MTD129 Digitale Medientechnik 2

Übung 6 – Run-Length Codierung von Binärbildern

W. Burger

2017-04-27

Lernziel dieser Übung: Praktische Umsetzung des Run-Length Codings anhand von Binärbildern (als Beispiel für einfache verlustfreie Kompression bei Bildern).

Installation/Setup

Wir verwenden für diese (und die nächste) Übung **ImageJ** (IJ), eine Java-basierte Umgebung zur Bildverarbeitung, die sich sehr leicht durch eigene Funktionen (Plugins) erweitern lässt.[1] Für dieser Übung legen wir ein **neues** Eclipse-Projekt an, das bereits passend vorkonfiguriert ist:

1. Expandiere `DMT2-Eclipse-IJ.zip` an geeigneter Stelle. Diese Datei enthält ein neues (vollständiges) Eclipse-Projekt.
2. Starte Eclipse und lege mit File » New » Java Project ein neues Java-Projekt an (mit ☐ **Use default location** NICHT selektiert).
3. Wähle den Ordner `DMT2-Eclipse-IJ` als Quellverzeichnis (Location) für das neue Projekt und drücke Finish.
4. Starte die `main`-Methode der Klasse `src/StartImageJ`. ImageJ sollte nun mit einem eigenen GUI laufen; mache Dich damit vertraut.

Alternativ kann man ImageJ auch **direkt starten**. In diesem Fall können Plugin-Files in Eclipse parallel editiert und ausgeführt werden, ohne dass man ImageJ jeweils neu starten muss (es kann jeweils nur *eine* Instanz von IJ laufen).

- **Windows:** `ImageJ.exe` ausführen (ImageJ konfiguiert sich dann weitgehend selbst);
- **MacOs:** einfach `ij.jar` als Java-Applikation ausführen.

[1] Umfassende Informationen zu ImageJ (Anleitungen, Beispiele) findet man hier:
Homepage: `http://rsb.info.nih.gov/ij/`
API Docs: `http://rsb.info.nih.gov/ij/developer/api/`

ImageJ Plugins: Die Datei `Demo_Plugin.java` definiert ein einfaches ImageJ-Plugin, das ein 8-Bit Grauwertbild invertiert. Um das Plugin unter ImageJ ausführen zu können, muss die erzeugte Binärdatei (`Demo_Plugin.class`) im lokalen Unterverzeichnis `plugins/` liegen (wie im Eclipse-Projekt bereits eingestellt). Dann kann das Plugin aus ImageJ über das Menü[2]

Plugins » Demo Plugin

direkt ausgeführt werden (sofern ein passendes Grauwertbild geöffnet ist). Weitere Details dazu werden in der Übung erklärt.

ImageJ Plugin-Beispiel: Hier ist ein Beispiel für ein vollständiges ImageJ-Plugin, das ein **8-Bit Grauwertbild** (`DOES_8G`) invertiert (file `Demo_Plugin.java`):

```
import ij.ImagePlus;
import ij.io.LogStream;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;

public class Demo_Plugin implements PlugInFilter {

  public int setup(String args, ImagePlus image) {
    LogStream.redirectSystem();
    return DOES_8G;
  }

  public void run(ImageProcessor ip) {
    System.out.println("Hello Hagenberg!");
    int w = ip.getWidth();
    int h = ip.getHeight();

    for (int x = 0; x < w; x++) {      // from left to right
      for (int y = 0; y < h; y++) {    // from top to bottom
        int val = ip.getPixel(x, y);   // get pixel value at (x,y)
        ip.putPixel(x, y, 255 - val); // set pixel value at (x,y)
      }
    }
  }
}
```

Wie man sieht, ist das alles extrem einfach ...

[2] Underscores im Java-Namen werden im Menü durch Leerzeichen ersetzt.

