

LAPORAN TUGAS KECIL 2
IF2211 STRATEGI ALGORITMA

Kompresi Gambar dengan Metode Quadtree



Disusun Oleh:

Rafen Max Alessandro 13523031

Grace Evelyn Simon 13523087

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

2025

DAFTAR ISI

BAB 1	
DESKRIPSI TUGAS	4
BAB 2	
ALGORITMA DIVIDE AND CONQUER	7
2.1. Algoritma Divide and Conquer	7
2.2. Langkah Kompresi Gambar dengan Algoritma Divide and Conquer	7
BAB 3	
IMPLEMENTASI DAN PENGUJIAN	10
3.1. Source Code Program	10
3.1.1. Main.java	11
3.1.2. QuadTree / QuadTreeBuilder.java	15
3.1.3. QuadTree / QuadTreeNode.java	19
3.1.4. QuadTree / QuadTreeProcessor.java	22
3.1.5. QuadTree / ErrorCalculator.java	27
3.2. Pengujian Program	34
3.3. Analisis Pengujian Program	45
3.4. Implementasi Bonus	46
3.4.1. Bonus SSIM	46
3.4.2. Bonus GIF	48
LAMPIRAN	51
Tautan Repository Github	51
Hasil Akhir Tugas Kecil 2	51

DAFTAR GAMBAR

Gambar 1. Perhitungan SSIM untuk suatu kanal warna	47
Gambar 2. Perhitungan SSIM untuk suatu pixel	47
Gambar 3. Perbandingan hasil kompresi MAD (gambar b) dengan SSIM (gambar c)	48
Gambar 4. Contoh GIF Hasil Kompresi QuadTree	50

DAFTAR TABEL

Tabel 1. Struktur keseluruhan program	11
Tabel 2. Struktur Main.java	12
Tabel 3. Penjelasan Metode Main.java	15
Tabel 4. Struktur QuadTreeBuilder.java	16
Tabel 5. Penjelasan Metode QuadTreeBuilder.java	19
Tabel 6. Penjelasan Atribut QuadTreeNode.java	20
Tabel 7. Struktur QuadTreeNode.java	21
Tabel 8. Penjelasan Metode QuadTreeNode.java	22
Tabel 9. Penjelasan Atribut QuadTreeProcessor.java	23
Tabel 10. Struktur QuadTreeProcessor.java	24
Tabel 11. Penjelasan Metode QuadTreeProcessor.java	28
Tabel 12. Struktur ErrorCalculator.java	29
Tabel 13. Penjelasan Metode ErrorCalculator.java	34
Tabel 14. Pengujian 1	35
Tabel 15. Pengujian 2	37
Tabel 16. Pengujian 3	38
Tabel 17. Pengujian 4	40
Tabel 18. Pengujian 5	41
Tabel 19. Pengujian 6	43
Tabel 20. Pengujian 7	44

BAB 1

DESKRIPSI TUGAS

Quadtree adalah struktur data hierarkis yang digunakan untuk membagi ruang atau data menjadi bagian yang lebih kecil, yang sering digunakan dalam pengolahan gambar. Dalam konteks kompresi gambar, Quadtree membagi gambar menjadi blok-blok kecil berdasarkan keseragaman warna atau intensitas piksel. Prosesnya dimulai dengan membagi gambar menjadi empat bagian, lalu memeriksa apakah setiap bagian memiliki nilai yang seragam berdasarkan analisis sistem warna RGB, yaitu dengan membandingkan komposisi nilai merah (R), hijau (G), dan biru (B) pada piksel-piksel di dalamnya. Jika bagian tersebut tidak seragam, maka bagian tersebut akan terus dibagi hingga mencapai tingkat keseragaman tertentu atau ukuran minimum yang ditentukan. Dalam implementasi teknis, sebuah Quadtree direpresentasikan sebagai simpul (*node*) dengan maksimal empat anak (*children*). Simpul daun (*leaf*) merepresentasikan area gambar yang seragam, sementara simpul internal menunjukkan area yang masih membutuhkan pembagian lebih lanjut. Setiap simpul menyimpan informasi seperti posisi (x, y), ukuran (width, height), dan nilai rata-rata warna atau intensitas piksel dalam area tersebut. Struktur ini memungkinkan pengkodean data gambar yang lebih efisien dengan menghilangkan redundansi pada area yang seragam. QuadTree sering digunakan dalam algoritma kompresi lossy karena mampu mengurangi ukuran file secara signifikan tanpa mengorbankan detail penting pada gambar.

Pada tugas kecil 2, terdapat beberapa prosedur pada program kompresi gambar, yakni:

1. Inisialisasi dan Persiapan Data

Masukkan gambar yang akan dikompresi akan diolah dalam format matriks piksel dengan nilai intensitas berdasarkan sistem warna RGB. Berikut adalah parameter-parameter yang dapat ditentukan oleh pengguna saat ingin melakukan kompresi gambar:

- a. Metode perhitungan variansi: pilih metode perhitungan variansi berdasarkan opsi yang tersedia pada tabel.

- b. Threshold variansi: nilai ambang batas untuk menentukan apakah blok akan dibagi lagi.
 - c. Minimum block size: ukuran minimum blok piksel yang diperbolehkan untuk diproses lebih lanjut.
2. Perhitungan Error

Untuk setiap blok gambar yang sedang diproses, hitung nilai variansi menggunakan metode yang dipilih sesuai tabel.

3. Pembagian Blok

Bandingkan nilai variansi blok dengan threshold:

- a. Jika variansi di atas threshold, ukuran blok lebih besar dari minimum block size, dan ukuran blok setelah dibagi menjadi empat tidak kurang dari minimum block size, blok tersebut dibagi menjadi empat sub-blok, dan proses dilanjutkan untuk setiap sub-blok.
- b. Jika salah satu kondisi di atas tidak terpenuhi, proses pembagian dihentikan untuk blok tersebut.

4. Normalisasi Warna

Untuk blok yang tidak lagi dibagi, lakukanlah normalisasi warna blok sesuai dengan rata-rata nilai RGB blok.

5. Rekursi dan Penghentian

Proses pembagian blok dilakukan secara rekursif untuk setiap sub-blok hingga semua blok memenuhi salah satu dari dua kondisi berikut:

- a. Error blok berada di bawah *threshold*.
 - b. Ukuran blok setelah dibagi menjadi empat kurang dari minimum block size.
6. Penyimpanan dan Output

Rekonstruksi gambar dilakukan berdasarkan struktur QuadTree yang telah dihasilkan selama proses kompresi. Gambar hasil rekonstruksi akan disimpan sebagai file

terkompresi. Selain itu, persentase kompresi akan dihitung dan disertakan dengan rumus sesuai dengan yang terlampir pada dokumen ini. Persentase kompresi ini memberikan gambaran mengenai efisiensi metode kompresi yang digunakan.

Tugas kecil meminta untuk membentuk program sederhana dalam bahasa C/C#/C++/Java (CLI) yang mengimplementasikan algoritma Divide and Conquer untuk melakukan kompresi gambar berbasis Quadtree yang mengimplementasikan seluruh parameter yang telah disebutkan sebagai user input. Berikut alur program yang harus dibuat:

1. [INPUT] alamat absolut gambar yang akan dikompresi.
2. [INPUT] metode perhitungan error (gunakan penomoran sebagai input).
3. [INPUT] ambang batas (pastikan range nilai sesuai dengan metode yang dipilih).
4. [INPUT] ukuran blok minimum.
5. [INPUT] Target persentase kompresi (floating number, 1.0 = 100%), beri nilai 0 jika ingin menonaktifkan mode ini, jika mode ini aktif maka nilai threshold bisa menyesuaikan secara otomatis untuk memenuhi target persentase kompresi (bonus).
6. [INPUT] alamat absolut gambar hasil kompresi.
7. [INPUT] alamat absolut gif (bonus).
8. [OUTPUT] waktu eksekusi.
9. [OUTPUT] ukuran gambar sebelum.
10. [OUTPUT] ukuran gambar setelah.
11. [OUTPUT] persentase kompresi.
12. [OUTPUT] kedalaman pohon.
13. [OUTPUT] banyak simpul pada pohon.
14. [OUTPUT] gambar hasil kompresi pada alamat yang sudah ditentukan.
15. [OUTPUT] GIF proses kompresi pada alamat yang sudah ditentukan (bonus).

BAB 2

ALGORITMA DIVIDE AND CONQUER

2.1. Algoritma *Divide and Conquer*

Divide and Conquer merupakan strategi pemecahan masalah yang melibatkan pembagian masalah kompleks menjadi bagian-bagian yang lebih kecil dan mudah dikelola. Strategi Divide and Conquer terdiri atas Divide, Conquer, dan Combine. Divide merupakan langkah membagi persoalan menjadi beberapa upa-persoalan yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya setiap upa-persoalan berukuran hampir sama). Conquer merupakan langkah menyelesaikan masing-masing upa-persoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar). Sedangkan Combine merupakan langkah menggabungkan solusi masing-masing upa-persoalan sehingga membentuk solusi persoalan semula. Algoritma Divide and Conquer memiliki beberapa keunggulan, seperti penyelesaian sub-masalah yang efisien dan kemudahan dalam paralelisasi. Dalam beberapa kasus, seperti pada algoritma pengurutan mergesort atau quicksort, penyelesaian masalah dilakukan secara efisien dengan kompleksitas waktu $O(n \log n)$, yang lebih baik daripada algoritma lain yang memiliki kompleksitas $O(n^2)$. Sub-masalah juga dapat diselesaikan secara bersamaan oleh beberapa prosesor atau thread, yang meningkatkan kecepatan eksekusi pada sistem dengan banyak prosesor.

2.2. Langkah Kompresi Gambar dengan Algoritma *Divide and Conquer*

Tahapan dalam melakukan kompresi gambar menggunakan algoritma *divide and conquer* adalah sebagai berikut

1. Melakukan *Preprocessing*

Bertujuan untuk mengubah data gambar menjadi format yang dapat diolah secara numerik. Program akan membaca file gambar (JPEG, PNG, JPG) menggunakan `BufferedImage`. Kemudian didapatkan dimensi gambar (`width x height`) yang dikonversi ke matriks 2D piksel dengan format `int[height][width]`.

2. Pembangunan QuadTree (Divide Phase)

Merupakan proses membangun QuadTree dengan algoritma rekursif:

a. Base Case

Algoritma berhenti membagi blok dan membuat leaf node ketika ukuran blok terlalu kecil ($\leq \text{minBlockSize}$) atau nilai error sudah cukup kecil ($\leq \text{threshold}$).

b. Recursive Case

- i. Ambil blok (x, y, width, height) dari gambar
- ii. Hitung nilai error dari blok ini menggunakan salah satu metode *error calculation*.
- iii. Bandingkan nilai error dengan threshold:
 - Jika error $\leq \text{threshold}$ atau ukuran blok terlalu kecil ($\leq \text{minBlockSize}$), berhenti membagi.
 - Jika error $> \text{threshold}$ dan ukuran cukup besar, bagi blok menjadi 4 kuadran, yakni NW: kiri atas, NE: kanan atas, SW: kiri bawah, SE: kanan bawah
- iv. Panggil algoritma ini secara rekursif untuk setiap sub-blok.
- v. Kembalikan node dengan 4 anak, kemudian simpan strukturnya dalam pohon QuadTree.

3. Langkah Rekonstruksi (Conquer Phase)

Selanjutnya, QuadTree yang dibangun direkonstruksi untuk mendapatkan hasil kompresi gambar, berikut langkah-langkahnya:

- a. Telusuri pohon Quadtree secara rekursif.
- b. Untuk setiap leaf node, ambil koordinat (x, y, width, height) dan warna rata-rata.
- c. Warnai kembali bagian tersebut pada gambar output. Seluruh blok akan diisi dengan satu warna rata-rata.

4. Langkah Output

- a. Hasil gambar yang direkonstruksi akan disimpan kembali sebagai file gambar.

