

Tugas Kecil 2 IF2211

Strategi Algoritma

Kompresi Gambar Dengan

Algoritma Divide And Conquer

Menggunakan Metode Quadtree



Oleh:
Peter Wongsoredjo - 13523039

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132
2025

BAB I

Deskripsi Singkat Tugas

Di era digital ini, data visual seperti gambar dan juga video mendominasi lalu lintas informasi. Namun, seringkali ukuran file media yang terlalu besar akan menjadi hambatan dalam penyimpanan dan pengiriman data. Salah satu pendekatan untuk mengatasi hal ini adalah melalui kompresi gambar, dengan menyederhanakan representasi visual suatu gambar, dan menarik suatu gambar baru dengan harapan dapat mencerminkan informasi penting yang dimiliki oleh gambar original.

Tugas kecil ini didesain untuk mengajarkan implementasi salah satu strategi fundamental dalam desain algoritma, yaitu Divide and Conquer, yang dalam konteks ini digunakan untuk melakukan kompresi gambar menggunakan struktur Quadtree. Gambar berwarna akan dianalisis dan dibagi-bagi secara rekursif menjadi 4 block terus menerus, dengan block yang cukup kompleks akan terus terbagi, sedangkan block yang dianggap sudah seragam akan dikompresi menjadi satu block warna rata-rata.

Program yang dibentuk untuk tugas kecil ini tidak hanya menghasilkan gambar terkompresi, namun juga memiliki fitur tambahan berupa visualisasi animasi GIF, yang menampilkan proses kompresi gambar secara bertahap dari gambar yang sangat kasar hingga mendekati detail gambar kompresi asli. Dengan demikian, tugas ini menguji pemahaman terhadap rekursi dan algoritma Divide and Conquer yang disatukan dengan pengolahan citra digital.

BAB II

Penjelasan Algoritma Divide and Conquer

Strategi Divide and Conquer merupakan salah satu paradigma penyelesaian masalah yang bekerja dengan cara membagi sebuah masalah besar menjadi beberapa sub-masalah yang kecil, dan kemudian menyelesaikan masing masing sub-masalah tersebut (rekursif), dan menggabungkan semua hasilnya untuk memperoleh solusi dari masalah semula. Strategi Divide and Conquer dibagi menjadi tiga tahap utama, yaitu:

1. Divide: membagi input masalah menjadi dua atau lebih bagian yang lebih kecil dan serupa.
2. Conquer: menyelesaikan setiap bagian permasalahan, umumnya dengan memanggil kembali algoritma yang sama secara rekursif.
3. Combine: menggabungkan solusi solusi dari semua sub-masalah menjadi solusi akhir

Strategi Divide and Conquer dalam tugas kali ini diaplikasikan untuk menyelesaikan permasalahan kompresi gambar dengan menggunakan struktur Quadtree. Inti dari strategi ini sangatlah sederhana, berupa :

1. Membaca dan mengubah gambar input menjadi block (matrix) pixel

Untuk dapat memindai, menghitung, dan nantinya melakukan kompresi gambar dari gambar input, maka tahap inisialisasi dari algoritma akan berupa pemrosesan gambar menjadi data angka yang dapat dibaca, dalam konteks ini warna tiap pixel pada gambar dan dibuat dalam bentuk matriks.

- Gambar input akan dibaca dan setiap pixel yang terdapat di gambar input nantinya akan dikonversi ke dalam tiga matriks 2D berbeda (R,G,B), hal ini dicapai melalui kelas ImageProcessor dan Block (notasi pseudocode tambahan terdapat di bawah).
- Block, pada intinya adalah tipe data untuk menyimpan suatu pixel, dengan atribut red, green, blue (matriks 2D warna), dan width serta height pixel tersebut.
- ImageProcessor akan membaca sebuah image, dan untuk setiap pixel (x,y), setiap RGB nya akan dicatat di block block yang ada.

2. Hitung keseragaman block berdasarkan threshold

Untuk setiap block yang terbentuk dari pemrosesan gambar, dilakukan perhitungan nilai error menggunakan salah satu metode yang akan dipilih: Variance, MAD, Max Difference, atau Entropy. Dengan melakukan perhitungan tersebut, block akan ditentukan apabila dirinya seragam atau tidak. Algoritma lebih lanjut terdapat di pseudocode ErrorMeasurement di bagian setelah ini.

3. Evaluasi kondisi base case

Setelah nilai error diperoleh, akan dilakukan evaluasi mengenai block tersebut, apakah block akan menjadi suatu simpul daun (leaf) dalam Quadtree. Kondisi berhenti / base case akan dicapai ketika:

- Luas block (lebar x tinggi) lebih kecil atau sama dengan minBlockSize
- Luas sub-block (block yang nantinya dibagi menjadi 4) lebih kecil dari minBlockSize
- Nilai error berada di bawah threshold (block dianggap homogen / seragam)

Jika salah satu saja dari ketiga kondisi di atas sudah terpenuhi, maka proses rekursi akan dihentikan, dan node akan disimpan sebagai suatu leaf dengan warna rata-rata dari block tersebut.

4. Divide: Membagi Block menjadi empat sub-block

Sebuah block yang dapat melewati kondisi base case tersebut (block masih terlalu besar, dan tidak cukup seragam), maka sekarang block akan dibagi lagi menjadi empat sub-block kuadran: kiri atas, kanan atas, kiri bawah, dan kanan bawah. Untuk setiap sub-block yang nantinya terbentuk, masing masing akan di *Conquer*, dan nantinya di *Combine* untuk mendapatkan solusi akhir. Proses ini dilakukan dengan membentuk empat sub-block, masing masing dengan startX, startY, width, dan height nya (penjelasan lebih lanjut terdapat di notasi algoritma di bawah).

5. Conquer: Memproses setiap sub-block secara rekursif

Setiap sub-block yang terbentuk setelah *Divide*, akan diproses kembali menggunakan langkah langkah dari nomor 2 hingga nomor 5 ini secara rekursif. Melalui rekursi ini, algoritma akan menelusuri lebih lanjut, bagian bagian gambar yang lebih kecil, dan akan melakukan pembagian lebih lanjut jika memang nantinya dibutuhkan. Proses ini dilakukan dengan membuat node leaf, dari keempat sub-block yang masing masing akan di recursiveCompress lagi.

6. Combine: Membentuk simpul induk dan struktur pohon

Setelah seluruh sub-block selesai diproses dan masing masing menghasilkan node (leaf maupun tidak), seluruh node tersebut nantinya akan dihubungkan kembali ke simpul induknya. Proses ini dilakukan dengan menyusun pointer anak (leaf) ke dalam objek *QuadTreeNode* sebagai root nya. Dengan demikian, suatu struktur Quadtree yang mencerminkan gambar awal, dari akar hingga daun dapat terbentuk, beserta segala data dan informasi warna yang disimpan.

Keseluruhan langkah-langkah tersebut, diimplementasikan pada algoritma nantinya melalui fungsi *recursiveCompress*, yang kurang lebih akan merepresentasikan algoritma Divide and Conquer, dengan pseudocode seperti berikut:

Pseudocode Divide and Conquer

```
QuadTreeNode recursiveCompress(input B : Block, x, y, width, height, minBlocksize:  
integer, threshold: real, method: ErrorMethod) → QuadTreeNode
```

{Menyelesaikan kompresi gambar dengan strategi Divide and Conquer berbasis Quadtree
Masukan: Block B, dengan koordinat awal (x, y), ukuran block(width, height), dan
parameter ukuran block minimum, serta metode yang digunakan untuk valuasi error
Keluaran: Simpul node Quadtree hasil yang mencerminkan block B}

Deklarasi

```
error : real  
avgRGB : array[1..3] of real  
avgRed, avgGreen, avgBlue, newWidth, newHeight, area, newArea : integer  
child[1..4], parentNode : QuadTreeNode
```

Algoritma

```
{Evaluasi Kondisi Base Case}
```

```
error ← calculateError(B, method)  
avgRGB ← getAverageRGB(B)  
avgRed ← round(avgRGB[0])  
avgGreen ← round(avgRGB[1])  
avgBlue ← round(avgRGB[2])
```

```
newWidth ← width / 2  
newHeight ← height / 2  
area ← width × height  
newArea ← newWidth × newHeight
```

```
{Kondisi Base Case}
```

```
if (area ≤ minBlockSize) or (newArea < minBlockSize) or (error ≤ threshold) then  
    → QuadTreeNode (x, y, width, height, avgRed, avgGreen, avgBlue)
```

