

Datenstrukturen und Algorithmen

Cedric Aehi 17-103-235

Nicolas Müller 17-122-094

Datenstrukturen und Algorithmen

Übung 9, Frühling 2018

26. April 2018

Abgabe: Diese Übung muss zu Beginn der Übungsstunde bis spätestens um 16 Uhr 15 am 3. Mai abgegeben werden. Die Abgabe der DA Übungen erfolgt immer in schriftlicher Form auf Papier. Programme müssen zusammen mit der von Ihnen erzeugten Ausgabe abgegeben werden. Drucken Sie wenn möglich platzsparend 2 Seiten auf eine A4-Seite aus. Falls Sie mehrere Blätter abgeben heften Sie diese bitte zusammen (Büroklammer, Bostitch, Mäppchen). *Der gesammte Sourcecode muss ausserdem elektronisch über Ilias abgegeben werden.*

Die Übung sollte vorzugsweise in Zweiergruppen bearbeitet werden, kann aber auch einzeln abgegeben werden. Vergessen Sie nicht, Ihren Namen und Ihre Matrikelnummer auf Ihrer Abgabe zu vermerken. Jede Übungsserie gibt 10 Punkte. Im Durchschnitt müssen Sie 7 von 10 Punkten erreichen, um die Testatbedingungen zu erfüllen.

Theoretische Aufgaben

1. In der Vorlesung auf der Folie 14 wurde behauptet, dass ein rekursiver Algorithmus zur Berechnung der optimalen Zerteilung von Eisenstangen exponentielle Laufzeit hat. Die Anzahl Aufrufe der CUT-ROD Funktion ist durch

$$T(0) = 1$$
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

beschrieben. Zeige, dass daraus $T(n) = 2^n$ folgt.

Hinweise: 1) Induktionsbeweis mit der Substitutionsmethode, 2) Geometrische Reihe.

1 Punkt

2. Die Fibonacci-Zahlen sind durch die Rekursionsgleichung

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}$$

definiert. Geben Sie ein dynamisches Programm an, das in Zeit $O(n)$ die n -te Fibonacci-Zahl berechnet. Zeichnen sie den Teilproblem-Graph (siehe Vorbesprechung). Wieviele Knoten und Kanten enthält der Graph? **1 Punkt**

3. Bestimmen Sie eine längste gemeinsame Teilsequenz (LCS) von $\langle 1, 0, 1, 1, 0, 1, 0, 0 \rangle$ und $\langle 0, 1, 0, 0, 1, 1, 0, 1 \rangle$. **1 Punkt**
4. Ein Taxifahrer in New York hat sich in ein gefährliches Quartier verfahren. Er befindet sich in der linken oberen Ecke des Strassengitters, das unten dargestellt ist. Das Ziel des Taxifahrers ist es, in die untere rechte Ecke zu gelangen, wo er das gefährliche Quartier verlässt. Jeder Strassenabschnitt hat ein gewisses Risiko für einen Überfall, das mit ganzen Zahlen angegeben ist. Alle Straßen sind Einbahnstraßen, welche von links nach rechts oder von oben nach unten verlaufen. Gesucht ist ein Weg, der das Gesamtrisiko für einen Überfall, also die Summe der Risiken aller passierten Strassenabschnitte, minimiert.

Nehmen Sie an, die Risiken seien in einem zweidimensionalen Feld r gespeichert. Das heißt $r[i, j]$ ist das Risiko für den Strassenabschnitt in Zeile i und Spalte j des Strassengitters. Die Startposition ist also $i = 1, j = 1$, und die Zielposition $i = 5, j = 7$.

