

GEWICHTETES INTERVALL-SCHEDULING

Eingabe: n Aufgaben A_1, A_2, \dots, A_n mit Startpunkten $s_1, s_2, \dots, s_n \in \mathbb{N}$, Endpunkten $e_1, e_2, \dots, e_n \in \mathbb{N}$ im Intervall $[0, T]$, und Längen $\ell_i = e_i - s_i$

Ausgabe: Ein Schedule auf *einem* Prozessor ohne Überlappung, mit maximaler *Gesamtlänge* der ausgeführten Aufgaben

Algorithmus:

sei $L(t)$ die optimale Gesamtlänge im Intervall $[0, t] \subseteq [0, T]$

$$L(0) := 0$$

FOR $t=1$ to T DO

$$L(t) := \max\{L(t-1), \max\{\ell_i + L(s_i) \mid \text{alle Aufgaben mit } e_i = t\}\}$$

(wir füllen eine Tabelle mit $L(0), L(1), L(2), \dots, L(T)$ und benutzen die Werte immer wieder für die Berechnung neuer Werte)

DYNAMISCHE PROGRAMMIERUNG

Das RUCKSACK Problem

Eingabe: eine Gewichtsschranke G , und n Objekte
mit Gewichten $g_1, g_2, \dots, g_n \in \mathbb{N}$
und Werten $w_1, w_2, \dots, w_n \in \mathbb{R}$

(wir nehmen $g_i \leq G$ an)

Ausgabe: Bepacke einen Rucksack mit Objekten maximaler Gesamtwert
so dass es die *Gewichtsschranke* G nicht übersteigt

(die Entscheidungsvariante ist NP-vollständig)

Dynamische Programmierung für das RUCKSACK Problem mit ganzzahligen Gewichten

Sei $W_i(g)$ der *maximale Wert* einer Auswahl von den ersten i Objekten mit Gesamtgewicht *genau* g .

(setze $W_i(g) = 0$ für $g = 0$ und $W_i(g) = -\infty$ sonst)

FOR $i = 1$ TO n DO

FOR $g = 0$ TO G DO

$$W_i(g) = \max\{W_{i-1}(g), w_i + W_{i-1}(g - g_i)\}$$

Theorem: Das RUCKSACK Problem mit ganzzahligen Gewichten g_i und Gewichtsschranke G kann in Zeit $\mathcal{O}(n \cdot G)$ gelöst werden.

Definition: Pseudopolynomielle Algorithmen

Aber: G ist nicht polynomiell in der Eingabelänge!

Ein Algorithmus heißt *pseudopolynomiell* wenn seine Laufzeit durch $\text{Poly}(n, Z)$ beschränkt ist wobei n die Eingabelänge und Z die (in Absolutwert) größte Zahl in der Eingabe ist (falls alle in der Eingabe vorkommende Zahlen natürlich sind).



Anders formuliert:

wenn seine Laufzeit polynomiell ist in der Eingabelänge,
falls die Eingabe unär kodiert ist.

(die Laufzeit $\mathcal{O}(n \cdot G)$ für RUCKSACK ist pseudopolynomiell)

Dynamische Programmierung (Wiederholung)

Das Problem wird in kleinere **Teilprobleme** aufgebrochen. Optimale Werte einfacherer Teilprobleme werden in einer **'Tabelle'** gespeichert und für die Lösung schwierigerer Teilprobleme verwendet (mit Hilfe einer **rekursiven Definition**).

Wann wird dynamische Programmierung benutzt?

- optimale Teilstruktur die optimale Lösung eines Problems enthält *optimale* Lösungen seiner Teilprobleme
- überlappende Teilprobleme relativ wenige Teilprobleme die während der Berechnung immer wieder vorkommen

Siehe Cormen-Leiserson-Rivest-Stein Introduction to Algorithms Chapter 15. Dynamic Programming

Dynamische Programmierung

(Vergleich mit Divide & Conquer)

- man kann die Lösung von Teilproblemen als einen *gerichteten, azyklischen Graphen (DAG)* modellieren;

$P_i \longrightarrow P_j$ bedeutet dass die Lösung von P_i in der Lösung von P_j verwendet wird;

- In Divide & Conquer ist der Graph ein (gerichteter) *Baum*, und die Teillösungen werden nicht gespeichert; (rekursive Berechnung = top-down)

