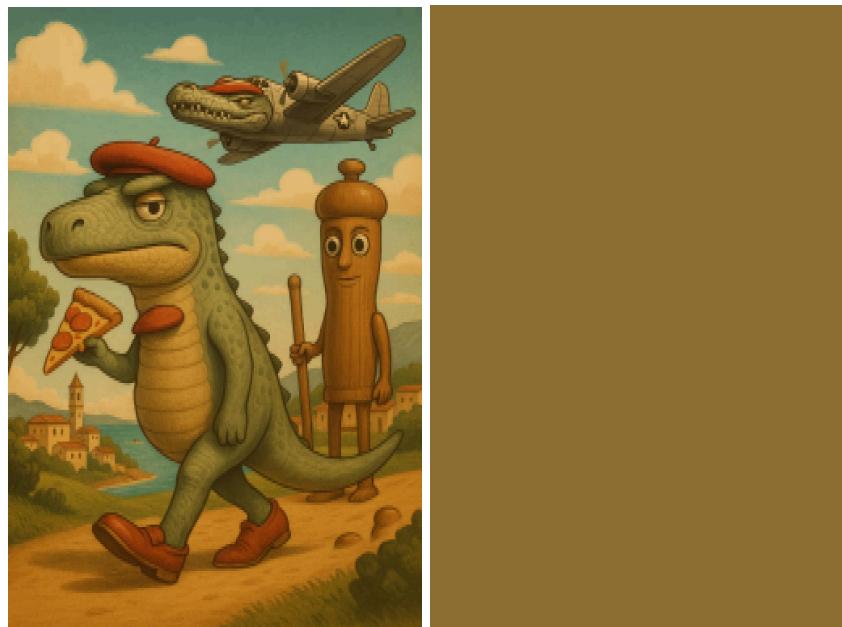


TUGAS KECIL 2
IF2211 STRATEGI ALGORITMA

Kompresi Gambar dengan Metode Quadtree



Oleh:

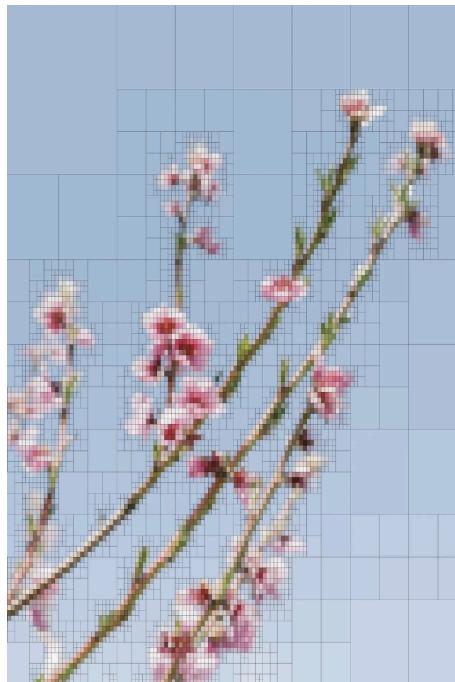
13523022 - Kenneth Ricardo Chandra
13523033 - Alvin Christopher Santausa

Daftar Isi

Daftar Isi	2
BAB I	
DESKRIPSI MASALAH	3
BAB II	
DASAR TEORI	4
2.1 Algoritma Divide and Conquer	4
2.2 Quadtree	4
2.3 Error Measurement Method	4
2.3.1 Variance	4
2.3.2 Mean Absolute Deviation (MAD)	5
2.3.3 Max Pixel Difference	6
2.3.4 Entropy	6
2.3.5 Structural Similarity Index (SSIM)	6
BAB III	
IMPLEMENTASI	8
3.1 Implementasi Algoritma Divide and Conquer pada program Kompresi Gambar Dengan Metode Quadtree	8
3.2 Implementasi program dalam bahasa java	9
3.2.1 Main.java	9
3.2.2 QuadTreeNode.java	12
3.2.3 QuadTree.java	14
3.2.4 Method.java	20
3.2.5 FileProcessor.java	27
3.2.6 GifSequenceWriter.java	28
3.3 Penjelasan Implementasi Spesifikasi Bonus yang Dikerjakan	31
3.3.1 SSIM	31
3.3.2 Pembuatan Gif	33
3.3.3 Target Persentase Kompresi	34
BAB IV	
PENGUJIAN	37
4.1 Gambar 1 - pattern.jpg	37
4.2 Gambar 2 - ranger.png	40
4.3 Gambar 3 - ferrari.jpg	45
BAB V	
PENUTUP	50
5.1 Analisis Percobaan Algoritma Divide and Conquer dalam Kompresi Gambar dengan Metode Quadtree	50
5.2 Kesimpulan	50

BAB I

DESKRIPSI MASALAH



Gambar 1. Quadtree dalam Kompresi Gambar

(Sumber: <https://medium.com/@tannerwyork/quadtrees-for-image-processing-302536c95c00>)

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.

BAB II

DASAR TEORI

2.1 Algoritma Divide and Conquer

Algoritma Divide and Conquer merupakan sebuah algoritma yang dapat digunakan untuk menyelesaikan berbagai masalah dengan cara membagi masalah tersebut menjadi beberapa bagian yang lebih kecil, lalu satu per satu diselesaikan. Hasil dari masalah yang telah diperkecil akan digabung kembali menjadi satu sehingga menjadi solusi dari permasalahan tersebut.

Untuk menerapkan algoritma ini terhadap suatu masalah, masalah tersebut harus dapat dibagi menjadi beberapa bagian yang kecil, yang dapat dilakukan secara rekursif agar bagian-bagian yang masih cukup besar dapat semakin diperkecil. Dengan membagi permasalahan menjadi beberapa bagian yang lebih kecil, permasalahan ini diharapkan dapat lebih mudah diselesaikan dan meningkatkan efisiensi pekerjaan. Salah satu penggunaan algoritma Divide and Conquer adalah dengan menggunakan Quadtree

2.2 Quadtree

Quadtree adalah sebuah metode yang digunakan dalam algoritma Divide and Conquer dengan membuat sebuah struktur data bertingkat layaknya sebuah pohon yang akan dibagi menjadi empat bagian atau kuadran secara rekursif. Nama Quadtree sendiri berasal dari kata “quad” yang berarti empat karena dapat menghasilkan empat “anak”.

Dalam hal ini, Quadtree digunakan untuk melakukan kompresi terhadap gambar. Quadtree akan membagi sebuah gambar menjadi 4 bagian yang mengandung value RGB. Quadtree akan secara rekursif membagi bagian-bagian dari gambar tersebut hingga mendapatkan suatu value RGB yang sama pada setiap pixel dari bagian tersebut sama. Sebuah bagian dari gambar tersebut disebut sebuah node dan bagian yang seluruhnya memiliki value RGB yang sama disebut sebagai leaf atau daun.

2.3 Error Measurement Method

Hasil dari kompresi suatu gambar pasti berbeda dari gambar aslinya, dari ukurannya hingga ke pixelnya. Maka dari itu, terdapat pula sejumlah error (perbedaan) dari gambar aslinya. Untuk mengukur jumlah dari error tersebut, maka terdapat beberapa metode yang dapat digunakan seperti Variance, Mean Absolute Deviation atau MAD, Max Pixel Difference, Entropy dan Structural Similarity Index.

2.3.1 Variance

Metode Variance adalah salah satu metode untuk menghitung error yang dihasilkan dari sebuah gambar yang dikompresi dengan mengukur seberapa menyebarluhnya nilai-nilai pixel dalam sebuah block gambar dari nilai rata-ratanya. Apabila nilai variansi ini tinggi, berarti perbedaan pixel antar block itu besar, yang mengakibatkan block tersebut kurang homogen. Rumusnya sendiri demikian

$$\sigma_c^2 = \frac{1}{N} \sum_{i=1}^N (P_{i,c} - \mu_c)^2$$

Dengan keterangan :

- σ_c^2 = Variansi tiap kanal warna c (R, G, B) dalam satu blok
- $P_{i,c}$ = Nilai piksel pada posisi i untuk kanal warna c
- μ_c = Nilai rata-rata tiap piksel dalam satu blok
- N = Banyaknya piksel dalam satu blok

Setelah mendapatkan variansi tiap warna (R,G,B), ketiganya akan dicari rata-ratanya agar mendapatkan variansi untuk setiap warna.

$$\sigma_{RGB}^2 = \frac{\sigma_R^2 + \sigma_G^2 + \sigma_B^2}{3}$$

Hasil inilah yang akan menjadi jumlah error dari gambar yang telah di kompresi.

2.3.2 Mean Absolute Deviation (MAD)

Metode Mean Absolute Deviation atau MAD adalah metode lain yang dapat menghitung error yang dihasilkan dari mengkompresi gambar. Metode ini mirip dengan metode Variance, namun bedanya metode ini menghitung ukuran rata-rata dari selisih absolut setiap nilai pixel terhadap rata-ratanya. Rumusnya demikian

$$MAD_c = \frac{1}{N} \sum_{i=1}^N |P_{i,c} - \mu_c|$$

Dengan keterangan :

- MAD_c = Mean Absolute Deviation tiap kanal warna c (R, G, B) dalam satu blok
- $P_{i,c}$ = Nilai piksel pada posisi i untuk kanal warna c
- μ_c = Nilai rata-rata tiap piksel dalam satu blok
- N = Banyaknya piksel dalam satu blok

Sama seperti pada metode Variance, setelah mendapatkan MAD dari masing-masing warna, metode ini juga akan mencari rata-rata dari ketiga MAD yang telah diperoleh.

$$MAD_{RGB} = \frac{MAD_R + MAD_G + MAD_B}{3}$$

Hasil inilah yang akan digunakan untuk menjadi error dari hasil kompresi gambar.

