forall i' \geq i$,
  $m[i, w', v] =$ true $\forall w' \geq w$,
  $m[i, w, v'] =$ true $\forall v' \leq w$.
- für $m[i, w, v] =$ false gilt:
  $m[i', w, v] =$ false $\forall i' \leq i$,
  $m[i, w', v] =$ false $\forall w' \leq w$,
  $m[i, w, v'] =$ false $\forall v' \geq w$.

Das ist ein starker Hinweis darauf, dass wir keine 3d-Tabelle benötigen.

DP Tabelle mit 2 Dimensionen

Tabelleneintrag $t[i, w]$ enthält statt Wahrheitswerten das jeweils grösste v, das erreichbar ist[28] mit

- den Gegenständen $1, \ldots, i$ $(0 \leq i \leq n)$
- bei höchstem zulässigen Gewicht w $(0 \leq w \leq W)$.

[28] So etwas ähnliches hätten wir beim Subset Sum Problem auch machen können, um die dünnbesetzte Tabelle etwas zu verkleinern

Berechnung

Initial

- $t[0, w] \leftarrow 0$ für alle $w \geq 0$.

Berechnung

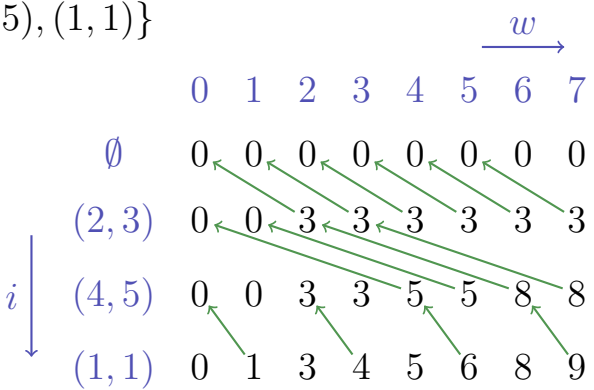
$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{falls } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach w.

Lösung steht in $t[n, w]$

Beispiel

$E = \{(2,3),(4,5),(1,1)\}$



Auslesen der Lösung: wenn $t[i,w] = t[i-1,w]$ dann Gegenstand i nicht benutzt und bei $t[i-1,w]$ weiterfahren, andernfalls benutzt und bei $t[i-1,s-w_i]$ weiterfahren.

Analyse

Die beiden Algorithmen für das Rucksackproblem haben eine Laufzeit in $\Theta(n \cdot W \cdot \sum_{i=1}^{n} v_i)$ (3d-Tabelle) und $\Theta(n \cdot W)$ (2d-Tabelle) und sind beide damit pseudopolynomiell, liefern aber das bestmögliche Resultat.

Der greedy Algorithmus ist sehr schnell, liefert aber unter Umständen beliebig schlechte Resultate.

Im folgenden beschäftigen wir uns mit einer Lösung dazwischen.

Approximation

Sei ein $\varepsilon \in (0, 1)$ gegeben. Sei I_{opt} eine bestmögliche Auswahl.

Suchen eine gültige Auswahl I mit

$$\sum_{i \in I} v_i \geq (1 - \varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Summe der Gewichte darf W natürlich in keinem Fall überschreiten.

Andere Formulierung des Algorithmus

Bisher: Gewichtsschranke $w \rightarrow$ maximaler Nutzen v

Umkehrung: Nutzen $v \rightarrow$ minimales Gewicht w

$\Rightarrow$ Alternative Tabelle: $g[i, v]$ gibt das minimale Gewicht an, welches

- eine Auswahl der ersten i Gegenstände ($0 \leq i \leq n$) hat, die
- einen Nutzen von genau v aufweist ($0 \leq v \leq \sum_{i=1}^{n} v_i$).

Berechnung

Initial

- $g[0,0] \leftarrow 0$
- $g[0,v] \leftarrow \infty$ (Nutzen v kann mit 0 Gegenständen nie erreicht werden.).