{DIVIDE}

```
B1 ← getSubBlock(B, 0, 0, newWidth, newHeight) { top-left }  
B2 ← getSubBlock(B, newWidth, 0, width - newWidth, newHeight) { top-right }  
B3 ← getSubBlock(B, 0, newHeight, newWidth, height - newHeight) { bottom-left }  
B4 ← getSubBlock(B, newWidth, newHeight, width - newWidth, height - newHeight)  
{ bottom-right }
```

{CONQUER}

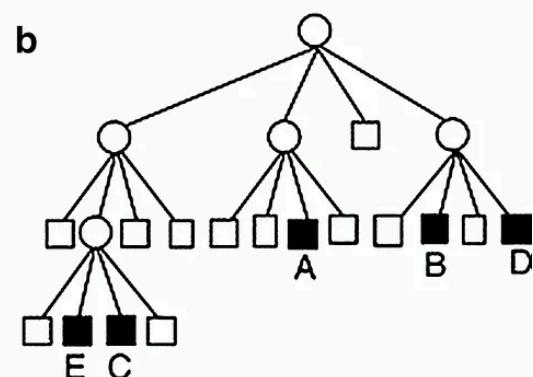
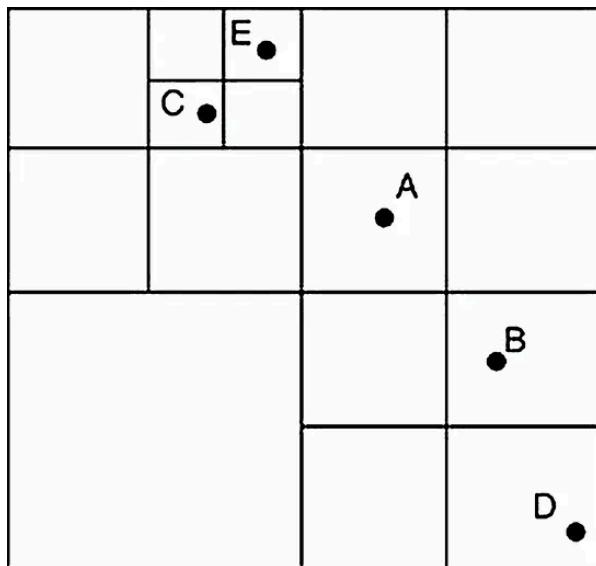
```
child[1] ← recursiveCompress(B1, x, y, newWidth, newHeight, minBlockSize,  
threshold, method)  
child[2] ← recursiveCompress(B2, x + newWidth, y, width - newWidth, newHeight,  
minBlockSize, threshold, method)  
child[3] ← recursiveCompress(B3, x, y + newHeight, newWidth, height - newHeight,  
minBlockSize, threshold, method)  
child[4] ← recursiveCompress(B4, x + newWidth, y + newHeight, width - newWidth,  
height - newHeight, minBlockSize, threshold, method)
```

```

{COMBINE}
parentNode ← node quadtree induk (x, y, width, height, avgRed, avgGreen, avgBlue)
setChildren(parentNode, child)
→ parentNode

```

atau ketika pseudocode divisualisasikan akan seperti berikut:



Tentunya pseudocode di atas tidak akan berfungsi tanpa adanya bantuan kode dari pseudocode di bawah ini:

getAverageRGB() dari class Block

```

function getAverageRGB(Block) → array[1..3] of real
    totalPixels ← width × height
    average[0..2] ← 0
    for i ← 0 to width - 1 do
        for j ← 0 to height - 1 do
            average[0] ← average[0] + red[i][j]
            average[1] ← average[1] + green[i][j]
            average[2] ← average[2] + blue[i][j]
    for k ← 0 to 2 do
        average[k] ← average[k] ÷ totalPixels
    → average

```

getSubBlock() dari kelas Block

```

function getSubBlock(startX, startY, subWidth, subHeight: integer) → Block
    subRed[subWidth][subHeight],

```

```

subGreen[subWidth][subHeight],
subBlue[subWidth][subHeight]
for i  $\leftarrow$  0 to subWidth - 1 do
    for j  $\leftarrow$  0 to subHeight - 1 do
        subRed[i][j]  $\leftarrow$  red[startX + i][startY + j]
        subGreen[i][j]  $\leftarrow$  green[startX + i][startY + j]
        subBlue[i][j]  $\leftarrow$  blue[startX + i][startY + j]
    → Block(subRed, subGreen, subBlue)

```

QuadTreeNode constructor

```

QuadTreeNode(x, y, width, height, avgRed, avgGreen, avgBlue)
    this.x  $\leftarrow$  x
    this.y  $\leftarrow$  y
    this.width  $\leftarrow$  width
    this.height  $\leftarrow$  height
    this.avgRed  $\leftarrow$  avgRed
    this.avgGreen  $\leftarrow$  avgGreen
    this.avgBlue  $\leftarrow$  avgBlue
    this.children  $\leftarrow$  null

```

calculateError() dari ErrorMeasurement

```

function calculateError(block : Block, method : ErrorMethod)  $\rightarrow$  double
    switch method do
        case VARIANCE:
             $\rightarrow$  VARIANCE(block)
        case MAD:
             $\rightarrow$  MAD(block)
        case MAX_DIFFERENCE:
             $\rightarrow$  MAX_PIXEL_DIFFERENCE(block)
        case ENTROPY:
             $\rightarrow$  ENTROPY(block)

```

Penjelasan secara menyeluruh dari setiap kelas, dan juga program perhitungan error tidak akan dimasukkan ke dalam penjelasan di bab ini, dan dapat dilihat langsung saja di bagian selanjutnya, karena akan terlalu panjang dan redundant.

BAB III

Implementasi

Pengerjaan Tugas Kecil 2 ini dikerjakan dengan menggunakan bahasa pemrograman Java, dengan kode sebagai berikut:

Block.java

```
public class Block {  
    private int [][] red;  
    private int [][] green;  
    private int [][] blue;  
    private int width;  
    private int height;  
  
    public Block(int [][] red, int [][] green, int [][] blue) {  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
        this.width = red.length;  
        this.height = red[0].length;  
    }  
  
    public int[][] getRed() {  
        return red;  
    }  
  
    public int[][] getGreen() {  
        return green;  
    }  
  
    public int[][] getBlue() {  
        return blue;  
    }  
  
    public int getWidth() {  
        return width;  
    }  
}
```

```
public int getHeight() {
    return height;
}

public int getPixelRed(int x, int y) {
    return red[x][y];
}

public int getPixelGreen(int x, int y) {
    return green[x][y];
}

public int getPixelBlue(int x, int y) {
    return blue[x][y];
}

public double[] getAverageRGB(){
    double[] average = new double[3];
    int totalPixels = width * height;
    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            average[0] += red[i][j];
            average[1] += green[i][j];
            average[2] += blue[i][j];
        }
    }
    average[0] /= totalPixels;
    average[1] /= totalPixels;
    average[2] /= totalPixels;
    return average;
}

public Block getSubBlock(int startX, int startY, int subWidth, int subHeight){
    int[][][] subRed = new int[subWidth][subHeight];
    int[][][] subGreen = new int[subWidth][subHeight];
    int[][][] subBlue = new int[subWidth][subHeight];

    for(int i = 0; i < subWidth; i++){
        for(int j = 0; j < subHeight; j++){
            subRed[i][j] = red[startX + i][startY + j];
        }
    }
}
```

```
        subGreen[i][j] = green[startX + i][startY + j];
        subBlue[i][j] = blue[startX + i][startY + j];
    }
}
return new Block(subRed, subGreen, subBlue);
}
}
```

ImageProcessor.java

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class ImageProcessor {
    public static Block imageToBlock(String path) throws IOException{
        BufferedImage image = ImageIO.read(new File(path));
        int width = image.getWidth();
        int height = image.getHeight();
        int[][][] red = new int[width][height];
        int[][][] green = new int[width][height];
        int[][][] blue = new int[width][height];