T	1	9	2	3	5	7
8	5	3	15	2	7	9
7	10	4	15	5	6	6
12	3	5	9	15	14	7
10	2	3	1	7	4	5

- Entwerfen Sie einen Algorithmus nach dem Muster der dynamischen Programmierung, der das Gesamtrisiko eines sichersten Weges berechnet. Beschreiben Sie den Algorithmus mit Pseudocode und skizzieren Sie die Lösungstabelle. Geben Sie die Laufzeit Ihres Algorithmus an. **1 Punkt**
- Beschreiben Sie in Worten, wie durch Rückverfolgung in der Lösungstabelle, die Ihr Algorithmus aus a) berechnet, die traversierten Zellen eines sichersten Weges gefunden werden können. Zeichnen Sie den Weg in der Lösungstabelle ein. Geben Sie die Laufzeit für die Rückverfolgung an. **1 Punkt**

Praktische Aufgaben

In dieser Aufgabe verwenden Sie dynamische Programmierung, um die Grösse von Rasterbildern zu verändern. Die Grösse von Bildern kann am einfachsten durch lineare Skalierung oder durch Abschneiden (cropping) verändert werden. Skalierung führt aber zu Verzerrungen, und Abschneiden kann zum Verlust von wichtigen Bildteilen führen. Sie werden in dieser Übung ein Verfahren implementieren, das versucht, den Inhalt des Bildes so wenig wie möglich zu verändern. Der Algorithmus wurde unter der Bezeichnung *Seam Carving* bekannt¹.

Die Idee von Seam Carving ist, im Bild Schritt für Schritt eine Pixel breite Nähre (*seams*) zu entfernen, bis die gewünschte Bildgrösse erreicht ist. Eine Naht ist entweder *vertikal*, um die Breite des Bildes zu verändern, oder *horizontal*, um die Höhe zu verändern. Die Idee ist nun, jeweils die “beste” Naht zu bestimmen, die möglichst unauffällig aus dem Bild entfernt werden kann. Dies ist in der Abbildung 1 illustriert.

¹http://en.wikipedia.org/wiki/Seam_carving



Abbildung 1: Beispiel für Seam Carving mit einer vertikalen Naht. Links: Eingabebild. Mitte: Visualisierung der optimalen vertikalen Naht. Rechts: Ausgabe nach sukzessiver Entfernung von 100 Nähten.

Um die Kosten (=“Qualitäts/Informationsverlust”) zu bestimmen, die durch das entfernen einer Naht anfallen, wird zuerst eine Energiefunktion $e(x, y)$ für einzelne Pixel (x, y) definiert. Diese Energiefunktion beschreibt die Kosten die entstehen, wenn der Pixel aus dem Bild entfernt wird. Intuitiv sollen die Kosten hoch sein, wenn sich das Bild an diesem Pixel stark ändert. In dem Fall führt das Entfernen des Pixels zu starken Kanten. Deshalb macht es Sinn, die Funktion e über die Ableitung des Bildes zu berechnen. Wir stellen eine Implementation dieser Funktion zur Verfügung (die Methode *energy*). Die Kosten für eine Naht sind nun einfach die Summe der Kosten über alle Pixel in der Naht.

In dieser Übung beschränken wir uns darauf, optimale *vertikale* Nähte zu finden. Der Algorithmus um die optimale Naht zu finden basiert auf dynamischer Programmierung und läuft wie folgt ab: Wir berechnen zuerst eine Tabelle M , welche für jeden Pixel die Kosten der optimalen Naht enthält, die durch diesen Pixel geht. Im zweiten Schritt wird aus der Tabelle die Naht selbst rekonstruiert. Im letzten Schritt wird das Ausgabebild ohne die Pixel der Naht erzeugt. Die ersten zwei Schritte sollen Sie implementieren, der dritte ist bereits im auf Ilias bereitgestellten Code vorhanden (die Methode *removeSeam*).

Um das Berechnen der Kosten zu vereinfachen definieren wir eine vertikale Naht als eine Reihe von Pixeln, je ein Pixel pro Bildzeile, welche von Zeile zu Zeile um *höchstens einen* Pixel horizontal verschoben sind. Damit lassen sich die Kosten $M(x, y)$ für die optimale Naht durch Pixel (x, y) leicht durch folgende Rekursionsformel bestimmen:

$$M(x, y) = e(x, y) + \min(M(x - 1, y - 1), M(x, y - 1), M(x + 1, y - 1)).$$

Die Tabelle wird in der Zeile $y = 0$ mit den Werten $e(x, 0)$ initialisiert.