2.3.3 Max Pixel Difference

Metode Max Pixel Difference, sesuai namanya adalah metode yang menghitung perbedaan terbesar nilai suatu pixel dengan nilai pixel rata-rata atau pixel lainnya. Hal ini akan dilakukan untuk setiap jenis kanal warna (R,G,B) Rumusnya demikian

$$D_c = \max(P_{i,c}) - \min(P_{i,c})$$

Dengan keterangan :

D_c	=	Selisih antara piksel dengan nilai max dan min tiap kanal warna c (R, G, B) dalam satu blok
$P_{i,c}$	=	Nilai piksel pada posisi i untuk channel warna c

Apabila seluruh selisih terbesar sudah didapatkan, maka akan dicari rata-ratanya dan nilai itulah yang akan menjadi hasil error dari kompresi gambar.

$$D_{RGB} = \frac{D_R + D_G + D_B}{3}$$

2.3.4 Entropy

Metode Entropy adalah metode lain yang dapat digunakan untuk mengukur error dari kompresi gambar dengan mengukur tingkat ketidakteraturan atau informasi yang terdapat dalam suatu block gambar. Nilai yang tinggi berarti gambar kompleks atau memiliki banyak variasi warna. Rumusnya demikian

$$H_c = - \sum_{i=1}^N P_c(i) \log_2(P_c(i))$$

Dengan keterangan :

H_c	=	Nilai entropi tiap kanal warna c (R, G, B) dalam satu blok
$P_c(i)$	=	Probabilitas piksel dengan nilai i dalam satu blok untuk tiap kanal warna c (R, G, B)

Apabila seluruh nilai entropi dari tiap kanal warna (R,G,B) telah didapatkan, maka akan dicari nilai rata-ratanya untuk mendapatkan nilai error dari kompresi gambar yang dilakukan

$$H_{RGB} = \frac{H_R + H_G + H_B}{3}$$

2.3.5 Structural Similarity Index (SSIM)

Metode Structural Similarity Index atau SSIM adalah metode yang mengecek apakah suatu gambar memiliki struktur yang sama dengan gambar lain atau tidak. Metode ini selain menggunakan intensitas, juga menggunakan kontras dan struktur lokal dari gambar yang

sedang diperiksa. Pada hal ini, SSIM yang dibandingkan adalah SSIM dari gambar sebelum dan sesudah kompresi. Metode ini menggunakan rumus seperti demikian

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

Dengan keterangan :

μ_x, μ_y : rata-rata intensitas gambar x dan y

σ_x^2, σ_y^2 : variansi gambar x dan y

σ_{xy} : kovarians antara x dan y

C_1, C_2 : konstanta untuk stabilisasi perhitungan

Metode ini memiliki konstanta untuk stabilisasi perhitungan yang bisa didapatkan dari rentang dinamis dari pixel gambar. Untuk metode ini, dianggap gambar yang diproses adalah gambar 24-bit RGB dengan 8-bit per kanal sehingga dari situ, didapatkan nilai konstanta $C_1 = 6.5025$ dan $C_2 = 58.5225$. Hal ini didapatkan dari rumus demikian

$$C_1 = (K_1 \cdot L)^2, \quad C_2 = (K_2 \cdot L)^2$$

Dengan keterangan :

L sebagai nilai maksimal pixel atau 255

$K_1 \approx 0.01$, $K_2 \approx 0.03$ (nilai default berdasarkan paper asli SSIM)

Untuk perhitungan intensitas gambar x dan y, dibutuhkan nilai rata-ratanya yang bisa didapat dari rumus ini

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Rumus ini berasal dari konversi warna RGB ke grayscale atau luminance(Y), dengan konstanta tersebut merupakan standar ITU-R BT.601 yang merupakan standar untuk video encoding dan mendefinisikan konversi warna dari RGB ke grayscale. Warna merah berpengaruh 29.9%, hijau berpengaruh 58.7% dan biru berpengaruh 11.4% yang dilihat dari pengaruh warna tersebut ke mata manusia. Rumus ini juga yang digunakan untuk menghitung SSIM terakhir setelah digabung dari penghitungan masing-masing RGB

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

BAB III

IMPLEMENTASI

3.1 Implementasi Algoritma Divide and Conquer pada program Kompresi Gambar Dengan Metode Quadtree

Implementasinya dalam pseudocode

```
// Asumsi fungsi yang tidak di tulis di sini sudah terdefinisi (karena di ??  
// sini hanya fokus membahas algoritma divide and conquer untuk pembentukan  
// quad tree-nya saja)  
// fungsi createQuadTree membuat QuadTree dari sebuah gambar dengan  
// threshold dan minBlockSize tertentu (terlihat dari parameter image,  
// minBlockSize, threshold)  
  
function CreateQuadTree(image, minBlockSize, threshold, errorMethod) ->  
    QuadTree  
    // menginisialisasi node asumsi sudah ada constructornya  
    root <- new QuadTreeNode(0, 0, image.width, image.height)  
  
    // anggap class QuadTree memiliki matrix red, green, blue  
    red, green, blue <- ExtractColorArrays(image)  
    // menginisialisasi atribut avgColor pada node root dengan warna rata-rata  
    // keseluruhan  
    root.avgColor <- CalculateAverageColor(0, 0, image.width, image.height)  
  
    // Melanjutkan membuat pohon  
    BuildTree(root, 0)  
  
    -> QuadTree  
End function  
  
// Asumsi fungsi yang tidak di tulis di sini sudah terdefinisi (karena di ??  
// Pada fungsi ini, Tree akan terus membentuk cabang (split) jika kondisi  
// terpenuhi (shouldSplit). Kondisi tersebut adalah jika error/variansi di  
// atas threshold dan ukuran block saat ini di atas minBlockSize dan ukuran  
// keempat block hasil bagi block saat ini juga di atas minBlockSize  
// fungsi Split diasumsikan sudah terdefinisi dan akan membagi blok saat ini  
// menjadi 4 bagian sama besar. Kemudian akan dihitung avgColornya  
// berdasarkan matrix RGB yang didapat dari image original  
  
procedure BuildTree(node, depth) ->  
    // Update maximum tree depth  
    treeDepth <- MAX(treeDepth, depth)  
  
    // Check if this node should be split further
```

```

if ShouldSplit(node) then
    // Split the node into four children
    SplitNode(node, minBlockSize)

        for EACH child in node.children
            child.avgColor <- CalculateAverageColor(child.x, child.y,
child.width, child.height)
            nodeCount <- nodeCount + 1
            BuildTree(child, depth + 1)
        end for
    end if
END FUNCTION

```

Penerapan algoritma Divide and Conquer pada program kompresi gambar kami akan membagi gambar yang dimasukan kedalam program menjadi bagian-bagian yang lebih kecil secara rekursif menggunakan Quadtree. Program pertama-tama akan meminta beberapa input seperti gambar yang ingin di kompres, metode pengukuran error, nilai threshold (untuk menentukan batas apakah bagian image dapat dibagi lagi/tidak), ukuran block minimum, serta output hasil kompresi.

Setelah itu gambar akan dikonversi menjadi sebuah matriks yang memiliki atribut warna RGB sebagai tahapan inisialisasi dari algoritma Divide and Conquer dan disimpan pada class QuadTree. Setelah itu, Quadtree akan menginisialisasikan ‘root’ (node 1)nya dengan avgColor dari matrix RGB ukuran asli. Kemudian, root/node tersebut akan dicek kehomogenannya dan melakukan pemecahan. Bila variansi root/node tersebut masih di atas threshold yang ditentukan dan block tersebut masih dapat dibagi menjadi lebih kecil (di atas ukuran block minimum), maka root/node tersebut akan dipecah menjadi 4 bagian node sama besar dan proses ini diulang kepada setiap anak dari node hingga pada akhirnya sudah menyentuh batas minimum block atau threshold.

3.2 Implementasi program dalam bahasa java

3.2.1 Main.java

Implementasi dari Main.java

```

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

```

```

Scanner scanner = new Scanner(System.in);
long startTime, endTime;

try {
    System.out.print("Masukkan absolute path ke input image: ");
    String inputPath = scanner.nextLine();
    BufferedImage originalImage =
FileProcessor.loadImage(inputPath);

    System.out.print(
        "Masukkan target compression rate (0-1, 0 untuk
menggunakan threshold dan minimum block size secara manual): ");
    double targetCompressionRate =
Double.parseDouble(scanner.nextLine());

    int errorMethod = 1;
    double threshold = 0;
    int minBlockSize = 4;

    if (targetCompressionRate == 0) {
        System.out.println("Select error measurement method:");
        System.out.println("1. Variance");
        System.out.println("2. Mean Absolute Deviation (MAD)");
        System.out.println("3. Max Pixel Difference");
        System.out.println("4. Entropy");
        System.out.println("5. Structural Similarity Index (SSIM)");
        System.out.print("Masukkan pilihan (1-5): ");

        errorMethod = Integer.parseInt(scanner.nextLine());

        System.out.print("Masukkan nilai threshold: ");
        threshold = Double.parseDouble(scanner.nextLine());

        System.out.print("Masukkan ukuran minimum block: ");
        minBlockSize = Integer.parseInt(scanner.nextLine());
    }

    System.out.print("Masukkan absolute path ke output image: ");
    String outputPath = scanner.nextLine();

    System.out.print(
        "\nApakah Anda ingin membuat GIF yang menunjukkan
kompresi dari setiap level kedalaman? (y/n): ");
    String generateGif = scanner.nextLine().trim().toLowerCase();

    String gifPath = null;
    if (generateGif.equals("y")) {
        System.out.print("Masukkan absolute path ke output GIF: ");