- b. Untuk output, terdapat beberapa metrik yang perlu dihitung, seperti:
 - i. Waktu eksekusi
 - ii. Ukuran gambar sebelum
 - iii. Ukuran gambar setelah
 - iv. Persentase kompresi
 - v. Kedalaman pohon Quadtree
 - vi. Banyak simpul pada pohon

BAB 3

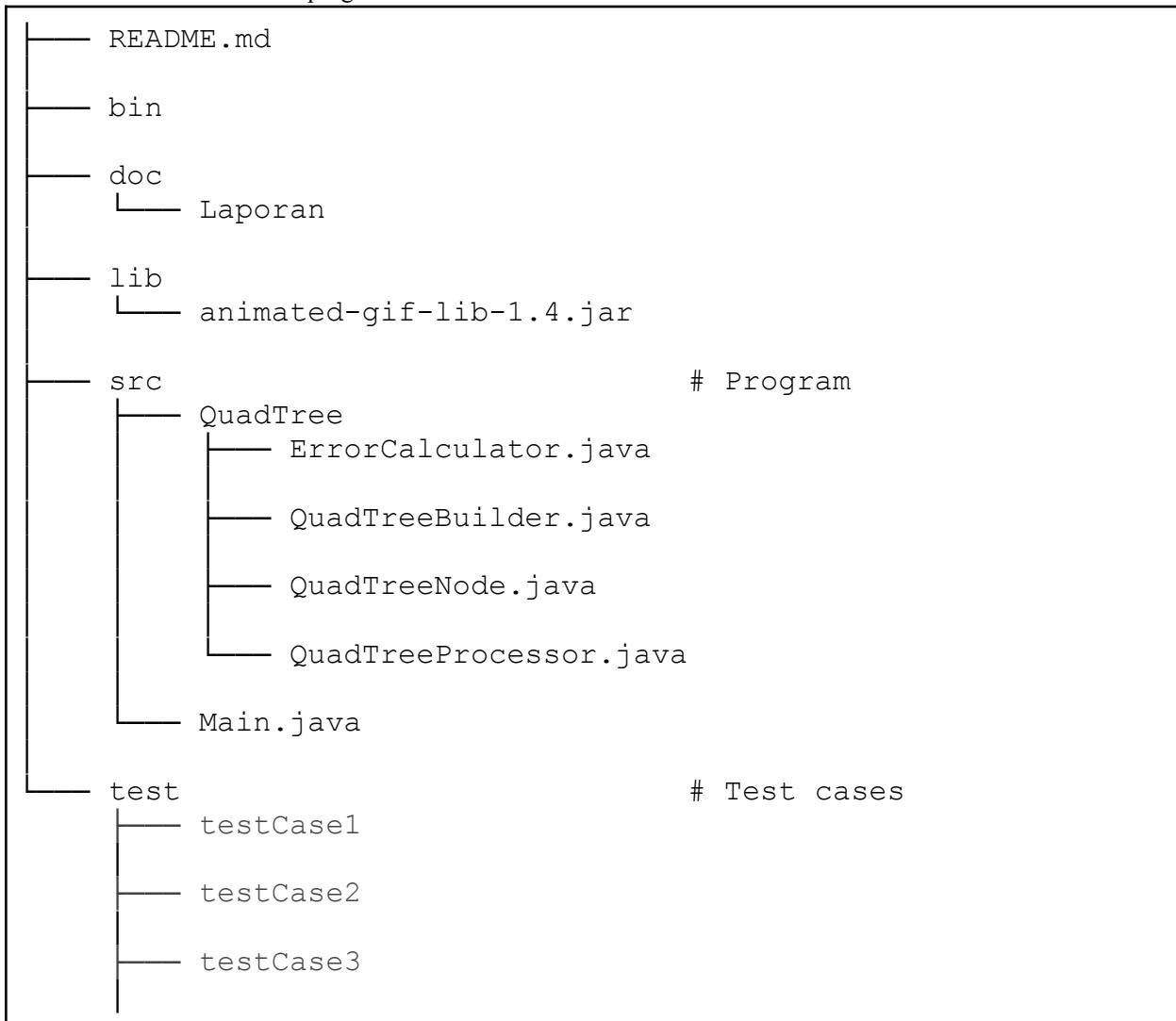
IMPLEMENTASI DAN PENGUJIAN

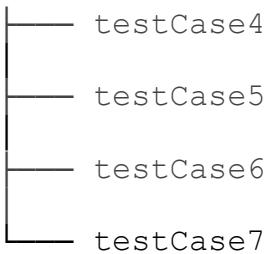
3.1. *Source Code Program*

Pemilihan bahasa Java sebagai bahasa pemrograman untuk program didasarkan terhadap kemudahan penggunaannya, terutama dalam pengelolaan struktur program. Java memungkinkan pengembangan yang lebih efisien tanpa memperhatikan detail mendalam seperti pembentukan *constructor* untuk setiap kelas, manajemen memori secara eksplisit, dan detail eksplisit lainnya. Hal ini selaras dengan tujuan program yang lebih menekankan fokus terhadap perancangan dan penerapan algoritma *divide and conquer* dalam melangsungkan proses pengolahan gambar.

Keseluruhan program memiliki struktur sebagai berikut:

Tabel 1. Struktur keseluruhan program





dengan penjelasan untuk setiap *file* adalah sebagai berikut.

3.1.1. Main.java

Main merupakan kelas yang menerima *input* dari pengguna mengenai parameter-parameter yang dibutuhkan untuk melakukan kompresi gambar. Main juga melangsungkan validasi data untuk memastikan parameter yang dimasukkan pengguna tidak akan menimbulkan kesalahan pada program. Main.java tidak memiliki atribut karena keseluruhan *input* dari pengguna disimpan dalam variabel lokal untuk kemudian digunakan sebagai parameter dari metode kelas QuadTreeProcessor yang memulai proses kompresi gambar.

Tabel 2. Struktur Main.java

```

import QuadTree.QuadTreeProcessor;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.InvalidPathException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;
import javax.imageio.ImageIO;

public class Main {

    public static void main(String[] args) {
        System.out.println("                                     ");
        System.out.println(" |");
        System.out.println(" |   Selamat datang di QuadTree Compression!");
        System.out.println(" |");
        System.out.println(" |   ( /^-^) /          \\"(^-^\\)");
        System.out.println(" |");
        System.out.println(" |   Tekan ENTER untuk memulai kompresi gambar! ");
        System.out.println(" ");

        Scanner scan = new Scanner(System.in);
        scan.nextLine();
    }
}
  
```

```

String inputPath;
int errorMethod;
double threshold;
int minBlockSize;
String outputImagePath;
String outputGifPath = "";

while (true) {
    System.out.print("\nalamat absolut gambar yang akan dikompresi:");
}

    inputPath = scan.nextLine();

    File inputFile = new File(inputPath);
    if (!inputFile.exists()) {
        System.out.println("Path yang kamu masukkan tidak valid!");
    } else {
        continue;
    }

    try {
        BufferedImage testImage = ImageIO.read(inputFile);
        if (testImage == null) {
            System.out.println("Path yang kamu masukkan tidak valid!");
        } else {
            continue;
        }
        break;
    } catch (IOException e) {
        System.out.println("Terjadi kesalahan saat membaca file
gambar.");
    }
}

System.out.println("Metode Perhitungan Error");
System.out.println("1. variance");
System.out.println("2. mean absolute deviation");
System.out.println("3. max pixel difference");
System.out.println("4. entropy");
System.out.println("5. SSIM");

while (true) {
    System.out.print("\nmetode perhitungan error: ");

    if (scan.hasNextInt()) {
        errorMethod = scan.nextInt();
    } else {
        scan.next();
        System.out.println("Nilai yang kamu masukkan tidak valid!");
    } else {
        continue;
    }
}

```

```
        }

        if (errorMethod >= 1 && errorMethod <= 5) {
            scan.nextLine();
            break;
        } else {
            System.out.println("Nilai yang kamu masukkan tidak valid!
○(T-T)o");
            continue;
        }
    }

    while (true) {
        System.out.print("\nambah batas: ");

        if (scan.hasNextDouble()) {
            threshold = scan.nextDouble();
        } else {
            scan.next();
            System.out.println("Nilai yang kamu masukkan tidak valid!
○(T-T)o");
            continue;
        }

        if (threshold >= 0) {
            scan.nextLine();
            break;
        } else {
            System.out.println("Nilai yang kamu masukkan tidak valid!
○(T-T)o");
            continue;
        }
    }

    while (true) {
        System.out.print("\nukuran blok minimum: ");

        if (scan.hasNextInt()) {
            minBlockSize = scan.nextInt();
            scan.nextLine();
            break;
        } else {
            scan.next();
            System.out.println("Nilai yang kamu masukkan tidak valid!
○(T-T)o");
            continue;
        }
    }

    while (true) {
        System.out.print("\nalamat absolut gambar hasil kompresi: ");
        outputPath = scan.nextLine();

        if (!isValidPath(outputImagePath, true)) {
            System.out.println("Path yang kamu masukkan tidak valid untuk
gambar! ○(T-T)o");
        }
    }
}
```

```

        continue;
    }

    break;
}

System.out.print("\nbuat GIF animasi? (y/n): ");
String createGif = scan.nextLine().toLowerCase();

if (createGif.equals("y")) {
    while (true) {
        System.out.print("\nalamat absolut GIF hasil kompresi: ");
        outputGifPath = scan.nextLine();

        if (!isValidPath(outputGifPath, false)) {
            System.out.println("Path yang kamu masukkan tidak valid
untuk GIF! o(T-T)o");
            continue;
        }

        break;
    }
}

scan.close();

QuadTreeProcessor processor = new QuadTreeProcessor();
processor.processImage(inputPath, outputImagePath, errorMethod,
threshold, minBlockSize, outputGifPath);
}

public static boolean isValidPath(String path, boolean gambar) {
    try {
        Path filePath = Paths.get(path);
        Path parentDirectory = filePath.getParent();

        if (parentDirectory != null &&
!Files.isDirectory(parentDirectory)) {
            return false;
        }

        if (path.lastIndexOf(".") == -1) {
            return false;
        }

        String extension = path.substring(path.lastIndexOf(".") +
1).toLowerCase();

        if (gambar) {
            return extension.equals("jpg") || extension.equals("jpeg") ||
extension.equals("png");
        } else {
            return extension.equals("gif");
        }
    } catch (InvalidPathException e) {

```

```
        return false;
    }
}
```

Tabel 3. Penjelasan Metode Main.java

Nama Metode	Deskripsi
void main()	Memulai program, menerima <i>input</i> dari pengguna, melakukan validasi terhadap setiap <i>input</i> , dan melanjutkan <i>input</i> ke metode untuk memulai proses kompresi gambar
boolean isValidPath()	Memberikan <i>true</i> jika alamat absolut yang diberikan pengguna sebagai alamat <i>output</i> hasil kompresi gambar sebagai <i>image</i> dan GIF bersifat valid (<i>image</i> memiliki ekstensi .jpg, .jpeg, atau .png dan GIF memiliki ekstensi .gif)

3.1.2. QuadTree / QuadTreeBuilder.java

QuadTreeBuilder bertanggung jawab sebagai kelas yang membangun struktur QuadTree. Menggunakan metode utama buildQuadTree() dan buildQuadTreeWithSSIM(), QuadTreeBuilder mengaplikasikan algoritma *divide and conquer* dalam bentuk fungsi rekursif. QuadTreeBuilder mengubah representasi gambar dalam bentuk *array of array of integer* yang menggambarkan intensitas kanal setiap pixel menjadi QuadTree sebagai dasar melakukan algoritma *divide and conquer* dalam mengkompresi gambar.

Sebagai kelas yang memiliki tujuan utama untuk membangun struktur QuadTree, QuadTreeBuilder tidak menyimpan nilai apapun sebagai suatu atribut.