Run-Length Coding

Das Ziel ist die Realisierung eines **Run-Length Encoders** und **Decoders** für **Binärbilder** (s. Beispiel in Abb. 1). Dabei sollen folgende Annahmen gelten:

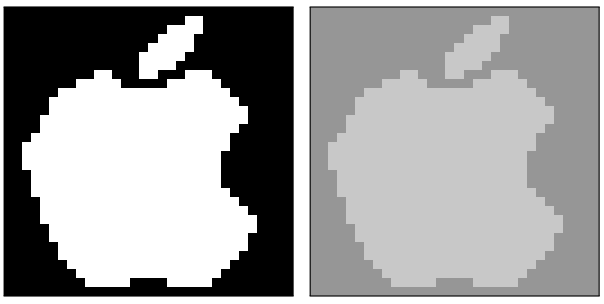
1. Die verwendeten Binärbilder sind 8-Bit Grauwertbilder, bei denen die 0/1-Werte durch die Pixelwerte 0 (schwarz) und 255 (weiß) repräsentiert sind.
2. Die Run-Length Codierung erfolgt zeilenweise von links oben nach rechts unten.
3. Die Codierung besteht aus einer Folge von Lauflängen für Runs mit abwechselnden 0/1-Werten. Zum Beispiel ergibt sich für die binäre Bildzeile (links) die zugehörige Codefolge

$$[\texttt{0001011111...000011}] \longrightarrow (3, 1, 1, 5, \ldots, 4, 2).$$

4. Der *erste* Run einer Zeile repräsentiert immer den Pixelwert 0. Beginnt die Zeile mit 1, so wird am Anfang ein leerer 0-Run (der Länge 0) eingefügt, d. h.

$$[\texttt{1110100000...111100}] \longrightarrow (0, 3, 1, 1, 5, \ldots, 4, 2).$$

5. Die einzelnen Run-Codes sollen mit 8-Bit Zeichen, also mit Werten von $0, \ldots, 255$ codiert werden.
6. Der Wert 255 wird als spezieller Marker-Code reserviert, der am Ende jeder Zeile eingefügt wird (damit man bei der Decodierung die Bildgröße rekonstruieren kann).
7. Somit ist die maximal mögliche Run-Länge 254. Runs mit mehr als 254 Elementen müssen in mehrere Runs zerlegt werden, zwischen die jeweils Dummy-Runs der Länge 0 eingefügt werden.



(a) (b)

Abbildung 1: Originalbild (`apple-32x32i.png`) der Größe 32×32 (a), in dem *Schwarz* dem Pixelwert 0 und *Weiß* dem Pixelwert 255 entspricht. Testbild mit dunkelgrau/hellgrau markierten Runs (b).

Aufg. 6.1 Auffinden der einzelnen Runs

A. Erstelle zunächst ein ImageJ-Plugin, das **die einzelnen Runs in einem Binärbild identifiziert und markiert**. Erstelle dazu eine Methode

```
int findRun(ImageProcessor ip, int x_start, int y),
```

die jenen Run identifiziert, der an der horiz. Position `x_start` in Zeile y des Bilds `ip` beginnt und die horiz. Endposition (`x_end`) des Runs zurückgibt. Verwende diese Methode in der run()-Methode des Plugins etwa so:

```
public void run(ImageProcessor ip) {
  int w = ip.getWidth();
  int h = ip.getHeight();

  for (int y = 0; y < h; y++) {
    int x = 0;
    while (x < w) {
        int x_end = findRun(ip, x, y);
        x = x_end + 1;
  }
}
```

Gib für jede Zeile die Zeilennummer und die Länge der gefundenen Runs an, wie beispielsweise in Abb. 2 gezeigt.

```
0: 32              | 8: 8 5 5 6 8     | 16: 2 22 8     | 24: 5 23 4
1: 20 2 10         | 9: 6 19 7        | 17: 2 22 8     | 25: 5 22 5
2: 18 4 10         | 10: 5 21 6       | 18: 3 21 8     | 26: 6 21 5
3: 17 4 11         | 11: 5 22 5       | 19: 3 21 8     | 27: 6 20 6
4: 16 5 11         | 12: 4 23 5       | 20: 3 22 7     | 28: 7 18 7
5: 15 5 12         | 13: 4 22 6       | 21: 4 22 6     | 29: 8 16 8
6: 15 4 13         | 14: 3 22 7       | 22: 4 23 5     | 30: 9 5 4 5 9
7: 10 2 3 2 3 3 9  | 15: 2 23 7       | 23: 4 24 4     | 31: 32
```

Abbildung 2: Auflistung der Runs für `apple-32x32i.png` in Abb. 1 b.

B. Ergänze das Plugin durch eine weitere Methode

```
void markRun(ImageProcessor ip, int x_start, int x_end, int y),
```

die (zur Kontrolle) die einzelnen Runs im Bild `ip` mit einem entsprechenden Grauwert (**100** für 0-Pixel bzw. **200** für 1-Pixel) markiert, wie in Abb. 1 b gezeigt. Wende dieses Plugin auf die weiteren Testbilder an und überprüfe die Ergebnisse.

