

Übungsaufgaben II, SBV1

Lukas Fiel, Lisa Panholzer

November 19, 2018

2 Übungsaufgaben II

2.1 Resampling und Interpolation

a) Implementierung Resampling

Die Implementierung des Resampling Filters wurde in diesem Abschnitt anhand der Nearest Neighbour Interpolation umgesetzt. Bevor der Filter ausgeführt wird, wird zuerst der Skalierungsfaktor bei dem Benutzer abgefragt. Gibt der Benutzer einen Wert ein, der über 1.0 liegt, wird das Bild vergrößert. Gibt er einen Wert ein, der unter 1.0 liegt wird das Eingangsbild verkleinert.

In dieser Implementierung wird die Umrechnung der Koordinaten anhand der Variante B umgesetzt. Dies bedeutet, dass der Skalierungsfaktor bereits vor der Neuberechnung der Koordinaten angepasst wird, in dem von diesem 1 subtrahiert wird. Das heißt, die neue Koordinate vom skalierten Bild B wird aus der Multiplikation der Koordinate aus dem Originalbild A um den adaptierten Skalierungsfaktor s' berechnet. Dies hat zur Folge, dass sich die Indizes in der Mitte zentrieren. Der Anfang bzw. das Ende des Bild Arrays bleibt hierbei aber weiterhin unterrepräsentiert.

```
        ; columns

import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;

import java.awt.*;

import ij.gui.GenericDialog;

public class Resample_ implements PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about"))
            {showAbout(); return DONE;}
        return DOES_SG+DOES_STACKS+SUPPORTS_MASKING;
    } //setup

    public void run(ImageProcessor ip) {
        byte[] pixels = (byte[])ip.getPixels();
        int width = ip.getWidth();
        int height = ip.getHeight();
        int tgtRadius = 4;

        int newWidth = width;
        int newHeight = height;

        int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
            ↪ height);

        //first request target scale factor from user
        GenericDialog dialog = new GenericDialog("user_input");
        dialog.addNumericField("scale_factor:", 1.0, 2);
        dialog.showDialog();
    }
}
```

```

        if(dialog.wasCanceled()) {
            return;
        }

        double tgtScaleFactor = dialog.getNextNumber();
        if(tgtScaleFactor < 0.01 || tgtScaleFactor > 10) {
            return;
        }

        newWidth = (int)(width * tgtScaleFactor + 0.5);
        newHeight = (int)(height * tgtScaleFactor + 0.5);

        // variant A of transformation of coordinates
        //double scaleFactorX = newWidth / (double)(width);
        //double scaleFactorY = newHeight / (double)(height);

        // variant B of transformation of coordinates
        double scaleFactorX = (double)(newWidth - 1.0) / (double)(width - 1.0);
        double scaleFactorY = (double)(newHeight - 1.0) / (double)(height - 1.0);
        ↪ ;

        //information output
        System.out.println("tgtScale:_" + tgtScaleFactor + ",_sX:_" +
            ↪ scaleFactorX + ",_sY:_" + scaleFactorY);
        System.out.println("new_width:_" + newWidth + ",_new_height:_" +
            ↪ newHeight);

        int [][] scaledImage = new int[newWidth][newHeight];

        //iterate over all pixel of the scaled image
        for(int x = 0; x < newWidth; x++) {
            for (int y = 0 ; y < newHeight; y++) {
                //calculate new scaled x and y coordinates
                double newX = (double)(x) / scaleFactorX;
                double newY = (double)(y) / scaleFactorY;

                //calculate new result value
                int resultVal = GetNNinterpolatedValue(inDataArrInt,
                    ↪ newX, newY, width, height);

                //set new value
                scaledImage[x][y] = resultVal;
            }
        }

        //show new image
        ImageJUtility.showNewImage(scaledImage, newWidth, newHeight, "scaled_img");
    } //run

    void showAbout() {
        IJ.showMessage("About_Template...",
            "this_is_a_PluginFilter_template\n");
    } //showAbout

    public int GetNNinterpolatedValue(int [][] inImg, double x, double y, int width,
        ↪ int height) {
        //round x and y position
        int xPos = (int) (x + 0.5);
        int yPos = (int) (y + 0.5);

        //safety check
        if(xPos >= 0 && xPos < width && yPos >= 0 && yPos < height) {
            return inImg[xPos][yPos];
        }
        return 0;
    }

    public int GetBilinearinterpolatedValue(int [][] inImg, double x, double y, int
        ↪ width, int height) {
        //implemented in separate java file
    }
}