1. Implementieren Sie im Java Code die Methode `computeCosts`, mit der die Tabelle M berechnet wird. *Hinweis: Initialisieren Sie zuerst die Zeile $y = 0$, und berechnen Sie dann die Tabelle Zeile für Zeile indem Sie bei $y = 1$ anfangen.* **2 Punkte**

2. Implementieren Sie im Java Code die Methode `computeSeam`, mit der aus der Tabelle M eine optimale Naht rekonstruiert wird. *Hinweis: Finden Sie in der Tabelle M zuerst denjenigen Pixel auf der untersten Zeile mit den kleinsten Kosten. Dies ist der Endpunkt der Naht. Verfolgen Sie dann die Naht zurück zur Zeile $y = 0$.* **2 Punkte**
3. Demonstrieren Sie Ihre Implementation mit zwei bis drei Beispielbildern. **1 Punkt**

Vergessen Sie nicht Ihren Sourcecode innerhalb der Deadline über die Ilias Aufgabenseite einzureichen.

1. In der Vorlesung auf der Folie 14 wurde behauptet, dass ein rekursiver Algorithmus zur Berechnung der optimalen Zerteilung von Eisenstangen exponentielle Laufzeit hat. Die Anzahl Aufrufe der CUT-ROD Funktion ist durch

$$T(0) = 1$$

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

beschrieben. Zeige, dass daraus $T(n) = 2^n$ folgt.

Hinweise: 1) Induktionsbeweis mit der Substitutionsmethode, 2) Geometrische Reihe.

1 Punkt

Induktionsverankerung: $T(0) = 1$ ✓

Induktion behauptung (zu zeigen): $T(n) = 2^n$

Induktions schritt: $n \rightsquigarrow n+1$

Annahme: $T(k) = 2^k$ für $k \leq n$ $T(n+1) = 2^{n+1}$

Es gilt:

$$\begin{aligned} T(n+1) &= 1 + \sum_{k=0}^n T(k) \\ &= 1 + \sum_{k=0}^n 2^k \\ &= 1 + \underbrace{\frac{2^{n+1} - 1}{2 - 1}}_{\text{geometrische Reihe}} = 2^{n+1} \quad \square. \end{aligned}$$

2. Die Fibonacci-Zahlen sind durch die Rekursionsgleichung

$$F_0 = 0$$

$$F_1 = 1$$

$$F_k = F_{k-1} + F_{k-2}$$

definiert. Geben Sie ein dynamisches Programm an, das in Zeit $O(n)$ die n -te Fibonacci-Zahl berechnet. Zeichnen sie den Teilproblem-Graph (siehe Vorbesprechung). Wieviele Knoten und Kanten enthält der Graph? **1 Punkt**

$\text{fib}(n)$

new Array $f[n+2]$;

$f[0] = 0;$

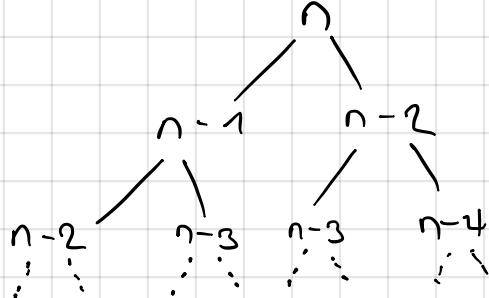
$f[1] = 1;$

for ($i = 2$; $i \leq n$; $i++$)

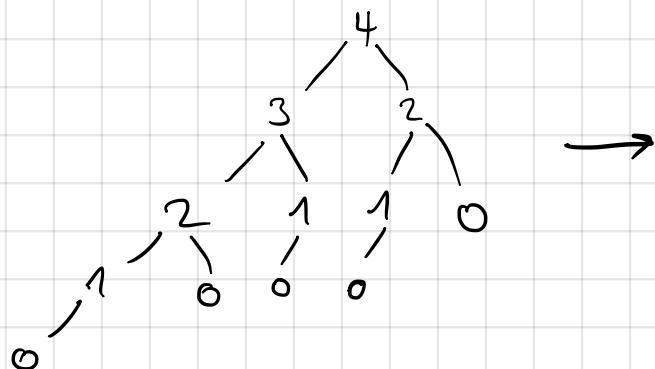
$f[i] = f[i-2] + f[i-1];$

return $f[n];$

Graph:



für $n=4$



5 Knoten

7 Kanten

3. Bestimmen Sie eine längste gemeinsame Teilsequenz (LCS) von $\langle 1, 0, 1, 1, 0, 1, 0, 0 \rangle$ und $\langle 0, 1, 0, 0, 1, 1, 0, 1 \rangle$. **1 Punkt**

