

Beispiellösung zu den Übungen  
**Datenstrukturen und Algorithmen**

SS 2008

Blatt 14

**AUFGABE 1** (6 Punkte):

Betrachten wir das *Problem der chromatischen Zahl*. Gegeben sei ein ungerichteter Graph  $G$ . Jeder Knoten in  $G$  hat maximal  $d$  Nachbarn. Gesucht ist die minimale Anzahl  $\chi(G)$  an Farben, die notwendig ist, um die Knoten des Graphen derart zu färben, dass adjazente Knoten unterschiedliche Farben haben. Diese minimale Anzahl wird die chromatische Zahl von  $G$  genannt. Es soll der Einfachheit halber Farben durch natürliche Zahlen dargestellt werden.

- a) Entwerfen Sie einen Greedy-Algorithmus mit Laufzeit  $\mathcal{O}(|V| + |E|)$  zur Bestimmung einer Färbung mit maximal  $d + 1$  Farben. Arbeiten Sie dabei vor allem den gierigen Charakter Ihrerer Strategie heraus. Hierbei ist *kein Pseudocode* verlangt.
- b) Beweisen Sie, dass der Algorithmus eine korrekte Färbung mit maximal  $d + 1$  Farben liefert.

*Hinweis:* Bisher ist es noch niemanden gelungen, einen Algorithmus zur Bestimmung der chromatischen Zahl anzugeben, welcher Laufzeit  $\mathcal{O}(n^c)$  mit  $n = |V| + |E|$  und konstantem  $c$  hat. Sollte Ihr Algorithmus derart schnell sein, so wird es wohl spezielle Graphen geben, auf denen die chromatische Zahl kleiner als die von Ihrem Algorithmus bestimmte Anzahl ist. (Können Sie einen solchen Graphen für Ihren Algorithmus angeben?)

Lösung:

- a) Der Algorithmus lautet:
  1. Wähle einen beliebigen Knoten  $v$  und setze  $farbe[v] = 1$ . (Beachte: *farbe* und *color* (aus dem Breiten- oder Tiefensuchalgorithmus) haben nichts miteinander zu tun.)
  2. Durchlaufe die Knoten von  $v$  aus mittels Breiten- oder Tiefensuche. In jedem neuen Knoten  $w$  setze  $farbe[w]$  auf die kleinste, von seinen direkten Nachbarn noch nicht verwendete Farbe. Das ist greedy, weil aus der lokalen Sicht eines Knoten der kleinstmögliche Wert genommen wird.
  3. Bestimme anschließend die größte verwendete Farbe und liefere diesen Wert zurück.
- b) Die Überprüfung der Nachbarknoten und die eigene Färbung lässt sich komplett in die Breiten- und Tiefensuche einbetten. Die anschließende Suche in Schritt 3 braucht  $\mathcal{O}(|V|)$  Laufzeit. Insgesamt bleibt die Laufzeit  $\mathcal{O}(|V| + |E|)$ .

**AUFGABE 2** (6 Punkte):

Es ist ein Grubenunglück passiert, bei dem das ganze Schachtsystem einer Mine zusammengestürzt ist. Einige Minenarbeiter konnten sich in einen Hohlraum retten und hoffen jetzt auf schnelle Hilfe. Es muss von einem entfernten Schacht (*Start*) aus eine Verbindung zu den Verschütteten (*Ziel*) gegraben werden, weil unglücklicherweise eine direkte Bohrung von oben nicht möglich ist. Aus alten Unterlagen wissen Sie die Zusammensetzung des umliegenden Erdreichs und können berechnen, wie lange man für eine Grabung durch einen gewissen Bereich benötigen würde. Ihre Berechnungen ergeben folgende Tabelle:

<i>Start</i>	15	3	31	2
41	5	42	4	24
78	49	106	51	87
15	22	13	36	<i>Ziel</i>

Aus Sicherheitsgründen sind nur Grabungen nach unten oder in Richtung der Verschütteten, also in diesem Fall nach rechts, erlaubt.

*Beispiel:* Ein möglicher Weg ist:  $P = \{r, r, u, u, u, r, r\}$ . Dieser Weg würde  $15 + 3 + 42 + 106 + 13 + 36 = 215$  Zeiteinheiten benötigen.

- Formulieren Sie zuerst die rekursive Berechnung der optimal benötigten Zeit. Entwickeln sie daraufhin einen Algorithmus, der nach Eingabe der  $N \times M$ -Matrix mit dynamischer Programmierung sowohl die optimale Zeit als auch den Weg zu den verschütteten Minenarbeitern berechnet.
- Zeigen Sie die Korrektheit Ihres Algorithmus und bestimmen Sie die Laufzeit.

Lösung:

Die Lösung ist sehr simpel. Man erstellt eine zweite  $N \times M$ -Matrix in der für jedes Feld die minimal benötigte Zeit zu diesem Feld berechnet wird. Initial berechnet man die Werte der ersten Spalte und der ersten Zeile. Dies kann man einfach machen, weil es nur einen Weg zu diesen Feldern gibt, nämlich nur „nach unten“ bzw. nur „nach rechts“. Die Werte für die Felder sind jeweils die Summe der Werte der bisher besuchten Felder plus den Wert des eigenen Feldes. Als nächster Schritt wird ab Zeile 2 Zeile für Zeile entsprechenden Werte für die Felder berechnet. Dies macht man, indem man jeweils den Wert des Feldes (diesen entnimmt man aus der Originalmatrix) auf den Wert des oberen Feldes und des linken Feldes addiert, beide Summen werden verglichen und das Minimum von beiden in das aktuelle Feld eingetragen, formal:  $wert(i, j) = \min(wert(i-1, j) + wert(i, j), wert(i, j-1) + wert(i, j))$  (dies ist auch gleichzeitig die rekursive Berechnung der optimalen Zeit). Der berechnete Wert ist korrekt, weil man nur aus den beiden betrachteten Richtungen Richtungen zu diesem Feld kommen kann. Diese Berechnung wird fortgesetzt, bis man zum Zielfeld angekommen ist. Der Pfad kann bestimmt werden, indem man vom Startfeld aus  $(N-1) \cdot (M-1)$  mal die Felder (soweit vorhanden) rechts und links betrachtet und zu dem Feld mit den kleineren Wert geht.