Tabel 4. Struktur QuadTreeBuilder.java

```
package QuadTree;

import java.util.*;

public class QuadTreeBuilder {
    // menyimpan urutan node quadtree berdasarkan level traversal (untuk keperluan GIF)
    private final List<List<QuadTreeNode>> renderSteps = new ArrayList<>();

    // mengembalikan daftar renderSteps dalam bentuk per-level (BFS)
    public List<List<QuadTreeNode>> getRenderSteps() {
        return renderSteps;
    }

    // mengembalikan daftar renderSteps dalam bentuk list datar (flat)
```

```

public List<QuadTreeNode> getRenderStepsFlat() {
    List<QuadTreeNode> flat = new ArrayList<>();
    for (List<QuadTreeNode> level : renderSteps) {
        flat.addAll(level);
    }
    return flat;
}

// membentuk quadtree berdasarkan metode error tertentu, threshold, dan
// ukuran minimum blok
public QuadTreeNode buildQuadtree(int[][] image, int x, int y, int width,
int height, int errorMethod, double threshold, int minBlockSize) {
    renderSteps.clear();
    return buildRecursive(image, x, y, width, height, errorMethod,
threshold, minBlockSize, 0);
}

// fungsi rekursif utama untuk pembentukan quadtree berdasarkan error umum
// (bukan SSIM)
private QuadTreeNode buildRecursive(int[][] image, int x, int y, int
width, int height, int errorMethod, double threshold, int minBlockSize, int
depth) {
    ErrorCalculator errorCalculator = new ErrorCalculator();
    double error = errorCalculator.calculateError(image, x, y, width,
height, errorMethod);

    QuadTreeNode node;

    // jika error lebih kecil dari threshold atau blok terlalu kecil,
    // hentikan pembagian
    if (error <= threshold || width <= minBlockSize || height <=
minBlockSize) {
        int[] color = calculateAverageColor(image, x, y, width, height);
        node = new QuadTreeNode(x, y, width, height, color);
    }
    else {
        int halfWidth = width / 2;
        int halfHeight = height / 2;
        int remainingWidth = width - halfWidth;
        int remainingHeight = height - halfHeight;

        // bagi blok menjadi 4 sub-blok
        QuadTreeNode[] children = new QuadTreeNode[4];
        children[0] = buildRecursive(image, x, y, halfWidth, halfHeight,
errorMethod, threshold, minBlockSize, depth + 1);
        children[1] = buildRecursive(image, x + halfWidth, y,
remainingWidth, halfHeight, errorMethod, threshold, minBlockSize, depth + 1);
        children[2] = buildRecursive(image, x, y + halfHeight, halfWidth,
remainingHeight, errorMethod, threshold, minBlockSize, depth + 1);
        children[3] = buildRecursive(image, x + halfWidth, y + halfHeight,
remainingWidth, remainingHeight, errorMethod, threshold, minBlockSize, depth +
1);

        node = new QuadTreeNode(x, y, width, height, children);
    }
}

```

```

        // simpan node ke dalam renderSteps berdasarkan level traversal
        while (renderSteps.size() <= depth) {
            renderSteps.add(new ArrayList<>());
        }
        renderSteps.get(depth).add(node);

        return node;
    }

    // membentuk quadtree menggunakan metode error SSIM, dengan logika khusus
    // perbandingan blok
    public QuadTreeNode buildQuadtreewithSSIM(int[][] imageInput, int x, int
y, int width, int height, double threshold, int minBlockSize) {
        renderSteps.clear();
        return buildRecursiveWithSSIM(imageInput, x, y, width, height,
threshold, minBlockSize, 0);
    }

    // fungsi rekursif utama untuk SSIM, membandingkan blok rata-rata dengan
    // blok asli
    private QuadTreeNode buildRecursiveWithSSIM(int[][] imageInput, int x, int
y, int width, int height, double threshold, int minBlockSize, int depth) {
        int[][] imageOutput = new int[imageInput.length][];
        for (int i = 0; i < imageInput.length; i++) {
            imageOutput[i] = imageInput[i].clone();
        }

        int[] color = calculateAverageColor(imageOutput, x, y, width, height);
        for (int i = y; i < y + height && i < imageOutput.length; i++) {
            for (int j = x; j < x + width && j < imageOutput[i].length; j++) {
                int pixel = (255 << 24) | (color[0] << 16) | (color[1] << 8) |
color[2];
                imageOutput[i][j] = pixel;
            }
        }

        ErrorCalculator errorCalculator = new ErrorCalculator();
        double error = errorCalculator.SSIM(imageInput, imageOutput, x, y,
width, height);

        QuadTreeNode node;
        // jika error cukup rendah dan ukuran blok masih bisa dibagi,
        lanjutkan pembagian
        if (error < threshold && (width * height) > minBlockSize) {
            int halfWidth = (int) Math.ceil(width / 2.0);
            int halfHeight = (int) Math.ceil(height / 2.0);

            QuadTreeNode[] children = new QuadTreeNode[4];
            children[0] = buildRecursiveWithSSIM(imageInput, x, y, halfWidth,
halfHeight, threshold, minBlockSize, depth + 1);
            children[1] = buildRecursiveWithSSIM(imageInput, x + halfWidth, y,
width - halfWidth, halfHeight, threshold, minBlockSize, depth + 1);
            children[2] = buildRecursiveWithSSIM(imageInput, x, y +
halfHeight, halfWidth, height - halfHeight, threshold, minBlockSize, depth +
1);
            children[3] = buildRecursiveWithSSIM(imageInput, x + halfWidth, y

```

```

+ halfHeight, width - halfWidth, height - halfHeight, threshold, minBlockSize,
depth + 1);

        node = new QuadTreeNode(x, y, width, height, children);
    }
    else {
        node = new QuadTreeNode(x, y, width, height, color);
    }

    // simpan node ke dalam renderSteps berdasarkan level traversal
    while (renderSteps.size() <= depth) {
        renderSteps.add(new ArrayList<>());
    }
    renderSteps.get(depth).add(node);

    return node;
}

// menghitung rata-rata nilai RGB pada blok (x, y, width, height)
private int[] calculateAverageColor(int[][] image, int x, int y, int
width, int height) {
    long[] sum = new long[3];
    int count = 0;

    for (int i = y; i < y + height && i < image.length; i++) {
        for (int j = x; j < x + width && j < image[i].length; j++) {
            int pixel = image[i][j];
            sum[0] += (pixel >> 16) & 0xFF;
            sum[1] += (pixel >> 8) & 0xFF;
            sum[2] += pixel & 0xFF;
            count++;
        }
    }

    return new int[] {
        (int) (sum[0] / count),
        (int) (sum[1] / count),
        (int) (sum[2] / count)
    };
}
}
}

```

Tabel 5. Penjelasan Metode QuadTreeBuilder.java

Nama Metode	Deskripsi
List<List<QuadTreeNode>> getRenderSteps()	Mengembalikan struktur renderSteps, yaitu daftar node yang disusun berdasarkan level traversal QuadTree (level 0 = root, level 1 = anak, dst)
List<QuadTreeNode> getRenderStepsFlat()	Mengembalikan seluruh node dari renderSteps dalam satu list datar (flattened)

QuadTreeNode buildQuadtree()	Membentuk QuadTree dengan melakukan divide and conquer sesuai dengan batasan parameter (metode perhitungan error, threshold, dan batas ukuran blok minimal)
QuadTreeNode buildQuadtreeWithSSIM()	Fungsi turunan dari buildQuadtree() yang dibentuk karena metode perhitungan error SSIM memerlukan algoritma divide and conquer yang sedikit berbeda dengan metode perhitungan error lainnya
int[] calculateAverageColor()	Mengembalikan array integer nilai hasil normalisasi seluruh pixel dalam suatu blok menggunakan rata-rata dari setiap kanal warna
QuadTreeNode buildRecursive()	(private) Fungsi rekursif internal untuk membangun QuadTree berdasarkan metode perhitungan error umum (non-SSIM), mencatat setiap node ke dalam renderSteps berdasarkan depth
QuadTreeNode buildRecursiveWithSSIM()	(private) Fungsi rekursif internal untuk membangun QuadTree menggunakan metode SSIM, membandingkan blok asli dengan versi rata-rata warnanya, dan menyimpan hasil per level di renderSteps

3.1.3. QuadTree / QuadTreeNode.java

QuadTreeNode merupakan kelas yang merepresentasikan struktur data simpul dalam sebuah QuadTree. Setiap simpul merepresentasikan sebuah blok gambar, dan QuadTreeNode dapat berupa *leaf* sebagai representasi blok yang tidak dibagi lagi, atau simpul dalam sebagai representasi blok yang masih dapat dibagi menjadi 4 sub-blok.

Tabel 6. Penjelasan Atribut QuadTreeNode.java

Nama Atribut	Deskripsi
int x, y	Koordinat kiri atas dari blok gambar yang direpresentasikan oleh simpul
int width, height	Ukuran lebar dan tinggi blok gambar
int[] color	Rata-rata warna blok yang direpresentasikan dalam bentuk <i>array</i> untuk setiap kanal ([red, green, blue]). Hanya dimiliki oleh <i>leaf</i>

QuadTreeNode[] children	<i>Array yang menyimpan referensi ke empat simpul anak yang membagi blok menjadi empat bagian sama besar, yaitu blok kiri atas, blok kanan atas, blok kiri bawah, dan blok kanan bawah. Hanya dimiliki oleh simpul dalam</i>
-------------------------	--

Tabel 7. Struktur QuadTreeNode.java

```

package QuadTree;

public class QuadTreeNode {
    private int x, y;                                // koordinat kiri atas blok gambar
    private int width, height;                        // ukuran blok gambar
    private int[] color;                             // rata-rata warna blok (R, G, B)
    private QuadTreeNode[] children;                 // children dari internal nodes

    /**
     * Konstruktor leaf nodes
     * Menyimpan informasi tentang blok gambar yang tidak dibagi lebih lanjut
     * @param x          koordinat horizontal kiri atas blok gambar
     * @param y          koordinat vertikal kiri atas blok gambar
     * @param width      lebar blok gambar
     * @param height     tinggi blok gambar
     * @param color      rata-rata warna blok gambar dalam format [Red, Green,
Blue]
     */
    public QuadTreeNode(int x, int y, int width, int height, int[] color) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.color = color;
        this.children = null;
    }

    /**
     * Konstruktor internal nodes
     * Node internal menyimpan referensi ke 4 anak yang membentuk
bagian-bagian gambar lebih kecil
     * @param x          koordinat horizontal kiri atas blok gambar
     * @param y          koordinat vertikal kiri atas blok gambar
     * @param width      lebar blok gambar
     * @param height     tinggi blok gambar
     * @param children   array berisi 4 anak node
     */
    public QuadTreeNode(int x, int y, int width, int height, QuadTreeNode[] children) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.color = null;
        this.children = children;
    }
}

```

```
}

/**
 * Mengambil koordinat X (horizontal) dari node
 * @return koordinat X dari node
 */
public int getX() {
    return x;
}

/**
 * Mengambil koordinat Y (vertikal) dari node
 * @return koordinat Y dari node
 */
public int getY() {
    return y;
}

/**
 * Mengambil lebar blok gambar yang diwakili oleh node
 * @return lebar blok gambar
 */
public int getWidth() {
    return width;
}

/**
 * Mengambil tinggi blok gambar yang diwakili oleh node
 * @return tinggi blok gambar
 */
public int getHeight() {
    return height;
}

/**
 * Mengambil warna rata-rata blok gambar untuk leaf node
 * @return array berisi warna rata-rata dalam format [Red, Green, Blue]
 */
public int[] getColor() {
    return color;
}

/**
 * Mengambil referensi ke child node jika node ini adalah internal node
 * @return array berisi referensi ke 4 anak node
 */
public QuadTreeNode[] getChildren() {
    return children;
}

/**
 * Memeriksa apakah node ini adalah leaf node
 * @return true jika node adalah leaf node, false jika node adalah
internal node
 */
public boolean isLeaf() {
```

```

        return children == null;
    }
}

```

Tabel 8. Penjelasan Metode QuadTreeNode.java

Nama Metode	Deskripsi
QuadTreeNode dengan parameter int[]	Konstruktor <i>leaf</i> , menyimpan atribut QuadTreeNode dan informasi mengenai blok yang tidak dibagi lebih lanjut dan hanya terdiri atas satu warna
QuadTreeNode dengan parameter QuadTreeNode[]	Konstruktor simpul dalam, menyimpan atribut QuadTreeNode dan informasi mengenai referensi kepada empat simpul anak
int getX()	Mengembalikan integer nilai atribut X, yaitu koordinat horizontal dari simpul
int getY()	Mengembalikan integer nilai atribut Y, yaitu koordinat vertikal dari simpul
int getWidth()	Mengembalikan integer nilai atribut width, yaitu lebar blok dari simpul
int getHeight()	Mengembalikan integer nilai atribut height, yaitu tinggi blok dari simpul
int[] getColor()	Mengembalikan <i>array of integer</i> nilai atribut color, yaitu rata-rata kanal warna ([red, green, blue] untuk <i>leaf</i>)
QuadTreeNode[] getChildren()	Mengembalikan atribut children, yaitu referensi kepada empat simpul anak
boolean isLeaf()	Mengembalikan <i>true</i> jika QuadTreeNode merupakan <i>leaf</i>

3.1.4. QuadTree / QuadTreeProcessor.java

Setelah Main.java melakukan validasi terhadap setiap *input* yang diberikan pengguna. QuadTreeProcessor menerima *input-input* tersebut untuk kemudian memulai proses pembentukan QuadTree untuk melangsungkan kompresi gambar. QuadTreeProcessor akan membentuk QuadTree, merekonstruksi data *array* setiap simpul menjadi gambar, menyimpan hasil ke *path* sesuai *input* pengguna, dan membentuk GIF proses kompresi gambar.

Tabel 9. Penjelasan Atribut QuadTreeProcessor.java

Nama Atribut	Deskripsi
long startTime, endTime	Menyimpan waktu mulai dan berakhirnya proses kompresi gambar untuk menghitung lama waktu proses
int totalNodes	Jumlah simpul yang telah terbentuk dalam QuadTree
int treeDepth	Kedalaman QuadTree selain <i>root</i>

Tabel 10. Struktur QuadTreeProcessor.java

```

package QuadTree;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import com.madgag.gif.fmsware.AnimatedGifEncoder;

public class QuadTreeProcessor {
    private long startTime;
    private long endTime;
    private int totalNodes = 0;
    private int treeDepth = 0;
    private AnimatedGifEncoder AnimatedGifEncoder;
    private boolean createGif;
    private int frameCount = 0;

    /**
     * proses utama untuk membaca gambar, mengompresinya dengan QuadTree,
     menyimpan hasil,
     * dan menghasilkan GIF animasi subdivisi jika diinginkan
     */
    public void processImage(String inputImagePath, String outputImagePath,
    int errorMethod, double threshold, int minBlockSize, String gifOutputPath) {
        this.createGif = (gifOutputPath != null && !gifOutputPath.isEmpty());

        try {
            startTime = System.nanoTime();
            System.out.println("\n(^-^)/ memulai proses kompresi gambar...");

            BufferedImage image = ImageIO.read(new File(inputImagePath));
            int width = image.getWidth();
            int height = image.getHeight();
            System.out.println("(^-^)/ gambar berhasil dibaca! ukuran: " +
width + "x" + height);

            if (createGif) {
                System.out.println("(^-^)/ menyiapkan pembuatan GIF

```

```

animasi...");
        AnimatedGifEncoder = new AnimatedGifEncoder();
        AnimatedGifEncoder.start(gifOutputPath);
        AnimatedGifEncoder.setRepeat(0);
        AnimatedGifEncoder.setDelay(1000);
    }

    int[][] imagePixels = convertImageToPixels(image, width, height);

    System.out.println("(^-^)/ sedang membangun quadtree...");
    QuadTreeBuilder builder = new QuadTreeBuilder();
    QuadTreeNode root = builder.buildQuadtree(imagePixels, 0, 0,
width, height, errorMethod, threshold, minBlockSize);
    calculateTreeDepthAndNodeCount(root, 0);

    if (createGif) {
        List<List<QuadTreeNode>> stepsByLevel =
builder.getRenderSteps();
        renderGIFInSubdivisionOrder(stepsByLevel, width, height);
        AnimatedGifEncoder.finish();
        System.out.println("(^-^)/ GIF animasi berhasil dibuat di: "
+ gifOutputPath);
        System.out.println(" total frame : " + frameCount);
    }

    System.out.println("(^-^)/ merekonstruksi gambar akhir...");
    BufferedImage reconstructedImage = reconstructImage(root, width,
height);
    ImageIO.write(reconstructedImage, "png", new
File(outputImagePath));

    long originalSize = Files.size(Paths.get(inputImagePath));
    long compressedSize = Files.size(Paths.get(outputImagePath));
    double compressionPercentage = (double) compressedSize /
originalSize * 100;

    endTime = System.nanoTime();
    long executionTime = (endTime - startTime) / 1_000_000;

    System.out.println("\n(^-^)/ hasil kompresi \\\\"(^-^\\\")");
    System.out.println("=====");
    System.out.println(" waktu proses : " + executionTime + " "
ms);
    System.out.println(" ukuran asli : " + originalSize + " "
bytes);
    System.out.println(" ukuran kompresi : " + compressedSize + " "
bytes);
    System.out.println(" rasio kompresi : " +
String.format("%.2f", compressionPercentage) + "%");
    System.out.println(" kedalaman pohon : " + treeDepth);
    System.out.println(" jumlah node : " + totalNodes);
    if (createGif) {
        System.out.println(" output GIF : " + gifOutputPath);
    }
    System.out.println("=====");
    System.out.println("\n(^-^)/ kompresi selesai! hasil disimpan di:");

```

```

" + outputPath);

        } catch (Exception e) {
            System.out.println("Ups! Terjadi kesalahan saat memproses
gambar:");
            e.printStackTrace();
        }
    }

/**
 * mengubah objek BufferedImage menjadi matriks pixel (RGB integer)
 */
private int[][] convertImageToPixels(BufferedImage image, int width, int
height) {
    int[][] pixels = new int[height][width];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            pixels[y][x] = image.getRGB(x, y);
        }
    }
    return pixels;
}

/**
 * menghitung total node dan kedalaman maksimum dari struktur Quadtree
 */
private void calculateTreeDepthAndNodeCount(QuadTreeNode node, int depth)
{
    if (node == null) return;
    treeDepth = Math.max(treeDepth, depth);
    totalNodes++;
    if (!node.isLeaf()) {
        for (QuadTreeNode child : node.getChildren()) {
            calculateTreeDepthAndNodeCount(child, depth + 1);
        }
    }
}

/**
 * menghasilkan ulang gambar dari struktur Quadtree hasil kompresi
 */
private BufferedImage reconstructImage(QuadTreeNode root, int width, int
height) {
    BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
    reconstructImageRecursive(root, image);
    return image;
}

/**
 * fungsi rekursif untuk menggambar warna blok-blok hasil QuadTree pada
BufferedImage
 */
private void reconstructImageRecursive(QuadTreeNode node, BufferedImage
image) {
    if (node.isLeaf()) {

```

```

        int rgb = (node.getColor()[0] << 16) | (node.getColor()[1] << 8) |
node.getColor()[2];
        for (int y = node.getY(); y < node.getY() + node.getHeight(); y++)
{
        for (int x = node.getX(); x < node.getX() + node.getWidth();
x++) {
            if (x < image.getWidth() && y < image.getHeight()) {
                image.setRGB(x, y, rgb);
            }
        }
    } else {
        for (QuadTreeNode child : node.getChildren()) {
            reconstructImageRecursive(child, image);
        }
    }
}

/**
 * merender proses subdivisi QuadTee berdasarkan level (BFS) ke dalam GIF
animasi
 */
private void renderGIFInSubdivisionOrder(List<List<QuadTreeNode>>
stepsByLevel, int width, int height) {
    BufferedImage canvas = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);

    int white = (255 << 16) | (255 << 8) | 255;
    for (int y = 0; y < height; y++)
        for (int x = 0; x < width; x++)
            canvas.setRGB(x, y, white);

    if (createGif) {
        BufferedImage firstFrame = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
        firstFrame.getGraphics().drawImage(canvas, 0, 0, null);
        AnimatedGifEncoder.addFrame(firstFrame);
        frameCount++;
    }

    for (List<QuadTreeNode> levelNodes : stepsByLevel) {
        for (QuadTreeNode node : levelNodes) {
            int[] color = node.getColor();
            if (color == null) continue;

            int rgb = (color[0] << 16) | (color[1] << 8) | color[2];

            for (int y = node.getY(); y < node.getY() + node.getHeight();
y++) {
                for (int x = node.getX(); x < node.getX() +
node.getWidth(); x++) {
                    if (x < width && y < height) {
                        canvas.setRGB(x, y, rgb);
                    }
                }
            }
        }
    }
}

```

```

        }

        if (createGif) {
            BufferedImage frame = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
            frame.getGraphics().drawImage(canvas, 0, 0, null);
            AnimatedGifEncoder.addFrame(frame);
            frameCount++;
        }
    }
}

```

Tabel 11. Penjelasan Metode QuadTreeProcessor.java

Nama Metode	Deskripsi
void processImage()	Menjalankan seluruh proses, mulai dari membaca gambar, membentuk QuadTree, menulis output, dan membuat GIF animasi subdivisi jika dipilih pengguna
int[][] convertImageToPixels()	Mengubah objek BufferedImage menjadi matriks pixel RGB untuk pemrosesan QuadTree
void calculateTreeDepthAndNodeCount()	Menghitung total node dan kedalaman maksimum dari pohon QuadTree
BufferedImage reconstructImage()	Merekonstruksi gambar akhir dari struktur QuadTreeNode hasil kompresi
void reconstructImageRecursive()	Menggambar ulang setiap blok QuadTree ke BufferedImage secara rekursif
void renderGIFInSubdivisionOrder()	Merender setiap level subdivisi QuadTree (BFS) ke dalam frame GIF animasi

3.1.5. QuadTree / ErrorCalculator.java

ErrorCalculator merupakan kelas yang dibentuk untuk menyediakan metode-metode perhitungan *error* sebagai parameter kompresi gambar. Kelas lain akan memanggil metode calculateError beserta *key* yang memetakan angka ke suatu metode perhitungan *error* untuk kemudian dibandingkan dengan nilai *threshold* sebagai pernyataan apakah suatu blok pixel sudah dianggap cukup seragam.

Sebagai kelas yang berperan sebagai wadah bagi metode-metode, kelas ErrorCalculator tidak memiliki atribut.

Tabel 12. Struktur ErrorCalculator.java

```
package QuadTree;

import java.util.HashMap;
import java.util.Map;

public class ErrorCalculator {

    /**
     * API ke kelas QuadTreeBuilder
     * 1 - Variance
     * 2 - Mean Absolute Deviation (MAD)
     * 3 - Max Pixel Difference
     * 4 - Entropy
     */
    public double calculateError(int[][][] image, int startX, int startY, int width, int height, int errorMethod) {
        switch (errorMethod) {
            case 1:
                return variance(image, startX, startY, width, height);
            case 2:
                return MAD(image, startX, startY, width, height);
            case 3:
                return MPD(image, startX, startY, width, height);
            case 4:
                return entropy(image, startX, startY, width, height);
            default:
                return 0;
        }
    }

    // mengambil nilai rgb dari sebuah pixel
    public int getColor(int pixel, String color) {
        switch (color) {
            case "red":
                return (pixel >> 16) & 0xFF;
            case "green":
                return (pixel >> 8) & 0xFF;
            case "blue":
                return pixel & 0xFF;
            default:
                return 0;
        }
    }

    // menghitung rata-rata sebuah color dalam sebuah blok
    public double[] meanColor(int[][][] image, int startX, int startY, int width, int height) {
        double[] mean = new double[3];
        int totalPixels = 0;

        for (int y = startY; y < startY + height && y < image.length; y++) {
            for (int x = startX; x < startX + width && x < image[y].length;
x++) {
                int pixel = image[y][x];
                mean[0] += pixel >> 16;
                mean[1] += pixel >> 8;
                mean[2] += pixel & 0xFF;
                totalPixels++;
            }
        }
        for (int i = 0; i < 3; i++) {
            mean[i] /= totalPixels;
        }
        return mean;
    }
}
```

```

        mean[0] += getColor(pixel, "red");
        mean[1] += getColor(pixel, "green");
        mean[2] += getColor(pixel, "blue");
        totalPixels++;
    }
}

mean[0] /= totalPixels;
mean[1] /= totalPixels;
mean[2] /= totalPixels;

return mean;
}

// menghitung variansi sebuah color dalam sebuah blok
public double[] varianceColor(int[][] image, int startX, int startY, int width, int height) {
    double[] variance = new double[3];
    double[] mean = meanColor(image, startX, startY, width, height);
    int totalPixels = 0;

    for (int y = startY; y < startY + height && y < image.length; y++) {
        for (int x = startX; x < startX + width && x < image[y].length;
x++) {
            int pixel = image[y][x];
            variance[0] += Math.pow((getColor(pixel, "red")) - mean[0],
2);
            variance[1] += Math.pow((getColor(pixel, "green")) - mean[1],
2);
            variance[2] += Math.pow((getColor(pixel, "blue")) - mean[2],
2);
            totalPixels++;
        }
    }

    variance[0] /= totalPixels;
    variance[1] /= totalPixels;
    variance[2] /= totalPixels;

    return variance;
}

// menghitung variansi dalam sebuah blok
public double variance(int[][] image, int startX, int startY, int width,
int height) {
    double[] variance = varianceColor(image, startX, startY, width,
height);
    return (variance[0] + variance[1] + variance[2]) / 3;
}

// menghitung kovariansi sebuah color antara dua buah blok
public double[] covarianceColor(int[][] imageX, int[][] imageY, int
startX, int startY, int width, int height) {
    double[] covariance = new double[3];
    double[] meanX = meanColor(imageX, startX, startY, width, height);
    double[] meanY = meanColor(imageY, startX, startY, width, height);
}

```

```

        int totalPixels = 0;

        for (int y = startY; y < startY + height && y < imageX.length; y++) {
            for (int x = startX; x < startX + width && x < imageX[y].length;
x++) {
                covariance[0] += (getColor(imageX[y][x], "red") - meanX[0]) *
(getColor(imageY[y][x], "red") - meanY[0]);
                covariance[1] += (getColor(imageX[y][x], "green") - meanX[1])
* (getColor(imageY[y][x], "green") - meanY[1]);
                covariance[2] += (getColor(imageX[y][x], "blue") - meanX[2]) *
(getColor(imageY[y][x], "blue") - meanY[2]);
                totalPixels++;
            }
        }

        covariance[0] /= totalPixels;
        covariance[1] /= totalPixels;
        covariance[2] /= totalPixels;

        return covariance;
    }

    // menghitung MAD dalam sebuah blok
    public double MAD(int[][] image, int startX, int startY, int width, int
height) {
        double[] MAD = new double[3];
        double[] mean = meanColor(image, startX, startY, width, height);
        int totalPixels = 0;

        for (int y = startY; y < startY + height && y < image.length; y++) {
            for (int x = startX; x < startX + width && x < image[y].length;
x++) {
                MAD[0] += Math.abs((getColor(image[y][x], "red")) - mean[0]);
                MAD[1] += Math.abs((getColor(image[y][x], "green")) -
mean[1]);
                MAD[2] += Math.abs((getColor(image[y][x], "blue")) - mean[2]);
                totalPixels++;
            }
        }

        MAD[0] /= totalPixels;
        MAD[1] /= totalPixels;
        MAD[2] /= totalPixels;

        return (MAD[0] + MAD[1] + MAD[2]) / 3;
    }

    // menghitung MPD dalam sebuah blok
    public double MPD(int[][] image, int startX, int startY, int width, int
height) {
        int minRed = 255, maxRed = 0;
        int minGreen = 255, maxGreen = 0;
        int minBlue = 255, maxBlue = 0;

        for (int y = startY; y < startY + height && y < image.length; y++) {
            for (int x = startX; x < startX + width && x < image[y].length;

```

```

x++) {
    int pixel = image[y][x];
    int red = getColor(pixel, "red");
    int green = getColor(pixel, "green");
    int blue = getColor(pixel, "blue");

    maxRed = Math.max(maxRed, red);
    minRed = Math.min(minRed, red);
    maxGreen = Math.max(maxGreen, green);
    minGreen = Math.min(minGreen, green);
    maxBlue = Math.max(maxBlue, blue);
    minBlue = Math.min(minBlue, blue);
}
}

return ((maxRed - minRed) + (maxGreen - minGreen) + (maxBlue -
minBlue)) / 3.0;
}

// menghitung probabilitas distribusi warna dalam sebuah blok
public double[][] probability(int[][] image, int startX, int startY, int
width, int height) {
    Map<Integer, Integer> redFrequency = new HashMap<>();
    Map<Integer, Integer> greenFrequency = new HashMap<>();
    Map<Integer, Integer> blueFrequency = new HashMap<>();
    int totalPixels = 0;

    for (int y = startY; y < startY + height && y < image.length; y++) {
        for (int x = startX; x < startX + width && x < image[y].length;
x++) {
            int pixel = image[y][x];
            int redValue = getColor(pixel, "red");
            int greenValue = getColor(pixel, "green");
            int blueValue = getColor(pixel, "blue");

            redFrequency.put(redValue, redFrequency.getOrDefault(redValue,
0) + 1);
            greenFrequency.put(greenValue,
greenFrequency.getOrDefault(greenValue, 0) + 1);
            blueFrequency.put(blueValue,
blueFrequency.getOrDefault(blueValue, 0) + 1);
            totalPixels++;
        }
    }

    double[][] probability = new double[3][256];

    for (Map.Entry<Integer, Integer> e : redFrequency.entrySet()) {
        probability[0][e.getKey()] = (double) e.getValue() / totalPixels;
    }

    for (Map.Entry<Integer, Integer> e : greenFrequency.entrySet()) {
        probability[1][e.getKey()] = (double) e.getValue() / totalPixels;
    }

    for (Map.Entry<Integer, Integer> e : blueFrequency.entrySet()) {

```

```

        probability[2][e.getKey()] = (double) e.getValue() / totalPixels;
    }

    return probability;
}

// menghitung entropy dalam sebuah blok
public double entropy(int[][] image, int startX, int startY, int width,
int height) {
    double[] entropy = new double[3];
    double[][] probability = probability(image, startX, startY, width,
height);

    for (int i = 0; i < 256; i++) {
        double r = probability[0][i], g = probability[1][i], b =
probability[2][i];

        if (r > 0) {
            entropy[0] += r * (Math.log(r) / Math.log(2));
        }

        if (g > 0) {
            entropy[1] += g * (Math.log(g) / Math.log(2));
        }

        if (b > 0) {
            entropy[2] += b * (Math.log(b) / Math.log(2));
        }
    }

    return ((entropy[0] * -1) + (entropy[1] * -1) + entropy[2] * -1) / 3;
}

// menghitung SSIM antara dua buah blok berukuran sama
public double SSIM(int[][] imageX, int[][] imageY, int startX, int startY,
int width, int height) {
    double[] SSIM = new double[3];
    double[] meanX = meanColor(imageX, startX, startY, width, height);
    double[] meanY = meanColor(imageY, startX, startY, width, height);
    double[] varianceX = varianceColor(imageX, startX, startY, width,
height);
    double[] varianceY = varianceColor(imageY, startX, startY, width,
height);
    double[] covariance = covarianceColor(imageX, imageY, startX, startY,
width, height);

    double c1 = Math.pow((0.01 * 255), 2);
    double c2 = Math.pow((0.03 * 255), 2);

    for (int i = 0; i < 3; i++) {
        SSIM[i] = (((2 * meanX[i] * meanY[i]) + c1) * ((2 * covariance[i])
+ c2)) /
        (((meanX[i] * meanX[i]) + (meanY[i] * meanY[i]) + c1) *
(varianceX[i] + varianceY[i] + c2));
    }
}

```

```

        return (0.299 * SSIM[0] + 0.587 * SSIM[1] + 0.114 * SSIM[2]) * 100;
    }
}

```

Tabel 13. Penjelasan Metode ErrorCalculator.java

Nama Metode	Deskripsi
double calculateError	Metode yang menghubungkan metode-metode perhitungan <i>error</i> pada kelas ErrorCalculator dengan kelas lainnya. Memetakan empat metode <i>error</i> dengan angka 1 hingga 4, dengan pemetaan sebagai berikut: 1 - Variance 2 - Mean Absolute Deviation (MAD) 3 - Max Pixel Difference 4 - Entropy
int getColor	Mengembalikan integer nilai intensitas kanal red, green, dan blue dari sebuah pixel
double[] meanColor	Mengembalikan array double nilai rata-rata intensitas setiap kanal warna (red, green, blue) dalam sebuah blok pixel
double[] varianceColor	Mengembalikan array double nilai variansi setiap kanal warna (red, green, blue) dalam sebuah blok pixel
double variance	Mengembalikan double nilai Variance dalam sebuah blok pixel
double MAD	Mengembalikan double nilai Mean Absolute Deviation (MAD) dalam sebuah blok pixel
double MPD	Mengembalikan double nilai Max Pixel Difference (MPD) dalam sebuah blok pixel
double[][] probability	Mengembalikan <i>array of array of double</i> nilai probabilitas berdasarkan distribusi setiap nilai intensitas (0 - 255) setiap kanal warna (red, green, blue) dalam sebuah blok pixel
double entropy	Mengembalikan double nilai Entropy dalam sebuah blok pixel
double SSIM	Mengembalikan double nilai SSIM antara dua

	buah blok pixel
--	-----------------

3.2. Pengujian Program

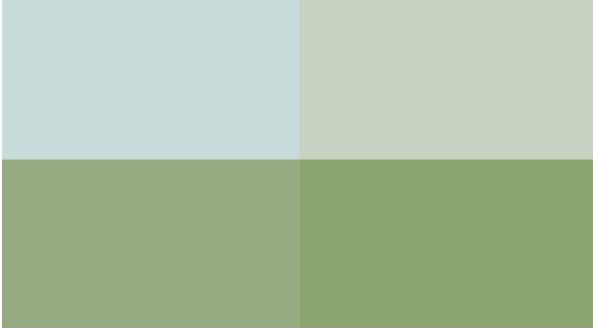
Tabel 14. Pengujian 1

 <p>Gambar asli</p> <p>Parameter: <i>threshold</i>: 50 ukuran blok minimum: 100 </p>	 <p>Variance</p> <pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 548 ms ukuran asli : 1027174 bytes ukuran kompresi : 54204 bytes rasio kompresi : 5.28% kedalaman pohon : 7 jumlah node : 9629 =====</pre>
 <p>Mean Absolute Deviation</p>	 <p>Max Pixel Difference</p>

<pre>(/^-^)/ hasil kompresi \(^-^\ =====</pre> <table border="0"> <tr><td>waktu proses</td><td>:</td><td>265 ms</td></tr> <tr><td>ukuran asli</td><td>:</td><td>1027174 bytes</td></tr> <tr><td>ukuran kompresi</td><td>:</td><td>4113 bytes</td></tr> <tr><td>rasio kompresi</td><td>:</td><td>0.40%</td></tr> <tr><td>kedalaman pohon</td><td>:</td><td>0</td></tr> <tr><td>jumlah node</td><td>:</td><td>1</td></tr> </table> <pre>=====</pre>	waktu proses	:	265 ms	ukuran asli	:	1027174 bytes	ukuran kompresi	:	4113 bytes	rasio kompresi	:	0.40%	kedalaman pohon	:	0	jumlah node	:	1	<pre>(/^-^)/ hasil kompresi \(^-^\ =====</pre> <table border="0"> <tr><td>waktu proses</td><td>:</td><td>302 ms</td></tr> <tr><td>ukuran asli</td><td>:</td><td>1027174 bytes</td></tr> <tr><td>ukuran kompresi</td><td>:</td><td>54676 bytes</td></tr> <tr><td>rasio kompresi</td><td>:</td><td>5.32%</td></tr> <tr><td>kedalaman pohon</td><td>:</td><td>7</td></tr> <tr><td>jumlah node</td><td>:</td><td>9005</td></tr> </table> <pre>=====</pre>	waktu proses	:	302 ms	ukuran asli	:	1027174 bytes	ukuran kompresi	:	54676 bytes	rasio kompresi	:	5.32%	kedalaman pohon	:	7	jumlah node	:	9005
waktu proses	:	265 ms																																			
ukuran asli	:	1027174 bytes																																			
ukuran kompresi	:	4113 bytes																																			
rasio kompresi	:	0.40%																																			
kedalaman pohon	:	0																																			
jumlah node	:	1																																			
waktu proses	:	302 ms																																			
ukuran asli	:	1027174 bytes																																			
ukuran kompresi	:	54676 bytes																																			
rasio kompresi	:	5.32%																																			
kedalaman pohon	:	7																																			
jumlah node	:	9005																																			
 <p>Entropy</p>	 <p>SSIM</p>																																				
<pre>(/^-^)/ hasil kompresi \(^-^\ =====</pre> <table border="0"> <tr><td>waktu proses</td><td>:</td><td>862 ms</td></tr> <tr><td>ukuran asli</td><td>:</td><td>1027174 bytes</td></tr> <tr><td>ukuran kompresi</td><td>:</td><td>57433 bytes</td></tr> <tr><td>rasio kompresi</td><td>:</td><td>5.59%</td></tr> <tr><td>kedalaman pohon</td><td>:</td><td>7</td></tr> <tr><td>jumlah node</td><td>:</td><td>10269</td></tr> </table> <pre>=====</pre>	waktu proses	:	862 ms	ukuran asli	:	1027174 bytes	ukuran kompresi	:	57433 bytes	rasio kompresi	:	5.59%	kedalaman pohon	:	7	jumlah node	:	10269	<pre>(/^-^)/ hasil kompresi \(^-^\ =====</pre> <table border="0"> <tr><td>waktu proses</td><td>:</td><td>5792 ms</td></tr> <tr><td>ukuran asli</td><td>:</td><td>1027174 bytes</td></tr> <tr><td>ukuran kompresi</td><td>:</td><td>51681 bytes</td></tr> <tr><td>rasio kompresi</td><td>:</td><td>5.03%</td></tr> <tr><td>kedalaman pohon</td><td>:</td><td>7</td></tr> <tr><td>jumlah node</td><td>:</td><td>9145</td></tr> </table> <pre>=====</pre>	waktu proses	:	5792 ms	ukuran asli	:	1027174 bytes	ukuran kompresi	:	51681 bytes	rasio kompresi	:	5.03%	kedalaman pohon	:	7	jumlah node	:	9145
waktu proses	:	862 ms																																			
ukuran asli	:	1027174 bytes																																			
ukuran kompresi	:	57433 bytes																																			
rasio kompresi	:	5.59%																																			
kedalaman pohon	:	7																																			
jumlah node	:	10269																																			
waktu proses	:	5792 ms																																			
ukuran asli	:	1027174 bytes																																			
ukuran kompresi	:	51681 bytes																																			
rasio kompresi	:	5.03%																																			
kedalaman pohon	:	7																																			
jumlah node	:	9145																																			

Tabel 15. Pengujian 2

 <p>Gambar asli</p> <p>Parameter: <i>threshold:</i> 40 <i>ukuran blok minimum:</i> 50</p>	 <p>Variance</p>
--	--

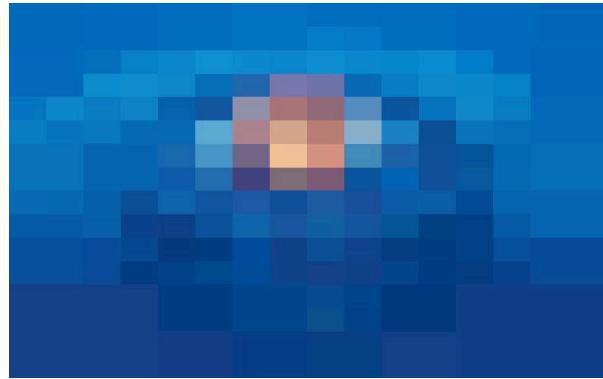
	<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 500 ms ukuran asli : 76035 bytes ukuran kompresi : 38307 bytes rasio kompresi : 50.38% kedalaman pohon : 8 jumlah node : 6197 =====</pre>
	
Mean Absolute Deviation	Max Pixel Difference
<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 427 ms ukuran asli : 76035 bytes ukuran kompresi : 16421 bytes rasio kompresi : 21.60% kedalaman pohon : 1 jumlah node : 5 =====</pre>	<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 408 ms ukuran asli : 76035 bytes ukuran kompresi : 39607 bytes rasio kompresi : 52.09% kedalaman pohon : 8 jumlah node : 6285 =====</pre>
	
Entropy	SSIM
<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 1272 ms ukuran asli : 76035 bytes ukuran kompresi : 44794 bytes rasio kompresi : 58.91% kedalaman pohon : 8 jumlah node : 18313 =====</pre>	<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 7212 ms ukuran asli : 76035 bytes ukuran kompresi : 38098 bytes rasio kompresi : 50.11% kedalaman pohon : 8 jumlah node : 6069 =====</pre>

Tabel 16. Pengujian 3



Gambar asli

Parameter:
threshold: 70
ukuran blok minimum: 4000



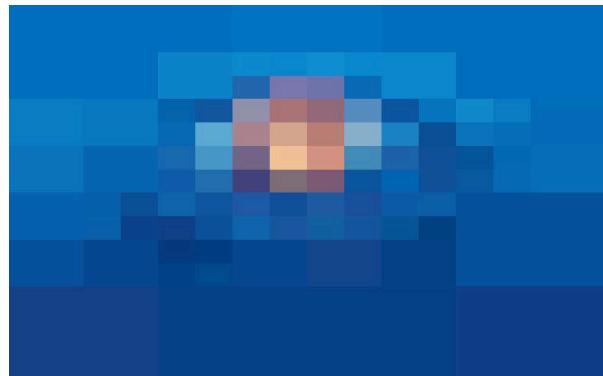
Variance

```
(/^-^)/ hasil kompresi \(^-^\  
=====  
waktu proses : 311 ms  
ukuran asli : 30782 bytes  
ukuran kompresi : 12254 bytes  
rasio kompresi : 39.81%  
kedalaman pohon : 4  
jumlah node : 209  
=====
```



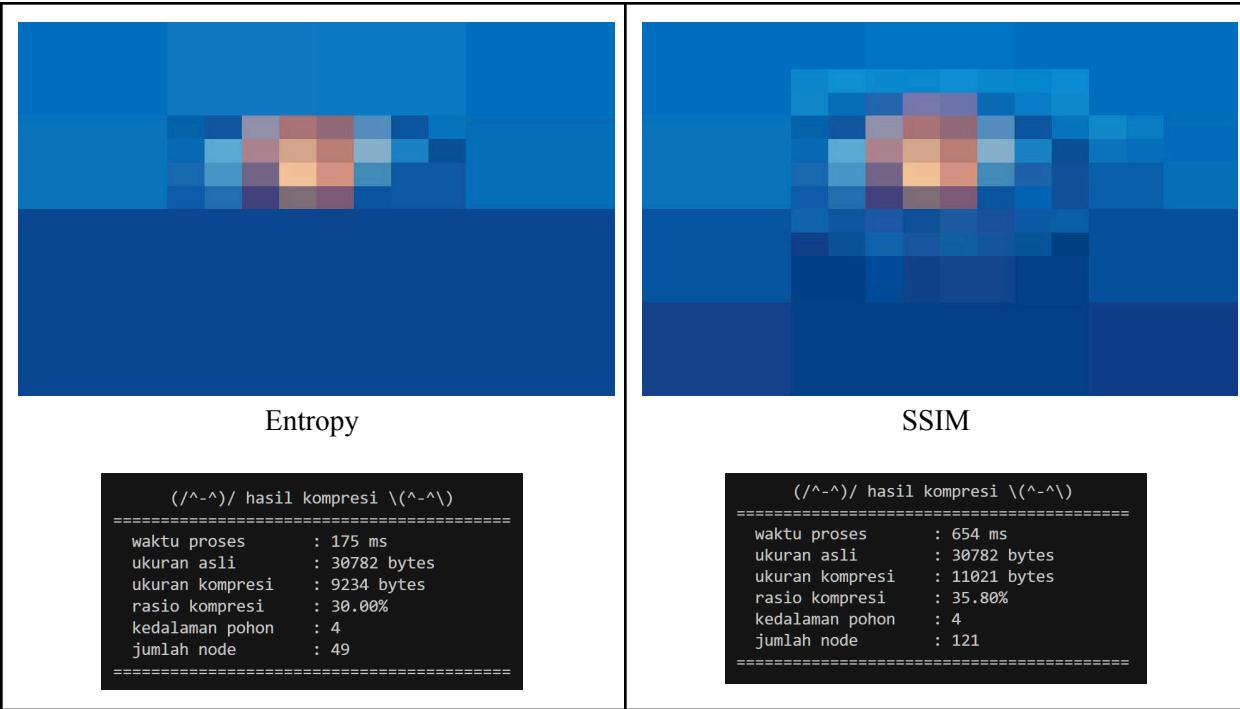
Mean Absolute Deviation

```
(/^-^)/ hasil kompresi \(^-^\  
=====  
waktu proses : 229 ms  
ukuran asli : 30782 bytes  
ukuran kompresi : 5965 bytes  
rasio kompresi : 19.38%  
kedalaman pohon : 0  
jumlah node : 1  
=====
```

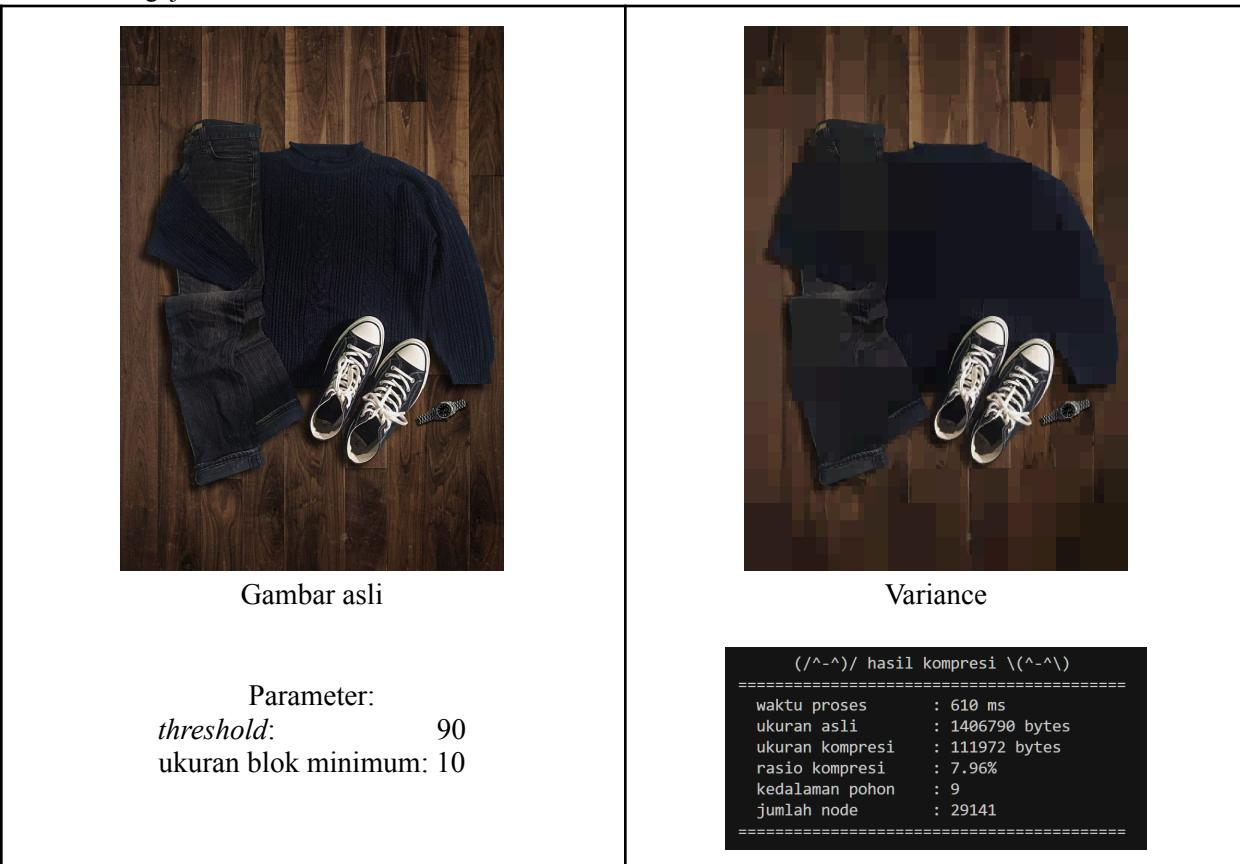


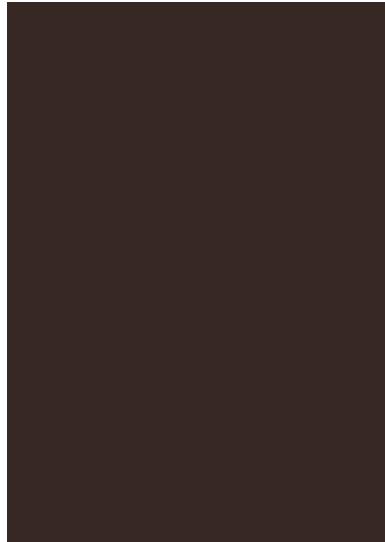
Max Pixel Difference

```
(/^-^)/ hasil kompresi \(^-^\  
=====  
waktu proses : 240 ms  
ukuran asli : 30782 bytes  
ukuran kompresi : 11078 bytes  
rasio kompresi : 35.99%  
kedalaman pohon : 4  
jumlah node : 129  
=====
```



Tabel 17. Pengujian 4





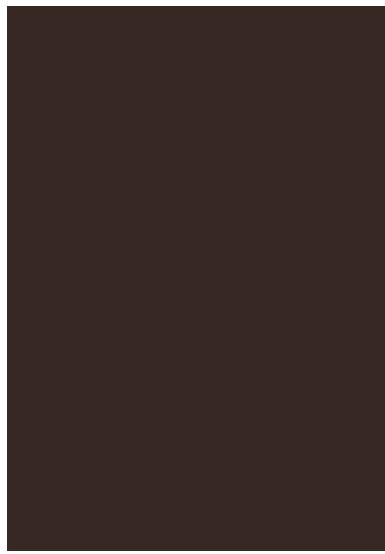
Mean Absolute Deviation

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 299 ms
ukuran asli      : 1406790 bytes
ukuran kompresi   : 4644 bytes
rasio kompresi    : 0.33%
kedalaman pohon   : 0
jumlah node       : 1
=====
```

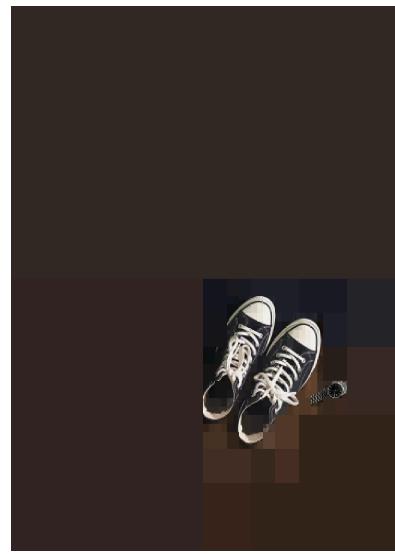


Max Pixel Difference

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 406 ms
ukuran asli      : 1406790 bytes
ukuran kompresi   : 63312 bytes
rasio kompresi    : 4.50%
kedalaman pohon   : 9
jumlah node       : 10737
=====
```



Entropy



SSIM

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses : 394 ms
ukuran asli : 1406790 bytes
ukuran kompresi : 4644 bytes
rasio kompresi : 0.33%
kedalaman pohon : 0
jumlah node : 1
=====
```

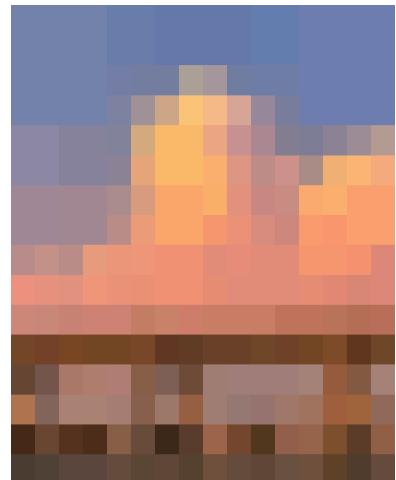
```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses : 7670 ms
ukuran asli : 1406790 bytes
ukuran kompresi : 40879 bytes
rasio kompresi : 2.91%
kedalaman pohon : 9
jumlah node : 9493
=====
```

Tabel 18. Pengujian 5



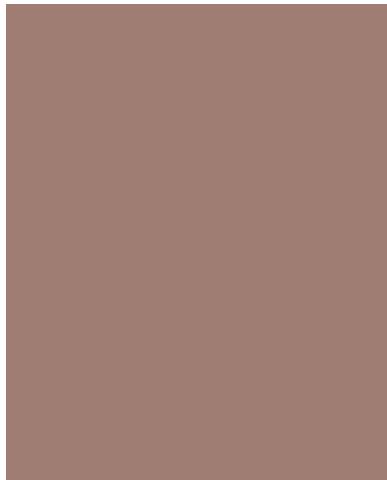
Gambar asli

Parameter:
threshold: 55
 ukuran blok minimum: 5500

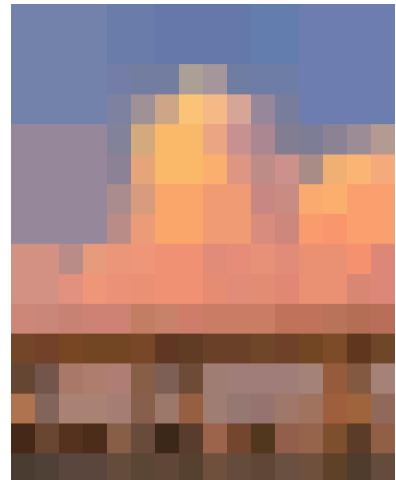


Variance

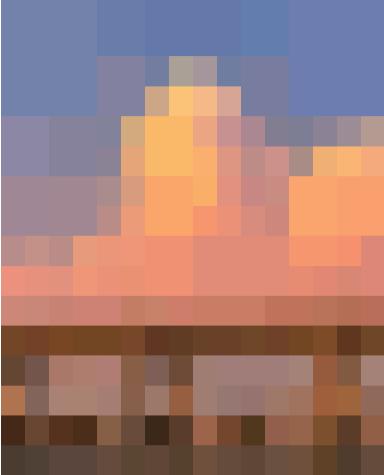
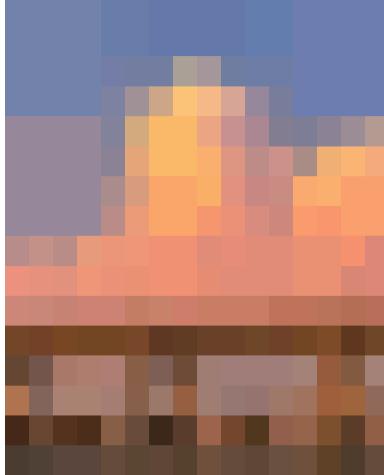
```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses : 372 ms
ukuran asli : 842057 bytes
ukuran kompresi : 30429 bytes
rasio kompresi : 3.61%
kedalaman pohon : 4
jumlah node : 249
=====
```



Mean Absolute Deviation



Max Pixel Difference

<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 230 ms ukuran asli : 842057 bytes ukuran kompresi : 4100 bytes rasio kompresi : 0.49% kedalaman pohon : 0 jumlah node : 1 =====</pre>	<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 301 ms ukuran asli : 842057 bytes ukuran kompresi : 29759 bytes rasio kompresi : 3.53% kedalaman pohon : 4 jumlah node : 237 =====</pre>
 <p style="text-align: center;">Entropy</p>	 <p style="text-align: center;">SSIM</p>
<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 308 ms ukuran asli : 842057 bytes ukuran kompresi : 28949 bytes rasio kompresi : 3.44% kedalaman pohon : 4 jumlah node : 233 =====</pre>	<pre>(/^-^)/ hasil kompresi \(^-^\ ===== waktu proses : 336 ms ukuran asli : 842057 bytes ukuran kompresi : 4100 bytes rasio kompresi : 0.49% kedalaman pohon : 0 jumlah node : 1 =====</pre>

Tabel 19. Pengujian 6

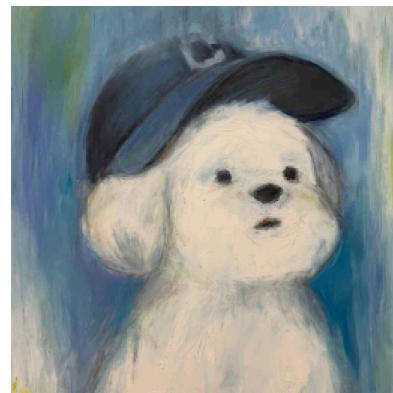
 <p style="text-align: center;">Gambar asli</p> <p style="text-align: center;">Parameter: threshold: 10</p>	 <p style="text-align: center;">Variance</p>
--	--

ukuran blok minimum: 10

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 381 ms
ukuran asli       : 1147984 bytes
ukuran kompresi    : 166685 bytes
rasio kompresi     : 14.52%
kedalaman pohon    : 8
jumlah node        : 58617
=====
```



Mean Absolute Deviation



Max Pixel Difference

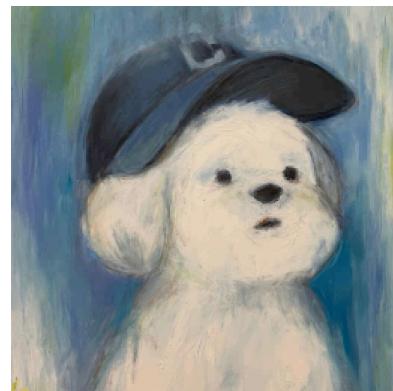
```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 321 ms
ukuran asli       : 1147984 bytes
ukuran kompresi    : 44923 bytes
rasio kompresi     : 3.91%
kedalaman pohon    : 8
jumlah node        : 5661
=====
```

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 334 ms
ukuran asli       : 1147984 bytes
ukuran kompresi    : 182057 bytes
rasio kompresi     : 15.86%
kedalaman pohon    : 8
jumlah node        : 67565
=====
```



Entropy

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 498 ms
ukuran asli       : 1147984 bytes
ukuran kompresi    : 210178 bytes
rasio kompresi     : 18.31%
kedalaman pohon    : 8
jumlah node        : 87309
=====
```



SSIM

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 18365 ms
ukuran asli       : 1147984 bytes
ukuran kompresi    : 181233 bytes
rasio kompresi     : 15.79%
kedalaman pohon    : 8
jumlah node        : 66589
=====
```

Tabel 20. Pengujian 7



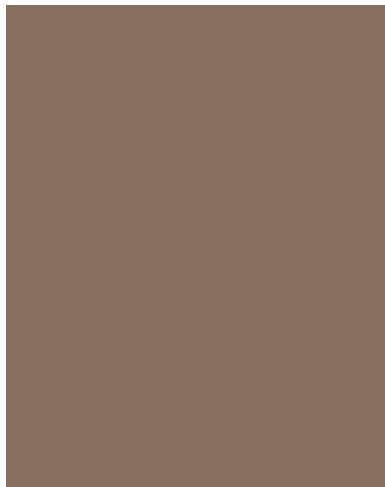
Gambar asli

Parameter:
threshold: 70
ukuran blok minimum: 150



Variance

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 166 ms
ukuran asli      : 804888 bytes
ukuran kompresi   : 22082 bytes
rasio kompresi    : 2.74%
kedalaman pohon   : 4
jumlah node       : 341
=====
```



Mean Absolute Deviation



Max Pixel Difference

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 87 ms
ukuran asli      : 804888 bytes
ukuran kompresi   : 2182 bytes
rasio kompresi    : 0.27%
kedalaman pohon   : 0
jumlah node       : 1
=====
```

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 99 ms
ukuran asli      : 804888 bytes
ukuran kompresi   : 22082 bytes
rasio kompresi    : 2.74%
kedalaman pohon   : 4
jumlah node       : 341
=====
```



Entropy

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 269 ms
ukuran asli       : 804888 bytes
ukuran kompresi    : 22747 bytes
rasio kompresi     : 2.83%
kedalaman pohon    : 6
jumlah node        : 5329
=====
```



SSIM