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                int rgb = image.getRGB(x, y);
                red[x][y] = (rgb >> 16) & 0xFF;
                green[x][y] = (rgb >> 8) & 0xFF;
                blue[x][y] = rgb & 0xFF;
            }
        }

        return new Block(red, green, blue);
    }

    private static void drawNode(BufferedImage image, QuadTreeNode node) {
        int x = node.getX();
        int y = node.getY();
        int width = node.getWidth();
        int height = node.getHeight();
        int rgb = (node.getAvgRed() << 16) | (node.getAvgGreen() << 8) |
node.getAvgBlue();

        if(node.isLeaf()){
            for (int i = x; i < x + width; i++) {
                for (int j = y; j < y + height; j++) {
                    image.setRGB(i, j, rgb);
                }
            }
        }
    }
}
```

```
        }
    }
    else {
        for (QuadTreeNode child : node.getChildren()) {
            drawNode(image, child);
        }
    }
}

public static BufferedImage renderImage(QuadTreeNode root, int width,
int height) {
    BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
    drawNode(image, root);
    return image;
}

public static void saveImage(BufferedImage image, String path) throws
IOException {
    File outputfile = new File(path);
    ImageIO.write(image, "png", outputfile);
}
}
```

QuadTreeNode.java

```
public class QuadTreeNode {
    private int x, y, width, height;
    private int avgRed, avgGreen, avgBlue;
    private QuadTreeNode[] children; // ada 4 children kalo punya

    public QuadTreeNode(int x, int y, int width, int height, int avgRed, int
avgGreen, int avgBlue) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.avgRed = avgRed;
        this.avgGreen = avgGreen;
        this.avgBlue = avgBlue;
        this.children = null; // Awal awal ngga ada children
    }

    public void setChildren(QuadTreeNode[] children) {
        this.children = children;
    }

    public boolean isLeaf(){
        return children == null;
    }

    public QuadTreeNode[] getChildren() {
        return children;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getWidth() {
        return width;
    }
```

```
}

public int getHeight() {
    return height;
}

public int getAvgRed() {
    return avgRed;
}

public int getAvgGreen() {
    return avgGreen;
}

public int getAvgBlue() {
    return avgBlue;
}

public int getDepth(){
    if(isLeaf()){
        return 0;
    }
    int maxDepth = 0;
    for(QuadTreeNode child : children){
        maxDepth = Math.max(maxDepth, child.getDepth());
    }
    return maxDepth + 1;
}

public int countNodes(){
    if(isLeaf()){
        return 1;
    }
    int count = 1; // hitung diri sendiri
    for(QuadTreeNode child : children){
        count += child.countNodes();
    }
    return count;
}
}
```

ErrorMeasurement.java

```
import java.util.Arrays;

public class ErrorMeasurement {
    // Variance
    private static double varianceChannel(int[][][] channel){
        int height = channel[0].length;
        int width = channel.length;
        int N = height * width;
        double mean = 0;
        double sum = 0;
        double sumSquared = 0;
        for(int i = 0; i < width; i++){
            for(int j = 0; j < height; j++){
                sum += channel[i][j];
            }
        }
        mean = sum / N;
        for(int i = 0; i < width; i++){
            for(int j = 0; j < height; j++){
                sumSquared += Math.pow(channel[i][j] - mean, 2);
            }
        }
        double variance = sumSquared / N;
        return variance;
    }

    private static double variance(Block block){
        return (varianceChannel(block.getRed()))
            + varianceChannel(block.getGreen())
            + varianceChannel(block.getBlue()) / 3;
    }

    // MAD
    private static double madChannel(int[][][] channel){
        int height = channel.length;
        int width = channel[0].length;
        double sum = 0;
        int N = height * width;
        double mean = Arrays.stream(channel)
```

```

        .flatMapToInt((Arrays::stream)
        .average().orElse(0);

    for (int[] row : channel) {
        for (int value : row) {
            sum += Math.abs(value - mean);
        }
    }

    return sum / N;
}

private static double mad(Block block){
    return (madChannel(block.getRed())
        + madChannel(block.getGreen())
        + madChannel(block.getBlue())) / 3;
}

// Max Pixel Difference
private static double maxPixelDifferenceChannel(int[][][] channel){
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;

    for (int[] row : channel) {
        for (int value : row) {
            max = Math.max(max, value);
            min = Math.min(min, value);
        }
    }

    return max - min;
}

private static double maxPixelDifference(Block block){
    return (maxPixelDifferenceChannel(block.getRed())
        + maxPixelDifferenceChannel(block.getGreen())
        + maxPixelDifferenceChannel(block.getBlue())) / 3;
}

// Entropy
private static double entropyChannel(int[][][] channel){

```

```

int[] histogram = new int[256];
int total = 0;

for (int[] row : channel) {
    for (int value : row) {
        histogram[value]++;
        total++;
    }
}

double entropy = 0.0;
for (int i = 0; i < histogram.length; i++) {
    if (histogram[i] > 0) {
        double probability = (double) histogram[i] / total;
        entropy -= probability * Math.log(probability) /
Math.log(2);
    }
}
return entropy;
}

private static double entropy(Block block){
    return (entropyChannel(block.getRed())
    + entropyChannel(block.getGreen())
    + entropyChannel(block.getBlue())) / 3;
}

// Helper
public enum ErrorMethod {
    VARIANCE,
    MAD,
    MAX_DIFFERENCE,
    ENTROPY
}

public static double calculateError(Block block, ErrorMethod method) {
    switch (method) {
        case VARIANCE:
            return variance(block);
        case MAD:
            return mad(block);
    }
}

```

```
        case MAX_DIFFERENCE:
            return maxPixelDifference(block);
        case ENTROPY:
            return entropy(block);
        default:
            throw new IllegalArgumentException("Metode apa ini
Kapten?");
    }
}
```

Compressor.java

```
public class Compressor {
    private Block Image;
    private ErrorMeasurement.ErrorMethod method;
    private double threshold;
    private int minBlockSize;

    public Compressor(Block image, ErrorMeasurement.ErrorMethod method,
double threshold, int minBlockSize) {
        this.Image = image;
        this.method = method;
        this.threshold = threshold;
        this.minBlockSize = minBlockSize;
    }

    public QuadTreeNode compress(){
        int width = Image.getWidth();
        int height = Image.getHeight();
        return recursiveCompress(0, 0, width, height, Image);
    }

    public QuadTreeNode recursiveCompress(int x, int y, int width, int
height, Block block){
        double error = ErrorMeasurement.calculateError(block, method);

        // <= threshold brarti seragam
        boolean seragam = error <= threshold;
        double[] avgRGB = block.getAverageRGB();
        int avgRed = (int) Math.round(avgRGB[0]);
        int avgGreen = (int) Math.round(avgRGB[1]);
        int avgBlue = (int) Math.round(avgRGB[2]);

        // Kalo ngga seragam, ya baju bebas (bagi empat)
        int newWidth = width / 2;
        int newHeight = height / 2;

        int area = width * height;
        int newArea = newWidth * newHeight;

        if(area <= minBlockSize || newArea < minBlockSize || seragam) {
```

```

        //System.out.println("Block (" + x + ", " + y + ", " + width +
", " + height + ") is uniform with error: " + error);
        return new QuadTreeNode(x, y, width, height, avgRed, avgGreen,
avgBlue);
    }

    Block topLeft = block.getSubBlock(0, 0, newWidth, newHeight);
    Block topRight = block.getSubBlock(newWidth, 0, width - newWidth,
newHeight);
    Block bottomLeft = block.getSubBlock(0, newHeight, newWidth, height -
newHeight);
    Block bottomRight = block.getSubBlock(newWidth, newHeight, width -
newWidth, height - newHeight);

    QuadTreeNode topLeftNode = recursiveCompress(x, y, newWidth,
newHeight, topLeft);
    QuadTreeNode topRightNode = recursiveCompress(x + newWidth, y, width -
newWidth, newHeight, topRight);
    QuadTreeNode bottomLeftNode = recursiveCompress(x, y + newHeight,
newWidth, height - newHeight, bottomLeft);
    QuadTreeNode bottomRightNode = recursiveCompress(x + newWidth, y +
newHeight, width - newWidth, height - newHeight, bottomRight);