Bottom-up Strategie:

4. Ein Taxifahrer in New York hat sich in ein gefährliches Quartier verfahren. Er befindet sich in der linken oberen Ecke des Strassengitters, das unten dargestellt ist. Das Ziel des Taxifahrers ist es, in die untere rechte Ecke zu gelangen, wo er das gefährliche Quartier verlässt. Jeder Strassenabschnitt hat ein gewisses Risiko für einen Überfall, das mit ganzen Zahlen angegeben ist. Alle Straßen sind Einbahnstraßen, welche von links nach rechts oder von oben nach unten verlaufen. Gesucht ist ein Weg, der das Gesamtrisiko für einen Überfall, also die Summe der Risiken aller passierten Straßenabschnitte, minimiert.

Nehmen Sie an, die Risiken seien in einem zweidimensionalen Feld r gespeichert. Das heißt $r[i, j]$ ist das Risiko für den Straßenabschnitt in Zeile i und Spalte j des Strassengitters. Die Startposition ist also $i = 1, j = 1$, und die Zielposition $i = 5, j = 7$.

T	1	9	2	3	5	7
8	5	3	15	2	7	9
7	10	4	15	5	6	6
12	3	5	9	15	14	7
10	2	3	1	7	4	5

- a) Entwerfen Sie einen Algorithmus nach dem Muster der dynamischen Programmierung, der das Gesamtrisiko eines sichersten Weges berechnet. Beschreiben Sie den Algorithmus mit Pseudocode und skizzieren Sie die Lösungstabelle. Geben Sie die Laufzeit Ihres Algorithmus an. **1 Punkt**
- b) Beschreiben Sie in Worten, wie durch Rückverfolgung in der Lösungstabelle, die Ihr Algorithmus aus a) berechnet, die traversierten Zellen eines sichersten Weges gefunden werden können. Zeichnen Sie den Weg in der Lösungstabelle ein. Geben Sie die Laufzeit für die Rückverfolgung an. **1 Punkt**

a) `int minCost (cost[][] , m , n) // m,n states which field you want to go to`
`new Array tc[m][n];`
`tc[1][1] = cost[1][1];`

`for i=2 to m`

`tc[i][1] = tc[i-1][1] + cost[i][1];`

`for j=2 to n`

`tc[1][j] = tc[1][j-1] + cost[1][j];`

`for i=2 to m`

`for j=2 to n`

`tc[i][j] = min(tc[i-1][j-1], tc[i-1][j],`
`tc[i][j-1]) + cost[i][j];`

`return tc[n][n];`

T	0	1	9	10	2	12	3	15	5	20	7	27	
8	8	5	5	3	4	15	13	2	14	7	21	9	29
7	15	10	11	4	8	15	19	5	19	6	20	6	26
12	27	3	18	5	13	9	17	15	32	14	33	7	27
10	37	2	20	3	16	1	17	7	24	4	28	5	32

$$tc = \underline{32}$$

zu a) Rot ist die Lösungstabelle / Laufzeit: $O(mn)$

zu b) Man startet bei $tc[m][n]$ und vergleicht

$tc[i-1][j-1]$, $tc[i-1][j]$ und $tc[i][j-1]$ und wählt den kleinsten Wert. Dies macht man bis zu $[1][1]$, somit erhält man den Weg.

(Grün ist der Lösungsweg).

