Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

Hausarbeit Algorithmen und Datenstrukturen
Studiengang Informatik
Studienrichtung Informatik

Anwendung des JPEG–Verfahrens

Eingereicht von: Leon Baumgarten
5CS22-1
Matrikelnummer: 5002213

16. Mai 2024

Inhaltsverzeichnis

1 Einleitung

Das JPEG-Verfahren (*Joint Photographic Experts Group*) ist eine weitverbreitete Methode zur Kompression digitaler Bilder, die eine signifikante Reduzierung der Dateigröße bei gleichzeitig geringem Verlust an optischer Qualität ermöglicht. Diese Hausarbeit demonstriert die Anwendung des JPEG-Algorithmus anhand eines gegebenen Graustufenbildes im Byte-Format mit einer Größe von 512×512 Pixeln.

Der Algorithmus beginnt mit der Aufteilung des Bildes in 8×8 Blöcke. Anschließend wird auf jeden Block die diskrete Kosinustransformation (DCT) angewendet. Die resultierenden Koeffizienten werden quantisiert, wobei drei verschiedene Quantisierungsmatrizen (Q20, Q50, Q90) verwendet werden. Nach der Quantisierung erfolgt die Rücktransformation durch Multiplikation mit der jeweiligen Quantisierungsmatrix und inverse DCT, gefolgt von der Rekonstruktion des Bildes.

In der vorliegenden wissenschaftlichen Hausarbeit werden die Schritte des JPEG-Verfahrens nachvollzogen, die Ergebnisse für verschiedene Quantisierungsmatrizen grafisch dargestellt und analysiert, sowie die Kompressionsleistung anhand des prozentualen Anteils der Nullen nach der Quantisierung bewertet.
– Die in der Hausarbeit beschriebene Implementierung zeigt somit die praktische Anwendung numerischer Methoden in der Bildverarbeitung. Durch die Zerlegung des Bildes in 8x8-Blöcke und die Umwandlung der Pixelwerte in das RGB-Format werden die Daten für den Einsatz des JPEG-Algorithmus vorbereitet, was einen wesentlichen Aspekt der modernen digitalen Bildkompression darstellt. Dieses Vorgehen veranschaulicht nicht nur die theoretischen Konzepte hinter der Bildkompression, sondern bietet auch Einblicke in die praktische Umsetzung numerischer Algorithmen in Software.

2 Aufgaben

1. Es ist ein Datensatz *picdat.dat* gegeben. Er liegt für ein Graustufenbild im Byte–Format vor, d.h. die Werte pro Pixel liegen im Bereich $0 \leq px \leq 255$.
Das Bild hat die Dimension 512×512. Lesen Sie den Datensatz ein und stellen Sie das Bild grafisch dar.

2. Der JPEG–Algorithmus wird auf 8×8 Matrizen angewendet. Zerlegen Sie also die 512×512 Datenmatrix in 64 8×8 Untermatrizen. .

3. Berechnen Sie die 8×8 orthogonale Transformationsmatrix der DCT

4.Auf jede Untermatrix wenden Sie den JPEG Algorithmus an:

- Elementweise Subtraktion von 128
- Anwendung der DCT
- Quantisierung mit der Matrix Q
- Rücktransformation (elementweise Multiplikation mit Q, inverse DCT, elementweise Addition von 128)

5. Bilden Sie wieder eine 512×512 Matrix

6. Es sind drei Q–Matrizen (8×8) gegeben: Q20 (Q20.dat), Q50 (Q50.dat) und Q90 (Q90.dat). Der JPEG–Algorithmus soll für alle drei Qi Matrizen ausgeführt werden. Für diese Matrizen geben Sie jeweils an:

- Wie hoch (prozentual) ist der Anteil der Nullen nach der Quantisierung
- Geben sie die resultierenden Bildern grafisch wieder und vergleichen Sie diese mit dem Originalbild

3 Lösungen

3.1 Einlesen & Ausgabe des Algorithmus

Zunächst wird in der main-Methode der Dateipfad festgelegt und die Bilddaten aus der Datei *picdat.dat* gelesen. Diese Daten werden zeilenweise verarbeitet, wobei jede Zeile einen Grauwert darstellt, der in ein Byte-Array umgewandelt wird. Anschließend wird aus diesem Array ein BufferedImage des Typs RGB erstellt, indem jeder Grauwert auf die Rot-, Grün- und Blau-Komponenten verteilt wird. (px, px, px)Schließlich wird das Bild in einem JFrame angezeigt, indem ein JPanel verwendet wird, das das Bild zeichnet.