    QuadTreeNode parentNode = new QuadTreeNode(x, y, width, height,
avgRed, avgGreen, avgBlue);
    parentNode.setChildren(new QuadTreeNode[]{topLeftNode, topRightNode,
bottomLeftNode, bottomRightNode});
    return parentNode;
}

}

```

Main.java

```

// Main.java (Improved CLI with visual formatting)
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class Main {

```

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    printHeader();

    try {
        System.out.print("Masukkan path gambar: ");
        String path = scanner.nextLine();

        System.out.println("\n|-----|");
        System.out.println(" | PILIH METODE ERROR |");
        System.out.println(" |-----|");
        System.out.println(" | 1. Variance |");
        System.out.println(" | 2. MAD (Mean Abs Deviation) |");
        System.out.println(" | 3. Max Pixel Difference |");
        System.out.println(" | 4. Entropy |");
        System.out.println(" |-----|");

        System.out.print("Pilihan (1-4): ");
        int methodChoice = scanner.nextInt();
        scanner.nextLine();

        ErrorMeasurement.ErrorMethod method = switch (methodChoice) {
            case 1 -> ErrorMeasurement.ErrorMethod.VARIANCE;
            case 2 -> ErrorMeasurement.ErrorMethod.MAD;
            case 3 -> ErrorMeasurement.ErrorMethod.MAX_DIFFERENCE;
            case 4 -> ErrorMeasurement.ErrorMethod.ENTROPY;
            default -> throw new IllegalArgumentException("Pilihan tidak valid.");
        };

        System.out.print("Masukkan threshold: ");
        double threshold = scanner.nextDouble();

        System.out.print("Masukkan ukuran minimum blok (berbasis luas area): ");
        int minBlockSize = scanner.nextInt();
        scanner.nextLine();

        System.out.print("Masukkan path untuk menyimpan gambar hasil kompresi (PNG): ");
        String outputPath = scanner.nextLine();
```

```
System.out.print("Apakah Anda ingin menghasilkan GIF juga?  
(y/n): ");
String gifChoice = scanner.nextLine().trim().toLowerCase();

String gifPath = null;
String frameFolderPath = new
File(System.getProperty("user.dir"), "frames").getAbsolutePath();

if (gifChoice.equals("y")) {
    System.out.print("Masukkan output path GIF (contoh:  
hasil.gif): ");
    gifPath = scanner.nextLine();
}

long startRead = System.currentTimeMillis();
Block image = ImageProcessor.imageToBlock(path);
long endRead = System.currentTimeMillis();

long start = System.currentTimeMillis();
Compressor compressor = new Compressor(image, method, threshold,
minBlockSize);
QuadTreeNode root = compressor.compress();
long end = System.currentTimeMillis();

BufferedImage outputImage = ImageProcessor.renderImage(root,
image.getWidth(), image.getHeight());
ImageProcessor.saveImage(outputImage, outputPath);
System.out.println("\nGambar hasil kompresi disimpan di: " +
outputPath);

if (gifChoice.equals("y")) {
    System.out.println("\nMembuat GIF dari beberapa tingkat
kompresi...");;
    GifGenerator.generateFramesByBlockSize(image, method,
threshold, minBlockSize);
    ImageProcessor.saveImage(outputImage, frameFolderPath +
"/frame_final.png");
    GifGenerator.generateGifFromFolder(frameFolderPath, gifPath,
500);
    System.out.println("GIF berhasil dibuat di: " + gifPath);
```

```
}

long originalSize = new File(path).length();
long compressedSize = new File(outputPath).length();

double compressionRatio = 100.0 * (1.0 - ((double)
compressedSize / originalSize));
int depth = root.getDepth();
int nodeCount = root.countNodes();

System.out.println("\n=====
=====");
System.out.println("                               STATISTIK KOMPRESI
");

System.out.println("=====
=====");
System.out.printf("Waktu load gambar      : %d ms\n", (endRead -
startRead));
System.out.printf("Waktu kompresi          : %d ms\n", (end -
start));
System.out.printf("Ukuran gambar awal     : %d bytes\n",
originalSize);
System.out.printf("Ukuran gambar hasil    : %d bytes\n",
compressedSize);
System.out.printf("Persentase kompresi    : %.2f%%\n",
compressionRatio);
System.out.printf("Kedalaman pohon        : %d\n", depth);
System.out.printf("Jumlah simpul quadtree: %d\n", nodeCount);

System.out.println("=====
=====");

} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
} finally {
    scanner.close();
}
}
```

```
private static void printHeader() {  
  
    System.out.println("||");  
    System.out.println("|| IMAGE COMPRESSOR  
||");  
  
    System.out.println("||");  
    System.out.println("|| Kompresi gambar + GIF  
||");  
  
    System.out.println("||\n");  
}  
}
```

Implementasi Bonus

GifGenerator.java

```
import javax.imageio.ImageIO;
import javax.imageio.ImageWriter;
import javax.imageio.metadata.IIOMetadata;
import javax.imageio.metadata.IIOMetadataNode;
import javax.imageio.stream.ImageOutputStream;
import javax.imageio.IIOImage;
import javax.imageio.ImageTypeSpecifier;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Locale;

public class GifGenerator {

    public static void generateGifFromFolder(String inputFolderPath, String
outputGifPath, int delayMs) throws IOException {
        File folder = new File(inputFolderPath);
        File[] imageFiles = folder.listFiles((dir, name) ->
name.toLowerCase().endsWith(".png") || name.toLowerCase().endsWith(".jpg"));

        if (imageFiles == null || imageFiles.length == 0) {
            throw new IOException("No image files found in the folder.");
        }

        ImageWriter gifWriter =
ImageIO.getImageWritersByFormatName("gif").next();
        ImageOutputStream output = ImageIO.createImageOutputStream(new
File(outputGifPath));
        gifWriter.setOutput(output);
        gifWriter.prepareWriteSequence(null);

        for (int i = 0; i < imageFiles.length; i++) {
            BufferedImage frame = ImageIO.read(imageFiles[i]);
            IIOMetadata metadata = getFrameMetadata(gifWriter, frame,
delayMs, i == 0);
            gifWriter.writeToSequence(new IIOImage(frame, null, metadata),
null);
        }
    }
}
```

```
    }

    gifWriter.endWriteSequence();
    output.close();
}

private static IIOMetadata getFrameMetadata(ImageWriter writer,
BufferedImage image, int delayTimeMs, boolean isFirst) throws IOException {
    ImageTypeSpecifier type =
ImageTypeSpecifier.createFromRenderedImage(image);
    IIOMetadata metadata = writer.getDefaultImageMetadata(type, null);
    String metaFormat = metadata.getNativeMetadataFormatName();
    IIOMetadataNode root = (IIOMetadataNode)
metadata.getAsTree(metaFormat);

    IIOMetadataNode gce = getOrCreateNode(root,
"GraphicControlExtension");
    gce.setAttribute("disposalMethod", "none");
    gce.setAttribute("userInputFlag", "FALSE");
    gce.setAttribute("transparentColorFlag", "FALSE");
    gce.setAttribute("delayTime", Integer.toString(delayTimeMs / 10));
    gce.setAttribute("transparentColorIndex", "0");

    if (isFirst) {
        IIOMetadataNode appExtensions = getOrCreateNode(root,
"ApplicationExtensions");
        IIOMetadataNode appNode = new
IIOMetadataNode("ApplicationExtension");
        appNode.setAttribute("applicationID", "NETSCAPE");
        appNode.setAttribute("authenticationCode", "2.0");
        appNode.setUserObject(new byte[]{1, 0, 0});
        appExtensions.appendChild(appNode);
    }

    metadata.setFromTree(metaFormat, root);
    return metadata;
}

private static IIOMetadataNode getOrCreateNode(IIOMetadataNode root,
String name) {
    for (int i = 0; i < root.getLength(); i++) {
```

```

        if (root.item(i).getNodeName().equalsIgnoreCase(name)) {
            return (IIOMetadataNode) root.item(i);
        }
    }
    IIOMetadataNode node = new IIOMetadataNode(name);
    root.appendChild(node);
    return node;
}

public static void generateFramesByBlockSize(Block image,
ErrorMeasurement.ErrorMethod method, double threshold, int minBlockSize)
throws IOException {
    int width = image.getWidth();
    int height = image.getHeight();
    int maxBlockArea = (height/2) * ((width+2)/2); // min area untuk
gambar 1 pixel

    //inputan
    Compressor baseCompressor = new Compressor(image, method, threshold,
minBlockSize);
    QuadTreeNode fullTree = baseCompressor.compress();
    int maxDepth = fullTree.getDepth();

    String folderPath = new File(System.getProperty("user.dir"),
"frames").getAbsolutePath();
    File folder = new File(folderPath);
    if (!folder.exists()) {
        folder.mkdirs();
    }

    double logStart = Math.log(minBlockSize);
    double logEnd = Math.log(maxBlockArea);
    double logStep = (logStart - logEnd) / (maxDepth - 1);

    for (int i = 0; i < maxDepth; i++) {
        double logValue = logStart - (maxDepth - i) * logStep;
        int currentMinBlockArea = (int) Math.exp(logValue);
        //System.out.println("Current min block area: " +
currentMinBlockArea);
        Compressor compressor = new Compressor(image, method, threshold,
currentMinBlockArea);
    }
}

```

```

        QuadTreeNode tree = compressor.compress();
        BufferedImage frame = ImageProcessor.renderImage(tree, width,
height);
        String filename = String.format(Locale.US, "%s/frame_%02d.png",
folderPath, i);
        ImageProcessor.saveImage(frame, filename);
    }
}
}
}

```

GifGenerator bekerja dalam menghasilkan visualisasi dari proses kompresi citra yang terjadi secara bertahap. Proses ini memiliki dua peran utama: membentuk serangkaian frame hasil kompresi dan menyusunnya menjadi satu file animasi GIF. Langkah-langkah yang dilakukan GifGenerator.java adalah sebagai berikut:

1. Membentuk Frame Akhir (Gambar Hasil sesuai input user):

Langkah pertama yang dilakukan adalah membuat hasil akhir kompresi berdasarkan parameter masukan dari pengguna, yaitu threshold dan minBlockSize. Kompresor utama akan membentuk QuadTree dengan kedalaman tertentu, yang akan menjadi referensi dalam pembentukan GIF.

2. Menentukan Nilai minBlockSize Terbesar:

Sistem akan menghitung nilai ukuran BlockSize minimum untuk membuat gambar menjadi 1 pixel (ukuran tetap sama, namun semua pixel memiliki warna yang sama). Nilai ini dihitung berdasarkan ukuran BlockSize yang lebih besar dari area potensi anak block ($\text{BlockSize} = ((\text{width}+2)/2) \times (\text{height}/2)$) karena $\text{newArea} < \text{blockSize}$ akan ter-trigger.

3. Mengambil Nilai depth dari Pohon Kompresi:

Melalui nilai kedalaman dari pohon kompresi yang telah kita temukan sebelumnya, depth pohon akan digunakan sebagai jumlah frame GIF yang akan dibentuk. Makin dalam pohon yang terbentuk, makin banyak pula iterasi pembentukan frame, sehingga transisi tetap terlihat mulus.

4. Menghitung Distribusi Nilai minBlockSize dengan Logaritma:

Supaya hasil kompresi dari setiap frame meningkat secara berkala dan terlihat mulus, nilai minBlockSize diatur dengan skala logaritmik, dengan setiap iterasi, BlockSize yang digunakan sebagai parameter tidak akan tumbuh secara linear, melainkan secara eksponensial, ketika menuju ke skala normal. Perubahan minBlockSize akan mengikuti kurva eksponensial terbalik, sehingga:

- Di awal, perubahan antara 1 frame ke frame berikutnya tetap terasa besar (karena perubahan besar tidak terlalu terasa secara visual)
- Di akhir, perubahan kecil akan tetap terasa halus dan bertahap (karena perbedaan warna dan detail sudah signifikan)

5. Membentuk Setiap Frame dan Menyimpannya di Folder frames/:

Setelah iterasi pembentukan tiap frame, semua frame (gambar dikompres) ini akan disimpan ke dalam folder frames.

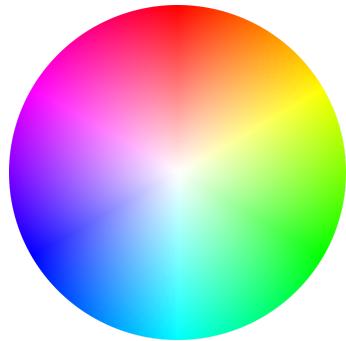
6. Menggabungkan Seluruh Frame Menjadi GIF:

Setelah seluruh frame terbentuk, seluruh file gambar (default png) di folder frames akan dibentuk menjadi file GIF animasi, dengan urutan dari gambar paling terkompres (satu blok besar) hingga ke gambar akhir yang merupakan gambar hasil algoritma terhadap input user.

BAB IV

Pengujian dan Analisis

1. RGB Test



Spesifikasi Input: 1200 x 1200, 582 Kb, minBlockSize = 4

Kualitas	Variance	MAD	Max Difference	Entropy
High Quality	Threshold: 15 Rate: 75.99%	Threshold: 5 Rate: 83.48%	Threshold: 10 Rate: 73.16%	Threshold: 0.1 Rate: 33.94%
Mid Quality	Threshold: 100 Rate: 88.86%	Threshold: 15 Rate: 91.27%	Threshold: 50 Rate: 88.90%	Threshold: 2 Rate: 83.92%
Low Quality	Threshold: 300 Rate: 91.02%	Threshold: 50 Rate: 96.44%	Threshold: 100 Rate: 90.65%	Threshold: 4 Rate: 92.76%

2. Black and White Test



Spesifikasi Input: 256x256, 22 Kb, minBlockSize = 4

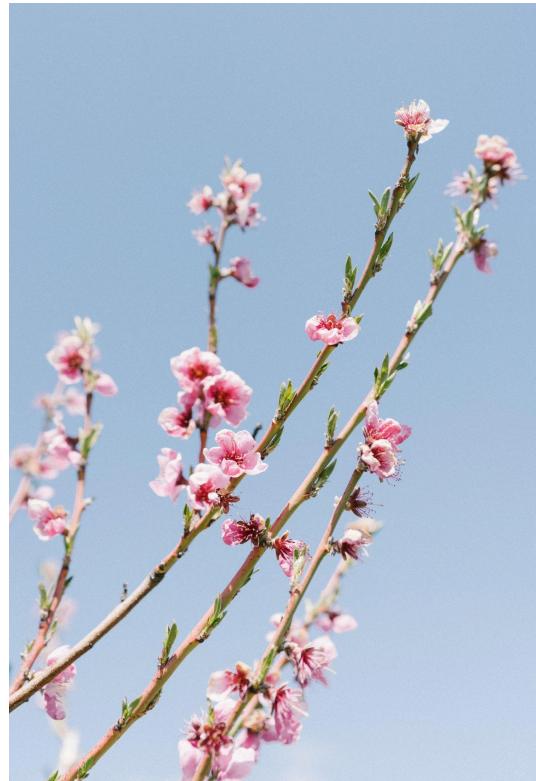
Kualitas	Variance	MAD	Max Difference	Entropy
High Quality	 Threshold: 100 Rate: 45.94%	 Threshold: 5 Rate: 35.06%	 Threshold: 50 Rate: 46.05%	 Threshold: 3 Rate: 23.69%
Mid Quality	 Threshold: 500 Rate: 61.28%	 Threshold: 15 Rate: 59.83%	 Threshold: 100 Rate: 57.45%	 Threshold: 4 Rate: 67.08%
Low Quality	 Threshold: 1000 Rate: 74.58%	 Threshold: 20 Rate: 73.19%	 Threshold: 150 Rate: 75.86%	 Threshold: 5 Rate: 78.96%

Berdasarkan pengujian yang dilakukan terhadap kedua test RGB dan juga Black and White (BnW), dapat terlihat melalui keseluruhan nilai threshold, ketika menguji gambar yang memiliki warna BnW, membutuhkan threshold yang lebih tinggi untuk melakukan kompresi terhadap gambar tersebut. Hal ini dikarenakan pixel hitam akan bernilai 0, dan pixel putih akan bernilai 255 pada skala RGB, sehingga suatu gambar dengan pixel hitam dan putih saja akan memiliki jarak yang sangat jauh dalam skala RGB. Jarak yang jauh inilah yang akan membuat perhitungan error menjadi lebih tinggi, menyebabkan gambar lebih sering dibelah lagi. Meskipun RGB yang diuji merupakan spektrum warna dengan banyaknya warna warna kontras, gambar BnW masih mengungguli nilai threshold yang dimiliki gambar BnW, menunjukkan tingginya nilai error di gambar BnW. Sehingga, untuk mengimbangi error yang semakin tinggi, maka threshold juga harus semakin tinggi, dan hal ini terbukti di semua metode. Sedangkan untuk analisis mendalam terhadap masing masing metode:

- Variance: mengukur penyebaran nilai pixel terhadap rata rata, sehingga jika hanya salah satu pixel saja jauh berbeda dari rata-rata, variance bisa langsung naik drastis. Oleh karena variance sangat sensitif terhadap outlier, skala nilai threshold nya bisa mencapai ribuan untuk block yang besar, sehingga nilai threshold harus tinggi agar tidak over-split.
- MAD: menghitung rata-rata dari selisih absolut terhadap mean. MAD lebih tahan terhadap outlier, karena representasi yang lebih stabil dari persebaran nilai. Dengan skala yang biasanya lebih kecil, MAD tidak terlalu agresif memecah block, dan threshold bisa lebih rendah dari variance.
- MPD: mengukur jarak antara pixel maximum dan pixel minimum dalam satu block. Dengan hanya mengambil dua nilai ekstrim, MPD tidak peduli berapa banyak nilai ekstrem yang muncul. MPD masih saja bisa tinggi walau hanya satu pixel yang berbeda, sehingga walaupun tidak sebesar variance, threshold idealnya berada di angka yang relatif cukup tinggi.
- Entropy: mengukur keragaman distribusi nilai pixel (histogram), dan oleh karena itu nilai threshold akan rendah jika hanya sedikit nilai RGB yang muncul, dan bisa tinggi jika banyak nilai piksel unik yang tersebar merata. Threshold untuk entropi adalah yang paling rendah diantara metode lainnya, dan cenderung stabil.

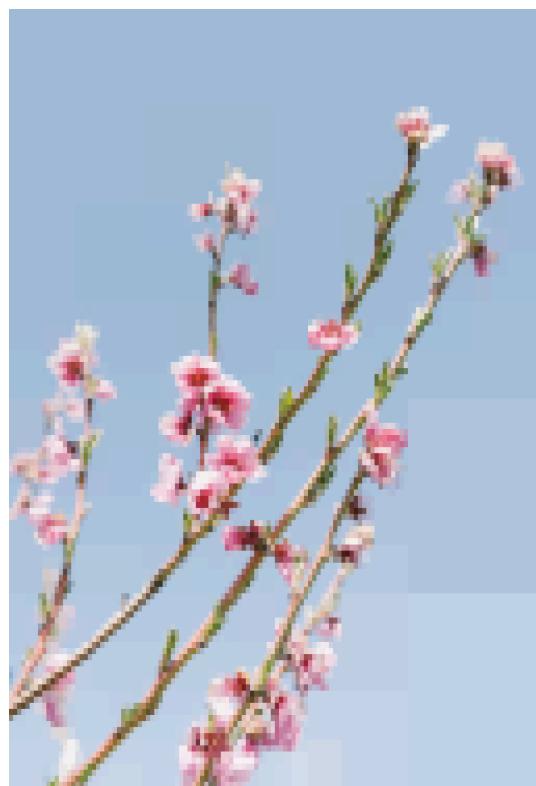
Informasi tersebut berpengaruh ke algoritma Divide and Conquer, dengan pembagian yang berdasarkan threshold, kita harus dapat memilih angka threshold yang baik untuk dapat mendapatkan hasil gambar kompresi yang terbaik. Dengan demikian, sekarang kita dapat melakukan tes terhadap suatu variabel konstan untuk menguji algoritma Divide and Conquer:

3. Flowers.jpg



Flowers.jpg akan dijadikan standar pengujian gambar, dengan ukuran 3322 x 4878 dan ukuran sekitar 2.6 MB, berikut adalah pengujian untuk setiap metode:

Variance

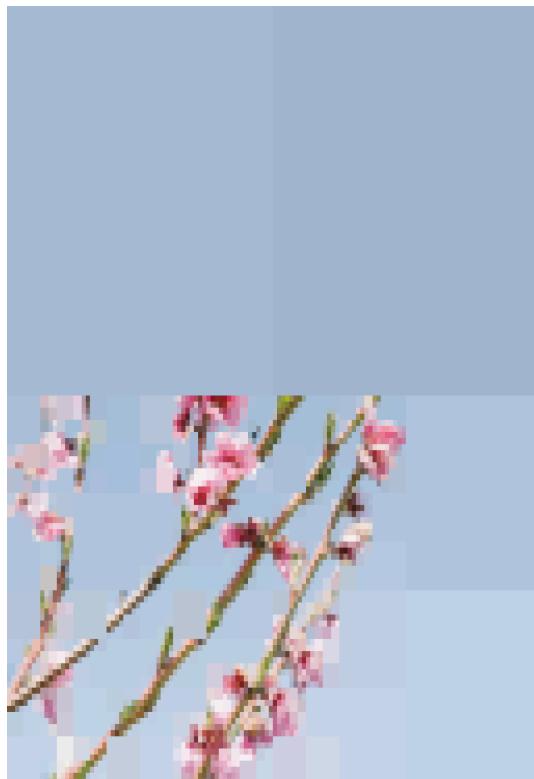


Dengan threshold 100, dan minBlockSize 512 (karena ukuran gambar yang besar)

STATISTIK KOMPRESI

Waktu load gambar	:	1563 ms
Waktu kompresi	:	1299 ms
Ukuran gambar awal	:	2667516 bytes
Ukuran gambar hasil	:	684676 bytes
Persentase kompresi	:	74.33%
Kedalaman pohon	:	7
Jumlah simpul quadtree	:	6489

Mean Absolute Deviation (MAD)

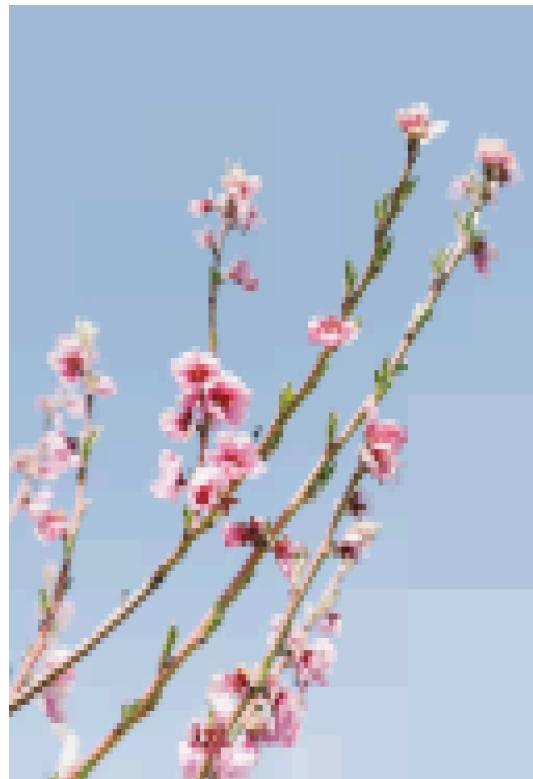


Ketika dicoba untuk melakukan kompresi dengan menggunakan MAD pada flowers.jpg, dengan threshold 15, dan juga minBlockSize 512, maka didapatkan gambar berikut, dengan setengah dari bagian gambar hilang karena kompresi. Hal ini mungkin disebabkan oleh karena sedikitnya kontras gambar pada setengah atas bagian gambar, dengan sedikitnya warna hijau dan juga pink, menyebabkan error yang dicapai tidak melebihi threshold, sehingga gambar kemudian disatukan dengan warna rata rata yaitu biru langit.

STATISTIK KOMPRESI

Waktu load gambar : 2106 ms
Waktu kompresi : 761 ms
Ukuran gambar awal : 2667516 bytes
Ukuran gambar hasil : 411309 bytes
Persentase kompresi : 84.58%
Kedalaman pohon : 7
Jumlah simpul quadtree: 3217

Dengan mengganti gambar menjadi kompresi tinggi, kita bisa mendapatkan gambar yang sebenarnya serupa.



