Laporan Tugas Kecil 2

IF2211 Strategi Algoritma

Kompresi Gambar Dengan Metode Quadtree

Disusun oleh:

Ziyan Agil Nur Ramadhan

13622076

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

BAB I
DESKRIPSI TUGAS

Tugas Kecil 2 IF2211 Strategi Algoritma bertujuan untuk mengimplementasikan algoritma *Divide and Conquer* pada kompresi gambar dengan metode Quadtree. 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.

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.

Ide pada tugas kecil 2 ini cukup sederhana, seperti pada pembahasan sebelumnya mengenai Quadtree. Berikut adalah prosedur pada program kompresi gambar yang akan dibuat dalam Tugas Kecil 2 (Divide and Conquer):

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 1.
   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 1.

3. Pembagian Blok

   Bandingkan nilai variansi blok dengan threshold:

   - Jika variansi di atas threshold (cek kasus khusus untuk metode bonus), 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.
   - 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.

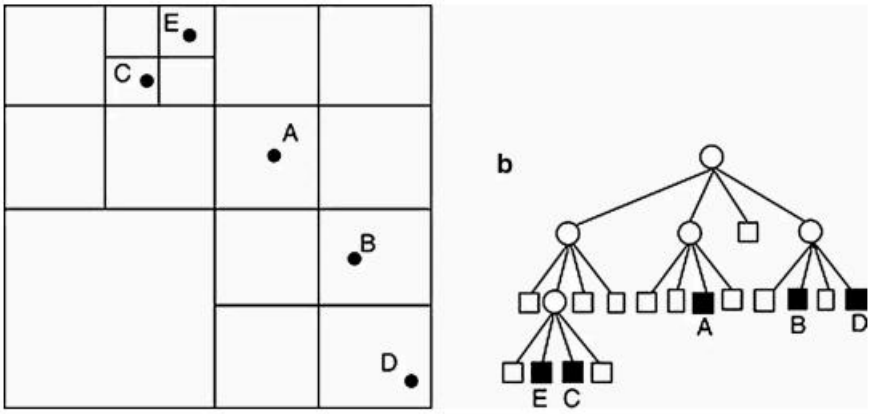
5. Rekursi dan Penghentian

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

   - Error blok berada di bawah threshold.
   - 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.



Gambar 3. Struktur Data Quadtree dalam Kompresi Gambar
(Sumber: https://medium.com/@tannerwyork/quadtrees-for-image-processing-302536c95c00
)

Parameter:

1. Error Measurement Methods (Metode Pengukuran Error)

   Metode yang digunakan untuk menentukan seberapa besar perbedaan dalam satu blok gambar. Jika error dalam blok melebihi ambas batas (threshold), maka blok akan dibagi menjadi empat bagian yang lebih kecil.

Tabel 1. Metode Pengukuran Error

Metode	Formula
Variance	$\sigma_c^2 = \frac{1}{N}\sum_{i=1}^{N}(P_{i,c} - \mu_c)^2$
	$\sigma_{RGB}^2 = \frac{\sigma_R^2 + \sigma_G^2 + \sigma_B^2}{3}$
	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

Mean Absolute Deviation (MAD)	$$MAD_c = \frac{1}{N}\sum_{i=1}^{N} \lvert P_{i,c} - \mu_c \rvert$$
	$$MAD_{RGB} = \frac{MAD_R + MAD_G + MAD_B}{3}$$
	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
Max Pixel Difference	$$D_c = max(P_{i,c}) - min(P_{i,c})$$
	$$D_{RGB} = \frac{D_R + D_G + D_B}{3}$$
	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*
Entropy	$$H_c = -\sum_{i=1}^{N} P_c(i) log_2(P_c(i))$$
	$$H_{RGB} = \frac{H_R + H_G + H_B}{3}$$
	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)
[Bonus] Structural Similarity Index (SSIM) (Referensi tambahan)	$$SSIM_c(x,y) = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(2\sigma_{xy,c} + C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + \sigma_{y,c}^2 + C_2)}$$
	$$SSIM_{RGB} = w_R \cdot SSIM_R + w_G \cdot SSIM_G + w_B \cdot SSIM_B$$
	Nilai SSIM yang dibandingkan adalah antara blok gambar sebelum dan sesudah dikompresi. Silakan lakukan eksplorasi untuk memahami serta memperoleh nilai konstanta pada formula SSIM, asumsikan gambar yang akan diuji adalah 24-bit RGB dengan 8-bit per kanal.