```

```

        gifPath = scanner.nextLine();
    }

    startTime = System.currentTimeMillis();

    QuadTree quadTree;

    if (targetCompressionRate > 0 && targetCompressionRate <= 1) {

        File tempDir = new File(new File(outputPath).getParent(),
        "temp");
        if (!tempDir.exists()) {
            tempDir.mkdirs();
        }
        String fileExtension =
FileProcessor.getFileExtension(inputPath);

        quadTree = Method.findOptimalParameters(originalImage, 1,
targetCompressionRate, outputPath,
                    outputPath, tempDir.getAbsolutePath(),
fileExtension);

        FileProcessor.deleteDirectory(tempDir);
    } else {
        quadTree = new QuadTree(originalImage, minBlockSize,
threshold, errorMethod);
    }

    BufferedImage compressedImage =
quadTree.generateCompressedImage();

    if (gifPath != null) {
        try {
            QuadTree.generateCompressionGif(quadTree, gifPath);
            System.out.println("GIF berhasil dibuat di: " +
gifPath);
        } catch (Exception e) {
            System.err.println("Error creating GIF: " +
e.getMessage());
        }
    }

    endTime = System.currentTimeMillis();

    FileProcessor.saveImage(compressedImage, outputPath);

    long originalSize = FileProcessor getFileSize(inputPath);
    long compressedSize = FileProcessor getFileSize(outputPath);

```

```

        double compressionPercentage =
quadTree.calculateCompressionPercentage(originalSize, compressedSize);

        System.out.println("\nResults:");
        System.out.println("Waktu eksekusi: " + (endTime - startTime) +
" ms");
        System.out.println("Ukuran asli image: " + originalSize + " bytes");
        System.out.println("Ukuran setelah dikompresi: " +
compressedSize + " bytes");
        System.out.println("Persentase kompresi: " +
String.format("%.2f", compressionPercentage) + "%");
        System.out.println("Kedalaman tree (depth): " +
quadTree.getTreeDepth());
        System.out.println("jumlah nodes: " + quadTree.getNodeCount());
        System.out.println("Alamat output: " + outputPath);
        System.out.println("Alamat gif: " + gifPath);

    } catch (IOException e) {
        System.err.println("Error processing image: " + e.getMessage());
        e.printStackTrace();
    } catch (Exception e) {
        System.err.println("An error occurred: " + e.getMessage());
        e.printStackTrace();
    } finally {
        scanner.close();
    }
}
}

```

3.2.2 QuadTreeNode.java

Implementasi dari QuadTreeNode.java

```

import java.awt.Color;

public class QuadTreeNode {
    private int x, y;
    private int width, height;
    private Color avgColor;
    private QuadTreeNode[] children;
    private boolean isLeaf;

    public QuadTreeNode(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
    }
}

```

```
    this.width = width;
    this.height = height;
    this.isLeaf = true;
    this.children = null;
    this.avgColor = Color.BLACK;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public int getWidth() {
    return width;
}

public int getHeight() {
    return height;
}

public Color getAvgColor() {
    return avgColor;
}

public void setAvgColor(Color avgColor) {
    this.avgColor = avgColor != null ? avgColor : Color.BLACK;
}

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

public boolean isLeaf() {
    return isLeaf;
}

public void setLeaf(boolean isLeaf) {
    this.isLeaf = isLeaf;
}

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

3.2.3 QuadTree.java

Implementasi dari QuadTree.java

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import javax.imageio.stream.ImageOutputStream;
import javax.imageio.stream.FileImageOutputStream;
import java.io.IOException;
import java.io.File;

public class QuadTree {
    BufferedImage originalImage;
    private QuadTreeNode root;
    private int minBlockSize;
    private double threshold;
    private int treeDepth;
    private int nodeCount;
    private int errorMethod;

    private int[][][] red;
    private int[][][] green;
    private int[][][] blue;

    public QuadTree(BufferedImage image, int minBlockSize, double threshold,
int errorMethod) {
        originalImage = image;
        this.minBlockSize = minBlockSize;
        this.threshold = threshold;
        this.treeDepth = 0;
        this.nodeCount = 1;
        this.errorMethod = errorMethod;
        root = new QuadTreeNode(0, 0, image.getWidth(), image.getHeight());

        this.red = new int[root.getHeight()][root.getWidth()];
        this.green = new int[root.getHeight()][root.getWidth()];
        this.blue = new int[root.getHeight()][root.getWidth()];

        for (int y = 0; y < root.getHeight(); y++) {
            for (int x = 0; x < root.getWidth(); x++) {
                Color color = new Color(image.getRGB(x, y));
                red[y][x] = color.getRed();
                green[y][x] = color.getGreen();
                blue[y][x] = color.getBlue();
            }
        }
    }

    root.setAvgColor(getAverageColor(0, 0, root.getWidth()),
```

```

root.getHeight()));

        buildTree(root, 0);
    }

private void buildTree(QuadTreeNode node, int depth) {
    this.treeDepth = Math.max(treeDepth, depth);

    if (shouldSplit(node)) {
        splitNode(node, minBlockSize);

        for (QuadTreeNode child : node.getChildren()) {
            child.setAvgColor(getAverageColor(child.getX(),
child.getY(), child.getWidth(), child.getHeight()));
            nodeCount++;
            buildTree(child, depth + 1);
        }
    }
}

// hitung warna rata-rata dri matrix red green blue
public Color getAverageColor(int startX, int startY, int regionWidth,
int regionHeight) {
    long sumR = 0, sumG = 0, sumB = 0;
    int count = 0;

    int endX = Math.min(startX + regionWidth, root.getWidth());
    int endY = Math.min(startY + regionHeight, root.getHeight());

    for (int y = startY; y < endY; y++) {
        for (int x = startX; x < endX; x++) {
            sumR += red[y][x];
            sumG += green[y][x];
            sumB += blue[y][x];
            count++;
        }
    }

    if (count > 0) {
        int avgR = (int) (sumR / count);
        int avgG = (int) (sumG / count);
        int avgB = (int) (sumB / count);
        return new Color(avgR, avgG, avgB);
    } else {
        return Color.BLACK;
    }
}

```

```

public void splitNode(QuadTreeNode node, int minBlockSize) {
    QuadTreeNode[] children = new QuadTreeNode[4];

    int minDimension = (int) Math.ceil(Math.sqrt(minBlockSize));

    int newWidth1 = calculateSplitSize(node.getWidth(), minDimension);
    int newHeight1 = calculateSplitSize(node.getHeight(), minDimension);

    int newWidth2 = node.getWidth() - newWidth1;
    int newHeight2 = node.getHeight() - newHeight1;

    int x = node.getX();
    int y = node.getY();

    children[0] = new QuadTreeNode(x, y, newWidth1, newHeight1);
    children[1] = new QuadTreeNode(x + newWidth1, y, newWidth2,
newHeight1);
    children[2] = new QuadTreeNode(x, y + newHeight1, newWidth1,
newHeight2);
    children[3] = new QuadTreeNode(x + newWidth1, y + newHeight1,
newWidth2, newHeight2);

    node.setChildren(children);
    node.setLeaf(false);
}

// Moved from QuadTreeNode
private int calculateSplitSize(int totalSize, int minBlockSize) {
    int halfSize = totalSize / 2;

    if (totalSize <= minBlockSize) {
        return totalSize;
    }

    return halfSize;
}

// samain region/node dengan warna tertentu
public void fillRegion(int startX, int startY, int regionWidth, int
regionHeight, Color color) {
    int endX = Math.min(startX + regionWidth, root.getWidth());
    int endY = Math.min(startY + regionHeight, root.getHeight());

    for (int y = startY; y < endY; y++) {
        for (int x = startX; x < endX; x++) {
            red[y][x] = color.getRed();
            green[y][x] = color.getGreen();
            blue[y][x] = color.getBlue();
        }
    }
}

```

```

        }
    }

// untuk cek apakah bisa dipecah?
private boolean shouldSplit(QuadTreeNode node) {
    if ((node.getWidth() / 2 * node.getHeight() / 2) < minBlockSize) {
        return false;
    }

    double error = calculateError(node);

    return error > threshold;
}

// menghitung error untuk dibandingkan dengan threshold
private double calculateError(QuadTreeNode node) {
    int x = node.getX();
    int y = node.getY();
    int width = node.getWidth();
    int height = node.getHeight();

    switch (errorMethod) {
        case 1:
            return Method.calculateVariance(this, x, y, width, height);
        case 2:
            return Method.calculateMAD(this, x, y, width, height);
        case 3:
            return Method.calculateMaxPixelDifference(this, x, y, width,
height);
        case 4:
            return Method.calculateEntropy(this, x, y, width, height);
        case 5:
            double ssim = Method.calculateSSIM(this, x, y, width,
height);
            return 1.0 - ssim; //1 - SSIM untuk mendapat error
        default:
            return Method.calculateVariance(this, x, y, width, height);
    }
}

// reconstruct hasil kompresi dengan matrix
public BufferedImage generateCompressedImageAtDepth(int depth) {
    BufferedImage outputImage = new BufferedImage(root.getWidth(),
root.getHeight(), BufferedImage.TYPE_INT_RGB);

    this.red = new int[root.getHeight()][root.getWidth()];
    this.green = new int[root.getHeight()][root.getWidth()];
}