Berechnung

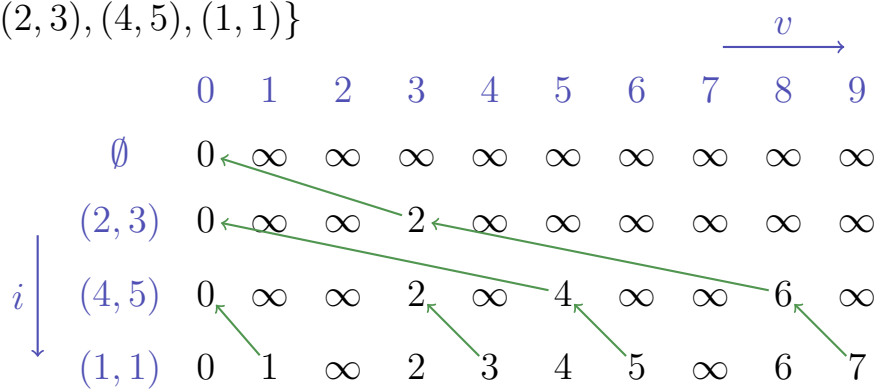
$$g[i,v] \leftarrow \begin{cases} g[i-1,v] & \text{falls } v < v_i \\ \min\{g[i-1,v], g[i-1,v-v_i]+w_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach v.

Lösung ist der grösste Index v mit $g[n,v] \leq w$.

Beispiel

$E = \{(2,3),(4,5),(1,1)\}$



Auslesen der Lösung: wenn $g[i,v] = g[i-1,v]$ dann Gegenstand i nicht benutzt und bei $g[i-1,v]$ weiterfahren, andernfalls benutzt und bei $g[i-1,b-v_i]$ weiterfahren.

Der Approximationstrick

Pseudopolynomielle Laufzeit wird polynomiell, wenn vorkommenden Werte in Polynom der Eingabelänge beschränkt werden können.

Sei $K > 0$ *geeignet* gewählt. Ersetze die Nutzwerte v_i durch "gerundete Werte" $\tilde{v_i} = \lfloor v_i / K \rfloor$ und erhalte eine neue Eingabe $E' = (w_i, \tilde{v_i})_{i=1\ldots n}$.

Wenden nun den Algorithmus auf Eingabe E' mit derselben Gewichtsschranke W an.

Idee

Beispiel $K = 5$

Eingabe Nutzwerte

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots, 98, 99, 100$$
$$\rightarrow$$
$$0, 0, 0, 0, 1, 1, 1, 1, 1, 2, \ldots, 19, 19, 20$$

Offensichtlich weniger unterschiedliche Nutzwerte

Eigenschaften des neuen Algorithmus

- Auswahl von Gegenständen aus E' ist genauso gültig wie die aus E. Gewicht unverändert!
- Laufzeit des Algorithmus ist beschränkt durch $\mathcal{O}(n^2 \cdot v_{\max}/K)$ ($v_{\max} := \max\{v_i | 1 \leq i \leq n\}$)

Wie gut ist die Approximation?

Es gilt

$$v_i - K \leq K \cdot \left\lfloor \frac{v_i}{K} \right\rfloor = K \cdot \tilde{v}_i \leq v_i$$

Sei I'_{opt} eine optimale Lösung von E'. Damit

$$\left(\sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K \overset{|I_{\text{opt}}| \leq n}{\leq} \sum_{i \in I_{\text{opt}}} (v_i - K) \leq \sum_{i \in I_{\text{opt}}} (K \cdot \tilde{v}_i) = K \sum_{i \in I_{\text{opt}}} \tilde{v}_i$$

$$\underset{I'_{\text{opt}} \text{optimal}}{\leq} K \sum_{i \in I'_{\text{opt}}} \tilde{v}_i = \sum_{i \in I'_{\text{opt}}} K \cdot \tilde{v}_i \leq \sum_{i \in I'_{\text{opt}}} v_i.$$

Wahl von K

Forderung:

$$\sum_{i \in I'} v_i \geq (1 - \varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Ungleichung von oben:

$$\sum_{i \in I'_{\text{opt}}} v_i \geq \left(\sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K$$

Also: $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$.