Bei Fehlern während des Lesens der Datei oder der Bildgenerierung werden Fehlermeldungen in der Konsole ausgegeben. Der Code illustriert grundlegende Techniken der Datei- und Bildverarbeitung sowie der GUI-Anzeige in Java.

3.2 Aufteilen in 8x8 Matrizen

Das 512x512 Pixel große Bild wird in 8x8 Pixel große Blöcke unterteilt, ein Schritt, der grundlegend für den JPEG-Komprimierungsalgorithmus ist. Dabei wird zuerst die Blockgröße auf 8 Pixel festgelegt. Mit Hilfe einer Methode *splitImageIntoBlocks* wird das Bild in ein zweidimensionales Array von BufferedImage-Objekten zerlegt. Dies geschieht durch das Durchlaufen des Bildes mit zwei verschachtelten Schleifen, die jeweils die Startpunkte der 8x8 Blöcke bestimmen und diese mittels der *getSubimage*-Funktion extrahieren. Das Ergebnis ist ein Array, in dem jedes Element einem 8x8-Block des Originalbildes entspricht

3.3 Berechnung orthogonale Transformationsmatrix der DCT

Die diskrete Kosinustransformation (DCT) hilft dabei, das Bildsignal in Frequenzkomponenten zu zerlegen, wobei niedrige Frequenzen (Bereiche langsamer Änderung im Bild) von hohen Frequenzen (Bereiche schneller Änderung im Bild) getrennt werden. Dies ermöglicht eine effizientere Speicherung, da die visuell weniger wichtigen hohen Frequenzen stärker komprimiert werden können.

Die diskrete Kosinustransformation (DCT) für eine $n \times n$-Matrix wird durch die folgende Transformationsmatrix T repräsentiert:

$$T_{ij}(n) = \begin{cases} \frac{1}{\sqrt{n}} & \text{für } i = 0, \\ \sqrt{\frac{2}{n}} \cos\left(\frac{\pi i(2j+1)}{2n}\right) & \text{sonst.} \end{cases}$$

Der Skalierungsfaktor α wird definiert als:

$$\alpha = \begin{cases} \frac{1}{\sqrt{n}} & \text{wenn } u = 0 \\ \sqrt{\frac{2}{n}} & \text{sonst} \end{cases}$$

Die Matrix T für $n = 8$ kann dann folgendermaßen geschrieben werden:

$$T = \begin{pmatrix} \alpha(0) \cdot \cos\left(\frac{(2\cdot 0+1)0\pi}{16}\right) & \alpha(0) \cdot \cos\left(\frac{(2\cdot 1+1)0\pi}{16}\right) & \cdots & \alpha(0) \cdot \cos\left(\frac{(2\cdot 7+1)0\pi}{16}\right) \\ \alpha(1) \cdot \cos\left(\frac{(2\cdot 0+1)1\pi}{16}\right) & \alpha(1) \cdot \cos\left(\frac{(2\cdot 1+1)1\pi}{16}\right) & \cdots & \alpha(1) \cdot \cos\left(\frac{(2\cdot 7+1)1\pi}{16}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha(7) \cdot \cos\left(\frac{(2\cdot 0+1)7\pi}{16}\right) & \alpha(7) \cdot \cos\left(\frac{(2\cdot 1+1)7\pi}{16}\right) & \cdots & \alpha(7) \cdot \cos\left(\frac{(2\cdot 7+1)7\pi}{16}\right) \end{pmatrix}$$