2. Threshold (Ambang Batas)

   Threshold adalah nilai batas yang menentukan apakah sebuah blok dianggap cukup seragam untuk disimpan atau harus dibagi lebih lanjut.

3. Minimum Block Size (Ukuran Blok Minimum)

   Minimum block size (luas piksel) adalah ukuran terkecil dari sebuah blok yang diizinkan dalam proses kompresi. Jika ukuran blok yang akan dibagi menjadi empat sub-blok berada di bawah ukuran minimum yang telah dikonfigurasi, maka blok tersebut tidak akan dibagi lebih lanjut, meskipun error masih di atas threshold.

4. Compression Percentage (Persentase Kompresi) [BONUS]

   Presentasi kompresi menunjukkan seberapa besar ukuran gambar berkurang dibandingkan dengan dengan ukuran aslinya setelah dikompresi menggunakan metode quadtree.

$$Persentase\ Kompresi = \left(1 - \frac{Ukuran\ Gambar\ Terkompresi}{Ukuran\ Gambar\ Asli}\right) \times 100\%$$

Spesifikasi Tugas Kecil 2:

- Buatlah 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.
- Alur program:

  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 II
ALGORITMA DIVIDE AND CONQUER

Kompresi gambar dengan metode Quadtree memanfaatkan algoritma divide and conquer. Algoritma ini diimplementasikan pada dua kelas yaitu kelas Node dan kelas Quadtree. Berikut merupakan penjelasan tiap kelas dan langkah-langkah divide and conquer yang terdapat pada kedua kelas tersebut.

1. Kelas Node

   Kelas ini merepresentasikan blok atau wilayah tertentu pada gambar. Setiap instance kelas Node mewakili sub-blok dari gambar serta menyimpan informasi koordinat (x, y), dimensi (width, height), serta warna rata-rata di wilayah tersebut. Langkah-langkah serta metode yang digunakan pada kelas Node adalah sebagai berikut.

   a. split()

      Metode ini membagi sebuah node yang tidak homogen menjadi empat sub-node. Langkah-langkah yang dilakukan yaitu mengubah status node saat ini menjadi bukan daun karena node telah dibagi. Kemudian, menghitung dimensi baru untuk setiap sub-blok dengan membagi lebar dan tinggi node asal dengan 2. Terakhir, akan terbentuk empat sub-node yang masing-masing memiliki koordinat yang disesuaikan.

   b. calculateAverageColor(image)

      Metode ini menghitung rata-rata warna untuk area yang diwakili oleh node. Langkah-langkahnya yaitu melakukan iterasi pada setiap piksel di dalam area node kemudian menjumlahkan nilai untuk masing-masing kanal RGB. Terakhir, menghitung nilai rata-rata tiap kanal dengan membagi total dengan jumlah piksel serta menyimpan hasil warna rata-rata sebagai representasi warna gambar akhir pada area tersebut.

   c. applyColorToImage(image)

      Metode ini akan menggunakan nilai rata-rata warna dari setiap node daun yang nantinya akan digunakan untuk mewarnai area yang mewakili blok tersebut pada output. Langkah-langkah yang dilakukan yaitu dengan mewarnai node menggunakan nilai warna rata-rata jika sebuah node merupakan daun. Metode ini akan dipanggil secara rekursif jika node bukan merupakan daun.