```

```

        this.blue = new int[root.getHeight()][root.getWidth()];

        drawQuadTreeAtDepth(root, 0, depth);

        for (int y = 0; y < root.getHeight(); y++) {
            for (int x = 0; x < root.getWidth(); x++) {
                Color color = new Color(red[y][x], green[y][x], blue[y][x]);
                outputImage.setRGB(x, y, color.getRGB());
            }
        }

        return outputImage;
    }
    // generate final compressed image
    public BufferedImage generateCompressedImage() {
        return generateCompressedImageAtDepth(treeDepth);
    }

    // mengisi region dengan warna average
    private void drawNode(QuadTreeNode node) {
        fillRegion(node.getX(), node.getY(), node.getWidth(),
node.getHeight(), node.getAvgColor());
    }

    // memindahkan warna quadtree dengan kedalaman tertentu pada matrix
    private void drawQuadTreeAtDepth(QuadTreeNode node, int currentDepth,
int maxDepth) {
        if (currentDepth >= maxDepth || node.isLeaf()) {
            drawNode(node);
        } else {
            for (QuadTreeNode child : node.getChildren()) {
                drawQuadTreeAtDepth(child, currentDepth + 1, maxDepth);
            }
        }
    }

    public static void generateCompressionGif(QuadTree quadTree, String
gifPath) throws IOException {
        int treeDepth = quadTree.getTreeDepth();

        BufferedImage[] frames = new BufferedImage[treeDepth + 1];
        for (int depth = 0; depth <= treeDepth; depth++) {
            frames[depth] = quadTree.generateCompressedImageAtDepth(depth);
        }

        createGif(frames, gifPath, 800);
    }
}

```

```

    private static void createGif(BufferedImage[] frames, String outputPath,
int delayMs) throws IOException {
    if (frames == null || frames.length == 0) {
        throw new IllegalArgumentException("No frames provided for GIF
creation");
    }

    File outputFile = new File(outputPath);
    File parentDir = outputFile.getParentFile();
    if (parentDir != null && !parentDir.exists()) {
        parentDir.mkdirs();
    }

    try (ImageOutputStream output = new
FileImageOutputStream(outputFile)) {
        GifSequenceWriter writer = new GifSequenceWriter(output,
frames[0].getType(), delayMs, true);

        for (BufferedImage frame : frames) {
            writer.writeToSequence(frame);
        }

        writer.close();
    }
}

// Menghitung persentase kompresi
public double calculateCompressionPercentage(long originalSize, long
compressedSize) {
    return (1.0 - ((double) compressedSize / originalSize)) * 100.0;
}

public int getTreeDepth() {
    return treeDepth;
}

public int getNodeCount() {
    return nodeCount;
}

public int getWidth() {
    return root.getWidth();
}

public int getHeight() {
    return root.getHeight();
}

```

```
}
```

3.2.4 Method.java

Implementasi dari Method.java

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

public class Method {

    // metode variance
    public static double calculateVariance(QuadTree quadTree, int x, int y,
    int width, int height) {
        java.awt.Color avgColor = quadTree.getAverageColor(x, y, width,
    height);
        double meanR = avgColor.getRed();
        double meanG = avgColor.getGreen();
        double meanB = avgColor.getBlue();

        double varR = 0, varG = 0, varB = 0;
        int count = 0;

        int endX = Math.min(x + width, quadTree.getWidth());
        int endY = Math.min(y + height, quadTree.getHeight());

        for (int cy = y; cy < endY; cy++) {
            for (int cx = x; cx < endX; cx++) {
                java.awt.Color pixelColor = quadTree.getAverageColor(cx, cy,
    1, 1); // Get single pixel color
                varR += Math.pow(pixelColor.getRed() - meanR, 2);
                varG += Math.pow(pixelColor.getGreen() - meanG, 2);
                varB += Math.pow(pixelColor.getBlue() - meanB, 2);
                count++;
            }
        }

        if (count > 0) {
            varR /= count;
            varG /= count;
            varB /= count;
        }

        return (varR + varG + varB) / 3.0;
    }
}
```

```

// metode mean absolute deviation
public static double calculateMAD(QuadTree quadTree, int x, int y, int
width, int height) {
    java.awt.Color avgColor = quadTree.getAverageColor(x, y, width,
height);
    double meanR = avgColor.getRed();
    double meanG = avgColor.getGreen();
    double meanB = avgColor.getBlue();

    double madR = 0, madG = 0, madB = 0;
    int count = 0;

    int endX = Math.min(x + width, quadTree.getWidth());
    int endY = Math.min(y + height, quadTree.getHeight());

    for (int cy = y; cy < endY; cy++) {
        for (int cx = x; cx < endX; cx++) {
            java.awt.Color pixelColor = quadTree.getAverageColor(cx, cy,
1, 1); // Get single pixel color
            madR += Math.abs(pixelColor.getRed() - meanR);
            madG += Math.abs(pixelColor.getGreen() - meanG);
            madB += Math.abs(pixelColor.getBlue() - meanB);
            count++;
        }
    }

    if (count > 0) {
        madR /= count;
        madG /= count;
        madB /= count;
    }

    return (madR + madG + madB) / 3.0;
}

// metode max pixel difference
public static double calculateMaxPixelDifference(QuadTree quadTree, int
x, int y, int width, int height) {

    java.awt.Color avgColor = quadTree.getAverageColor(x, y, width,
height);
    int meanR = avgColor.getRed();
    int meanG = avgColor.getGreen();
    int meanB = avgColor.getBlue();

    int maxDifference = 0;

    int endX = Math.min(x + width, quadTree.getWidth()));

```

```

        int endY = Math.min(y + height, quadTree.getHeight());

        for (int cy = y; cy < endY; cy++) {
            for (int cx = x; cx < endX; cx++) {
                java.awt.Color pixelColor = quadTree.getAverageColor(cx, cy,
1, 1);
                int diffR = Math.abs(pixelColor.getRed() - meanR);
                int diffG = Math.abs(pixelColor.getGreen() - meanG);
                int diffB = Math.abs(pixelColor.getBlue() - meanB);

                int pixelMax = Math.max(diffR, Math.max(diffG, diffB));
                maxDifference = Math.max(maxDifference, pixelMax);
            }
        }

        return maxDifference;
    }

    // metode entropy
    public static double calculateEntropy(QuadTree quadTree, int x, int y,
int width, int height) {
        int[] histogram = new int[256];
        int total = 0;

        int endX = Math.min(x + width, quadTree.getWidth());
        int endY = Math.min(y + height, quadTree.getHeight());

        for (int cy = y; cy < endY; cy++) {
            for (int cx = x; cx < endX; cx++) {
                java.awt.Color color = quadTree.getAverageColor(cx, cy, 1,
1);
                int gray = (color.getRed() + color.getGreen() +
color.getBlue()) / 3;
                histogram[gray]++;
                total++;
            }
        }

        double entropy = 0.0;
        for (int i = 0; i < 256; i++) {
            if (histogram[i] > 0) {
                double p = (double) histogram[i] / total;
                entropy -= p * (Math.log(p) / Math.log(2));
            }
        }
        return entropy;
    }
}

```

```

// metode structural similarity index (SSIM)
public static double calculateSSIM(QuadTree quadTree, int x, int y, int
width, int height) {
    int endX = Math.min(x + width, quadTree.getWidth());
    int endY = Math.min(y + height, quadTree.getHeight());
    int N = (endX - x) * (endY - y);

    if (N == 0) {
        return 1.0; // pixel 0
    }

    double[] origL = new double[N]; // original luminance
    double[] compL = new double[N]; // compressed luminance

    java.awt.Color avg = quadTree.getAverageColor(x, y, width, height);
    double avgLum = 0.299 * avg.getRed() + 0.587 * avg.getGreen() +
0.114 * avg.getBlue();
    // L = 0.299 * R + 0.587 * G + 0.114 * B

    int i = 0;
    for (int cy = y; cy < endY; cy++) {
        for (int cx = x; cx < endX; cx++) {
            java.awt.Color pix = quadTree.getAverageColor(cx, cy, 1, 1);
            double lum = 0.299 * pix.getRed() + 0.587 * pix.getGreen() +
0.114 * pix.getBlue();
            origL[i] = lum;
            compL[i] = avgLum;
            i++;
        }
    }

    return computeSSIM(origL, compL);
}

private static double computeSSIM(double[] x, double[] y) {
    int N = x.length;
    double C1 = 6.5025, C2 = 58.5225;
    // C1 = (K1*L)^2, C2 = (K2*L)^2
    // K1 = 0.01, K2 = 0.03, L = 255

    double meanX = 0, meanY = 0;
    for (int i = 0; i < N; i++) {
        meanX += x[i];
        meanY += y[i];
    }
    meanX /= N;
    meanY /= N;
}

```

```

        double varX = 0, varY = 0, covXY = 0;
        for (int i = 0; i < N; i++) {
            varX += Math.pow(x[i] - meanX, 2);
            varY += Math.pow(y[i] - meanY, 2);
            covXY += (x[i] - meanX) * (y[i] - meanY);
        }
        varX /= N;
        varY /= N;
        covXY /= N;

        double numerator = (2 * meanX * meanY + C1) * (2 * covXY + C2);
        double denominator = (meanX * meanX + meanY * meanY + C1) * (varX +
varY + C2);

        return numerator / denominator;
    }

    public static QuadTree findOptimalParameters(BufferedImage
originalImage, int errorMethod,
        double targetCompressionRate, String inputPath, String
outputPath, String tempDirPath,
        String fileExtension) throws IOException {
    if (targetCompressionRate > 0 && targetCompressionRate <= 1) {
        errorMethod = 1;
    }

    double minThreshold = 0.0;
    double maxThreshold = 255 * 255;

    int minBlockSizeStart = 1;
    int maxBlockSize = 64;

    double optimalThreshold = 0.0;
    int optimalMinBlockSize = minBlockSizeStart;
    double bestDifference = Double.MAX_VALUE;
    double epsilon = 0.001;

    int iterations = 0;

    while ((maxThreshold - minThreshold) > epsilon && iterations < 20) {
        iterations++;
        double midThreshold = (minThreshold + maxThreshold) / 2;

        String debugImgPath =
String.format("%s/temp_image%02d_t%.4f_bs%d%s",
                tempDirPath, iterations, midThreshold,
minBlockSizeStart, fileExtension);

```