```

```
} //class Resample_
```

b) Implementierung Bi-Lineare Interpolation

Die Implementierung des Resamplings Filters wurde in diesem Abschnitt anhand der Bilinearen Interpolation umgesetzt. Bevor der Filter ausgeführt wird, wird zuerst der Skalierungsfaktor bei dem Benutzer abgefragt. Gibt der Benutzer einen positiven Wert ein, der über 1.0 liegt, wird das Bild vergrößert. Gibt er einen darunterliegenden Wert ein wird das Eingangsbild verkleinert.

Danach wird die neue Höhe und Breite des Eingangsbildes anhand des Skalierungsfaktor berechnet. Zusätzlich wird der Skalierungsfaktor für die x und y Koordinaten separat berechnet und gespeichert. Anschließend wird anhand einer for-Schleife über alle Pixel des neu angelegten Arrays des skalierten Bildes iteriert.

Neben der neuen Koordinate wird der Pixelwert anhand der Methode `GetBilinearInterpolatedValue()` berechnet. In dieser werden unterschiedliche Fragmente für die Berechnung des gewichteten Mittelwert aufbereitet. Damit der neue skalare Wert berechnet werden kann, müssen zuerst 4 benachbarten Pixel aus dem Originalbild ermittelt werden. Die Nachbarpixel sind folgende: $p1(x+1,y)$, $p2(x,y+t1)$, $p3(x+1, y+1)$ und $p4(x,y)$.

Um den gewichtet Mittelwert für diesen Pixel zu erhalten werden anhand der Kalkulationsfragmente, den benachbarten skalaren Werten ($p1$ - $p4$) der gewichtete Mittelwert berechnet werden. Anschließend wird dieser Wert an die Methode retourniert und im skalierten Bild an der aktuellen Koordinate eingefügt.

Test der Implementierung

Um die Implementierung des Resampling Filters anhand der bi-linearen Interpolation zu prüfen und mit dem Resampling Filter und der Nearest Neighbor Interpolation zu vergleichen, wurden zwei Testbilder inklusive Differenzbild generiert.

Testbild zur bilinearen Interpolation (Skalierungsfaktor 2.0):



Figure 1: Resampling anhand bilinearer Interpolation und Skalierung um Faktor 2.0



Figure 2: Resampling anhand Nearest Neighbor Interpolation und Skalierung um Faktor 2.0

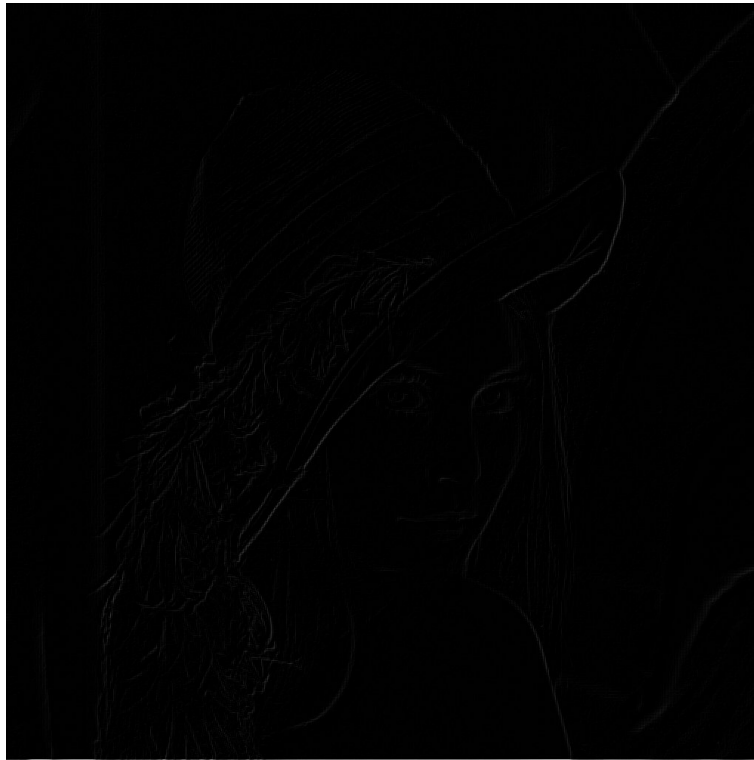


Figure 3: Differenzbild generiert aus Nearest Neighbor und bilinearer Interpolation