2. Kelas Quadtree

   Kelas ini akan mengatur keseluruhan proses pembagian dan penggabungan yang diterapkan pada gambar. Kelas ini menginisialisasi node akar yang mewakili seluruh gambar dan memulai proses rekursif untuk membangun Quadtree dengan beberapa kriteria error. Langkah-langkah serta metode yang digunakan pada kelas Node adalah sebagai berikut.

   a. compress()

      Metode ini akan memproses kompresi dengan membangun Quadtree dan mengaplikasikan warna rata-rata ke gambar output. Langkah-langkahnya dengan memanggil metode rekursif buildQuadTree(root, 0) untuk memproses node akar kemudia setelah pohon terbentuk akan memanggil root.applyColorToImage(compressedImage) untuk merekonstruksi gambar hasil kompresi dari node daun.

   b. buildQuadTree(Node node, int depth)

      Metode ini merupakan inti dari pendekatan divide and conquer karena melakukan pembagian secara rekursif pada node. Langkah-langkahnya yaitu dengan menghitung warna rata-rata node dengan node.calculateAverageColor(originalImage) serta menghitung eror di area node dengan metode yang diimplementasikan pada interface ErrorCalculator (seperti: Variance, MAD, Max Pixel Difference, Entropy, dan Structural Similarity Index). Jika eror melebihi threshold dan ukuran node cukup besar, akan dilakukan pemanggilan node.split() untuk membagi node menjadi empat sub-node. Node yang dibagi akan dilakukan pemanggilan secara rekursif dengan buildQuadTree(child, depth + 1). Jika node tidak memenuhi syarat pembagian, maka node akan dianggap sebagai daun dan tidak akan dibagi lagi