```

        double actualCompressionRate =
testCompressionRate(originalImage, errorMethod, midThreshold,
                     4, inputPath, debugImgPath);

        double difference = Math.abs(actualCompressionRate -
targetCompressionRate);

        // System.out.println(" Iteration " + iterations + ":" +
Threshold= " + String.format("%.4f", midThreshold) +
        //           ", BlockSize=" + minBlockSizeStart +
        //           ", Compression=" + String.format("%.2f",
actualCompressionRate * 100) + "%, " +
        //           "Diff=" + String.format("%.4f", difference * 100) +
"% " +
        //           ", Debug image: " + new
File(debugImgPath).getName());

        if (difference < bestDifference) {
            bestDifference = difference;
            optimalThreshold = midThreshold;
        }

        if (difference < 0.01) {
            break;
        }

        if (actualCompressionRate > targetCompressionRate) {
            maxThreshold = midThreshold;
        } else {
            minThreshold = midThreshold;
        }
    }

    // System.out.println("Optimal threshold found: " +
String.format("%.4f", optimalThreshold));

    // Reset best difference for block size search
bestDifference = Double.MAX_VALUE;

    int minBlockSizeMin = minBlockSizeStart;
    int minBlockSizeMax = maxBlockSize;
    iterations = 0;

    while (minBlockSizeMin <= minBlockSizeMax && iterations < 10) {
        iterations++;
        int midBlockSize = (minBlockSizeMin + minBlockSizeMax) / 2;

```

```

        String debugImgPath =
String.format("%s/temp_imageB%02d_t%.4f_bs%d%s",
                tempDirPath, iterations, optimalThreshold, midBlockSize,
fileExtension);

        double actualCompressionRate =
testCompressionRate(originalImage, errorMethod, optimalThreshold,
midBlockSize, inputPath, debugImgPath);

        double difference = Math.abs(actualCompressionRate -
targetCompressionRate);

        // System.out.println(" Iteration " + iterations + ":" +
BlockSize=" + midBlockSize +
        //           ", Threshold=" + String.format("%.4f",
optimalThreshold) +
        //           ", Compression=" + String.format("%.2f",
actualCompressionRate * 100) + "%, " +
        //           "Diff=" + String.format("%.4f", difference * 100) +
"%" +
        //           ", Debug image: " + new
File(debugImgPath).getName());

        if (difference < bestDifference) {
            bestDifference = difference;
            optimalMinBlockSize = midBlockSize;
        }

        if (difference < 0.01) {
            break;
        }

        if (actualCompressionRate > targetCompressionRate) {
            minBlockSizeMax = midBlockSize - 1;
        } else {
            minBlockSizeMin = midBlockSize + 1;
        }
    }

    return new QuadTree(originalImage, optimalMinBlockSize,
optimalThreshold, errorMethod);
}

// Test rate dengan target
private static double testCompressionRate(BufferedImage originalImage,
int errorMethod, double threshold,
int minBlockSize, String inputPath, String debugImgPath) throws
IOException {

```

```

        QuadTree testTree = new QuadTree(originalImage, minBlockSize,
threshold, errorMethod);
        BufferedImage compressedImage = testTree.generateCompressedImage();
        FileProcessor.saveImage(compressedImage, debugImgPath);

        long originalSize = FileProcessor.getFileSize(inputPath);
        long compressedSize = FileProcessor.getFileSize(debugImgPath);
        double compressionRate = 1.0 - ((double) compressedSize /
originalSize);

        // int treeDepth = testTree.getTreeDepth();
        // int nodeCount = testTree.getNodeCount();

        // System.out.println("    Details: Depth=" + treeDepth +
        //                     ", Nodes=" + nodeCount +
        //                     ", Original=" + originalSize + "B" +
        //                     ", Compressed=" + compressedSize + "B");

        return compressionRate;
    }
}

```

3.2.5 FileProcessor.java

Implementasi dari FileProcessor.java

```

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

public class FileProcessor {

    // load iamge
    public static BufferedImage loadImage(String filePath) throws
IOException {
        File file = new File(filePath);
        return ImageIO.read(file);
    }

    // save image
    public static void saveImage(BufferedImage image, String filePath)
throws IOException {
        File outputFile = new File(filePath);
        String extension = filePath.substring(filePath.lastIndexOf('.') +
1);
        ImageIO.write(image, extension, outputFile);
    }
}

```

```

}

// cek file size
public static long getFileSize(String filePath) {
    File file = new File(filePath);
    if (file.exists()) {
        return file.length();
    } else {
        return -1;
    }
}

//hapus directory
public static boolean deleteDirectory(File directory) {
    if (directory.exists()) {
        File[] files = directory.listFiles();
        if (files != null) {
            for (File file : files) {
                if (file.isDirectory()) {
                    deleteDirectory(file);
                } else {
                    file.delete();
                }
            }
        }
    }
    return directory.delete();
}

// dapeting file extension dri input user
public static String getFileExtension(String filePath) {
    String extension = "";
    int i = filePath.lastIndexOf('.');
    if (i > 0) {
        extension = filePath.substring(i);
    }
    return extension.toLowerCase();
}
}

```

3.2.6 GifSequenceWriter.java

Implementasi dari GifSequenceWriter.java

```

import javax.imageio.*;
import javax.imageio.metadata.*;
import javax.imageio.stream.*;

```

```

import java.awt.image.*;
import java.io.*;
import java.util.Iterator;

/**
 * GifSequenceWriter.java
 *
 * Creates a GIF file from a sequence of BufferedImages
 *
 * Original code by Elliot Kroo (elliot[at]kroo[dot]net)
 * Modified for this application
 */
public class GifSequenceWriter {
    protected ImageWriter gifWriter;
    protected ImageWriteParam imageWriteParam;
    protected IIOMetadata imageMetaData;

    /**
     * Creates a new GifSequenceWriter
     *
     * @param outputStream the ImageOutputStream to be written to
     * @param imageType one of the imageTypes specified in BufferedImage
     * @param timeBetweenFramesMS the time between frames in milliseconds
     * @param loopContinuously whether the GIF should loop repeatedly
     * @throws IIException if no GIF ImageWriters are found
     */
    public GifSequenceWriter(
        ImageOutputStream outputStream,
        int imageType,
        int timeBetweenFramesMS,
        boolean loopContinuously) throws IIException, IOException {

        // Get GIF writer
        gifWriter = getWriter();
        imageWriteParam = gifWriter.getDefaultWriteParam();

        // Create metadata
        ImageTypeSpecifier imageTypeSpecifier =
            ImageTypeSpecifier.createFromBufferedImageType(imageType);

        imageMetaData =
        gifWriter.getDefaultImageMetadata(imageTypeSpecifier, imageWriteParam);

        // Configure GIF metadata (loop count, timing)
        configureRootMetadata(timeBetweenFramesMS, loopContinuously);

        // Initialize writer
        gifWriter.setOutput(outputStream);
    }
}

```

```

        gifWriter.prepareWriteSequence(null);
    }

    private ImageWriter getWriter() throws IIoException {
        Iterator<ImageWriter> iter = ImageIO.getImageWritersBySuffix("gif");
        if (!iter.hasNext()) {
            throw new IIoException("No GIF Image Writers Found");
        }
        return iter.next();
    }

    private void configureRootMetadata(int timeBetweenFramesMS, boolean
loopContinuously) throws IIoException {
        String metaFormatName = imageMetaData.getNativeMetadataFormatName();
        IIOMetadataNode root = (IIOMetadataNode)
imageMetaData.getAsTree(metaFormatName);

        // Get Graphics Control Extension node
        IIOMetadataNode graphicsControlExtensionNode = getNode(root,
"GraphicControlExtension");
        graphicsControlExtensionNode.setAttribute("disposalMethod", "none");
        graphicsControlExtensionNode.setAttribute("userInputFlag", "FALSE");
        graphicsControlExtensionNode.setAttribute("transparentColorFlag",
"FALSE");
        graphicsControlExtensionNode.setAttribute("delayTime",
Integer.toString(timeBetweenFramesMS / 10)); // In 1/100 sec
        graphicsControlExtensionNode.setAttribute("transparentColorIndex",
"0");

        // Set loop count if requested
        if (loopContinuously) {
            IIOMetadataNode applicationExtensions = getNode(root,
"ApplicationExtensions");
            IIOMetadataNode applicationExtension = new
IIOMetadataNode("ApplicationExtension");

            applicationExtension.setAttribute("applicationID", "NETSCAPE");
            applicationExtension.setAttribute("authenticationCode", "2.0");

            // Loop continuously (0 means forever)
            byte[] loopBytes = new byte[] { 0x1, 0x0, 0x0 };
            applicationExtension.setUserObject(loopBytes);
            applicationExtensions.appendChild(applicationExtension);
        }

        try {
            imageMetaData.setFromTree(metaFormatName, root);
        } catch (IIOInvalidTreeException e) {

```

```

        throw new II0Exception("Invalid metadata tree", e);
    }

    private static IIOMetadataNode getNode(IIOMetadataNode rootNode, String
nodeName) {
    for (int i = 0; i < rootNode.getLength(); i++) {
        if (rootNode.item(i).getNodeName().equalsIgnoreCase(nodeName)) {
            return (IIOMetadataNode) rootNode.item(i);
        }
    }

    // Node doesn't exist, create and add it
    IIOMetadataNode node = new IIOMetadataNode(nodeName);
    rootNode.appendChild(node);
    return node;
}

/**
 * Writes a frame to the GIF sequence
 * @param img the image to write
 * @throws IOException if errors occur during writing
 */
public void writeToSequence(RenderedImage img) throws IOException {
    gifWriter.writeToSequence(
        new IIIOImage(img, null, imageMetaData),
        imageWriteParam
    );
}