Das in Figure 3 dargestellte Bild, stellt die Differenz zwischen dem Testbild das anhand der Nearest Neighbor und der bilinearen Interpolation erstellt wurde. Bei näherer Betrachtung sieht man, dass sich an den Kanten im Bild, weiße Ausprägungen wiederfinden. Dies verdeutlicht, dass die NN Interpolation aufgrund keiner Neuberechnung der Werte, diese nicht so gut wiedergibt. Diese Unterschiede in den skalaren Werten ergeben sich daraus, dass die bilineare Interpolation aufgrund der Verwendung des gewichteten Mittelwerts von 4 Pixeln einen ähnlicheren skalaren Wert ergibt, als bei der NN Interpolation.

```

; columns

import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;

import ij.gui.GenericDialog;

public class ResampleBilineareInterpolation_ implements PlugInFilter {

```

```

public int setup(String arg, ImagePlus imp) {
    if (arg.equals("about")) {
        showAbout();
        return DONE;
    }
    return DOES_SG + DOES_STACKS + SUPPORTS_MASKING;
} // setup

public void run(ImageProcessor ip) {
    byte[] pixels = (byte[]) ip.getPixels();
    int width = ip.getWidth();
    int height = ip.getHeight();

    int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
        ↪ height);

    int newWidth = width;
    int newHeight = height;

    // first request target scale factor from user
    GenericDialog dialog = new GenericDialog("user_input");
    dialog.addNumericField("scale_factor", 1.0, 2);
    dialog.showDialog();

    // if user has canceled the dialog
    if (dialog.wasCanceled()) {
        return;
    }

    // get user input of scale factor
    double tgtScaleFactor = dialog.getNextNumber();

    // check range
    if (tgtScaleFactor < 0.01 || tgtScaleFactor > 10) {
        return;
    }

    // calculate new width and height with scale factor
    newWidth = (int) (width * tgtScaleFactor + 0.5);
    newHeight = (int) (height * tgtScaleFactor + 0.5);

    // calculate scale factor per dimension (variant a)
    double scaleFactorX = newWidth / ((double) width);
    double scaleFactorY = newHeight / ((double) height);

    // information output
    System.out.println("tgtScale_=" + tgtScaleFactor + "sX=" + scaleFactorX
        ↪ + "sY=" + scaleFactorY);
    System.out.println("new_width_=" + newWidth + "new_height_=" +
        ↪ newHeight);

    int[][] scaledImg = new int[newWidth][newHeight];

    // fill new result image → iterate over result image
    for (int x = 0; x < newWidth; x++) {
        for (int y = 0; y < newHeight; y++) {
            // calculate new coordinate
            double newX = x / scaleFactorX;
            double newY = y / scaleFactorY;

            // get bilinear interpolated value
            double resultVal = GetBilinearInterpolatedValue(
                ↪ inDataArrInt, newX, newY, width, height);

            // set new rounded value for current location
            scaledImg[x][y] = (int) (resultVal + 0.5);
        }
    }

    // show new image
    ImageJUtility.showNewImage(scaledImg, newWidth, newHeight, "scaled_img_(
        ↪ bilinear_interpolation");
} // run

public double GetBilinearInterpolatedValue(int[][] inImg, double x, double y,

```

```

↪ int width, int height) {

    // calculate the delta for x and y
    double deltaX = x - Math.floor(x);
    double deltaY = y - Math.floor(y);

    // set calculation fregment
    int xPlus1 = (int) x + 1;
    int yPlus1 = (int) y + 1;

    //handling of image edge for x
    if (x + 1 >= width) {
        xPlus1 = (int) x;
    }

    //handling of image edge for y
    if (y + 1 >= height) {
        yPlus1 = (int) y;
    }

    // get 4 neighboring pixels
    int neighbor1 = inImg[xPlus1][((int) (y))];
    int neighbor2 = inImg[((int) (x))][yPlus1];
    int neighbor3 = inImg[xPlus1][yPlus1];
    int neighbor4 = inImg[((int) (x))][((int) (y))];

    // calculate weighted mean out of neighbors
    double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
        ↪ deltaX * (1 - deltaY) * neighbor1)
        + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
        ↪ * neighbor3);

    return weightedMean;
}

void showAbout() {
    IJ.showMessage("About_Template...", "this_is_a_PluginFilter_template\n"
        ↪ );
} // showAbout

} // class ResampleBilinearInterpolation_

```

c) Implementierung Checker-Board

Die Nearest Neighbor Interpolation bezieht sich in ihrer Berechnung auf keinen neuen skalaren Wert. Es werden jeglich die benachbarten Pixel geprüft, und der Wert der am Nächsten ist, für den neuen Pixel herangezogen. Die Neuzuweisung schlägt sich auf die Bildqualität wieder, dieser wird bei einer Skalierung pixeliger bzw. kantiger.