Laufzeit: $O(m+n)$

```

1 import java.awt.*;
2 import java.awt.image.*;
3 import java.io.File;
4 import java.util.Arrays;
5
6 import javax.imageio.ImageIO;
7
8 public class SeamCarving {
9
10    /**
11     * Computes a table that, for each pixel, stores the smallest cost for
12     * a vertical seam that goes through that pixel. Uses dynamic programming
13     * by stepping through all rows from top to bottom of the image. The cost
14     * for introducing a seam at each pixel is determined using the energy()
15     * method below.
16     *
17     * @param img the input image
18     * @return the table storing the costs
19     */
20    public static float[][] computeCosts(BufferedImage img)
21    {
22        int width = img.getWidth();
23        int height = img.getHeight();
24        float[][] costTable = new float[width][height];
25        for(int x = 0; x < width; x++){
26            for(int y = 0; y < height; y++)
27                costTable[x][y] = energy(img, x, y);
28        }
29        return costTable;
30    }
31
32    /**
33     * Reconstructs a vertical seam from the cost table. A vertical seam is stored
34     * as an array where element y in the array stores an index seam[y], which
35     * indicates
36     * that in row y the seam goes through column seam[y].
37     *
38     * @param costs the cost table
39     * @param width of the cost table
40     * @param height of the cost table
41     * @return the seam
42     */
43    public static int[] computeSeam(float[][] costs, int width, int height) {
44
45        int[] seam = new int[height];
46
47        //Get starting X coordinate
48        float tmpCost = 0;
49        int startX = 0;
50        for (int i = 0; i < width; i++) {
51            if (costs[i][0] > tmpCost) {
52                startX = i;
53                tmpCost = costs[i][0];
54            }
55        }
56        seam[0] = startX;
57
58        for(int i = 1; i < height-1; i++){
59            seam[i] = getLowestUp(seam[i-1], i-1, costs);
60        }
61        return seam;
62    }
63
64    public static int getLowestUp(int x, int y, float[][] map){
65        float[] costs = new float[3];
66
67        if(x > 0)
68            costs[0] = map[x-1][y+1];
69        else
70            costs[0] = Float.POSITIVE_INFINITY;
71        if(x < map.length-1)
72            costs[1] = map[x][y+1];
73        else
74            costs[1] = Float.POSITIVE_INFINITY;
75        if(x < map.length-1)

```

```

76         costs[2] = map[x+1][y+1];
77     else
78         costs[2] = Float.POSITIVE_INFINITY;
79
80     if(costs[0] < costs[1] && costs[0] < costs[2])
81         return x-1;
82     if(costs[1] < costs[0] && costs[1] < costs[2])
83         return x;
84     return x+1;
85 }
86
87
88 /**
89 * Removes a vertical seam from the image. The seam is an array that stores
90 * for each row y in the image the index of the column where the seam lies.
91 * The resulting image after removing the seam has one column less than
92 * the original (its width is reduced by one).
93 *
94 * @param img the input image
95 * @param seam the seam to be removed
96 * @return the new image
97 */
98 public static BufferedImage removeSeam(BufferedImage img, int[] seam)
99 {
100     int width, height;
101     width = img.getWidth();
102     height = img.getHeight();
103
104     // The width of the new image is reduced by one
105     BufferedImage newImg = new BufferedImage(width-1, height, BufferedImage.
TYPE_INT_RGB);
106
107     // For all rows in the image
108     for(int y=0; y<height; y++)
109     {
110         // Copy columns up to seam
111         for(int x=0; x<seam[y]; x++)
112         {
113             newImg.setRGB(x, y, img.getRGB(x,y));
114         }
115         // Skip seam and copy the rest of the columns
116         for(int x=seam[y]; x<width-1; x++)
117         {
118             newImg.setRGB(x, y, img.getRGB(x+1,y));
119         }
120     }
121     return newImg;
122 }
123
124 /**
125 * Computes the energy of a pixel in the image. This energy is used as the cost
126 * for introducing a seam at this pixel. The energy here approximates the sum
127 * of the absolute values of the first derivatives of the image in x and y
direction.
128 *
129 * @param img the input image
130 * @param x x-coordinate of the pixel
131 * @param y y-coordinate of the pixel
132 * @return energy of the pixel
133 */
134 public static float energy(BufferedImage img, int x, int y)
135 {
136     int width = img.getWidth();
137     int height = img.getHeight();
138
139     if(x<0 || x>=width || y<0 || y>=height)
140         return 0.f;
141
142     float c0[],c1[];
143     c0 = new float[3];
144     c1 = new float[3];
145     float didx = 0.f;
146     float didy = 0.f;
147
148     if(x+1<width)
149     {