Dengan threshold 5, dan minBlockSize 512, kita mampu mendapatkan kompresi yang mirip dengan variance.

STATISTIK KOMPRESI

Waktu load gambar	:	1402 ms
Waktu kompresi	:	1165 ms
Ukuran gambar awal	:	2667516 bytes
Ukuran gambar hasil	:	697662 bytes
Persentase kompresi	:	73.85%
Kedalaman pohon	:	7
Jumlah simpul quadtree	:	6777

Maximum Pixel Difference (MPD)



Menguji kembali kompresi medium di Maximum Pixel Difference dengan threshold 50, dan minBlockSize 512, kita dapatkan data berikut.

STATISTIK KOMPRESI

Waktu load gambar : 1687 ms
Waktu kompresi : 1133 ms
Ukuran gambar awal : 2667516 bytes
Ukuran gambar hasil : 730880 bytes
Persentase kompresi : 72.60%
Kedalaman pohon : 7
Jumlah simpul quadtree: 7445

Entropy



Dengan kompresi medium yang menggunakan threshold 2, dan juga minBlockSize 512, kita dapatkan statistika kompresi berikut

STATISTIK KOMPRESI

Waktu load gambar	:	1592 ms
Waktu kompresi	:	2030 ms
Ukuran gambar awal	:	2667516 bytes
Ukuran gambar hasil	:	1119405 bytes
Persentase kompresi	:	58.04%
Kedalaman pohon	:	7
Jumlah simpul quadtree	:	21845

Melihat bagaimana semua gambar mampu untuk dikompresikan ke kualitas medium, menggunakan standar yang telah ditentukan sebelumnya (dengan satu outlier di MAD). Dapat kita katakan bahwa algoritma Divide and Conquer, yang dibangun dengan struktur QuadTree mampu melakukan kompresi gambar dengan baik, dengan algoritma mampu menciptakan gambar yang serupa, dengan metode yang berbeda beda. Adanya algoritma divide and conquer ini mampu melakukan eksekusi penggeraan dalam waktu yang relatif cukup singkat, dan memberikan hasil yang juga memuaskan.

GIF

Tanpa GIF (threshold 50, minBlockSize 64)

IMAGE COMPRESSOR
Kompresi gambar + GIF

Masukkan path gambar: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGOR

PILIH METODE ERROR
1. Variance 2. MAD (Mean Abs Deviation) 3. Max Pixel Difference 4. Entropy

Pilihan (1-4): 1

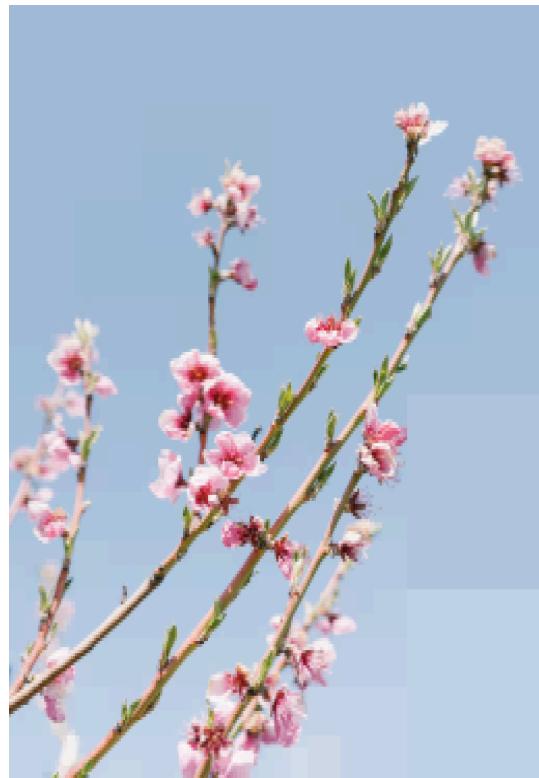
Masukkan threshold: 50

Masukkan ukuran minimum blok (berbasis luas area): 64

Masukkan path untuk menyimpan gambar hasil kompresi (PNG): C:\Users\ASU

Apakah Anda ingin menghasilkan GIF juga? (y/n): n

Gambar hasil kompresi disimpan di: C:\Users\ASUS\Documents\SEMESTER_4\S



STATISTIK KOMPRESI

Waktu load gambar	:	1426 ms
Waktu kompresi	:	1956 ms
Ukuran gambar awal	:	2667516 bytes
Ukuran gambar hasil	:	1110747 bytes
Persentase kompresi	:	58.36%
Kedalaman pohon	:	8
Jumlah simpul quadtree	:	22737

PS C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\SPEK\R

Dengan GIF (threshold 50, minBlockSize 64)

IMAGE COMPRESSOR
Kompresi gambar + GIF

Masukkan path gambar: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITM

PILIH METODE ERROR
1. Variance 2. MAD (Mean Abs Deviation) 3. Max Pixel Difference 4. Entropy

Pilihan (1-4): 1

Masukkan threshold: 50

Masukkan ukuran minimum blok (berbasis luas area): 64

Masukkan path untuk menyimpan gambar hasil kompresi (PNG): C:\Users\ASUS\T

Apakah Anda ingin menghasilkan GIF juga? (y/n): y

Masukkan output path GIF (contoh: hasil.gif): C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITM\hasil.gif

Gambar hasil kompresi disimpan di: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITM\hasil.png

Membuat GIF dari beberapa tingkat kompresi...

GIF berhasil dibuat di: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITM\hasil.gif

STATISTIK KOMPRESI

Waktu load gambar	:	1872 ms
Waktu kompresi	:	1552 ms
Ukuran gambar awal	:	2667516 bytes
Ukuran gambar hasil	:	1110747 bytes
Persentase kompresi	:	58.36%
Kedalaman pohon	:	8
Jumlah simpul quadtree:	:	22737

PS C:\Users\ASUS\Documents\SEMESTER 4\STRATEGI ALGORITMA\TUCIL 2\SPEK\Repc



Hasil Kompresi terus menerus dan mencetak frame, dengan jumlah MaxDepth, dengan final frame berupa gambar hasil kompresi biasa, mampu menciptakan suatu GIF yang bisa terlihat seperti pada gif di atas. Frame untuk gambar GIF tersimpan di bin/frames, dan terbukti bahwa frames juga terbuat dengan benar, sesuai iterasi algoritma.

▼ bin	●
▼ frames	●
frame_01.png	U
frame_02.png	U
frame_03.png	U
frame_04.png	U
frame_05.png	U
frame_06.png	U
frame_07.png	U
frame_final.png	U

4. Small size



Berikut merupakan gambar airplane.jpg yang berukuran 512 x 512, dengan ukuran 94 KB. Dengan menggunakan threshold 16, dan minBlockSize 16, dapat dibentuk gambar seperti di kanan, dengan statsistik berikut. Gambar terkompresi 55%, dan masih memiliki kualitas yang serupa dengan gambar di kiri.

```
PS C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\SPEK\Repo\Tucil2_13523039\bin> java Main
```

IMAGE COMPRESSOR
Kompresi gambar + GIF

```
Masukkan path gambar: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\airplane.jpg
```

PILIH METODE ERROR
1. Variance 2. MAD (Mean Abs Deviation) 3. Max Pixel Difference 4. Entropy

```
Pilihan (1-4): 1
```

```
Masukkan threshold: 16
```

```
Masukkan ukuran minimum blok (berbasis luas area): 16
```

```
Masukkan path untuk menyimpan gambar hasil kompresi (PNG): C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\airplane2.jpg  
Apakah Anda ingin menghasilkan GIF juga? (y/n): n
```

```
Gambar hasil kompresi disimpan di: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\airplane2.jpg
```

STATISTIK KOMPRESI

```
Waktu load gambar : 102 ms  
Waktu kompresi : 69 ms  
Ukuran gambar awal : 96076 bytes  
Ukuran gambar hasil : 43026 bytes  
Persentase kompresi : 55.22%  
Kedalaman pohon : 7  
Jumlah simpul quadtree: 12609
```

```
PS C:\Users\ASUS\Documents\SEMESTER 4\STRATEGI ALGORITMA\TUCIL 2\SPEK\Repo\Tucil2_13523039\bin> █
```