Theoretische Überlegung: Da bei dieser Strategie kein komplexer Algorithmus verwendet wird und nur eine Neuzuweisung statt - berechnung stattfindet, ist dieser schneller als die andere Strategie.

Die bilineare Interpolation hingegen bedient sich auch der benachbarten Pixel, berechnet jedoch auf Basis dieser den gewichteten Mittelwert. Dieser Wert ist ähnlicher als zum Beispiel die Zuweisung der NN Interpolation und resultiert daher in einem besseren Bildqualität.

Theoretische Überlegung: Aufgrund der Verwendung eines komplexeren Algorithmus ist die Laufzeit bei diesem auch länger.

2.2 Klassifizierung mittels Kompression

a) Klassifizierung von Texten

Idee

Aus Texten in 8 verschiedenen Sprachen soll mittels Kompression eine Klassifizierung stattfinden. Dazu wurden folgende Datensätze als *.txt Dateien vorbereitet:

- Abstract einer wissenschaftlichen Arbeit. Diese hatte den Vorteil dass es eine deutsche und englische Übersetzung gab. Alle weiteren Sprachen wurden aus der englischen Version mittels *google translate* generiert.
- Wörterbuch mit 10000 deutschen Wörtern. Dieser Datensatz wurde mittels *google translate* in alle anderen Sprachen übersetzt.
- Die erste Seite der Datenschutzrichtlinien von Facebook. Da die Datenschutzrichtlinien in sämtlichen Sprachen abrufbar sind, konnte für alle Sprachen ein passender Datensatz gefunden werden.
- Die erste Seite der Datenschutzrichtlinien von Google. Auch hier waren Daten in allen Sprachen verfügbar.
- Ein Witz der aus dem deutschen mittels *google translate* in alle andern Sprachen übersetzt wurde.

Da nach einer Übersetzung die Texte in verschiedenen Sprachen ungleich viele Buchstaben beinhalten ist auch die Dateigröße unterschiedlich. Dies könnte eventuell rechnerisch berücksichtigt werden. Viel einfacher aber ist es, die letzten Buchstaben jedes langen Textes zu ignorieren und so eine einheitliche Länge des Textes zu gewährleisten. Dies wurde mittels eines shell-Skripts erreicht, welches nur die ersten n Bytes eines Files speichert. So konnte für jeden Text eine Datei erzeugt werden die in allen Sprachen den selben Speicherbedarf hat. Der Verlust der letzten Byte ist bei einer Klassifizierung unwesentlich.

```
; columns
#!/bin/bash

# mkdir cutTestData
mkdir cutTestData/witzData/

for filename in TestData/witzData/*.txt; do
    dd bs=1 count=8200 if="$filename" of="cut$filename"
done
```

Nach einer solchen Normierung der Texte können diese miteinander verglichen werden. Dazu wurde ein Programm in *Octave* geschrieben (siehe Listing a) , welches die Texte der einzelnen Datensätze miteinander vergleicht und in einer Matrix darstellt. Eine qualitativ hochwertige Aussage ob die Ergebnisse statistische Aussagekraft haben, kann mit 5 Datensätzen nicht getroffen werden. Es ist aber sicherlich ein Trend erkennbar.

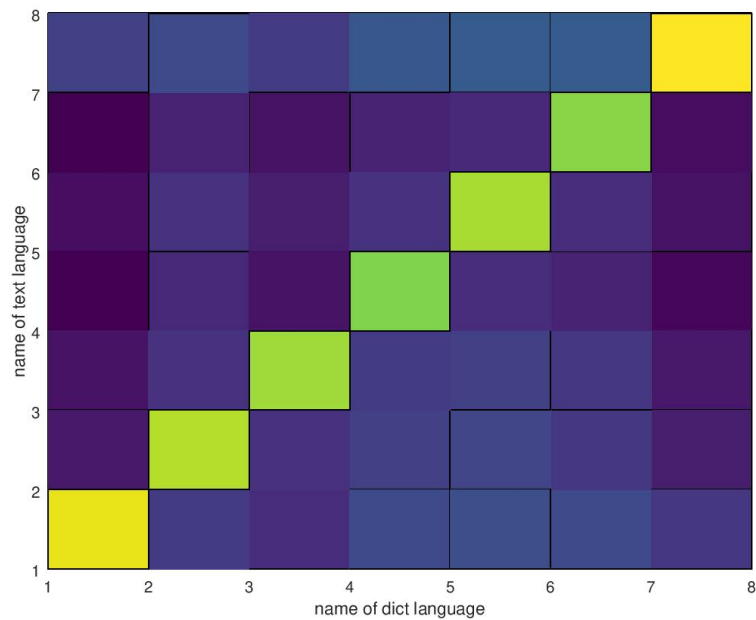


Figure 4: zipDataMatrix

Listing 1: Octave Script zur Darstellung der einzelnen Kompressionsraten.

```

; columns

# clear console, clear variables, close all open figures
clc
clear all
close all

# in this example we will use following languages
# languages = {"de","en","fr","es","po","un","bo","ne"};

# constants
# please note: copy-pasting windows paths: there are no backslashes in the path
folderPath = 'cutTestData/*';

# initialize result matrix
resultMatrix = zeros(8);

# loop through data folders
folders = glob(folderPath)
for x=1:numel(folders)
    [~, nameI] = fileparts (folders{x})
    for y=1:numel(folders)
        [~, nameI] = fileparts (folders{y})

        # just calculate matrix for different data sets
        if (!strcmp(folders{x},folders{y}))

            folderPath1 = [folders{x} , '/*'];
            folderPath2 = [folders{y} , '/*'];
            filesOfFolder1 = glob(folderPath1);
            filesOfFolder2 = glob(folderPath2);

            # for every element in data folder
            for i=1:numel(filesOfFolder1)
                [~, nameI] = fileparts (filesOfFolder1{i});
                #get file size
                [info, err, msg] = stat (filesOfFolder1{i});
                file1Size = info.size;

                #CREATE zip of file
                tmpFolderPath = nameI;
                mkdir(tmpFolderPath);
                copyfile(filesOfFolder1{i},tmpFolderPath);
                firstZipName = [nameI, '.zip'];
                zip(firstZipName,[tmpFolderPath,'/*']);
                #get zip size
                [info, err, msg] = stat (firstZipName);
                file1file2ZipSize = info.size;

                # calculate compression rate
                file1CompressionRate = file1file2ZipSize / file1Size;

                #delete zip and folder as we just need the size for calculation
                delete([tmpFolderPath,'/',nameI,'.txt']);
                rmdir(tmpFolderPath);
                delete(firstZipName);

                for j=1:numel(filesOfFolder2)
                    [~, nameJ] = fileparts (filesOfFolder2{j});
                    zipName = [nameI,nameJ,'.zip'];
                    #create tmp folder in testData folder
                    tmpFolderPath = [nameI,nameJ];
                    mkdir(tmpFolderPath);

                    #copy filesOfFolder1 to folder
                    ['copy_filesOfFolder1_', nameI, '__', nameJ , '_to_', tmpFolderPath];
                    copyfile(filesOfFolder1{i},tmpFolderPath);
                    copyfile(filesOfFolder2{j},tmpFolderPath);

                    # get folder size (add jokeFile size to filesOfFolder1size)
                    [info, err, msg] = stat (filesOfFolder2{i});
                    file2Size = file1Size + info.size;

                    #zip it and get size of zip
                    zip(zipName,[tmpFolderPath,'/*']);

```

```

[info, err, msg] = stat (zipName);
file2ZipSize = info.size;

#calculate compression rate
compressionRate = file2ZipSize / file2Size ;

#calculate expected rate
expectedfile2ZipSize = file2Size * file1CompressionRate;

%Matrix(i,j) = (expectedfile2ZipSize - file2ZipSize) / file2ZipSize;
kompressionDict = (file1Size - file1file2ZipSize) / file1Size;
kompressionBoth = (file2Size - file2ZipSize) / file2Size;
kompressionsDelta = abs(kompressionBoth - kompressionDict);
Matrix(i,j) = kompressionsDelta;

#cleanup - remove tmp folder
[remove_ tmpFolderPath];
delete([tmpFolderPath, '/', nameI, '.txt']);
delete([tmpFolderPath, '/', nameJ, '.txt']);
rmdir(tmpFolderPath);
delete(zipName);
endfor
endfor

resultMatrix = resultMatrix .+ Matrix;

else # dont calculate anything if the folders are the same
["skip " folders{x}]
endif

endfor
endfor

figure()
surf(resultMatrix)
view(2)
xlabel("name of dict language");
ylabel("name of text language");
zlabel("diff of compression rates");
'SUCCESS'

```

b) OPTIONAL – nur für Interessierte/Experten

2.3 Kompression und Code-Transformation

a) Lempel-Ziv Kompression einer Sequenz

Figure a) zeigt die händische Berechnung der Lemper Ziv Kompression. Beim Übertragen ins Protokoll wurde allerdings ein Fehler entdeckt, der in Tabelle 1 korrigiert wurde.

Aufgabe 2.3.a Lempel Ziv Kompression

Zeichenkette: abababbbbaaaabababcddda

aktuelles Zeichen	nächstes Zeichen	Ausgabe 2 im Wörterbuch?	ins Wörterbuch?
a	b	N → a	ab 256
b	a	N → b	ba 257
a	b	Y → 256	aba 258
b	b	Y → 256	abb 259
b	b	N → b	bbb 260
b	a	N → 257	baa 261
a	a	N → a	aaa 262
a	a	Y 262	aab 263
b	a	Y 257	bab 264
b	a	Y 257 Y 264	baac 265
c	c	N → c	cc 266
c	d	N → c	cd 267
d	d	N → d	dd 268
d	d	Y 268	dda 269

23 Zeichen im Ursprungstext
14 Zeichen im Resultat

Zeichenkette:

97, 98, 256, 256, 98, 257, 97, 262, 257, 257, 264, 99, 99,
↳ 100, 268

Kompressionsrate: $\frac{23}{14} = 1,64$

aktuelles Zeichen	nächstes Zeichen	Ausgabe (im Wörterbuch?)	ins Wörterbuch!	Speicher
a	b	N → a	ab	256
b	a	N → b	ba	257
a	b	Y → 256	aba	258
a	b	Y → 256	abb	259
b	b	N → b	bb	260
b	a	Y → 257	baa	261
a	a	N → a	aa	262
a	a	Y → 256	aab	263
b	a	Y → 257	bab	264
b	a	Y (257), Y → 264	bababc	265
c	c	N → c	cc	266
c	d	N → c	cd	267
d	d	N → d	dd	268
d	d	Y → 268	dda	269
a		N → a		

Table 1: Level Ziv Kompression

In der korrigierten Version ergibt die resultierende Zeichenkette:

97	98	256	256	98	257	97	262	257	264	99	99	100	268	269
----	----	-----	-----	----	-----	----	-----	-----	-----	----	----	-----	-----	-----

$$[H]KompressionsrateC = \frac{23}{15} = 1.5334 \quad (1)$$

b) Huffman Coding

Die nächste Seite zeigt die händische Berechnung des Huffman Baums zum gegebenen Beispiel. Die mittlere Codewortlänge liegt dabei bei $1.869bit$.

Weiters kann man auf der darauffolgenden Seite die manuelle Berechnung von 6 Testfällen finden. Es kann gesagt werden, dass die Huffman Kompression sehr gut geeignet wäre um Daten zu komprimieren, die sehr oft gleich sind. Dabei spielt homogenität keine Rolle, sondern rein die Häufigkeit des Auftretens der Sonderfälle. An dieser Stelle sei erwähnt, dass die Information

über solche Sonderfälle nicht verloren geht, sondern einfach mehr Speicher benötigt (verlustfreies Komprimieren).

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 a b a b a b b b a a a a b a b a b c c d d d a

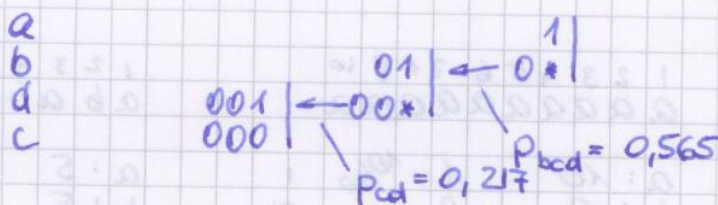
codierung $a = 00, b = 01, c = 10, d = 11$

23 Zeichen zu 2 bit ergeben eine Länge von 46 bit

Häufigkeiten

a: 10	$10/23 = 0,435$	} sort	a	1	1
b: 8	$8/23 = 0,348$		b	01	2
c: 2	$2/23 = 0,087$		c	000	4
d: 3	$3/23 = 0,130$		d	001	3

Baum



mittlere Codewortlänge

$$L = 1 \cdot 10 + 2 \cdot 0,348 + 3 \cdot 0,087 + 3 \cdot 0,130 = 1,782 \text{ bit}$$

Kompressionsrate $\frac{2 \text{ bit}}{1,782 \text{ bit}} = 1,122$

Binärlayout: $10 \cdot 1 + 8 \cdot 2 + 2 \cdot 3 + 3 \cdot 3 = 41 \text{ bit}$

Zeichen: $\{a, b, c, d\} \rightarrow 2\text{bit Darstellung } \{00, 01, 10, 11\}$

1 2 3 4 5 6 7 8 9 10
a b c d a b c d a b

a: 3 $3/10 = 0,3$ 1
b: 3 $3/10 = 0,3$ 01
c: 2 $2/10 = 0,2$ 001
d: 2 $2/10 = 0,2$ 000

$$L = 1 \cdot 0,3 + 2 \cdot 0,3 + 3 \cdot 0,2 \cdot 2 = 2,1$$

Kompressionsrate: 0,95
 \rightarrow keine Kompression

1 2 3 4 5 6 7 8 9 10
a a a a a a a b c d

a: 7 $7/10 = 0,7$ 1
b: 1 $1/10 = 0,1$ 01
c: 1 $1/10 = 0,1$ 001
d: 1 $1/10 = 0,1$ 000

$$L = 1 \cdot 0,7 + 2 \cdot 0,1 + 3 \cdot 0,1 \cdot 2 = 1,5$$

Kompressionsrate: 1,5 1,3

1 2 3 4 5 6 7 8 9 10
a a a a a a a a a a

a: 10 $1 = 10/10$ 1
b: 0 0 01
c: 0 0 001
d: 0 0 000

$$L = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot 0 \cdot 2 = 1$$

Kompressionsrate: 2

1 2 3 4 5 6 7 8 9 10
a b a b a b a b a b

a: 5 $5/10 = 0,5$ 1
b: 5 $5/10 = 0,5$ 01
c: 0 0 001
d: 0 0 000

$$L = 1 \cdot 0,5 + 2 \cdot 0,5 + 3 \cdot 0 = 1,5$$

Kompressionsrate: 1,3

1 2 3 4 5 6 7 8 9 10
b b b b b c b b b b

a: 0 0 000
b: 9 $9/10 = 0,9$ 1
c: 1 $1/10 = 0,1$ 01
d: 0 0 001

$$L = 1 \cdot 0,9 + 2 \cdot 0,1 + 3 \cdot 0 \cdot 2 = 1,1$$

Kompressionsrate: 1,82

1 2 3 4 5 6 7 8 9 10
c d c d c d c d c d

a: 0 0 000
b: 0 0 001
c: 5 $5/10 = 0,5$ 01
d: 5 $5/10 = 0,5$ 1

$$L = 1 \cdot 0,5 + 2 \cdot 0,5 + 3 \cdot 0 = 1,5$$

Kompressionsrate: 1,3

c) Komprimierung einer Sequenz mittels Runlength Coding

Berechnung der Kompressionsrate

Die nachfolgende Sequenz soll anhand von Runlength Coding händisch komprimiert und die Kompressionsrate ausgegeben werden:

010101111100000111100010101111 (30 Stellen, n=2 Symbole:0,1)

Die Sequenz wird von links nach rechts codiert, und anstatt der eigentlichen Zeichen die Häufigkeit dieser ausgegeben. Nach der händischen Komprimierung weist die Frequenz folgende Lauflänge auf:

11111554311114 (14 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$30/14 = \mathbf{2,14}$

Erweiterung der Symbolmenge

Die Komprimierung von Sequenzen anhand der RLC ist am effektivsten, je homogener der Informationsgehalt ist. Das bedeutet, dass bei vielen unterschiedlichen Zeichen die Komprimierungsmethode nicht mehr sinnvoll angewandt werden kann. Bei sehr kurzen Sequenzen kann es sogar zu einer Erhöhung der Zeichenanzahl in dieser kommen.

Wird nun die Anzahl der Symbole erhöht, muss beachtet werden, dass zusätzlich zu der Häufigkeit des Zeichens noch ein Trennsymbol, eine ID bzw. das Zeichen selbst mitgegeben werden muss. Dies wird anhand eines selbst gewählten Beispiels demonstriert.

Folgende Sequenz wird anhand von RLC komprimiert:

AAZZBBBBBBBBBCCCCCCCDEEEEEEEFFGHIJJJKLMN

(40 Stellen, n=14 Symbole:A,Z,B,C,D,E,F,G,H,I,J,K,L,M,N)

Die komprimierte Sequenz lautet:

A2Z2B8C7D1E7F2H1I2J3K1L1M1N1 (28 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$40/28=\mathbf{1,42}$

Wird eine sehr kurze Sequenz mit einer hohen Anzahl an Symbolen komprimiert, kann es aufgrund des hohen Informationsgehalts dazu kommen, dass die codierte Sequenz länger ist, als die Originale.

Folgende Sequenz wird anhand von RLC komprimiert:

AZBBCDEEFFFGHIIJJJKLLLMNOPQRSTU

(30 Stellen, n=21 Symbole:A,Z, B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T)

Die komprimierte Sequenz lautet:

A1Z1B2C1D1E2F3G1H1I2J3K1L3M1N1O1P1Q1R1S1T1 (42 Stellen)

Daraus ergibt sich folgende Kompressionsrate:

$30/42=0,71$

d) Entropieberechnung

Folgende Sequenz zur Entropieberechnung ist gegeben:

111122661112233334564511211111 (30 Stellen, n=6 Symbole: 1,2,3,4,5,6)

Entropieberechnung

Sequenz: 1 1 1 1 2 2 6 6 1 1 1 2 2 3 3 3 3 4 5 6 4 5 1 1 2 1 1 1
 → 30 Stellen

Häufigkeiten:

1 = 14	4 = 2
2 = 5	5 = 2
3 = 4	6 = 3

$$H = - \left(\left(\frac{14}{30} \cdot \log_2 \left(\frac{14}{30} \right) \right) + \left(\frac{5}{30} \cdot \log_2 \left(\frac{5}{30} \right) \right) + \right. \\
\left. \left(\frac{4}{30} \cdot \log_2 \left(\frac{4}{30} \right) \right) + \left(\frac{2}{30} \cdot \log_2 \left(\frac{2}{30} \right) \right) + \right. \\
\left. \left(\frac{2}{30} \cdot \log_2 \left(\frac{2}{30} \right) \right) + \left(\frac{3}{30} \cdot \log_2 \left(\frac{3}{30} \right) \right) \right) \\
= \underline{\underline{2,1846}}$$

minimale Entropie: 1 1 1 1 1 1 1 1 1 1 $H = 0$

maximale Entropie: 0 1 2 3 4 5 6 7 8 9 $H = 3,32$

Figure 5: manuelle Entropieberechnung

Nachdem die Häufigkeiten der Symbole erfasst wurde, wurde die Formel der Shannon-Entropie angewandt. Das Ergebnis der Berechnung lautet **2,1846**.

Minimale und Maximale Entropie

Bei folgender 10-stelliger Sequenz ist die Entropie minimal:

Sequenz: 1111111111 (10 Stellen, n=1 Symbol: 1)

$$H = 0$$

Bei folgender 10-stelliger Sequenz ist die Entropie maximal:

Sequenz: 0123456789 (10 Stellen, n=10 Symbole: 0,1,2,3,4,5,6,7,8,9)

$$H = 3,32$$

Auswirkung auf Kompressionsrate

Eine geringe Entropie bewirkt, dass die Sequenz besser komprimierbar ist. Je höher die Entropie, desto schlechter ist die Sequenz komprimierbar. Nicht nur die Auftrittswahrscheinlichkeit hat eine Auswirkung auf die erzielbare Kompressionsrate. Auch die Länge der Sequenz, sowie die Anzahl der Zeichen spielt eine wesentliche Rolle. Je größer diese Faktoren, desto schlechter ist eine Sequenz durch unterschiedliche Kompressionsverfahren bearbeitbar.