```

```

150         c0[0] = (float)(img.getRGB(x, y) & 0xFF);
151         c0[1] = (float)((img.getRGB(x, y) >> 8) & 0xFF);
152         c0[2] = (float)((img.getRGB(x, y) >> 16) & 0xFF);
153
154         c1[0] = (float)(img.getRGB(x+1, y) & 0xFF);
155         c1[1] = (float)((img.getRGB(x+1, y) >> 8) & 0xFF);
156         c1[2] = (float)((img.getRGB(x+1, y) >> 16) & 0xFF);
157     } else
158     {
159         c0[0] = (float)(img.getRGB(x-1, y) & 0xFF);
160         c0[1] = (float)((img.getRGB(x-1, y) >> 8) & 0xFF);
161         c0[2] = (float)((img.getRGB(x-1, y) >> 16) & 0xFF);
162
163         c1[0] = (float)(img.getRGB(x, y) & 0xFF);
164         c1[1] = (float)((img.getRGB(x, y) >> 8) & 0xFF);
165         c1[2] = (float)((img.getRGB(x, y) >> 16) & 0xFF);
166     }
167
168     for(int i=0; i<3; i++)
169     {
170         didx += (float)((c1[i]-c0[i])*(c1[i]-c0[i]));
171     }
172     didx = (float)Math.sqrt(didx);
173
174     if(y+1<height)
175     {
176         c0[0] = (float)(img.getRGB(x, y) & 0xFF);
177         c0[1] = (float)((img.getRGB(x, y) >> 8) & 0xFF);
178         c0[2] = (float)((img.getRGB(x, y) >> 16) & 0xFF);
179
180         c1[0] = (float)(img.getRGB(x, y+1) & 0xFF);
181         c1[1] = (float)((img.getRGB(x, y+1) >> 8) & 0xFF);
182         c1[2] = (float)((img.getRGB(x, y+1) >> 16) & 0xFF);
183     } else
184     {
185         c0[0] = (float)(img.getRGB(x, y-1) & 0xFF);
186         c0[1] = (float)((img.getRGB(x, y-1) >> 8) & 0xFF);
187         c0[2] = (float)((img.getRGB(x, y-1) >> 16) & 0xFF);
188
189         c1[0] = (float)(img.getRGB(x, y) & 0xFF);
190         c1[1] = (float)((img.getRGB(x, y) >> 8) & 0xFF);
191         c1[2] = (float)((img.getRGB(x, y) >> 16) & 0xFF);
192     }
193
194     for(int i=0; i<3; i++)
195     {
196         didy += (float)((c1[i]-c0[i])*(c1[i]-c0[i]));
197     }
198     didy = (float)Math.sqrt(didy);
199
200     return didx+didy;
201 }
202
203 public static void main(String[] args) {
204
205     int width;
206     int height;
207     BufferedImage img;
208     float costs[][][];
209     int seam[];
210
211     try {
212         img = ImageIO.read(new File("High_Sierra.png"));
213     } catch(Exception e)
214     {
215         System.out.printf("Could not read image file!\n");
216         return;
217     }
218
219     // Try different n between 1 and 100!
220     for(int n=0; n<100; n++)
221     {
222         width = img.getWidth();
223         height = img.getHeight();
224
225         // Compute costs for seams.

```

```
226         costs = computeCosts(img);
227         // Extract seam with lowest cost.
228         seam = computeSeam(costs, width, height);
229         // Remove the seam from the image.
230         img = removeSeam(img, seam);
231     }
232
233     try {
234         ImageIO.write(img, "bmp", new File("processed.bmp"));
235     } catch(Exception e)
236     {
237         System.out.printf("Could not write image file!\n");
238         return;
239     }
240
241 }
242 }
243
```

Original:



Bearbeitet:

