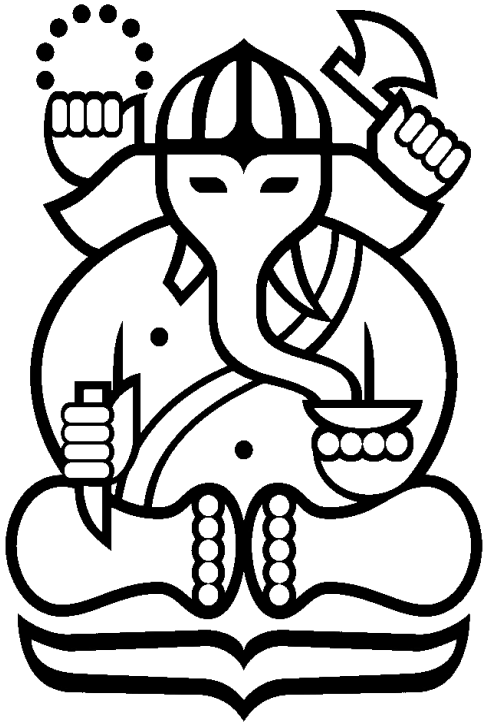
LAPORAN TUGAS KECIL 2
IF2211 STRATEGI ALGORITMA

Kompresi Gambar Dengan Metode Quadtree



Disusun oleh :

13523041 – Hanif Kalyana Aditya

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

BAB I
DESKRIPSI MASALAH DAN ALGORITMA

1.1 Algoritma Divide and Conquer

Penyelesaian sebuah masalah pada dasarnya dapat dipecahkan oleh cara tertentu. Dalam menyelesaikan masalah tersebut, terdapat sebuah cara atau algoritma yang sering digunakan. Algoritma tersebut adalah Algoritma Divide and Conquer. Algoritma Divide and Conquer adalah salah satu metode pendekatan yang cukup sederhana dan dapat memecahkan berbagai persoalan.

Algoritma Divide and Conquer pada dasarnya terdiri dari tiga tahap. Langkah pertama adalah *Divide* yang membagi persoalan menjadi beberapa subpersoalan yang berukuran lebih kecil. Subpersoalan dikelompokkan karena memiliki kemiripan dengan persoalan sebelumnya. Kedua adalah *Conquer* yaitu menyelesaikan semua subpersoalan. Terakhir, solusi dari semua subpersoalan digabungkan hingga terbentuk solusi persoalan awal atau disebut dengan *Combine*

1.2 Kompresi Gambar dengan Quadtree

Suatu gambar pada umumnya adalah kumpulan piksel atau kotak yang sangat kecil sebagai representasi warna yang disimpan dalam bentuk kombinasi warna dasar *red*, *green*, dan *blue* (RGB). Dari tiga warna tersebut nantinya dapat dikombinasikan sehingga menghasilkan warna-warna lain dengan mengatur intensitas masing-masing warna (0 – 255, 8 bit). Semakin banyak piksel maka semakin besar pula ukuran *file* gambar sehingga diperlukan suatu metode yang dapat mengompresi ukuran gambar tanpa menghilangkan sebagian besar isinya.

Salah satu metode yang dapat digunakan untuk mengompresi gambar adalah dengan menggunakan algoritma Divide and Conquer. Karena gambar sering disimpan dalam bentuk kotak/persegi, algoritma tersebut akan membagi suatu gambar menjadi empat buah subpersoalan atau blok yang ukurannya sama besar. Tujuannya adalah membagi gambar menjadi kumpulan suatu blok yang memiliki warna serupa dan direpresentasikan dalam bentuk *tree*. Dalam penerapan hal tersebut digunakan sebuah *threshold*, dan ukuran blok minimum agar terdapat batas bagi algoritma untuk berhenti membagi gambar/blok lagi. Oleh karena itu, sebuah perhitungan error yang menghasilkan nilai tertentu diperlukan untuk membandingkannya dengan nilai *threshold*. Pada tugas ini, digunakan empat buah metode perhitungan error sebagai berikut :

1) Variansi
2) Mean Absolute Deviation
3) Max Pixel Difference
4) Entropy

1.3 Algoritma Kompresi Gambar dengan Quadtree dan Pendekatan Divide and Conquer

Dalam mengompresi sebuah gambar secara algoritmik, penulis menggunakan pendekatan Divide and Conquer. Berikut langkah penyelesaian permasalahan dalam algoritma tersebut dari sudut pandang pengguna program :

