Algorithmen und Datenstrukturen

Beispiel: Lernplanung

Prof. Justus Piater, Ph.D.

4. Juli 2023

In den vorhergehenden Kapiteln haben wir klassische abstrakte Datentypen, Datenstrukturen und Algorithmen eingeführt und anhand typischer Anwendungen illustriert. In der Praxis ist die Situation genau andersherum: Sie sind mit einem Anwendungsproblem konfrontiert und müssen entscheiden, auf Basis welcher abstrakter Datentypen, Datenstrukturen und Algorithmen Sie es lösen. Hier betrachten wir nun eine solche Anwendung, und gehen es mit verschiedenen algorithmischen Ansätzen an.

Dieses Kapitel folgt einem Fachartikel des Autors, dessen ergänzende Lektüre empfohlen wird. [Piater 2009]

Inhaltsverzeichnis

1 Beispiel: Lernplanung

Sie möchten sich auf die Abschlussklausur vorbereiten. Sie haben N Tage Zeit, um K Kapitel durchzuarbeiten. Die Kapitel bauen aufeinander auf und sollten in der gegebenen Reihenfolge bearbeitet werden. Sie möchten jedes Kapitel jeweils komplett an einem Tag abarbeiten und es nicht über mehrere Tage aufteilen. Jedes Kapitel erfordert jeweils einen gegebenen Arbeitsaufwand.

Wie finden Sie eine Aufteilung der K Kapitel auf N Tage, die den Arbeitsaufwand möglichst gleichmäßig auf die N Tage verteilt?

Lernplanung [Slide 1]

Gegeben: Eine Sequenz S von K Elementen mit Gewichten w_k, $k = 1, ..., K$.

k_n ist das *letzte* Kapitel, das ich am Tag n lesen sollte, damit meine Arbeitsbelastung möglichst gleichmäßig verteilt ist:

Gesucht: Eine Partitionierung $0 = k_0 \leq k_1 \leq \cdots \leq k_N = K$ von S in N Teilsequenzen, die

$$\begin{aligned} C &= \sum_{n=1}^{N} \left(\bar{w} - \sum_{k=k_{n-1}+1}^{k_n} w_k \right)^2 \\ &\propto \sum_{n=1}^{N} \left(\sum_{k=k_{n-1}+1}^{k_n} w_k \right)^2 = \sum_{n=1}^{N} C_{k_{n-1}+1,k_n} \end{aligned}$$

minimiert, wobei $\bar{w} = \frac{1}{N} \sum_{k=1}^{K} w_k$.

Beweis der Proportionalität: Schreibe $C = \sum_n (c - v_n)^2$ und expandiere; $\sum_n v_n = \sum_{n=1}^{N} \sum_{k=k_{n-1}+1}^{k_n} w_k = \sum_{k=1}^{K} w_k$ ist unabhängig von der Partitionierung.

Anmerkung

Dies ist nur interessant, falls $K > N$.

Brute-Force-Algorithmus? [Slide 2]

Die Zahl der möglichen Partitionierungen ist

$$s(K,N) = \begin{cases} 1 & \text{falls } K \leq N \text{ oder } N \leq 1, \\ \sum_{k=1}^{K-N+1} s(K-k, N-1) & \text{andernfalls.} \end{cases}$$

- Gibt es weniger Kapitel als Tage, oder gibt es nur einen Tag, dann sind alle sinnvollen Lösungen äquivalent.
- Andernfalls beenden wir den ersten Tag nach Kapitel k, $k = 1, \ldots, k_{\max}$ (wobei $k_{\max}$ mindestens ein Kapitel für jeden verbleibenden Tag übrig lässt), und fahren rekursiv mit der um einen Tag verkürzten Sequenz und den verbleibenden Kapiteln fort.

Falls $K \gg N$, ist dies mindestens $\Omega\left(2^N\right)$.

Diese untere Grenze wird erreicht, falls es für jeden Tag n genau zwei unabhängige Möglichkeiten k_n gibt, das letzte Kapitel für diesen Tag zu wählen.

Anzahl der möglichen Partitionierungen [Slide 3]

	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$	$N=7$	$N=8$	$N=9$	$N=10$
$K=1$	1	1	1	1	1	1	1	1	1	1
$K=2$	1	1	1	1	1	1	1	1	1	1
$K=3$	1	2	1	1	1	1	1	1	1	1
$K=4$	1	3	3	1	1	1	1	1	1	1
$K=5$	1	4	6	4	1	1	1	1	1	1
$K=6$	1	5	10	10	5	1	1	1	1	1
$K=7$	1	6	15	20	15	6	1	1	1	1
$K=8$	1	7	21	35	35	21	7	1	1	1
$K=9$	1	8	28	56	70	56	28	8	1	1
$K=10$	1	9	36	84	126	126	84	36	9	1
$K=11$	1	10	45	120	210	252	210	120	45	10
$K=12$	1	11	55	165	330	462	462	330	165	55
$K=13$	1	12	66	220	495	792	924	792	495	220
$K=14$	1	13	78	286	715	1287	1716	1716	1287	715
$K=15$	1	14	91	364	1001	2002	3003	3432	3003	2002
$K=16$	1	15	105	455	1365	3003	5005	6435	6435	5005
$K=17$	1	16	120	560	1820	4368	8008	11440	12870	11440
$K=18$	1	17	136	680	2380	6188	12376	19448	24310	24310
$K=19$	1	18	153	816	3060	8568	18564	31824	43758	48620
$K=20$	1	19	171	969	3876	11628	27132	50388	75582	92378

Für die Planung dieses Kurses: Der Kurs enthält $K = 26$ unteilbare Abschnitte (ohne dieses Kapitel), die möglichst gleichmäßig auf $N = 12$ wöchentliche Unterrichtseinheiten zu verteilen sind: $s(26, 12) = 4457400$ verschiedene Möglichkeiten

Greedy-Algorithmus? [Slide 4]

Für $n = 1, ..., N$ wähle k_n so, dass

$$C_n = \left(\bar{w} - \sum_{k=k_{n-1}+1}^{k_n} w_k \right)^2$$

minimiert wird.

Laufzeit?

$O(K)$

Minimiert diese Lösung $C = \sum_{n=1}^{N} C_n$?

Beispiel:

$w_k = 3$, $K = 8$, $N = 6$

Divide-and-Conquer-Algorithmus? [Slide 5]

1. **Divide:** Falls $K \leq N$ oder falls $N \leq 1$, gibt es nur eine Lösung. Andernfalls finde

$$\begin{aligned} n_{\text{mid}} &= \left\lfloor \frac{N}{2} \right\rfloor \\ k_{\text{mid}} &= \operatorname*{argmin}_l \{C_{1,l} + C_{l+1,K}\} \end{aligned}$$

2. **Rekursion:** Löse die beiden Teilprobleme.

3. **Conquer:** Verkette die beiden Teillösungen.

Laufzeit?

Rekursionsbaum mit $O(K)$ Arbeitsschritten auf jeder Ebene (zum Finden der Mittelpunkte), und $\log N$ Ebenen, also $O(K \log N)$

Minimiert diese Lösung C?

Beispiel:

$S = \{2, 2, 2, 2, 8\}$, $N = 4$

Dynamic Programming [Slide 6]

Optimale Unterproblemstruktur: In einer optimalen Partitionierung der Länge N ist die Teilpartitionierung der ersten $N-1$ Elemente optimal.

Beweis?

Durch Widerspruch: Eine Verbesserung dieser Teilpartitionierung verbessert auch die gesamte Partitionierung.

Korollar: Um eine optimale Partitionierung der Länge N zu finden, genügt es, die beste Kombination einer optimalen Teilpartitionierung der Länge $N-1$ der $k < K$ ersten Elemente mit den verbleibenden $K-k$ Elementen des letzten Tages zu bestimmen.

Überlappung der Unterprobleme: Um die optimale Partitionierung der Länge N zu finden, berücksichtigen wir Teillösungen, die wir bereits für $N-1$ berechnet haben.

Genau dies war beim Brute-Force-Algorithmus nicht der Fall!

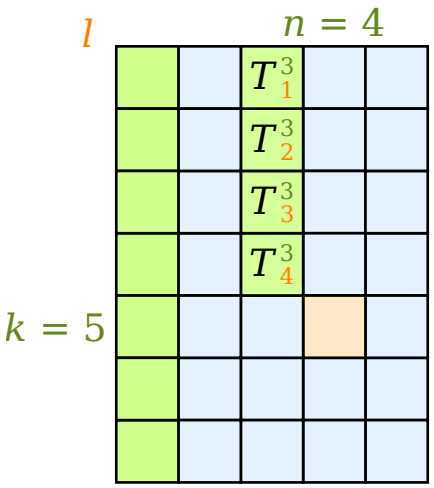
Einfacher DP-Algorithmus [Slide 7]

Fülle eine Tabelle T mit den minimalen Kosten einer Partitionierung der Länge n der Elemente $1, ..., k$:

$$T_k^n = \min_{1 \le l < k} \left\{ T_l^{n-1} + C_{l+1,k} \right\}$$

Dies ist exakt die mathematische Umsetzung des obigen Korollars.

(T besitzt einen unteren und einen oberen Index, keinen Exponenten.)



Laufzeit?

Tabelle mit $\Theta(KN)$ Einträgen, deren Berechnung jeweils eine Zeit von $\Theta(K)$ benötigt, also $\Theta(K^2N)$.

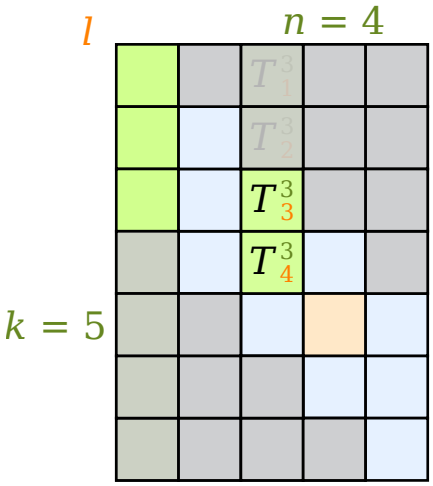
Wie finden wir die Partitionierung?

In einer Hilfstabelle L merken wir uns Für jeden Eintrag in T den Wert l, der die Teilkosten minimiert (also das letzte Kapitel des Vortags), und geben rekursiv alle optimalen Teilsequenzen aus.

Verbesserter DP-Algorithmus [Slide 8]

Beobachtung: Um T_k^n zu berechnen, ist es überflüssig,

- die $l < n-1$ zu betrachten, denn diese lassen Tage zu Beginn der Sequenz ungenutzt;
- die $k > K - N + n$ zu betrachten, denn diese lassen Tage am Ende der Sequenz ungenutzt.



Dies ergibt

$$T_k^n = \min_{n-1 \le l < k} \left\{ T_l^{n-1} + C_{l+1,k} \right\}$$

wobei man die Spalte n jeweils nur für die Zeilen k mit $n \le k \le K - N + n$ ausfüllt.

So enthält jede Spalte der Tabelle nur $K - N + 1$ Werte!

Laufzeit?

Tabelle mit $\Theta((K-N)N)$ Einträgen, deren Berechnung jeweils $\Theta(K-N)$ Zeit benötigt, also $\Theta\left((K-N)^2 N\right)$.

Beispiel [Slide 9]

Verteile $K = 6$ Kapitel mit Gewichten $w_{k=1,\dots,6} = 2, 5, 3, 4, 7, 6$ auf $N = 3$ Tage.

$$T_k^n = \begin{bmatrix} 4 & & \\ 49 & 29 & \\ \mathbf{100} & 58 & 38 \\ 196 & 98 & 74 \\ & \mathbf{221} & 147 \\ & & \mathbf{257} \end{bmatrix}, L_k^n = \begin{bmatrix} 0 & & \\ 0 & 1 & \\ \mathbf{0} & 2 & 2 \\ 0 & 2 & 3 \\ & \mathbf{3} & 4 \\ & & \mathbf{5} \end{bmatrix}$$

- Tag 1: Kapitel 1–3, $\sqrt{C_{1,3}} = 10$
- Tag 2: Kapitel 4–5, $\sqrt{C_{4,5}} = 11$
- Tag 3: Kapitel 6, $\sqrt{C_{6,6}} = 6$

Python-Implementation [Slide 10]

```
#!/usr/bin/env python3
# -*- python -*-

import sys
import numpy as np


def displayPartitions(W, Wcum, T, L, k, n):
    if n > 1:
        displayPartitions(W, Wcum, T, L, L[k, n], n - 1)

    print("%g:" % (Wcum[k] - Wcum[L[k, n]]), end=' ')
    for w in range(L[k, n] + 1, k + 1):
        print(W[w], end=' ')
    print()


def partition(W, K, N):
    T = np.zeros((K + 1, N + 1)) # cost table
    # To recover the  partitions :
    L = np.zeros((K + 1, N + 1), int) # table of min-cost separators

    Wcum = np.cumsum(W)

    for k in range(1, K - N + 1 + 1):
        T[k, 1] = Wcum[k] * Wcum[k]

    for n in range(2, N + 1):
        for k in range(n, K - N + n + 1):
            tMin = float('Inf')
            for l in range(n - 1, k):
                wRest = Wcum[k] - Wcum[l]
                t = T[l, n - 1] + wRest * wRest
                if t < tMin:
                    tMin = t
                    T[k, n] = t
                    L[k, n] = l
    print(T)
    print(L)

    displayPartitions(W, Wcum, T, L, K, N)


if len(sys.argv) < 2:
    print("Usage:   " + sys.argv[0] + " w1 w2 ... wK N")
    print("Example: " + sys.argv[0] + " 2 5 3 4 7    3")
    print("Example: " + sys.argv[0] + " 1 2 3 4 5 4 3 2 1   5")
    print("Example: " + sys.argv[0] + " 5 4 3 2 1 2 3 4 5   5")
    sys.exit(0)
```

```
# Mathematical indexing: All valid data start at index 1.
K = len(sys.argv) - 2        # number of chapters
N = int(sys.argv[K + 1])     # number of readings (= reading sessions)
W = [0.] + list(map(float, sys.argv[1:K + 1])) # chapter weights

if N > K:
    print("%d days > %d readings; read one a day." % (N, K))
    sys.exit(0)

partition(W, K, N)
```