```
(/^-^)/ hasil kompresi \(^-^\
=====
waktu proses      : 853 ms
ukuran asli       : 804888 bytes
ukuran kompresi    : 22698 bytes
rasio kompresi     : 2.82%
kedalaman pohon    : 6
jumlah node        : 5425
=====
```

3.3. Analisis Pengujian Program

Pada tahap ini, dilakukan pengujian terhadap algoritma kompresi gambar berbasis Quadtree yang telah diimplementasikan. Algoritma kompresi gambar menggunakan QuadTree pada program ini diterapkan dengan menggunakan prinsip *divide and conquer*, dimana area gambar dibagi berulang kali menjadi sub-blok berukuran lebih kecil hingga memenuhi kondisi homogenitas atau ukuran minimum. Dalam hal ini, kompleksitas waktu dari algoritma berada pada kisaran $O(n^2 \log n)$ untuk gambar berukuran $n \times n$. Hal ini karena pada setiap tingkat rekursi, seluruh area gambar tetap dianalisis ($O(n^2)$ total piksel), dan proses rekursi dilakukan hingga kedalaman logaritmik sesuai ukuran gambar ($\log n$). Pada setiap level rekursi, sebuah blok gambar akan dibagi menjadi empat sub-blok, dan masing-masing dianalisis secara terpisah. Jika dalam kasus terburuk setiap piksel dianggap tidak homogen, maka setiap piksel akan menjadi sebuah *leaf node*, menghasilkan kompleksitas ruang dan waktu mendekati $O(n^2)$. Namun, dalam kasus terbaik dimana gambar sangat homogen, algoritma hanya menghasilkan satu *node*, memberikan kompleksitas waktu dan ruang $O(1)$.

Perbedaan hasil kompresi setiap metode:

Masing-masing metode perhitungan *error* memiliki hasil yang berbeda karena setiap metode memiliki cara pandang yang berbeda terhadap “kemiripan warna” dalam blok piksel.

1. Variance

Variansi adalah metode statistik yang mengukur seberapa besar penyebaran nilai-nilai warna piksel dalam suatu blok dibandingkan dengan rata-ratanya. Semakin besar nilai variansi, berarti nilai warna piksel dalam blok sangat bervariasi atau menyebar jauh dari rata-rata. Variansi cocok digunakan untuk mendeteksi gradasi warna atau area dengan perubahan halus, namun sangat sensitif terhadap perbedaan kecil. Oleh sebab itu, melalui beberapa uji coba, metode variansi mampu menghasilkan gambar yang masih memiliki kemiripan tinggi dengan gambar aslinya. Hal ini menjadikan variansi sangat cocok digunakan apabila tujuan utama adalah mempertahankan kualitas gambar semirip mungkin dengan aslinya, walaupun harus mengorbankan efisiensi kompresi dan menambah jumlah simpul dalam pohon Quadtree secara signifikan.

2. MAD (Mean Absolute Difference)

Mean Absolute Deviation (MAD) adalah metode perhitungan error yang mengukur rata-rata dari nilai absolut selisih antara setiap piksel dengan rata-rata warna piksel dalam satu blok gambar. Secara prinsip, MAD merepresentasikan seberapa jauh, secara rata-rata, nilai-nilai piksel menyimpang dari nilai tengahnya, tanpa mengkuadratkan selisih tersebut seperti pada metode variansi. Dalam MAD, blok-blok gambar yang memiliki sedikit variasi bisa dianggap cukup homogen dan tidak perlu dibagi lebih lanjut. Akibatnya, jumlah node lebih sedikit, waktu proses kompresi lebih cepat, dan ukuran *file* hasil kompresi pun menjadi lebih kecil.

3. MPD (Max Pixel Difference)

Max Pixel Difference (MPD) adalah metode perhitungan error yang menilai seberapa besar jarak antara piksel paling terang dan piksel paling gelap dalam satu blok gambar. Dalam konteks gambar berwarna, MPD biasanya dihitung secara terpisah untuk tiap channel warna (merah, hijau, biru), lalu dirata-ratakan. Hasil akhir dari pengujian MPD menunjukkan struktur pohon yang cukup dalam. Meski demikian, karena MPD tidak mempertimbangkan keseluruhan distribusi data, kadang-kadang ia bisa melewatkannya variasi halus yang terdeteksi oleh metode seperti variansi atau entropy.

4. Entropy

Dalam implementasinya pada algoritma QuadTree, entropy menjadi metode yang adaptif, melihat seberapa signifikan variasi tersebut dalam konteks distribusinya. Hal ini membuat entropy menjadi penyeimbang antara metode sensitif seperti variansi dan metode kasar seperti MPD. Dari sisi struktur pohon, entropy cenderung menghasilkan pohon yang tidak terlalu dalam tapi tetap responsif terhadap variasi informasi visual.

5. SSIM (Structural Similarity Index)

SSIM adalah metode evaluasi kualitas gambar yang didasarkan pada persepsi visual manusia. Tidak seperti metode *error* lainnya yang hanya melihat perbedaan nilai piksel secara matematis, SSIM menilai kesamaan struktur visual antara dua gambar. Karena pendekatannya yang mempertimbangkan cara manusia memproses informasi visual, SSIM menjadi metode yang sangat efektif dalam mempertahankan kualitas perceptual saat gambar dikompresi atau direduksi. Hasil pengujian menunjukkan bahwa SSIM merupakan metode yang sangat efektif menjaga kualitas visual, bahkan ketika struktur pohon QuadTree tidak terlalu dalam.

3.4. Implementasi Bonus

3.4.1. Bonus SSIM

Structural Similarity Index merupakan salah satu metrik kuantitatif yang dapat digunakan untuk mengukur kemiripan antara dua gambar, dengan penilaian metrik didasarkan terhadap persepsi manusia dalam merespons intensitas warna dengan mempertimbangkan struktur, pencahayaan, dan kontras. SSIM memanfaatkan fakta bahwa tidak seperti mesin, manusia tidak melihat perbedaan pixel secara mentah, tetapi memperhatikan struktur, tekstur, dan pola-pola visual yang terdapat pada gambar tersebut. Maka, dapat dikatakan bahwa SSIM memperhitungkan pola struktural antara kedua gambar yang dibandingkan.

Sesuai dengan spek, perhitungan nilai SSIM dihitung menggunakan rumus

$$SSIM_c(x, y) = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(2\sigma_{xy,c} + C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + \sigma_{y,c}^2 + C_2)}$$

Gambar 1. Perhitungan SSIM untuk suatu kanal warna

$$SSIM_{RGB} = w_R \cdot SSIM_R + w_G \cdot SSIM_G + w_B \cdot SSIM_B$$

Gambar 2. Perhitungan SSIM untuk suatu pixel

Dimana C_n merupakan nilai konstanta $(K_n * L)^2$ untuk suatu nilai K_n kecil dan L nilai *range* maksimum pixel, serta w_n adalah kontribusi dari setiap kanal warna terhadap persepsi keseluruhan warna. C_n bertujuan untuk menjaga kestabilan nilai dengan memastikan tidak dilakukan pembagian dengan 0, dan w_n bertujuan untuk mencerminkan bagaimana manusia menilai kualitas gambar, yaitu lebih sensitif terhadap warna hijau.

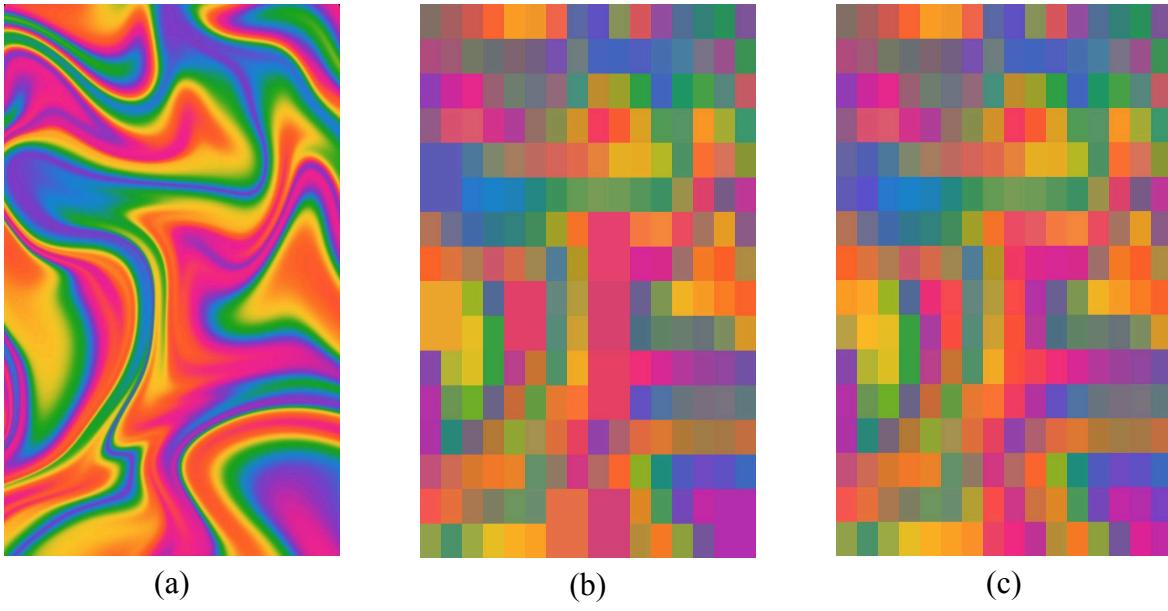
Dibandingkan dengan metode perhitungan *error* lainnya, menggunakan SSIM sebagai metode perhitungan *error* memiliki cara yang berbeda karena metode perhitungan *error* lainnya

melakukan perhitungan terhadap nilai pixel dalam satu blok, sedangkan SSIM membandingkan nilai pixel antara dua blok. Hal ini mengakibatkan terdapat perbedaan algoritma untuk melakukan kompresi menggunakan SSIM sebagai metode perhitungan *error*:

1. Setelah gambar di proses menjadi *array of array of integer* yang disebut dengan arrayInput, proses membentuk QuadTree berdasarkan SSIM diawali dengan melakukan *clone* terhadap *array* tersebut, membentuk array baru bernama arrayOutput.
2. Dilakukan normalisasi warna pada arrayOutput, sehingga setiap pixel pada arrayOutput sekarang bernilai sama
3. Perhitungan SSIM dilakukan antara arrayInput dan arrayOutput yang menghasilkan persentase kemiripan di antara keduanya.
4. Karena nilai SSIM menyatakan kemiripan antara dua blok, jika nilai 100 dikurangi SSIM sudah lebih kecil daripada *threshold*, maka proses diselesaikan dan arrayOutput menjadi hasil dari kompresi gambar.
5. Jika nilai 100 dikurangi SSIM masih lebih besar daripada *threshold*, maka arrayInput dibagi menjadi empat bagian sesuai dengan konsep QuadTree, dan setiap bagian diproses melalui tahapan yang sama seperti sebelumnya sebagai penerapan dari algoritma *divide and conquer*.

Karena SSIM mencerminkan persepsi visual manusia, maka hasil kompresi yang dihasilkan memiliki keuntungan dibandingkan metode perhitungan *error* lainnya. Kualitas gambar hasil kompresi lebih terjaga dan efisiensi program juga lebih optimal karena penggunaan SSIM mengurangi pembagian blok yang secara visual manusia dikatakan mirip.

Seperti pada contoh gambar X, untuk parameter *threshold* dan *minBlockSize* bernilai samma, metode perhitungan *error* Mean Absolute Deviation memberikan hasil kompresi yang ‘kusam’ karena melakukan kompresi hanya berdasarkan perhitungan. Sedangkan untuk metode perhitungan *error* SSIM, kecerahan dari gambar masih dipertahankan, menyesuaikan dengan gambar asli yang relatif cerah.



Gambar 3. Perbandingan hasil kompresi MAD (gambar b) dengan SSIM (gambar c)

3.4.2. Bonus GIF

GIF animasi dalam program ini bertujuan untuk memvisualisasikan proses kompresi gambar menggunakan algoritma QuadTree. Tidak hanya menyimpan hasil akhir gambar yang telah dikompresi, program juga dapat membuat animasi yang memperlihatkan bagaimana gambar secara bertahap dibagi menjadi blok-blok berdasarkan kemiripan warna atau struktur visual lainnya. Proses pembuatan GIF dilakukan hanya jika pengguna memberikan path output untuk GIF. Jika path tersebut valid, sistem akan mengaktifkan `createGif` dan mempersiapkan objek `AnimatedGifEncoder` untuk merekam setiap langkah pembentukan struktur QuadTree.

1. Pembentukan Struktur QuadTree dan Pencatatan Render Steps

Setelah membaca gambar input dan mengubahnya menjadi matriks piksel, program melanjutkan ke proses utama pembentukan struktur pohon QuadTree. Pada tahap ini, gambar dianalisis dan dibagi menjadi blok-blok, dan setiap blok diperiksa tingkat keseragaman warnanya berdasarkan metode error tertentu. Jika blok tidak cukup seragam dan ukurannya masih layak dibagi, maka blok tersebut dibagi menjadi empat sub-blok. Proses ini dilakukan secara rekursif dan tercatat dalam urutan level-order traversal atau *breadth-first search* (BFS). Informasi setiap subdivisi disimpan dalam struktur `renderSteps`, yaitu daftar per level dari node-node yang terbentuk, yang nantinya akan digunakan untuk merender animasi.

2. Render Animasi Berdasarkan Urutan Subdivisi

Setelah `renderSteps` terisi, sistem membuat animasi GIF berdasarkan urutan subdivisi. Proses ini diawali dengan pembuatan kanvas kosong berwarna putih, yang akan menjadi frame pembuka animasi. Setelah itu, program melakukan *loop* per level subdivisi, dan di

setiap level, semua blok yang terbentuk pada tingkat tersebut digambar pada kanvas menggunakan warna rata-rata blok tersebut. Setelah satu level selesai digambar, kanvas saat ini disalin dan ditambahkan sebagai satu frame ke dalam GIF animasi.

3. Efek Visual GIF

Efek visual yang dihasilkan dari proses ini adalah transisi bertahap dari gambar kosong ke gambar utuh, di mana bagian-bagian gambar secara perlahan “muncul” dalam bentuk blok-blok besar yang kemudian membelah diri menjadi blok-blok lebih kecil.

4. Penyelesaian dan Penyimpanan Hasil

Setelah semua level subdivisi selesai dirender, proses pembuatan GIF diakhiri dan file disimpan di lokasi output yang ditentukan oleh pengguna. Program juga menghitung dan menampilkan informasi penting seperti jumlah frame dalam GIF, ukuran file asli dan hasil kompresi, rasio kompresi, waktu proses, serta kedalaman dan jumlah node dalam struktur QuadTree.

Gambar 4. Contoh GIF Hasil Kompresi QuadTree

LAMPIRAN

Tautan Repository Github

https://github.com/graceevelyns/Tucil2_13523031_13523087

Hasil Akhir Tugas Kecil 2

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	✓	