Aufg. 6.2 Run-Length Encoder

A. Als Nächstes wenden wir uns der eigentlichen **Codierung der einzelnen Runs** zu. Erstelle dazu eine Methode

```
List<Integer> encodeRun(int length),
```

die eine Liste von `Integer`-Elementen[3] liefern soll, die jeweils **einen Run** repräsentieren. Wenn `length` ≤ 254,[4] so soll diese Liste nur *ein* Element mit dem Wert `length` enthalten. Andernfalls ist das Ergebnis eine Folge **mehrerer Runs** (jeweils mit max. Länge 254), zwischen die **Dummy-Runs der Länge 0** eingefügt sind.

B. Konstruiere damit eine weitere Methode, die das **gesamte Bild** `ip` in **eine** Liste von Run-Codes codiert:

```
List<Integer> encode(ImageProcessor ip),
```

Dabei sollte man nicht vergessen,

1. am Anfang der Zeile einen Run der Länge 0 einzufügen, falls die Zeile mit einem 1-Pixel beginnt und
2. am Ende jeder Zeile den **Abschlusscode 255** einzufügen.

Gib das Ergebnis für ausgewählte Bilder als Zahlenfolgen aus. Verwende dafür eine Methode ähnlich zu folgender:

```
void printCode(List<Integer> code) {
  for (int c : code) {
    System.out.print(c + " ");
  }
  System.out.println();
}
```

Beispiele:

`apple-32x32i.png` (32×32):
32 255 20 2 10 255 18 4 10 255 17 4 11 255 16 5 11 255 15 5 12 255 15 ...
... 5 255 6 21 5 255 6 20 6 255 7 18 7 255 8 16 8 255 9 5 4 5 9 255 32 255

`disney.png` (577×274):
0 254 0 254 0 69 255 0 254 0 254 0 69 255 0 254 0 254 0 69 255 0 254 0 ...
... 0 69 255 0 254 0 254 0 69 255 0 254 0 254 0 69 255 0 254 0 254 0 69 255

C. Ermittle die **Länge der erzeugten Run-Length Codes** (Anzahl der Bytes) und vergleiche sie mit der zugehörigen Bildgröße. Unter der Annahme, dass das unkomprimierte Speicherung eines Binärbilds nur 1 Bit pro Pixel erfordert: wie hoch ist der **Kompressionsfaktor** für die einzelnen Testbilder?

[3] Objekte vom Typ `Integer` repräsentieren `int`-Werte und können u. a. auch in Listen eingefügt werden.

[4] Den **Code 255** reservieren wir als Marker für das Zeilenende.

Aufg. 6.3 Run-Length Decoder

A. Normalerweise würde man nun die RL-Bytefolge in eine Datei schreiben, aus der man später das ursprüngliche Binärbild jederzeit rekonstruieren kann. Wir überspringen diesen Schritt und begnügen uns damit, das Bild direkt wieder aus der generierten Codefolge zu rekonstruieren. Erstelle dazu eine neue Methode

```
ImageProcessor decode(List<Integer> code),
```

die aus der Codefolge `code` das enthaltene Bild rekonstruiert. Dies erfordert folgende Schritte:

1. Bestimme aus `code` zunächst die Länge (`w`) und die Anzahl (`h`) der Bildzeilen (durch Auffinden/Zählen der Zeilenmarker **255**).
2. Erzeuge ein neues Bild durch[5]
   ```
   ImageProcessor ip2 = new ByteProcessor(w, h);
   ```
3. Befülle das Bild mit der aus `code` rekonstruierten Abfolge von 0/255-Werten.
4. Gib das fertige Bild aus der Methode zurück.

B. Verwende die Methode `decode()` in einem erweiterten Plugin, in dem Folgendes passiert:

1. Codiere das Input-Bild mit `encode(ip)` zu `code`.
2. Rekonstruiere mit `decode(code)` das Bild zu `ip2`.
3. Bringe das rekonstruierte Bild mit folgender Anweisung zur Anzeige:
   ```
   new ImagePlus("Reconstructed image", ip2).show();
   ```
4. Überprüfe anhand aller vorbereiteten Testbilder, ob das Ergebnis jeweils identisch zum Originalbild ist.
5. Wende dieses Plugin auch auf größere Binärbilder an (s. Abb 3).