Graph-Algorithmus [Slide 11]

Erstellen wir einen Graphen, dessen Knoten $v_{k,n}$ jeweils das letzte am Tag n bearbeitete Kapitel k repräsentieren, und dessen Kanten die von einem Tag zum nächsten bearbeiteten Kapitel repräsentieren.

- $V = \{k = 0, ..., K\} \times \{n = 0, ..., N\}$
- E enthält alle gerichteten Kanten von $v_{k,n}$ nach $v_{l,n+1}$, $k < l \leq K$, gewichtet mit $w(v_{k,n}, v_{l,n+1}) = \left(\sum_{i=k+1}^{l} w_i \right)^2 = C_{k+1,l}$.
- Berechne den kürzesten Weg von $v_{0,0}$ nach $v_{K,N}$.

Laufzeit?

$\Theta(KN)$ Knoten, und $\Theta(K^2)$ Kanten pro Tag, also $\Theta(K^2N)$ Kanten, deren Gewichte sich jeweils in konstanter Zeit berechnen lassen. Dijkstra ist also $O(K^2N \log(KN)) = O(K^2N \log K)$ mit einem *binary heap*, $\Theta(K^2N^2)$ mit einer unsortierten Liste (was vorteilhaft ist, wenn die Zahl der Kapitel exponentiell in der Anzahl der Tage ist), oder $O(K^2N + KN \log(KN)) = O(K^2N)$ mit einem Fibonacci heap.

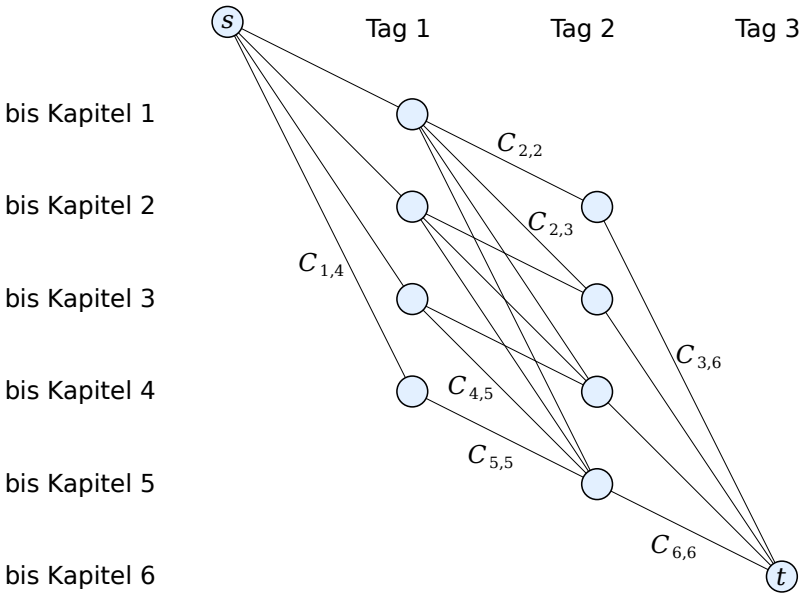
Dies entspricht dem einfachen DP-Algorithmus. Lassen wir die Knoten und Kanten trivial suboptimaler Konfigurationen weg, erhalten wir das Äquivalent zum verbesserten DP-Algorithmus, mit derselben asymptotischen Laufzeit wie diesem.

Platzbedarf?

Im Gegensatz zum DP-Algorithmus repräsentiert der Graph alle möglichen täglichen Lektüren explizit.

Beispiel [Slide 12]

Verteile $K = 6$ Kapitel auf $N = 3$ Tage.



Bibliographie [Slide 13]

Piater, Justus (2009). „Planning Readings: a Comparative Exploration of Basic Algorithms“. In: *Computer Science Education* 19.3, S. 179–192. DOI: 10.1080/08993400903255226. URL: https://iis.uibk.ac.at/public/papers/Piater-2009-CSE.pdf.