5. High Pixel

Ketika diberikan gambar yang sangat besar seperti foto fuji di bawah ini



Gambar original (kiri) merupakan gambar dengan ukuran 4894 x 2772, dengan ukuran kurang lebih 7 MB. Sebuah gambar pemandangan yang akan penuh dengan warna-warna yang mencolok, maka dapat kita gunakan entropi untuk tetap membagi dikarenakan banyaknya ragam warna. Dengan menggunakan threshold 1 dan minBlockSize 16, kita dapat melakukan kompresi hingga 80% dengan bentuk foto masih menyamai gambar original.

```
PS C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\SPEK\Repo\Tucil2_13523039\bin> java Main
    IMAGE COMPRESSOR
    Kompresi gambar + GIF

Masukkan path gambar: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\fiji.jpg

    PILIH METODE ERROR
    1. Variance
    2. MAD (Mean Abs Deviation)
    3. Max Pixel Difference
    4. Entropy

Pilihan (1-4): 4
Masukkan threshold: 1
Masukkan ukuran minimum blok (berbasis luas area): 16
Masukkan path untuk menyimpan gambar hasil kompresi (PNG): C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\fiji3.jpg
Apakah Anda ingin menghasilkan GIF juga? (y/n): n

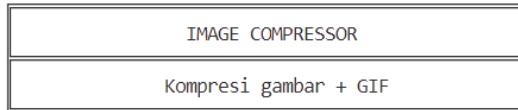
Gambar hasil kompresi disimpan di: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_ALGORITMA\TUCIL_2\testing\fiji3.jpg

    STATISTIK KOMPRESI
    Waktu load gambar : 1975 ms
    Waktu kompresi : 4146 ms
    Ukuran gambar awal : 7351334 bytes
    Ukuran gambar hasil : 1421896 bytes
    Persentase kompresi : 80.66%
    Kedalaman pohon : 9
    Jumlah simpul quadtree: 349525
```

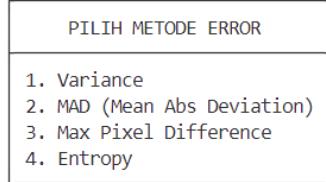
6. 8-Bit



Gambar Original (kiri) merupakan gambar 8-bit dengan ukuran 3000 x 2000 dengan besar file 338 KB. Dengan gambar yang sudah 8-bit sebelumnya, kita masih dapat melakukan kompresi terhadap gambar dengan menggunakan metode Max Pixel Difference, dengan threshold 5, dan minBlockSize 16 dan menghasilkan suatu gambar di kanan yang nyaris identik, namun dengan persentase kompresi 42.77%.



Masukkan path gambar: C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_A



Pilihan (1-4): 3

Masukkan threshold: 5

Masukkan ukuran minimum blok (berbasis luas area): 16

Masukkan path untuk menyimpan gambar hasil kompresi (PNG): C:\Users

Apakah Anda ingin menghasilkan GIF juga? (y/n): n

Gambar hasil kompresi disimpan di: C:\Users\ASUS\Documents\SEMESTER

STATISTIK KOMPRESI

Waktu load gambar : 2069 ms
Waktu kompresi : 1488 ms
Ukuran gambar awal : 346722 bytes
Ukuran gambar hasil : 198418 bytes
Persentase kompresi : 42.77%
Kedalaman pohon : 9
Jumlah simpul quadtree: 121441

DC C:\Users\ASUS\Documents\SEMESTER_4\STRATEGI_A\COMPRESS\TIKI\TIKI.DCN

Dengan melihat segala jenis test-case yang telah dilaksanakan oleh algoritma, dapat terlihat kinerja algoritma Divide and Conquer dalam melakukan kompresi gambar. Menggunakan berbagai metode untuk menentukan threshold, kita mampu menghasilkan gambar yang nyaris identik dengan gambar asli, namun dengan ukuran yang lebih rendah. Melalui pengujian ini, dapat dinyatakan bahwa algoritma Divide and Conquer mampu digunakan untuk menyelesaikan permasalahan seperti membagi sutau permasalahan (gambar), dan mencapai solusi dalam waktu yang relatif singkat, dan hasil yang sesuai.

Analisis Kompleksitas Waktu Algoritma

Algoritma kompresi yang direpresentasikan oleh [RecursiveCompress](#) mengikuti pola umum strategi Divide and Conquer, dimana gambar, yang didefinisikan menjadi block, bila memenuhi kondisi, akan dibagi menjadi empat sub-block secara rekursif hingga kondisi base case. Untuk mengetahui efisiensi dari algoritma ini, pertama kita hitung dulu fungsi-fungsi pendukung yaitu `getAverageRGB()`, `getSubBlock()`, dan `calculateError()`.

Setiap kali fungsi `recursiveCompress()` dipanggil, dilakukan sejumlah operasi bantuan terhadap blok berukuran $n \times n$:

- Perhitungan error (calculateError) dan rata-rata warna (getAverageRGB) melakukan iterasi dua dimensi terhadap pixel $\rightarrow O(n^2)$
- Pembagian block menjadi 4 sub-block menggunakan getSubBlock juga akan melakukan salinan terhadap pixel $\rightarrow O(n^2)$

Dengan setiap iterasi pemanggilan akan membagi block menjadi 4 sub-block yang rata, maka pemanggilan fungsi bersifat rekursif akan mengikuti teorema Master yang berbunyi:

$$T(n) = aT(n/b) + cn^d$$

yang dalam hal ini, $a = 4$ (membagi menjadi 4 upa-permasalahan)

$$b = 2$$

$$c = 1$$

$$d = 2 \text{ (operasi di luar rekursif karena } O(n^2))$$

Dengan demikian, maka kompleksitas menjadi

$$T(n) = 4T(n/2) + n^2,$$

dengan $a = b^d$, maka kompleksitas algoritma menjadi

$$T(n) = O(n^{\log_b a} \cdot \log n),$$

$$T(n) = O(n^2 \log n)$$

Dengan algoritma diluar algoritma Divide and Conquer, seperti pemrosesan gambar menjadi block, dan lainnya hanya mencapai $O(n^2)$, maka dapat ditarik kesimpulan bahwa program memiliki kompleksitas algoritma $O(n^2 \log n)$.

Disamping program wajib, GifGenerator juga memiliki kompleksitasnya sendiri, dengan mengukur kedalaman Tree final-frame, algoritma Divide and Conquer akan di loop dengan iterasi sebanyak d (depth), sehingga untuk kasus penggunaan gif, maka kompleksitas algoritma berubah menjadi $O(d \cdot n^2 \log n)$.

BAB V

Kesimpulan

Penerapan algoritma Divide and Conquer dalam kompresi gambar berbasis QuadTree membuktikan bahwa paradigma ini sangat efektif untuk menyederhanakan representasi data visual. Dengan membagi gambar menjadi 4 block yang lebih kecil terus menerus, algoritma mampu menyederhanakan bagian gambar yang cukup seragam sambil mempertahankan detail pada bagian kompleks, memungkinkan penurunan ukuran gambar dengan kualitas yang masih menyerupai.

Salah satu kekuatan dari algoritma Divide and Conquer adalah efisiensi algoritma, yang hanya melakukan pemrosesan lebih lanjut pada bagian gambar dengan variasi signifikan, meminimalkan redundansi dan mempercepat waktu komputasi. Dengan kompleksitas algoritma $O(n^2 \log n)$ yang dapat dianggap cukup efisien, Algoritma Divide and Conquer menunjukkan kemampuannya dalam menyelesaikan tantangan seperti kompresi citra digital, dengan performa yang lebih baik dari pada umumnya.

LAMPIRAN

Link Github: https://github.com/PeterWongsoredjo/Tucil2_13523039

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✓	
2. Program berhasil dijalankan	✓	
3. Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	✓	
4. Mengimplementasi seluruh metode perhitungan error wajib	✓	
5. [Bonus] Implementasi persentase kompresi sebagai parameter tambahan		✓
6. [Bonus] Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error		✓
7. [Bonus] Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar	✓	
8. Program dan laporan dibuat (kelompok) sendiri	✓	