Dynamische Programmierung

Weiterführende Literatur:

- *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 3*, Seite 10-12
- Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 230-235 (PDF 248-253)
- *Wikipedia-Artikel „Dynamische Programmierung“*

Dynamische Programmierung ist eine Methode zum algorithmischen Lösen eines Optimierungsproblems durch *Aufteilung in Teilprobleme* und *systematische Speicherung* von Zwischenresultaten.[1]

Aufteilung in Teilprobleme

systematische Speicherung

Dynamische Programmierung *vereint Aspekte* der drei bisher vorgestellten *Algorithmenmuster*. Vom Ansatz der *Greedy*-Algorithmen wird die *Wahl optimaler Teillösungen* übernommen, von *Divide-and-conquer* und *Backtracking* die *rekursive Herangehensweise* basierend auf einem Konfigurationsbaum. Während Divide-and-conquer-Verfahren unabhängige Teilprobleme durch rekursive Aufrufe lösen, werden bei der dynamischen Programmierung abhängige Teilprobleme optimiert gelöst, indem mehrfach auftretende Teilprobleme nur einmal gelöst werden.[2]

vereint Aspekte

Algorithmenmuster

Greedy

Wahl optimaler Teillösungen

Divide-and-conquer

Backtracking

rekursive Herangehensweise

Als Vorteil der dynamischen Programmierung kann eine *Verbesserung der Laufzeit-Effizienz* genannt werden. Es wird jedoch mehr *Speicherplatz* benötigt.[3]

Verbesserung der Laufzeit-Effizienz

Speicherplatz

Das Rucksack-Problem[4]

```
3  import org.bschlangaul.helfer.Konsole;
4
5  /**
6   * Die Lösung des Rucksack-Problems naiv (rekursiv) oder mit Dynamischer
7   * Programmierung. Quelle: <a href=
8   * "https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-
    ↪  10/">geeksforgeeks.org</a>
9   */
10 class Rucksack {
11
12   /**
13    * Helfer-Methode, die größere Zahl zurückgibt.
14    *
15    * @param a Die Zahl a.
16    * @param b Die Zahl b
17    *
18    * @return Die größere Zahl.
19    */
20   static int max(int a, int b) {
21     return (a > b) ? a : b;
22   }
23
24   /**
25    * Lösung des Rucksackproblems mit der Hilfe eines naiven, rekursiven Ansatzes.
```

[1] *Wikipedia-Artikel „Dynamische Programmierung“*.
[2] Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 230 (PDF 248).
[3] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 3*.
[4] Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 231-235.

```
26      *
27      * @param kapazität Die noch verhandene Kapazität des Rucksacks an Gewichten.
28      * @param gewichte  Die Gewichte der einzelnen Gegenstände.
29      * @param werte     Die Werte der einzelnen Gegenstände.
30      * @param n         Die Anzahl der Gegenstände, die sich noch nicht im Rucksack
31      *                  befinden.
32      *
33      * @return Die Summe der Werte der Gegenstände, die in den Rucksack gepackt
34      *         werden konnten.
35      */
36     static int rucksackNaiv(int kapazität, int gewichte[], int werte[], int n) {
37       if (n == 0 || kapazität == 0)
38         return 0;
39
40       // Wenn das Gewicht des n-ten Gegenstand größer als die Kapazität des
         ↪  Rucksacks
41       // ist, dann kann der Gegenstand nicht in die optimale Lösung mit einbezogen
42       // werden.
43       if (gewichte[n - 1] > kapazität)
44         return rucksackNaiv(kapazität, gewichte, werte, n - 1);
45
46       // Gib das Maxium dieser zwei Möglichkeiten zurück:
47       // (1) der n-te Gegenstand passt in den Rucksack.
48       // (2) passt nicht in den Rucksack.
49       else
50         return max(werte[n - 1] + rucksackNaiv(kapazität - gewichte[n - 1],
           ↪  gewichte, werte, n - 1),
51             rucksackNaiv(kapazität, gewichte, werte, n - 1));
52     }
53
54     /**
55      * Lösung des Rucksackproblems mit der Hilfe der Dynamischen Programmierung.
56      *
57      * @param kapazität Die noch verhandene Kapazität des Rucksacks an Gewichten.
58      * @param gewichte  Die Gewichte der einzelnen Gegenstände.
59      * @param werte     Die Werte der einzelnen Gegenstände.
60      * @param n         Die Anzahl der Gegenstände, die sich noch nicht im Rucksack
61      *                  befinden.
62      *
63      * @return Die Summe der Werte der Gegenstände, die in den Rucksack gepackt
64      *         werden konnten.
65      */
66     static int rucksackDP(int kapazität, int gewichte[], int werte[], int n) {
67       int i, w;
68       int k[][] = new int[n + 1][kapazität + 1];
69
70       for (i = 0; i <= n; i++) {
71         for (w = 0; w <= kapazität; w++) {
72           if (i == 0 || w == 0)
73             k[i][w] = 0;
74           else if (gewichte[i - 1] <= w)
75             k[i][w] = max(werte[i - 1] + k[i - 1][w - gewichte[i - 1]], k[i -
               ↪  1][w]);
76           else
77             k[i][w] = k[i - 1][w];
78         }
79       }
80
81       Konsole.zeige2DIntFeld(k);
82
83       return k[n][kapazität];
84     }
```

```
85
86    public static void main(String args[]) {
87      int werte[] = new int[] { 60, 100, 120 };
88      int gewichte[] = new int[] { 10, 20, 30 };
89      int kapazität = 50;
90      int n = werte.length;
91      System.out.println(rucksackNaiv(kapazität, gewichte, werte, n));
92      System.out.println(rucksackDP(kapazität, gewichte, werte, n));
93    }
94  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/muster/rucksack/Rucksack.java`

Literatur

[1] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 3*. `https://www.studon.fau.de/file2566440_download.html`.

[2] Gunter Saake und Kai-Uwe Sattler. *Algorithmen und Datenstrukturen. Eine Einführung in Java*. 2014.

[3] *Wikipedia-Artikel „Dynamische Programmierung“*. `https://de.wikipedia.org/wiki/Dynamische_Programmierung`.