/**
 * Closes the GIF writer
 * @throws IOException if an error occurs during closing
 */
public void close() throws IOException {
    gifWriter.endWriteSequence();
}
}

```

3.3 Penjelasan Implementasi Spesifikasi Bonus yang Dikerjakan

3.3.1 SSIM

Implementasi dari SSIM (pseudocode)

```

function calculateSSIM(quadTree, x, y, width, height):
    endX = min(x + width, quadTree.width)
    endY = min(y + height, quadTree.height)
    N = (endX - x) * (endY - y)

    if N == 0:
        return 1.0

    avgColor = quadTree.getAverageColor(x, y, width, height)
    avgLum = 0.299 * avgColor.R + 0.587 * avgColor.G + 0.114 * avgColor.B

    origL = array of size N
    compl = array of size N

    i = 0
    for cy from y to endY - 1:
        for cx from x to endX - 1:
            pixelColor = quadTree.getAverageColor(cx, cy, 1, 1)
            lum = 0.299 * pixelColor.R + 0.587 * pixelColor.G + 0.114 *
pixelColor.B
                origL[i] = lum
                compl[i] = avgLum
                i = i + 1

    return computeSSIM(origL, compl)

function computeSSIM(origL, compl):
    N = length of origL
    C1 = (0.01 * 255)^2
    C2 = (0.03 * 255)^2

    meanX = average of origL
    meanY = average of compl

    varX = variance of origL
    varY = variance of compl
    covXY = covariance between origL and compl

    numerator = (2 * meanX * meanY + C1) * (2 * covXY + C2)
    denominator = (meanX^2 + meanY^2 + C1) * (varX + varY + C2)

    return numerator / denominator

```

SSIM diimplementasikan dengan pertama mencari ujung sumbu X dan sumbu Y dari gambar yang akan di kompres. Setelah itu, nilai tersebut akan dicek dengan nilai X dan Y nya, bila hasilnya 0, maka gambar tersebut merupakan sebuah gambar dengan 0 pixel (gambar nya

tidak ada) dan nilai SSIM nya 1 atau sempurna. Setelah itu, akan terjadi pengambilan warna rata-rata dari block hasil Quadtree yang lalu akan dikonversi ke luminance. Lalu, untuk setiap pixel, akan diambil luminance nya dan akan dibandingkan dengan rata-rata luminance dari blok yang sudah di kompres. Lalu dengan menggunakan rumus, akan dihitung nilai SSIM nya.

3.3.2 Pembuatan Gif

Implementasi pembuatan Gif

```
CLASS GifSequenceWriter
VARIABLES:
    gifWriter          // ImageWriter for writing GIFs
    imageWriteParam   // Parameters for writing
    imageMetaData      // Metadata for each frame

    FUNCTION Constructor(outputStream, imageType, timeBetweenFramesMS,
loopContinuously)
        gifWriter ← getWriter() // Get available GIF writer
        imageWriteParam ← gifWriter.getDefaultWriteParam()
        imageTypeSpecifier ← createFromBufferedImageType(imageType)
        imageMetaData ←
        gifWriter.getDefaultImageMetadata(imageTypeSpecifier, imageWriteParam)

        // Configure metadata
        configureRootMetadata(timeBetweenFramesMS, loopContinuously)

        gifWriter.setOutput(outputStream)
        gifWriter.prepareWriteSequence(null)
    END FUNCTION

    FUNCTION getWriter()
        iter ← ImageIO.getWritersBySuffix("gif")
        IF iter has no next THEN
            THROW "No GIF Writer Found"
        RETURN iter.next()
    END FUNCTION

    FUNCTION configureRootMetadata(timeBetweenFramesMS, loopContinuously)
        root ← imageMetaData as tree
        graphicsControl ← getNode(root, "GraphicControlExtension")
        graphicsControl.set("disposalMethod", "none")
        graphicsControl.set("userInputFlag", "FALSE")
        graphicsControl.set("transparentColorFlag", "FALSE")
        graphicsControl.set("delayTime", timeBetweenFramesMS / 10)
        graphicsControl.set("transparentColorIndex", "0")

        IF loopContinuously IS TRUE THEN
            appExtensions ← getNode(root, "ApplicationExtensions")
```

```

        appExtension ← createNode("ApplicationExtension")
        appExtension.set("applicationID", "NETSCAPE")
        appExtension.set("authenticationCode", "2.0")
        appExtension.setUserObject([0x1, 0x0, 0x0]) // Loop forever
        appExtensions.appendChild(appExtension)
    END IF

    imageMetaData.setFromTree(root)
END FUNCTION

FUNCTION getNode(root, name)
    FOR each child IN root
        IF child.nodeName == name THEN
            RETURN child
        END FOR
    newNode ← createNode(name)
    root.appendChild(newNode)
    RETURN newNode
END FUNCTION

FUNCTION writeToSequence(image)
    gifWriter.writeToSequence(image with imageMetaData, imageWriteParam)
END FUNCTION

FUNCTION close()
    gifWriter.endWriteSequence()
END FUNCTION
END CLASS

```

Pembuatan Gif diimplementasikan dengan membuat metadata untuk image frames dan mengubah metadata tersebut agar membuat delay antar frame serta membuat animasi untuk mengeloa. Image yang diambil berasal dari node-node yang berada di Quadtree dan menjadi basis untuk Gif yang dibuat. Gif yang dihasilkan nantinya akan mulai dari yang image pada node bawah, hingga ke image pada node atas, yaitu hasil kompresi gambar

3.3.3 Target Persentase Kompresi

Implementasi Target Persentase Kompresi

```

function findOptimalParameters(originalImage, errorMethod,
targetCompressionRate, inputPath, outputPath, tempDir, ext):
    if targetCompressionRate in (0, 1]:
        errorMethod = 1 // Override if target compression rate is valid

    minThreshold = 0.0
    maxThreshold = 255 * 255
    optimalThreshold = 0.0

```

```

optimalBlockSize = 1
bestDiff = MAX_VALUE
epsilon = 0.001

// Step 1: Find optimal threshold (binary search)
repeat until (maxThreshold - minThreshold < epsilon) or max 20
iterations:
    midThreshold = (minThreshold + maxThreshold) / 2
    testPath = generateTempPath(tempDir, midThreshold)

    actualRate = testCompressionRate(originalImage, errorMethod,
midThreshold, blockSize=4, inputPath, testPath)
    diff = abs(actualRate - targetCompressionRate)

    if diff < bestDiff:
        bestDiff = diff
        optimalThreshold = midThreshold

    if diff < 0.01: break

    if actualRate > targetCompressionRate:
        maxThreshold = midThreshold
    else:
        minThreshold = midThreshold

// Step 2: Find optimal block size (binary search)
minBlockSize = 1
maxBlockSize = 64
bestDiff = MAX_VALUE

repeat until (minBlockSize > maxBlockSize) or max 10 iterations:
    midBlockSize = (minBlockSize + maxBlockSize) / 2
    testPath = generateTempPath(tempDir, optimalThreshold, midBlockSize)

    actualRate = testCompressionRate(originalImage, errorMethod,
optimalThreshold, midBlockSize, inputPath, testPath)
    diff = abs(actualRate - targetCompressionRate)

    if diff < bestDiff:
        bestDiff = diff
        optimalBlockSize = midBlockSize

    if diff < 0.01: break

    if actualRate > targetCompressionRate:
        maxBlockSize = midBlockSize - 1
    else:
        minBlockSize = midBlockSize + 1

```

```

    return new QuadTree(originalImage, optimalBlockSize, optimalThreshold,
errorMethod)

function testCompressionRate(image, method, threshold, minBlockSize,
inputPath, outputPath):
    tree = new QuadTree(image, minBlockSize, threshold, method)
    compressed = tree.generateCompressedImage()
    saveImage(compressed, outputPath)

    originalSize = getFileSize(inputPath)
    compressedSize = getFileSize(outputPath)

    return 1.0 - (compressedSize / originalSize)

```

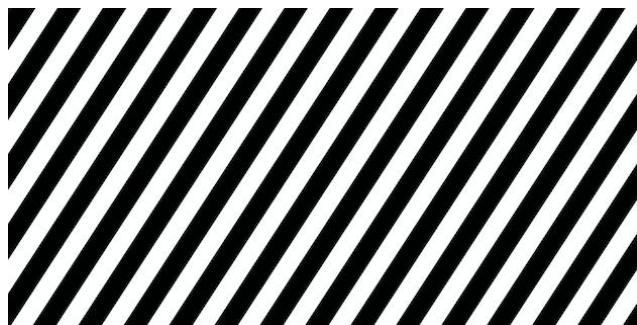
Target persentase kompresi ini diimplementasikan dengan pertama membuat sebuah program untuk membuat threshold secara otomatis, sesuai dengan target persentase kompresi yang diinginkan. Hal ini terdapat pada method.findOptimalParameters yang berfungsi untuk mencari parameter-parameter yang sesuai untuk mendapatkan persentase kompresi yang sesuai. Untuk threshold yang optimal, dilakukan binary search dengan block yang tetap terlebih dahulu. Binary search ini dilakukan dengan terus membuat Quadtree dan memproses gambar hingga akhirnya mendapat persentase kompresi yang sesuai. Data ini disimpan dan dibandingkan hingga mendapat yang optimal. Setelah itu, hal yang sama dilakukan, namun kali ini untuk mencari ukuran block yang optimal, sehingga yang tetap adalah thresholdnya, yang sudah optimal. Dengan begitu, pada akhirnya hasil kompresi yang keluar adalah sesuai dengan target persentase yang diinginkan. Hal ini dilakukan pada function testCompressionRate yang akan mengetes apakah hasil kompresinya sudah sesuai atau belum.

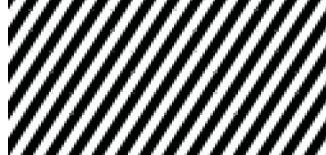
BAB IV

PENGUJIAN

Pengujian dilakukan terhadap 3 gambar dan setiap gambar diuji dengan 5 metode perhitungan variansi berbeda ditambah 1 metode target persentase. Kemudian setiap metode dilakukan sebanyak 3 variasi sehingga total percobaan adalah $3 \times 6 \times 3$ atau = 54 percobaan.

4.1 Gambar 1 - pattern.jpg



Metode Variance			
Percentage Method Threshold blocksize	0 1 10 5	0 1 20 5	0 1 10 10
statistik	Results: Waktu eksekusi: 651 ms Ukuran asli image: 150271 bytes Ukuran setelah dikompresi: 50058 bytes Percentase kompresi: 66.69% Kedalaman tree (depth): 7 jumlah nodes: 15717	Results: Waktu eksekusi: 707 ms Ukuran asli image: 150271 bytes Ukuran setelah dikompresi: 50066 bytes Percentase kompresi: 66.68% Kedalaman tree (depth): 7 jumlah nodes: 15429	Results: Waktu eksekusi: 767 ms Ukuran asli image: 150271 bytes Ukuran setelah dikompresi: 50884 bytes Percentase kompresi: 66.14% Kedalaman tree (depth): 7 jumlah nodes: 15497
output			
gif			

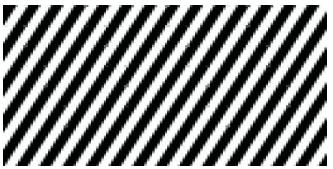
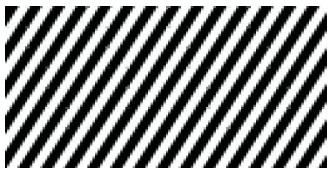
Metode MAD			
Percentage Method Threshold blocksize	0 2 25 10	0 2 50 10	0 2 25 20
statistik	<p>Results:</p> <p>Waktu eksekusi: 776 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 52466 bytes</p> <p>Persentase kompresi: 65.09%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 10721</p>	<p>Results:</p> <p>Waktu eksekusi: 700 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53713 bytes</p> <p>Persentase kompresi: 64.26%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 8405</p>	<p>Results:</p> <p>Waktu eksekusi: 690 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53193 bytes</p> <p>Persentase kompresi: 64.60%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 4761</p>
output			
gif			

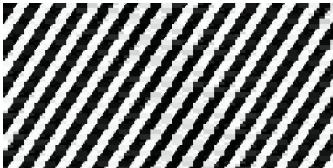
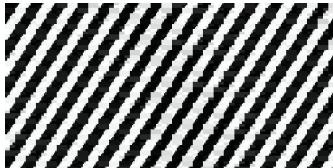
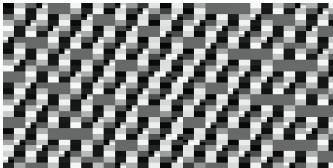
Metode Max Pixel Difference			
Percentage Method Threshold blocksize	0 3 50 25	0 3 100 25	0 3 50 50
statistik	<p>Results:</p> <p>Waktu eksekusi: 753 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53514 bytes</p> <p>Persentase kompresi: 64.39%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 5457</p>	<p>Results:</p> <p>Waktu eksekusi: 755 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53514 bytes</p> <p>Persentase kompresi: 64.39%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 5457</p>	<p>Results:</p> <p>Waktu eksekusi: 662 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 49851 bytes</p> <p>Persentase kompresi: 66.83%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 3165</p>
output			

gif			
-----	--	--	--

Metode Entropy			
Percentage Method Threshold blocksize	0 4 2 10	0 4 2.5 10	0 4 2 20
statistik	<p>Results:</p> <p>Waktu eksekusi: 855 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53468 bytes</p> <p>Persentase kompresi: 64.42%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 9149</p>	<p>Results:</p> <p>Waktu eksekusi: 833 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 51115 bytes</p> <p>Persentase kompresi: 65.98%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 4529</p>	<p>Results:</p> <p>Waktu eksekusi: 692 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 52312 bytes</p> <p>Persentase kompresi: 65.19%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 4169</p>
output			
gif			

Metode SSIM			
Percentage Method Threshold blocksize	0 5 0.4 10	0 5 0.6 10	0 5 0.4 30
statistik	<p>Results:</p> <p>Waktu eksekusi: 955 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 50904 bytes</p> <p>Persentase kompresi: 66.13%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 14565</p>	<p>Results:</p> <p>Waktu eksekusi: 906 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 50941 bytes</p> <p>Persentase kompresi: 66.10%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 13957</p>	<p>Results:</p> <p>Waktu eksekusi: 751 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 53514 bytes</p> <p>Persentase kompresi: 64.39%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 5457</p>

output			
gif			

Target Compression Percentage (Bonus)			
Percentage Method Threshold blocksize	0.4	0.5	0.7
statistik	<p>Results:</p> <p>Waktu eksekusi: 3902 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 56587 bytes</p> <p>Persentase kompresi: 62.34%</p> <p>Kedalaman tree (depth): 9</p> <p>Jumlah nodes: 11897</p>	<p>Results:</p> <p>Waktu eksekusi: 3709 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 56587 bytes</p> <p>Persentase kompresi: 62.34%</p> <p>Kedalaman tree (depth): 9</p> <p>Jumlah nodes: 11897</p>	<p>Results:</p> <p>Waktu eksekusi: 2070 ms</p> <p>Ukuran asli image: 150271 bytes</p> <p>Ukuran setelah dikompresi: 44601 bytes</p> <p>Persentase kompresi: 70.32%</p> <p>Kedalaman tree (depth): 6</p> <p>Jumlah nodes: 1609</p>
output			
gif			

4.2 Gambar 2 - ranger.png



Metode Variance			
Percentage Method Threshold blocksize	0 1 20 10	0 1 70 10	0 1 20 20
statistik	Results: Waktu eksekusi: 985 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 69894 bytes Persentase kompresi: 7,73% Kedalaman tree (depth): 8 jumlah nodes: 15929	Results: Waktu eksekusi: 987 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 64527 bytes Persentase kompresi: 14,81% Kedalaman tree (depth): 8 jumlah nodes: 13441	Results: Waktu eksekusi: 961 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 68393 bytes Persentase kompresi: 9,71% Kedalaman tree (depth): 7 jumlah nodes: 15461
output			
gif			

Metode MAD			
Percentage Method Threshold	0	0	0

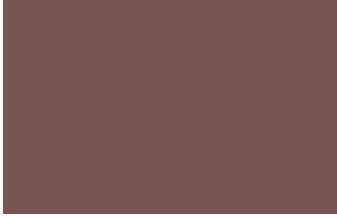
blocksize	2 25 10	2 50 10	2 25 20
statistik	<p>Results:</p> <p>Waktu eksekusi: 909 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 50889 bytes Persentase kompresi: 32,82% Kedalaman tree (depth): 8 jumlah nodes: 7245</p>	<p>Results:</p> <p>Waktu eksekusi: 979 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 32752 bytes Persentase kompresi: 56,76% Kedalaman tree (depth): 8 jumlah nodes: 2409</p>	<p>Results:</p> <p>Waktu eksekusi: 894 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 50270 bytes Persentase kompresi: 33,63% Kedalaman tree (depth): 7 jumlah nodes: 7089</p>
output			
gif			

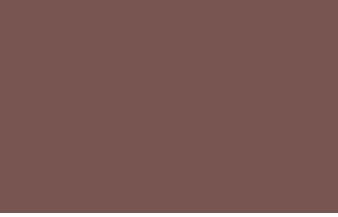
Metode Max Pixel Difference			
Percentage Method Threshold blocksize	0 3 50 25	0 3 100 25	0 3 50 10
statistik	<p>Results:</p> <p>Waktu eksekusi: 917 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 60525 bytes Persentase kompresi: 20,10% Kedalaman tree (depth): 7 jumlah nodes: 11605</p>	<p>Results:</p> <p>Waktu eksekusi: 890 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 56334 bytes Persentase kompresi: 25,63% Kedalaman tree (depth): 7 jumlah nodes: 9377</p>	<p>Results:</p> <p>Waktu eksekusi: 994 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 61214 bytes Persentase kompresi: 19,19% Kedalaman tree (depth): 8 jumlah nodes: 11869</p>
output			

gif			
-----	--	--	--

Metode Entropy			
Percentage Method Threshold blocksize	0 4 2 10	0 4 5 10	0 4 2 20
statistik	Results: Waktu eksekusi: 1050 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 72562 bytes Persentase kompresi: 4,20% Kedalaman tree (depth): 8 jumlah nodes: 19273	Results: Waktu eksekusi: 937 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 52865 bytes Persentase kompresi: 30,21% Kedalaman tree (depth): 8 jumlah nodes: 7289	Results: Waktu eksekusi: 933 ms Ukuran asli image: 75747 bytes Ukuran setelah dikompresi: 69972 bytes Persentase kompresi: 7,62% Kedalaman tree (depth): 7 jumlah nodes: 18545
output			
gif			

Metode SSIM			
Percentage Method Threshold blocksize	0 5 0.4 10	0 5 0.6 10	0 5 0.4 30

statistik	<p>Results:</p> <p>waktu eksekusi: 1010 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 65305 bytes Persentase kompresi: 13,79% Kedalaman tree (depth): 8 jumlah nodes: 13877</p>	<p>Results:</p> <p>waktu eksekusi: 941 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 62855 bytes Persentase kompresi: 17,02% Kedalaman tree (depth): 8 jumlah nodes: 12309</p>	<p>Results:</p> <p>waktu eksekusi: 898 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 64214 bytes Persentase kompresi: 15,23% Kedalaman tree (depth): 7 jumlah nodes: 13493</p>
output			
gif			

Target Compression Percentage (Bonus)			
Percentage Method Threshold blocksize	0.4	0.6	0.8
statistik	<p>Results:</p> <p>waktu eksekusi: 2656 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 45121 bytes Persentase kompresi: 40,43% Kedalaman tree (depth): 8 jumlah nodes: 5309</p>	<p>Results:</p> <p>waktu eksekusi: 1783 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 30709 bytes Persentase kompresi: 59,46% Kedalaman tree (depth): 7 jumlah nodes: 877</p>	<p>Results:</p> <p>waktu eksekusi: 1756 ms ukuran asli image: 75747 bytes ukuran setelah dikompresi: 14681 bytes Persentase kompresi: 80,62% Kedalaman tree (depth): 7 jumlah nodes: 217</p>
output			
gif			

4.3 Gambar 3 - ferrari.jpg



Metode Variance			
Percentage Method Threshold blocksize	0 1 20 10	0 1 50 10	0 1 20 30
statistik	Results: Waktu eksekusi: 13835 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 146475 bytes Persentase kompresi: 54.22% Kedalaman tree (depth): 9 jumlah nodes: 32457	Results: Waktu eksekusi: 13972 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 142399 bytes Persentase kompresi: 55.50% Kedalaman tree (depth): 9 jumlah nodes: 28001	Results: Waktu eksekusi: 13949 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 132928 bytes Persentase kompresi: 58.46% Kedalaman tree (depth): 8 jumlah nodes: 10433
output			
gif			

Metode MAD			
Percentage Method Threshold blocksize	0 2 5 5	0 2 10 5	0 2 5 100
statistik	<pre>Results: Waktu eksekusi: 12337 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 141710 bytes Persentase kompresi: 55.71% Kedalaman tree (depth): 9 jumlah nodes: 27497</pre>	<pre>Results: Waktu eksekusi: 12528 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 120726 bytes Persentase kompresi: 62.27% Kedalaman tree (depth): 9 jumlah nodes: 15073</pre>	<pre>Results: Waktu eksekusi: 9332 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 108481 bytes Persentase kompresi: 66.10% Kedalaman tree (depth): 7 jumlah nodes: 2885</pre>
output			
gif			

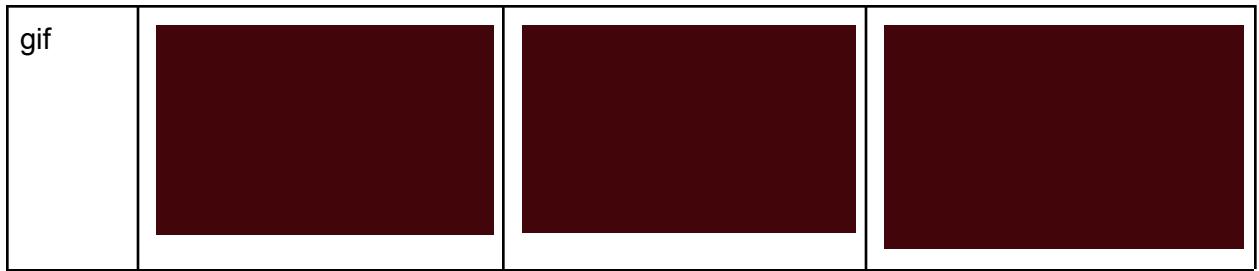
Metode Max Pixel Difference			
Percentage Method Threshold blocksize	0 3 20 20	0 3 100 20	0 3 20 100
statistik	<pre>Results: Waktu eksekusi: 11353 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 132739 bytes Persentase kompresi: 58.51% Kedalaman tree (depth): 8 jumlah nodes: 10305</pre>	<pre>Results: Waktu eksekusi: 10894 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 119392 bytes Persentase kompresi: 62.69% Kedalaman tree (depth): 8 jumlah nodes: 6025</pre>	<pre>Results: Waktu eksekusi: 11960 ms Ukuran asli image: 319968 bytes Ukuran setelah dikompresi: 112145 bytes Persentase kompresi: 64.95% Kedalaman tree (depth): 7 jumlah nodes: 3389</pre>

output			
gif			

Metode Entropy			
Percentage Method Threshold blocksize	0 4 1 10	0 4 3 10	0 4 1 70
statistik	<p>Results:</p> <p>Waktu eksekusi: 13654 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 152961 bytes</p> <p>Persentase kompresi: 52.19%</p> <p>Kedalaman tree (depth): 9</p> <p>Jumlah nodes: 163141</p>	<p>Results:</p> <p>Waktu eksekusi: 13381 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 147797 bytes</p> <p>Persentase kompresi: 53.81%</p> <p>Kedalaman tree (depth): 9</p> <p>Jumlah nodes: 36733</p>	<p>Results:</p> <p>Waktu eksekusi: 11080 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 117039 bytes</p> <p>Persentase kompresi: 63.42%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 16053</p>
output			
gif			

Metode SSIM			
Percentage Method Threshold blocksize	0 5 0.3 20	0 5 0.7 20	0 5 0.3 100
statistik	<p>Results:</p> <p>Waktu eksekusi: 10892 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 130073 bytes</p> <p>Persentase kompresi: 59.35%</p> <p>Kedalaman tree (depth): 8</p> <p>Jumlah nodes: 9469</p>	<p>Results:</p> <p>Waktu eksekusi: 12165 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 123754 bytes</p> <p>Persentase kompresi: 61.32%</p> <p>Kedalaman tree (depth): 8</p> <p>Jumlah nodes: 7217</p>	<p>Results:</p> <p>Waktu eksekusi: 10682 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 109647 bytes</p> <p>Persentase kompresi: 65.73%</p> <p>Kedalaman tree (depth): 7</p> <p>Jumlah nodes: 3061</p>
output			
gif			

Target Compression Percentage (Bonus)			
Percentage Method Threshold blocksize	0.35	0.55	0.75
statistik	<p>Results:</p> <p>Waktu eksekusi: 84858 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 183491 bytes</p> <p>Persentase kompresi: 42.65%</p> <p>Kedalaman tree (depth): 11</p> <p>Jumlah nodes: 2112349</p>	<p>Results:</p> <p>Waktu eksekusi: 33257 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 140987 bytes</p> <p>Persentase kompresi: 55.94%</p> <p>Kedalaman tree (depth): 10</p> <p>Jumlah nodes: 27573</p>	<p>Results:</p> <p>Waktu eksekusi: 23021 ms</p> <p>Ukuran asli image: 319968 bytes</p> <p>Ukuran setelah dikompresi: 77914 bytes</p> <p>Persentase kompresi: 75.65%</p> <p>Kedalaman tree (depth): 9</p> <p>Jumlah nodes: 1589</p>
output			



BAB V

PENUTUP

5.1 Analisis Percobaan Algoritma Divide and Conquer dalam Kompresi Gambar dengan Metode Quadtree

Salah satu implementasi algoritma divide and conquer adalah untuk membentuk quadtree yang digunakan untuk mengompresi gambar. Algoritma ini akan membagi gambar menjadi empat kuadran secara rekursif hingga menyentuh batas tertentu yang telah ditentukan.

Jika hanya berfokus pada pembentukan QuadTree dengan algoritma divide and conquer tanpa memerhatikan fungsi lain seperti penentuan avgColor dan sebagainya. Kondisi terbaik untuk algoritma terjadi saat variansi dari gambar asli berada di bawah threshold yang ditentukan atau ukuran block berada di bawah ukuran minimum yang telah ditentukan. Dengan kondisi itu maka algoritma akan memiliki Big-O notation $O(1)$. Kondisi terburuk jika gambar harus terus dipecah hingga pixel terkecil (mengunjungi semua pixel) sehingga algoritma akan memiliki Big-O notation $O(n^2)$. Namun karena ada threshold dan minblocksize yang ditentukan, maka pembagian tidak selalu terjadi hingga mengunjungi setiap pixel dan algoritma divide and conquer untuk membentuk quadTree ini cenderung memiliki Big-O notation $(N \log N)$.

5.2 Kesimpulan

Kesimpulan yang bisa didapatkan dari percobaan ini adalah bahwa metode Quadtree dapat melakukan kompresi gambar dengan berbagai parameter, dan untuk algoritma Divide and Conquernya sendiri, berhasil melakukan kompresi gambar dengan pertama membaginya gambar tersebut menjadi beberapa bagian yang lebih kecil untuk disesuaikan dengan threshold dan ukuran block minimumnya agar dapat dilakukan kompresi. Setelah itu, bagian-bagian tersebut akan disatukan kembali menjadi gambar yang baru yang telah dikompresi. Hasil dari kompresi ini juga lumayan baik, yaitu bisa $O(1)$, $O(n^2)$ atau $O(N \log N)$, tergantung gambar dan parameter yang diberikan. Selain itu, untuk mengukur error dari metode ini, juga dibuat beberapa errorMethod untuk mengecek seberapa besar error atau perubahan yang dibuat dari gambar aslinya.

Lampiran

Link Repository : https://github.com/Incheon21/Tucil2_13523033_13523022

Tabel Kelengkapan Spesifikasi

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	<input checked="" type="checkbox"/>	
2	Program berhasil dijalankan	<input checked="" type="checkbox"/>	
3	Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	<input checked="" type="checkbox"/>	
4	Mengimplementasi seluruh metode perhitungan error wajib.	<input checked="" type="checkbox"/>	
5	[Bonus] Implementasi persentase kompresi sebagai parameter tambahan.	<input checked="" type="checkbox"/>	
6	[Bonus] Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error.	<input checked="" type="checkbox"/>	
7	[Bonus] Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar.	<input checked="" type="checkbox"/>	
8	Program dan laporan dibuat (kelompok) sendiri.	<input checked="" type="checkbox"/>	