Daraus ergibt sich folgende Transformationsmatrix:

$$T = \begin{pmatrix} 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 \\ 0.490 & 0.416 & 0.278 & 0.098 & -0.098 & -0.278 & -0.416 & -0.490 \\ 0.462 & 0.191 & -0.191 & -0.462 & -0.462 & -0.191 & 0.191 & 0.462 \\ 0.416 & -0.098 & -0.490 & -0.278 & 0.278 & 0.490 & 0.098 & -0.416 \\ 0.354 & -0.354 & -0.354 & 0.354 & 0.354 & -0.354 & -0.354 & 0.354 \\ 0.278 & -0.490 & 0.098 & 0.416 & -0.416 & -0.098 & 0.490 & -0.278 \\ 0.191 & -0.462 & 0.462 & -0.191 & -0.191 & 0.462 & -0.462 & 0.191 \\ 0.098 & -0.278 & 0.416 & -0.490 & 0.490 & -0.416 & 0.278 & -0.098 \end{pmatrix}$$

3.4 Anwendung JPEG Algorithmus

In diesem Kapitel ist der Pseudocode dargestellt, der die Logik zum Erstellen von 10 zufälligen Schlüsseln und die Ausgabe der Vater-Sohn-Beziehung beschreibt. Für das Erstellen von 10 zufälligen Schlüsseln wurde die Klasse 'Random' importiert. Mit der folgenden for-Schleife werden die Schlüssel erstellt. Die Funktion 'printTree' dient dazu, die Struktur des Suchbaums auszugeben. Dabei wird zunächst überprüft, ob der Wert in dem ersten Knoten *null* ist. Wenn der aktuelle Knoten *null* ist, handelt es sich um das Ende des Teilbaums und die Funktion springt an das Ende. Danach wird überprüft, ob der aktuelle Teilbaum ein linkes und rechtes Kind hat. Die Ausgabe erfolgt per print-Anweisungen. Anschließend ruft sich die Funktion selbst wieder auf (Rekursion). So werden die linken und rechten Unterbäume durchlaufen, wodurch die Baumstruktur rekursiv ausgegeben wird.

10 zufällige Schlüssel erstellen:

```
1                          for ( i von 0 bis 9) {
2                                  Erzeuge eine neue zufaellige
                                      Zahl -> value
3                                  Fuege value in den Baum ein }
4                          Return baum
```

Darstellung der Vater-Sohn-Beziehung:

```
1                          Funktion printTree(root) {
2                                  Wenn root nicht null ist:
3                                  Gebe Wurzel aus
4
5                                  Wenn root.left nicht null ist
                                      :
6                                  Gebe Linkes Kind aus
7                                  Sonst
8                                  Gebe "kein␣linkes␣Kind" aus
9
10                                 Wenn root.right nicht null
                                      ist:
11                                 Gebe  Rechtes Kind aus
12                                 Sonst
13                                 Gebe "kein␣rechtes␣Kind␣" aus
14
15                                 Gebe root.left aus
16                                 Gebe root.right aus }
17
18                         Funktion tenRandom(baum)
19                         Erzeuge eine neue Zufallszahl
```

3.5 512 x 512 Matrix bilden

In diesem Kapitel ist der Pseudocode dargestellt, der 100 Datensätze mit 20 zufälligen Schlüsseln generiert und die durchschnittliche Höhe aller erstellten Suchbäume berechnet. Um 100 Datensätze zu generieren, wird die bereits in Aufgabe 4 (Kapitel 2.4) geschriebene Funktion 'generateRandomTree' genutzt und mithilfe einer for-Schleife 100 mal ausgeführt. Damit werden 100 Datensätze mit je 20 zufälligen Schlüsseln erstellt. Um die durchschnittliche Höhe aller erstellten Suchbäume zu errechnen, werden die Höhen der generierten Suchbäume in einer Variable gespeichert und zu der Variable 'totalHeight' addiert. Letztlich wird die gesamte Höhe aller Suchbäume durch die Anzahl geteilt und man bekommt die durchschnittliche Höhe heraus.

100 zufällige Datensätze erstellen:

```
1                          Ganzzahl numberOfTrees = 100
2
3                          Funktion generateRandomTree(baum,
                              numberOfKeys){
```

```
4                                   for (i von 1 bis numberOfKeys
                                       ){
5                                         Wert = Zufall.nextInt
                                             (500)
6                                         baum = Einfuegen(baum
                                             , Wert) }
7                                   Gib baum zurueck }
8
9                           for (i von 1 bis numberOfTrees) {
10                                  Baumknoten rootT = null
11                                  rootT = zufaelligen Baum(
                                        rootT, 20)
12                                  Aufruf Methode zum Berechnen
                                        der Hoehe }
```

Durchschnittliche Höhe berechnen:

```
1                           in heightT Hoehe von Funktion
                                findHeight speichern
2                           Ganzzahl totalHeight = 0
3
4                           totalHeight += Hoehe jedes Baumes
5                           Double durchschnittliche Hoehe =
                                totalHeight / numberOfTrees
```

Vergleich - durchschnittliche Höhe binärer Suchbaum mit AVL-Baum
In einem Vergleich zwischen einem binären Suchbaum (BST) und einem AVL-Baum hinsichtlich ihrer durchschnittlichen Höhen ergeben sich wesentliche Unterschiede aufgrund ihrer Struktur und Ausgeglichenheit.

Ein binärer Suchbaum ist stark von der Reihenfolge der Schlüssel beim Einfügen abhängig. Im ungünstigsten Fall, etwa wenn die Schlüssel in auf- oder absteigender Reihenfolge eingefügt werden, kann die Höhe des Baums bis zu $n-1$ betragen, wobei n die Anzahl der Schlüssel ist. Im besten Fall, wenn die Schlüssel zufällig eingefügt werden, kann die durchschnittliche Höhe bei etwa $O(\log n)$ liegen.
Im Gegensatz dazu sind AVL-Bäume darauf optimiert, eine ausgeglichene Struktur zu erhalten, indem sie sicherstellen, dass die Höhendifferenz zwischen den Teilbäumen maximal 1 beträgt. Dadurch wird eine maximale Höhe von $O(\log n)$ gewährleistet. Die durchschnittliche Höhe eines zufällig konstruierten AVL-Baums ist daher tendenziell kleiner als die eines zufällig konstruierten binären Suchbaums, insbesondere wenn die Schlüssel zufällig eingefügt wurden.
Die durchschnittliche Höhe eines BST liegt nach mehrmaligem Ausführen des Pro-

grammcodes größtenteils zwischen 6 und 7. Im Vergleich dazu liegt die durchschnittliche Höhe eines AVL-Baumes mit 20 zufälligen Schlüsseln bei ca. 3.8. Damit ist die theoretische Vermutung bestätigt, dass der AVL-Baum aufgrund seiner ausgeglichenen Struktur, bewerkstelligt durch einfache und doppelte Rotationen, eine kleinere durchschnittliche Höhe aufweist. [**?**]

4 Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Hausarbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

Leipzig, 16. Mai 2024 _____

Leon Baumgarten

Literatur

[1] Plato, Robert
Numerische Mathematik kompakt. Grundlagenwissen für Studium und Praxis..
Berlin, Germany: Springer Spektrum (Lehrbuch), 2021

[2] Schwarz, Hans Rudolf; Köckler, Norbert
Numerische Mathematik. [mit Online-Service.
akualisierte Aufl. Wiesbaden: Vieweg + Teubner, (2011)

[3] Holger Perlt
Skript Vorlesung Numerik.

[4] A.M.Raid, W.M.Khedr, M. A. El-dosuky, Wesam Ahmed *Skript Mansoura University.*
https://www.airccse.org/journal/ijcses/papers/5214ijcses04.pdf (2014)

[5] Saupe, D., Hamzaoui, R. *Bild- und Videokompression.*
https://www.degruyter.com/document/doi/10.1524/itit.45.5.245.22713/html
(2009)

5 Anhang

Imports:

```
1  import java.awt.image.BufferedImage;
2  import java.io.BufferedReader;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import java.awt.Graphics;
```

Einlesen Datensatz und Ausgabe:

```
1  public class GrayScaleImageBuilder {
2
3          private static final int WIDTH = 512;
4          private static final int HEIGHT = 512;
5
6          public static void main(String[] args) {
7                  String filePath = "/Users/
                       leonbaumgarten/eclipse-workspace/
                       Numerik/src/JPEG_Verfahren/picdat.
                       dat";
8                  try {
9                          // Auslesen der Bilddaten von
                               picdat.dat
10                         byte[] imageData =
                               readImageData(filePath);
11                         BufferedImage image =
                               createRGBImageFromGrayScale
                               (imageData); //
                               BufferedImage aus den
                               Grauwerten erstellen
12                         System.out.println("Bild␣
                               wurde␣erfolgreich␣erstellt
                               .");
13                         displayImage(image, "
                               Graustufenbild");
14                 } catch (IOException e) {
15                         // falls Fehler ->
                               Fehlermeldung ausgeben
16                         System.err.println("Es␣gibt␣
                               einen␣Fehler:␣" + e.
                               getMessage());
```

```
17                  }
18              }
19              // Daten aus Datei in byte-aArray speichern
20              private static byte[] readImageData(String
                    filePath) throws IOException {
21                  BufferedReader reader = new
                        BufferedReader(new FileReader(
                        filePath));
22                  byte[] imageData = new byte[WIDTH *
                        HEIGHT];
23                  String line;
24                  int index = 0;
25
26                  while ((line = reader.readLine()) !=
                        null && index < imageData.length)
                        {
27                      int grayValue = Integer.
                            parseInt(line.trim()); //
                            als Integer formatieren
28                      imageData[index++] = (byte)
                            grayValue; // Wert im byte
                            -Array speichern
29                  }
30
31                  reader.close();
32                  if (index != WIDTH * HEIGHT) {
33                      throw new IOException("Die␣
                            Datei␣hat␣nicht␣die␣
                            erwartete␣Anzahl␣von␣
                            Pixeln.");
34                  }
35
36                  return imageData;
37              }
38
39              private static BufferedImage
                    createRGBImageFromGrayScale(byte[]
                    grayData) {
40                  BufferedImage image = new
                        BufferedImage(WIDTH, HEIGHT,
                        BufferedImage.TYPE_INT_RGB);
41                  int index = 0;
42                  for (int y = 0; y < HEIGHT; y++) {
```

```
43                          for (int x = 0; x < WIDTH; x
                               ++) {
44                               int px = grayData[
                                    index++] & 0xFF;
                                    // Vorzeichen
                                    korrigieren
45                               int rgb = (px << 16)
                                    | (px << 8) | px;
                                    // Grauwerte in
                                    RGB-format
                                    umwandeln (weil
                                    RGB-BufferedImage)
46                               image.setRGB(x, y,
                                    rgb);
47                          }
48                     }
49                     return image;
50                }
51                // Anzeige des Bildes
52                private static void displayImage(
                     BufferedImage img, String title) {
53                     javax.swing.SwingUtilities.
                          invokeLater(() -> {
54                               JFrame frame = new JFrame(
                                    title);
55                               JPanel panel = new JPanel() {
56                                    @Override
57                                    protected void
                                         paintComponent(
                                         Graphics g) {
58                                         super.
                                              paintComponent
                                              (g);
59                                         g.drawImage(
                                              img, 0, 0,
                                               this);
60                                    }
61                               };
62                               panel.setPreferredSize(new
                                    java.awt.Dimension(img.
                                    getWidth(), img.getHeight
                                    ()));
63                               frame.
```

```
                                    setDefaultCloseOperation(
                                    JFrame.EXIT_ON_CLOSE);
64                                  frame.getContentPane().add(
                                    panel);
65                                  frame.pack();
66                                  frame.setLocationRelativeTo(
                                    null);
67                                  frame.setVisible(true);
68                          });
69                  }
70          }
```

Segmentierung in 8x8 große Blöcke:

```
1           // Groesse Untermatrizen
2           private static final int BLOCK_SIZE = 8;
3
4           BufferedImage[][] blocks = splitImageIntoBlocks(image
               );
5
6           private static BufferedImage[][] splitImageIntoBlocks
               (BufferedImage image) {
7                   int numBlocksPerDimension = WIDTH /
                       BLOCK_SIZE;
8                   BufferedImage[][] blocks = new BufferedImage[
                       numBlocksPerDimension][
                       numBlocksPerDimension];
9
10                  for (int i = 0; i < numBlocksPerDimension; i
                       ++) {
11                          for (int j = 0; j <
                               numBlocksPerDimension; j++) {
12                                  // 8x8 Bloecke aus Gesamtbild
                                        ziehen
13                                  blocks[i][j] = image.
                                    getSubimage(j * BLOCK_SIZE
                                    , i * BLOCK_SIZE,
                                    BLOCK_SIZE, BLOCK_SIZE);
14                          }
15                  }
16
17                  return blocks;
18          }
```