Wahl von K

Wähle $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$. Die optimale Summe ist aber unbekannt, daher wählen wir $K' = \varepsilon \frac{v_{\max}}{n}$.[29]

Es gilt $v_{\max} \leq \sum_{i \in I_{\text{opt}}} v_i$ und somit $K' \leq K$ und die Approximation ist sogar etwas besser.

Die Laufzeit des Algorithmus ist beschränkt durch

$$\mathcal{O}(n^2 \cdot v_{\max}/K') = \mathcal{O}(n^2 \cdot v_{\max}/(\varepsilon \cdot v_{\max}/n)) = \mathcal{O}(n^3/\varepsilon).$$

[29] Wir können annehmen, dass vorgängig alle Gegenstände i mit $w_i > W$ entfernt wurden.

FPTAS

Solche Familie von Algorithmen nennt man *Approximationsschema*: die Wahl von ε steuert Laufzeit und Approximationsgüte.

Die Laufzeit $\mathcal{O}(n^3/\varepsilon)$ ist ein Polynom in n und in $\frac{1}{\varepsilon}$. Daher nennt man das Verfahren auch ein voll polynomielles Approximationsschema *FPTAS - Fully Polynomial Time Approximation Scheme*

Optimale binäre Suchbäume

Gegeben: Suchwahrscheinlichkeiten p_i zu jedem Schlüssel k_i ($i = 1, \ldots, n$) und q_i zu jedem Intervall d_i ($i = 0, \ldots, n$) zwischen Suchschlüsseln eines binären Suchbaumes. $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.

Gesucht: Optimaler Suchbaum T mit Schlüsseltiefen $\mathrm{depth}(\cdot)$, welcher die erwarteten Suchkosten

$$\begin{aligned} C(T) &= \sum_{i=1}^{n} p_i \cdot (\mathrm{depth}(k_i) + 1) + \sum_{i=0}^{n} q_i \cdot (\mathrm{depth}(d_i) + 1) \\ &= 1 + \sum_{i=1}^{n} p_i \cdot \mathrm{depth}(k_i) + \sum_{i=0}^{n} q_i \cdot \mathrm{depth}(d_i) \end{aligned}$$

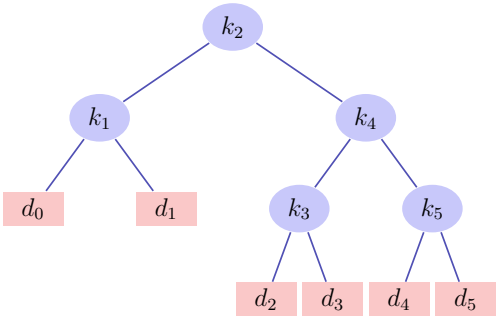
minimiert.