Im Dynamischen Programmieren ist der Graph ein DAG (directed acyclic graph); die Lösungen werden mehrfach benutzt, und deshalb gespeichert; (bottom-up Berechnung)

- *Tendenziell:* In der Rekursionsgleichung sind Teilprobleme nicht um einen multiplikativen Faktor (wie bei D&C), sondern nur um eine additive Konstante kleiner;

Ein volles Approximationschema für RUCKSACK

für *beliebige* Gewichte g_1, g_2, \dots, g_n und Werte w_1, w_2, \dots, w_n :

- setze $s = \frac{\varepsilon \cdot w_{\max}}{n}$ und sei $w_i^* = \lfloor w_i/s \rfloor$
- berechne eine exakte Lösung für die Instanz $(g_1, g_2, \dots, g_n; w_1^*, w_2^*, \dots, w_n^*)$
- sei B die gewählte Objektmenge
gib B (mit echtem Wert $\sum_{i \in B} w_i$) als Lösung aus

Theorem 1: Die Laufzeit ist $\mathcal{O}(n^3 \cdot \frac{1}{\varepsilon})$.

Theorem 2: Der Algorithmus ist $(1 + 2\varepsilon)$ -approximativ.

Beweis: Sei $B \subseteq \{1, 2, \dots, n\}$ die ausgegebene Bepackung und B_{OPT} eine optimale Bepackung.

$$\begin{aligned} OPT(I) &= \sum_{B_{OPT}} w_i = \sum_{B_{OPT}} s \cdot \frac{w_i}{s} \leq \sum_{B_{OPT}} s \cdot (\lfloor \frac{w_i}{s} \rfloor + 1) \leq \\ &\leq s \cdot \sum_{B_{OPT}} \lfloor \frac{w_i}{s} \rfloor + s \cdot n \leq s \cdot \sum_B \lfloor \frac{w_i}{s} \rfloor + s \cdot n \leq \\ &\leq s \cdot \sum_B \frac{w_i}{s} + s \cdot n = \sum_B w_i + \varepsilon \cdot w_{\max} \leq FPTAS(I) + \varepsilon OPT(I) \end{aligned}$$

$$FPTAS(I) \geq (1 - \varepsilon) \cdot OPT(I) \geq \frac{OPT(I)}{1 + 2\varepsilon}$$

Gewichtetes VERTEX COVER auf Bäumen

Eingabe: Ein ungerichteter Baum $T(V, E)$ mit einer Gewichtung der Knoten w_v für jeden $v \in V$.

Ausgabe: Eine Knotenüberdeckung mit minimalem Gewicht.

Ein effizienter Algorithmus mit Dynamischer Programmierung

- wähle einen beliebigen Knoten v_0 als Wurzel
(und richte alle Kanten weg von v_0)
jetzt hat jeder andere Knoten einen *Vater*, und ≥ 0 Kinder
- für jeden $v \in V$ sei T_v der Teilbaum mit Wurzel v

- für jeden Knoten $v \in V$ berechnen wir 'bottom-up'
 - $D_v(0)$: Gewicht einer minimalen Knotenüberdeckung C_v des Teilbaums T_v so dass $v \notin C_v$;
 - $D_v(1)$: Gewicht einer minimalen Knotenüberdeckung C_v des Teilbaums T_v so dass $v \in C_v$.
- für Blätter gilt $D_v(0) = 0$ und $D_v(1) = w_v$
- falls v die Kinder u_1, u_2, \dots, u_m hat, dann

$$D_v(0) = \sum_{i=1}^m D_{u_i}(1)$$

$$D_v(1) = w_v + \sum_{i=1}^m \min\{D_{u_i}(0), D_{u_i}(1)\}$$

- das optimale Gewicht für den ganzen Baum ist

$$\min\{D_{v_0}(0), D_{v_0}(1)\}$$

Die minimale Knotenüberdeckung (top-down)

sei $C = \emptyset$;

laufe von v_0 mit BFS oder DFS nach unten

IF $v = v_0$ or Vater(v) $\in C$ THEN

 IF $D_v(1) \leq D_v(0)$ THEN $C = C \cup \{v\}$

 ELSE verwirf v

IF Vater(v) $\notin C$ THEN $C = C \cup \{v\}$