**AUFGABE 3** (6 Punkte):

Betrachten Sie nun das *maximal Rechte Teilsummenproblem*. Dabei ist als Eingabe eine Folge  $A = \langle a_1, a_2, \dots, a_n \rangle$  ganzer Zahlen  $a_i \in \mathbb{Z}$  gegeben. Gesucht ist die maximale Summe einer

zusammenhängenden Teilfolge  $\langle a_l, a_{l+1}, \dots, a_r \rangle$  von A, die genau in  $a_k$  endet, also

$$R(k) = \max \left\{ \sum_{i=l}^k a_i \mid 1 \leq l \leq k \right\}$$

- a) Finden Sie eine rekursive Formulierung für  $R(k)$ .
- b) Geben Sie einen effizienten Algorithmus an, der mit dynamischer Programmierung  $R(k)$  berechnet. Zeigen sie zudem die Korrektheit.

*Hinweis:* Der naive Algorithmus, welcher alle  $\mathcal{O}(n^2)$  möglichen Blöcke jeweils in Zeit  $\mathcal{O}(n)$  aufsummiert und dann das Maximum bestimmt, löst das Problem in Zeit  $\mathcal{O}(n^3)$ . Das Ergebnis Ihrer dynamischen Programmierung muss asymptotisch schneller bestimmt werden.

Lösung:

a)

$$R(k) = \begin{cases} 0 & k = 0 \\ \max \{a_k, a_k + R(k-1)\} & k \geq 1 \end{cases}$$

Beweis per Induktion über  $k$ :

IA:  $R(0)$  trivial, weil es nichts zu summieren gibt.

IS: Fall 1:  $R(k-1) \leq 0$ . Das heißt, dass die größte Teilsumme, die auf  $a_k$  endet, nur die Zahl  $a_k$  enthält.  $R(k) = a_k$  wird korrekt zurückgegeben.

Fall 2:  $R(k-1) > 0$ . Dann bildet  $a_k$  zusammen mit der rechten Randsumme der ersten  $(k-1)$ -ten Elemente  $R(k)$ . Somit wird mit  $a_k + R(k-1)$  das korrekte Ergebnis zurückgeliefert.

b)

RANDSUMME( $k$ )

```

1   R[0] ← 0;
2   for i ← 1 to k do
3       R[i] ← max(a_i, a_i + R[i-1]);
4   return R[k]
```

Die Korrektheit folgt aus dem Beweis zur rekursiven Berechnung.

#### AUFGABE 4 (6 Punkte):

Betrachten Sie als etwas komplexeren Fall das *maximale Teilsummenproblem*. Dabei ist als Eingabe eine Folge  $A = \langle a_1, a_2, \dots, a_n \rangle$  ganzer Zahlen  $a_i \in \mathbb{Z}$  gegeben. Gesucht ist die maximale Summe einer zusammenhängenden Teilfolge  $\langle a_l, a_{l+1}, \dots, a_r \rangle$  von A, also

$$\max \left\{ \sum_{i=l}^r a_i \mid 1 \leq l \leq r \leq n \right\}$$

- a) Bezeichne  $S(k)$  den Wert der maximalen Teilsumme in  $\langle a_1, a_2, \dots, a_k \rangle$ . Finden Sie eine rekursive Formulierung für  $S(k)$ . Verwenden Sie dabei gegebenenfalls  $R$  aus der dritten Aufgabe.
- b) Geben Sie einen effizienten Algorithmus an, der mittels dynamischer Programmierung die maximale Teilsumme von  $A$  berechnet. Zeigen Sie zudem die Korrektheit.

*Hinweis:* Der naive Algorithmus, welcher alle  $\mathcal{O}(n^2)$  möglichen Blöcke jeweils in Zeit  $\mathcal{O}(n)$  aufsummiert und dann das Maximum bestimmt, löst das Problem in Zeit  $\mathcal{O}(n^3)$ . Das Ergebnis Ihrer dynamischen Programmierung sollte asymptotisch schneller bestimmt werden.

Lösung:

a)

$$S(k) = \begin{cases} 0 & k = 0 \\ \max \{S(k-1), R(k)\} & k \geq 1 \end{cases}$$

Beweis per Induktion über  $k$ :

IA trivial

IS Fall 1:  $a_k$  ist nicht in  $S(k)$  enthalten. Das bedeutet, dass die maximale Teilsumme  $\langle a_1, \dots, a_k \rangle$  gleich der maximalen Teilsumme  $\langle a_1, \dots, a_{k-1} \rangle$  ist, also  $S(k) = S(k-1)$ .

Fall 2:  $a_k$  gehört zu  $S(k)$ . Dann ist  $S(k)$  gleich der maximalen rechten Randsumme, die mit  $a_k$  endet, also  $S(k) = R(k)$ .

b)

TEILSUMME( $k$ )

```

1   R[0] ← 0;
2   S[0] ← 0;
3   for i ← 1 to k do
4       R[i] ← max(a_i, a_i + R[i - 1]);
5       S[i] ← max(S[i - 1], R[i]);
6   return S[k]
```

Die Korrektheit folgt aus der rekursiven Berechnung.