Aufg. 6.4 Bonusaufgabe: Huffman-Codierung des Run-Length Codes

Wende die Huffman-Codierung (aus der letzten Übung) auf den erzeugten RL-Code an. Das Alphabet besteht in diesem Fall aus den 8-Bit Zeichen $0, \ldots, 255$. Berechne die Gesamtzahl der generierten Code-Bits (!) für das Binärbild in Abb. 3 (`rhino.png`). Welcher Kompressionsfaktor ergibt sich insgesamt (RL + Huffman-Codierung)?

[5] Erfordert `import ij.process.ByteProcessor` (8-Bit Grauwertbild).

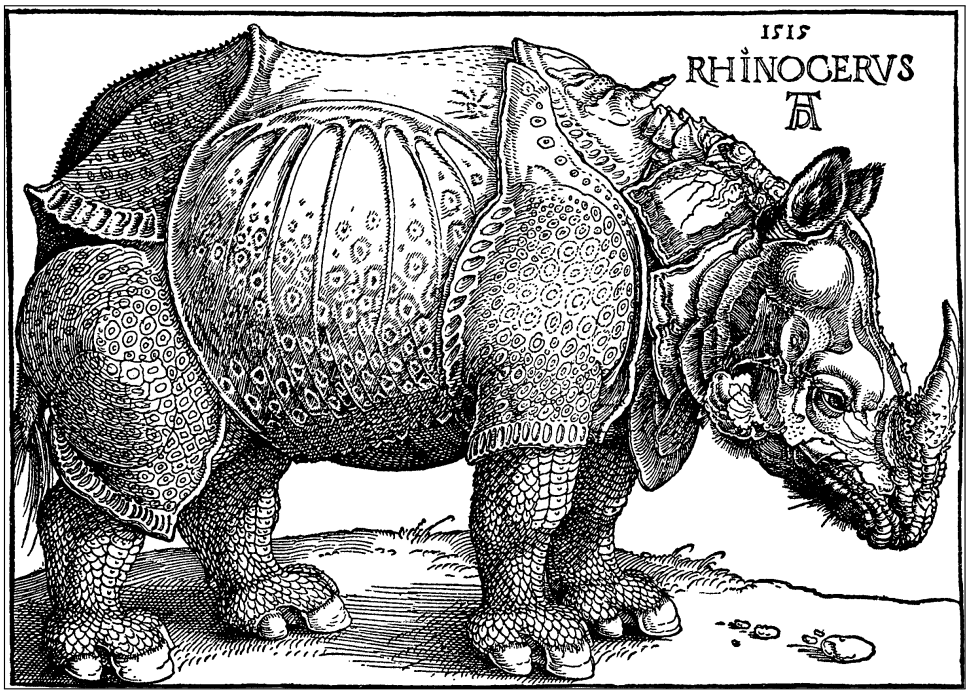


Abbildung 3: Binärbild `rhino.png` der Größe 2427×1725 (4.186.575 Pixel). Der zugehörige RL-Code nach obigem Schema enthält 344.201 Bytes. Zum Vergleich: die (komprimierte) PNG-Datei ist 247.014 Bytes groß.

Nützliche Java-Tipps (Verwendung von Listen)

- **Anlegen/Befüllen** einer Liste von `Integer`-Objekten:

```
import java.util.LinkedList;
import java.util.List;
...
List<Integer> listA = new LinkedList<>();
listA.add(10);
listA.add(3);
listA.add(7);
...
```

  Man beachte, dass die Werte vom (primitiven) Typ `int` hier automatisch zu Listenelementen vom Objekttyp `Integer` konvertiert werden. So etwas nennt man „Auto-Boxing“.

- **Aneinanderfügen** (Verketten) von zwei Listen:

```
List<Integer> listB = new LinkedList<>();
listB.add(20);
listB.add(30);
...
listA.addAll(listB);
// listA = (10,3,7,20,30), listB = (20,30)
```

- **Iteration** über eine Liste von `Integer`-Objekten:

```
for (int value : listA) {
  // process value ...
}
```

  Hier werden die Listenelemente vom Objekttyp `Integer` automatisch auf den primitiven Typ `int` konvertiert („Auto-Unboxing“).

- Beachte, dass beim **Import** das Package `java.util` verwendet wird!