Beispiel

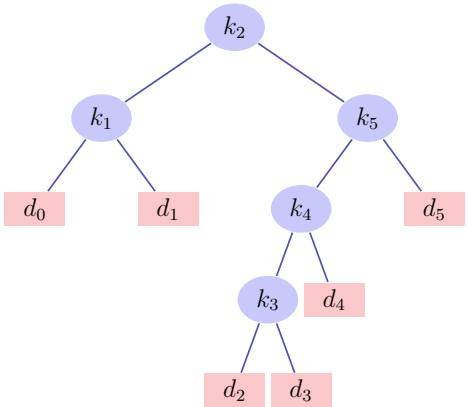
Erwartete Häufigkeiten

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Beispiel



Suchbaum mit erwarteten Kosten 2.8



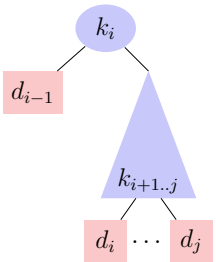
Suchbaum mit erwarteten Kosten 2.75

Struktur eines optimalen Suchbaumes

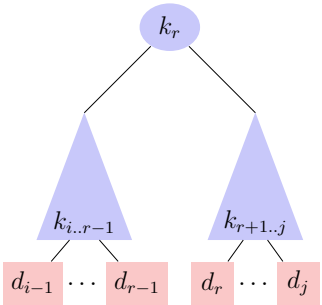
- Teilsuchbaum mit Schlüsseln $k_i, \ldots, k_j$ und Intervallschlüsseln $d_{i-1}, \ldots, d_j$ muss für das entsprechende Teilproblem optimal sein.[30]
- Betrachten aller Teilsuchbäume mit Wurzel k_r, $i \leq r \leq j$ und optimalen Teilbäumen $k_i, \ldots, k_{r-1}$ und $k_{r+1}, \ldots, k_j$

[30] Das übliche Argument: wäre er nicht optimal, könnte er durch eine bessere Lösung ersetzt werden, welche die Gesamtlösung verbessert.

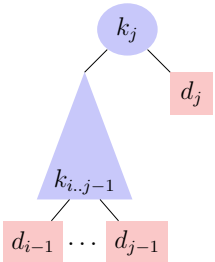
Teilsuchbäume



leerer linker Teilsuchbaum



Teilsuchbäume links und rechts nichtleer



leerer rechter Teilsuchbaum

Erwartete Suchkosten

Sei $\text{depth}_T(k)$ die Tiefe des Knotens im Teilbaum T. Sei k_r die Wurzel eines Teilbaumes T_r und T_{L_r} und T_{R_r} der linke und rechte Teilbaum von T_r. Dann

$$\text{depth}_T(k_i) = \text{depth}_{T_{L_r}}(k_i) + 1, \ (i < r)$$
$$\text{depth}_T(k_i) = \text{depth}_{T_{R_r}}(k_i) + 1, \ (i > r)$$

Erwartete Suchkosten

Seien $e[i, j]$ die Kosten eines optimalen Suchbaumes mit Knoten $k_i, \ldots, k_j$.

Basisfall: $e[i, i-1]$, erwartete Suchkosten d_{i-1}

Sei $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$.

Wenn k_r die Wurzel eines optimalen Teilbaumes mit Schlüsseln $k_i, \ldots, k_j$, dann

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

mit $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$:

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j).$$

Dynamic Programming

$$e[i,j] = \begin{cases} q_{i-1} & \text{falls } j = i-1, \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{falls } i \le j \end{cases}$$

Berechnung

Tabellen $e[1 \ldots n+1, 0 \ldots n], w[1 \ldots n+1, 0 \ldots m], r[1 \ldots n, 1 \ldots n]$
Initial

- $e[i, i-1] \leftarrow q_{i-1}$, $w[i, i-1] \leftarrow q_{i-1}$ für alle $1 \leq i \leq n+1$.

Berechnung

$$
\begin{aligned}
w[i,j] &= w[i,j-1] + p_j + q_j \\
e[i,j] &= \min_{i \leq r \leq j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} \\
r[i,j] &= \arg\min_{i \leq r \leq j} \{e[i,r-1] + e[r+1,j] + w[i,j]\}
\end{aligned}
$$

für Intervalle $[i,j]$ mit ansteigenden Längen $l = 1, \ldots, n$, jeweils für $i = 1, \ldots, n-l+1$. Resultat steht in $e[1,n]$, Rekonstruktion via r. Laufzeit $\Theta(n^3)$.

Beispiel

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

w

j \ i	1	2	3	4	5	6
0	0.05					
1	0.30	0.10				
2	0.45	0.25	0.05			
3	0.55	0.35	0.15	0.05		
4	0.70	0.50	0.30	0.20	0.05	
5	1.00	0.80	0.60	0.50	0.35	0.10

e

j \ i	1	2	3	4	5	6
0	0.05					
1	0.45	0.10				
2	0.90	0.40	0.05			
3	1.25	0.70	0.25	0.05		
4	1.75	1.20	0.60	0.30	0.05	
5	2.75	2.00	1.30	0.90	0.50	0.10

r

j \ i	1	2	3	4	5
1	1				
2	1	2			
3	2	2	3		
4	2	2	4	4	
5	2	4	5	5	5