1. Pengguna memilih salah satu dari dua opsi yaitu langsung memberikan alamat absolut file gambar yang akan dikompresi dan opsi keluar dari program.
2. Selanjutnya pengguna memasukkan beberapa parameter yang dibutuhkan untuk mengompresi gambar antara lain :
    1) Metode perhitungan error
    2) *Threshold*
    3) Ukuran minimum sebuah blok
    4) Alamat absolut file setelah dikompresi
3. Kemudian, program akan melanjutkan dengan memulai proses pengompresan gambar. Hal pertama yang dilakukan adalah menghitung error dari gambar yang dianggap sebagai node pertama *tree*. Lalu dilakukan pengecekan untuk menentukan apakah gambar akan dibagi lagi (tahap *divide*) menjadi empat buah blok (*child* node) atau tidak. Pembagian akan terus dilakukan apabila terpenuhi dari tiga kondisi berikut :
    1) Nilai error di atas threshold
    2) Ukuran blok lebih besar dari ukuran minimum blok
    3) Ukuran blok setelah dibagi menjadi empat subblok tidak kurang dari ukuran minimum blok

    Setelah salah satu dari ketiga kondisi tersebut tidak terpenuhi, maka pembagian berhenti dilakukan dan blok terakhir akan dianggap sebagai daun dari *tree*. Daun/blok tersebut nantinya akan diberi warna sesuai dengan RGB rata-rata dari blok tersebut (tahap *solve*).
4. Proses kemudian berlanjut untuk setiap blok lainnya dengan pendekatan rekursif sampai terbentuk sebuah *tree* yang daunnya menyimpan informasi sebuah warna. Dari tree tersebut nantinya akan di-*combine* dengan memberi warna tiap piksel sesuai dengan warna bloknya masing-masing.
5. Hasil akhir dari program ini adalah sebuah kumpulan blok kecil yang memiliki warna serupa dengan gambar aslinya. Dengan menggunakan mata, gambar ini akan terlihat sebagai gambar yang buram.
6. Terakhir, program akan melakukan penyimpanan file yang telah terkompresi.

BAB II
IMPLEMENTASI ALGORITMA DALAM BAHASA JAVA

Dalam pembuatan program ini, penulis menggunakan bahasa pemrograman Java. Struktur dari program ini terbagi menjadi tiga file, yaitu file Main.java, Block.java, dan Solver.java dengan pembagian fungsi sebagai berikut :

2.1 Main.java

File ini merupakan bagian utama dalam program dan berisi logika atau gambaran umum bagaimana program ini dijalankan.

2.2 Compressor.java

File ini berisi *method* yang berkaitan dengan algoritma utama Divide and Conquer. Berikut beberapa *method*-nya :

Method/Constructor	Deskripsi
Compressor(…)	Konstruktor sebuah objek yang menyimpan variabel inputFile, errorMethod, threshold, minBlockSize, compressedFile
compressImage(…)	Algoritma utama dari kompresi gambar dengan men-*divide* gambar dan melakukan *combine* blok
splitQuadtree(…)	Membagi sebuah blok secara rekursif
reconstructImage (…)	Merekonstruksi *tree* yang telah didapat
loadImage()	Memuat gambar sehingga dapat diproses
saveCompressedImage(...)	Menyimpan gambar pada alamat yang telah diberikan sebelumnya

2.3 QuadtreeNode.java

File ini berisi konstruktor blok sebagai sebuah *node* dan beberapa *method*-nya yang berkaitan dengan objek QuadtreeNode

Fungsi / Prosedur	Deskripsi
QuadtreeNode(…)	Konstruktor yang menyimpan atribut koordinat, rata-rata warna, panjang, lebar, dan *children* dari sebuah *node* atau blok
setChildren()	Melakukan inisialisasi pada atribut children sebuah node
normalizeColor(…)	Menentukan warna dari sebuah blok

2.4 ErrorCalculator.java

File ini berisi perhitungan error

Fungsi / Prosedur	Deskripsi
calculateError(…)	Method awal yang mengarahkan metode perhitungan error
variance(…)	Menghitung error dengan metode varians
meanAbsoluteDeviation(…)	Menghitung error dengan metode MAD
maxPixelDifference(…)	Menghitung error dengan metode Max Pixel Difference
entropy(…)	Menghitung error dengan metode entropy
calculateEntropy(…)	Menghitung nilai berdasarkan histogram yang diterima

2.5 Utils.java

File ini berisi method pendukung

Fungsi / Prosedur	Deskripsi
getRed(…)	Mengembalikan komponen merah dari RGB
getGrenn(…)	Mengembalikan komponen hijau dari RGB
getBlue(…)	Mengembalikan komponen biru dari RGB
compareRed(…)	Mengembalikan list dari nilai max dan min merah dalam suatu blok
compareGrenn(…)	Mengembalikan list dari nilai max dan min hijau dalam suatu blok
compareBlue(…)	Mengembalikan list dari nilai max dan min biru dalam suatu blok

2.6 IO.java

File ini berisi konstruktor blok sebagai sebuah *node* dan beberapa *method*-nya yang berkaitan dengan objek QuadtreeNode

Fungsi / Prosedur	**Deskripsi**
printCompressionStats(…)	Menampilkan statistik gambar yang telah dikompresi
getTreeDepth()	Menghitung kedalaman tree
printHeader (…)	Menampilkan header di awal program
printFooter(…)	Menampilkan footer di akhir program
getValidatedIntInput(…)	Mengembalikan nilai parameter bertipe integer yang valid
getValidatedDoubleInput(…)	Mengembalikan nilai parameter bertipe double yang valid
getValidatedStringInput(…)	Mengembalikan nilai parameter bertipe string yang valid
showErrorMethodMenut(…)	Menampilkan menu pilihan metode perhitungan error

BAB III
SOURCE CODE PROGRAM

3.1 Repository Program

Repository Program dapat diakses melalui tautan *Github* berikut:
https://github.com/hnfadtya/Tucil2_13523041/

3.2 Source Code Program

Main.java

```
import java.io.File;
import java.util.Scanner;

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

        while (true) {
            IO.printHeader();

            // 1. Input at the begining
            System.out.println("Enter the absolute path from image
    to compress..");
            System.out.println("Enter 'Exit' to quit from this
    program..");
            System.out.print(">>> ");
            String input = scanner.nextLine().trim();

            if ((input).equalsIgnoreCase("Exit")) {
                IO.printFooter();
                break;
            }

            // 2. Input required parameter
            File inputFile = new File(input);
            if (!inputFile.exists() || !inputFile.isFile()) {
                System.out.println("Error: Path '" + input + "'
    cannot be found!\n");
                continue;
            }

            IO.showErrorMethodMenu();
            int errorMethod = IO.getValidatedIntInput(scanner,
    "Enter method in (1-4): ", 1, 4);
            double threshold = IO.getValidatedDoubleInput(scanner,
    "\nEnter threshold value: ");
            int minBlockSize = IO.getValidatedIntInput(scanner,
    "Enter minimum block size: ", 1, Integer.MAX_VALUE);
            File outputFile = IO.getValidatedOutputFile(scanner,
    "Enter file output path: ");

            // 3. Compressing File
            Compressor compressor = new Compressor(inputFile,
    errorMethod, threshold, minBlockSize, outputFile);
            try {
                compressor.compressImage();
                System.out.println("\nSuccess compressing file");
            } catch (Exception e) {
                System.out.println("Compressing file failed: " +
    e.getMessage());
            }
        }
        scanner.close();
    }
}
```

```
System.out.println("\nMemuat file: " + filename);
```

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

public class Compressor {
    private final File inputFile;
    private final int errorMethod;
    private final double threshold;
    private final int minBlockSize;
    private final File compressedFile;

    private BufferedImage inputImage;
    private QuadtreeNode root;

    // Constructor
    public Compressor(File inputFile, int errorMethod, double
    threshold, int minBlockSize, File compressedFile) {
        this.inputFile = inputFile;
        this.errorMethod = errorMethod;
        this.threshold = threshold;
        this.minBlockSize = minBlockSize;
        this.compressedFile = compressedFile;
    }

    // Main processing method
    public void compressImage() {
        try {
            long start = System.currentTimeMillis();

            // load and compress image
            inputImage = loadImage(inputFile);
            BufferedImage compressedImage = new
    BufferedImage(inputImage.getWidth(), inputImage.getHeight(),
    BufferedImage.TYPE_INT_RGB);
            root = new QuadtreeNode(0, 0, inputImage.getWidth(),
    inputImage.getHeight());
            splitQuadtree(inputImage, root);

            // reconstruct (actually compress and reconstructing
    image could be done simultaneously)
            reconstructImage(compressedImage, root);
            saveCompressedImage(compressedImage, compressedFile);
            long end = System.currentTimeMillis();

            System.out.println("Execution Time: " + (end - start) +
    "ms");
            IO.printCompressionStats(inputFile, compressedFile,
    root);
        } catch (Exception e) {
            System.out.println("Error occur: " + e.getMessage());
        }
    }

    // Recursion for normalize block
    private void splitQuadtree(BufferedImage image, QuadtreeNode
    node) {
        int nodeSize = node.width * node.height;
```

```
import java.awt.Color;
import java.awt.image.BufferedImage;

public class QuadtreeNode {
    public int x, y, width, height;
    public int averageR;
    public int averageG;
    public int averageB;
    public Color averageColor;
    public boolean isLeaf;
    public QuadtreeNode[] children;

    public QuadtreeNode(int x, int y, int width, int height) { //
    constructor
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.averageR = 0;
        this.averageG = 0;
        this.averageB = 0;
        this.averageColor = new Color(0, 0, 0);
        this.isLeaf = false;
        this.children = new QuadtreeNode[4]; // NW, NE, SW, SE
    }

    public void setChildren() {
        int count = 0;
        int childWidth = this.width / 2;
        int childHeight = this.height / 2;
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                int X = this.x + childWidth * i;
                int Y = this.y + childHeight * j;
                this.children[count] = new QuadtreeNode(X, Y,
    childWidth, childHeight);
                count++;
            }
        }
    }

    public void normalizeColor(BufferedImage img) {
        for (int i = this.x; i < this.x + this.width; i++) {
            for (int j = this.y; j < this.y + this.height; j++) {
                img.setRGB(i, j, this.averageColor.getRGB());
            }
        }
    }

    public void calculateAverageColor(BufferedImage img) {
        int sumR = 0, sumG = 0, sumB = 0;
        int pixelCount = width * height;

        for (int i = this.x; i < this.x + this.width; i++) {
            for (int j = this.y; j < this.y + this.height; j++) {
                int rgb = img.getRGB(x, y);
                sumR += Utils.getRed(rgb);
```

```
import java.awt.image.BufferedImage;

public class ErrorCalculator {
    // all method here are static because they are only using
    parameter and returning the result

    // calculateError : deciding which error-method would be used
    (accessible -> public)
    public static double calculateError(int method, BufferedImage
    img, QuadtreeNode node) {
        double error = 0;

        if (method == 1) {error = variance(img, node);}
        if (method == 2) {error = meanAbsoluteDeviation(img, node);}
        if (method == 3) {error = maxPixelDifference(img, node);}
        if (method == 4) {error = entropy(img, node);}

        return error;
    }

    private static double variance(BufferedImage img, QuadtreeNode
    node){
        double varR = 0;
        double varG = 0;
        double varB = 0;
        double varRGB;

        for (int x = node.x; x < node.x + node.width; x++) {
            for (int y = node.y; y < node.y + node.height; y++) {
                int rgb = img.getRGB(x, y);
                varR += (double) (Math.pow((Utils.getRed(rgb)
    -  node.averageR), 2));
                varG += (double) (Math.pow((Utils.getGreen(rgb)
    -  node.averageG), 2));
                varB += (double) (Math.pow((Utils.getBlue(rgb)
    -  node.averageB), 2));
            }
        }

        varRGB = (varR + varG + varB) / (3 * (node.width *
    node.height));
        return varRGB;
    }

    private static double meanAbsoluteDeviation(BufferedImage img,
    QuadtreeNode node){
        double madR = 0;
        double madG = 0;
        double madB = 0;
        double madRGB;

        for (int x = node.x; x < node.x + node.width; x++) {
            for (int y = node.y; y < node.y + node.height; y++) {
                int rgb = img.getRGB(x, y);
                madR += (double) (Math.abs(Utils.getRed(rgb)
    -  node.averageR));
                madG += (double) (Math.abs(Utils.getGreen(rgb)
```

```
// using bitwise to improving performa (just experiment)
public class Utils {
    public static int getRed(int rgb) {
        return (rgb >> 16) & 0xff;
    }

    public static int getGreen(int rgb) {
        return (rgb >> 8) & 0xff;
    }

    public static int getBlue(int rgb) {
        return rgb & 0xff;
    }

    public static int[] compareRed(int rgb, int[] redMaxMin) {
        if (Utils.getRed(rgb) > redMaxMin[0]) {
            redMaxMin[0] = Utils.getRed(rgb);
        }
        if (Utils.getRed(rgb) < redMaxMin[1]) {
            redMaxMin[1] = Utils.getRed(rgb);
        }
        return redMaxMin;
    }

    public static int[] compareGreen(int rgb, int[] greenMaxMin) {
        if (Utils.getGreen(rgb) > greenMaxMin[0]) {
            greenMaxMin[0] = Utils.getGreen(rgb);
        }
        if (Utils.getGreen(rgb) < greenMaxMin[1]) {
            greenMaxMin[1] = Utils.getGreen(rgb);
        }
        return greenMaxMin;
    }

    public static int[] compareBlue(int rgb, int[] blueMaxMin) {
        if (Utils.getBlue(rgb) > blueMaxMin[0]) {
            blueMaxMin[0] = Utils.getBlue(rgb);
        }
        if (Utils.getBlue(rgb) < blueMaxMin[1]) {
            blueMaxMin[1] = Utils.getBlue(rgb);
        }
        return blueMaxMin;
    }
}
```

```
import java.io.File;
import java.util.Scanner;

public class IO {
    private static int countNode;

    public static void printCompressionStats(File originalFile, File
    compressedFile, QuadtreeNode root) {
        try {
            long originalSizeBytes = originalFile.length();
            long compressedSizeBytes = compressedFile.length();

            double originalSizeKB = originalSizeBytes / 1024.0;
            double compressedSizeKB = compressedSizeBytes / 1024.0;

            double compressionRatio = 1 - (compressedSizeKB /
    originalSizeKB);
            double compressionPercentage = compressionRatio * 100;

            countNode = 0;
            int treeDepth = getTreeDepth(root);

            System.out.printf("Image size before compressed: %.2f
    KB\n", originalSizeKB);
            System.out.printf("Image size after compressed: %.2f
    KB\n", compressedSizeKB);
            System.out.printf("Compression percentage: %.2f%%\n",
    compressionPercentage);
            System.out.printf("Tree depth: %d\n", treeDepth);
            System.out.printf("Total node(s): %d\n\n",
    countNode);
        } catch (Exception e) {
            System.err.println("Error reading file or calculating
    compression: " + e.getMessage());
        }
    }

    // Recursive helper method to get the depth of the Quadtree
    public static int getTreeDepth(QuadtreeNode node) {
        if (node == null || node.children == null ||
    node.children.length == 0) {
            return 1;
        }

        int maxDepth = 0;
        for (QuadtreeNode child : node.children) {
```

BAB IV

MASUKAN DAN LUARAN PROGRAM

4.1 Threshold dan Blocksize sama

Masukan:

```
Pilih metode perhitungan error:
[1] Variance
[2] Mean Absolute Deviation
[3] Max Pixel Difference
[4] Entropy
Enter method in (1-4): 4

Enter threshold value: 80
Enter minimum block size: 80
Enter file output path: C:\Users\ASUS\Stima\Tucil2_13523041\test\1-04_Image04.jpg
Execution Time: 277ms
Image size before compressed: 68,38 KB
Image size after compressed: 12,18 KB
Compression percentage: 82,19%
Tree depth: 2
Total node(s): 4
```

```
Pilih metode perhitungan error:
[1] Variance
[2] Mean Absolute Deviation
[3] Max Pixel Difference
[4] Entropy
Enter method in (1-4): 3

Enter threshold value: 80
Enter minimum block size: 80
Enter file output path: C:\Users\ASUS\Stima\Tucil2_13523041\test\03.jpg
Execution Time: 515ms
Image size before compressed: 203,40 KB
Image size after compressed: 91,65 KB
Compression percentage: 54,94%
Tree depth: 8
Total node(s): 21748
```

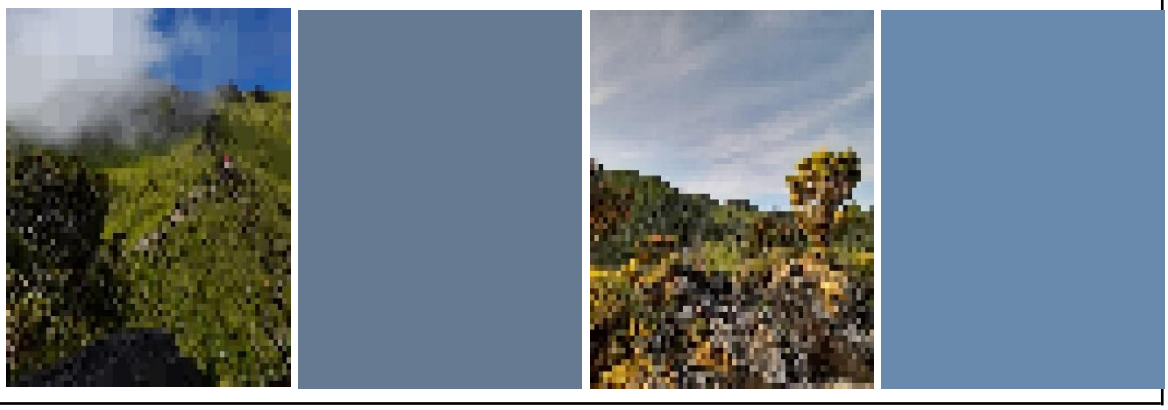
```
Pilih metode perhitungan error:
[1] Variance
[2] Mean Absolute Deviation
[3] Max Pixel Difference
[4] Entropy
Enter method in (1-4): 2

Enter threshold value: 80
Enter minimum block size: 80
Enter file output path: C:\Users\ASUS\Stima\Tucil2_13523041\test\02.jpg
Execution Time: 200ms
Image size before compressed: 180,60 KB
Image size after compressed: 19,36 KB
Compression percentage: 89,28%
Tree depth: 2
Total node(s): 4
```

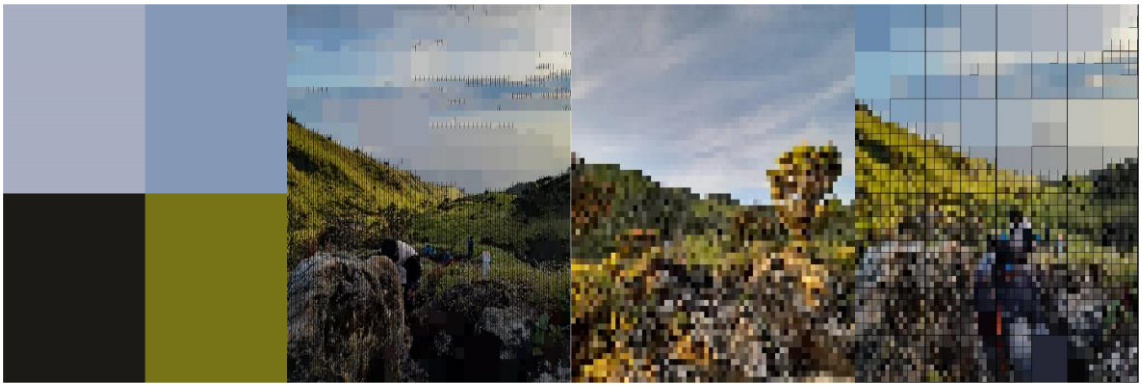
```
Pilih metode perhitungan error:
[1] Variance
[2] Mean Absolute Deviation
[3] Max Pixel Difference
[4] Entropy
Enter method in (1-4): 1

Enter threshold value: 80
Enter minimum block size: 80
Enter file output path: C:\Users\ASUS\Stima\Tucil2_13523041\test\01.jpg
Execution Time: 666ms
Image size before compressed: 187,73 KB
Image size after compressed: 83,48 KB
Compression percentage: 55,53%
Tree depth: 8
Total node(s): 17636
```

Keluaran:



4.2 Threshold sama

Masukan: 
Keluaran:

BAB V LAMPIRAN

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	✓	

Referensi

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025