BAB III

KODE PROGRAM

Kode Program (Java) yang Mengimplementasikan Algoritma Divide and Conquer

```java
// class Node
public class Node {
    private int x, y, width, height;
    private Node[] children;
    private Color averageColor;
    private boolean isLeaf;

    public Node(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;
    }

    public void split() {
        this.isLeaf = false;
        this.children = new Node[4];

        int newWidth = width / 2;
        int newHeight = height / 2;

        // Atas-kiri
        children[0] = new Node(x, y, newWidth, newHeight);
        // Atas-kanan
        children[1] = new Node(x + newWidth, y, newWidth, newHeight);
        // Bawah-kiri
        children[2] = new Node(x, y + newHeight, newWidth, newHeight);
        // Bawah-kanan
        children[3] = new Node(x + newWidth, y + newHeight, newWidth, newHeight);
    }

    public void calculateAverageColor(BufferedImage image) {
        int totalRed = 0;
        int totalGreen = 0;
        int totalBlue = 0;
        int pixelCount = 0;
```

```
        for (int j = y; j < y + height; j++) {
            for (int i = x; i < x + width; i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));
                    totalRed += pixelColor.getRed();
                    totalGreen += pixelColor.getGreen();
                    totalBlue += pixelColor.getBlue();
                    pixelCount++;
                }
            }
        }

        if (pixelCount > 0) {
            int avgRed = totalRed / pixelCount;
            int avgGreen = totalGreen / pixelCount;
            int avgBlue = totalBlue / pixelCount;
            this.averageColor = new Color(avgRed, avgGreen, avgBlue);
        } else {
            this.averageColor = Color.BLACK;
        }
    }

    public void applyColorToImage(BufferedImage image) {
        if (isLeaf) {
            for (int j = y; j < y + height; j++) {
                for (int i = x; i < x + width; i++) {
                    if (i < image.getWidth() && j < image.getHeight()) {
                        image.setRGB(i, j, averageColor.getRGB());
                    }
                }
            }
        } else {
            for (Node child : children) {
                child.applyColorToImage(image);
            }
        }
    }

    // Getters and setters
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
```

```
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

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

    public Color getAverageColor() {
        return averageColor;
    }

    public boolean isLeaf() {
        return isLeaf;
    }

    public void setAverageColor(Color averageColor) {
        this.averageColor = averageColor;
    }
}

// class QuadTree
public class QuadTree {
    private Node root;
    private BufferedImage originalImage;
    private BufferedImage compressedImage;
    private ErrorCalculator errorCalculator;
    private int minBlockSize;
    private double threshold;
    private int treeDepth;
    private int nodeCount;

    public QuadTree(BufferedImage image, ErrorCalculator errorCalculator, int minBlockSize, double
threshold) {
        this.originalImage = image;
        this.compressedImage = new BufferedImage(image.getWidth(), image.getHeight(),
BufferedImage.TYPE_INT_RGB);
        this.errorCalculator = errorCalculator;
        this.minBlockSize = minBlockSize;
```

```
        this.threshold = threshold;
        this.treeDepth = 0;
        this.nodeCount = 0;

        // Inisialisasi root node dengan dimensi seluruh gambar (mulai dari titik (0,0))
        this.root = new Node(0, 0, image.getWidth(), image.getHeight());
    }

    public void compress() {
        buildQuadTree(root, 0);
        root.applyColorToImage(compressedImage);
    }

    private void buildQuadTree(Node node, int depth) {
        // Memperbaharui statistik pohon
        nodeCount++;
        treeDepth = Math.max(treeDepth, depth);

        // Menghitung rata-rata warna pada node berdasarkan piksel yang ada
        node.calculateAverageColor(originalImage);

        // Menghitung error pada node menggunakan metode yang dipilih oleh pengguna
        double error = errorCalculator.calculateError(originalImage, node);

        // Mengecek apakah node perlu dibagi lebih lanjut
        if (error > threshold && node.getWidth() / 2 >= minBlockSize && node.getHeight() / 2 >=
minBlockSize) {
            node.split();
            for (Node child : node.getChildren()) {
                buildQuadTree(child, depth + 1);
            }
        }
    }

    public BufferedImage getCompressedImage() {
        return compressedImage;
    }

    public int getTreeDepth() {
        return treeDepth;
    }

    public int getNodeCount() {
        return nodeCount;
    }
```

```
    public Node getRoot() {
        return root;
    }

    // Method untuk menghitung persentase kompresi berdasarkan ukuran file asli dan hasil
    public double calculateCompressionPercentage(long originalSize, long compressedSize) {
        return (1.0 - ((double) compressedSize / originalSize)) * 100.0;
    }

    // Method untuk mendapatkan semua leaf node (digunakan misalnya untuk keperluan visualisasi)
    public List<Node> getAllLeafNodes() {
        List<Node> leafNodes = new ArrayList<>();
        collectLeafNodes(root, leafNodes);
        return leafNodes;
    }

    private void collectLeafNodes(Node node, List<Node> leafNodes) {
        if (node.isLeaf()) {
            leafNodes.add(node);
        } else {
            for (Node child : node.getChildren()) {
                collectLeafNodes(child, leafNodes);
            }
        }
    }
}
```

Kode Program Lainnya

```
public class CompressionParameter {
    private int errorMetricChoice;
    private double threshold;
    private int minBlockSize;
    private double targetCompressionPercentage;
    private String inputImagePath;
    private String outputImagePath;
    private String outputGifPath;

    public CompressionParameter(int errorMetricChoice, double threshold, int minBlockSize,
                                double targetCompressionPercentage, String inputImagePath,
                                String outputImagePath, String outputGifPath) {
        this.errorMetricChoice = errorMetricChoice;
        this.threshold = threshold;
        this.minBlockSize = minBlockSize;
        this.targetCompressionPercentage = targetCompressionPercentage;
```

```
        this.inputImagePath = inputImagePath;
        this.outputImagePath = outputImagePath;
        this.outputGifPath = outputGifPath;
    }

    public int getErrorMetricChoice() {
        return errorMetricChoice;
    }

    public double getThreshold() {
        return threshold;
    }

    public void setThreshold(double threshold) {
        this.threshold = threshold;
    }

    public int getMinBlockSize() {
        return minBlockSize;
    }

    public double getTargetCompressionPercentage() {
        return targetCompressionPercentage;
    }

    public String getInputImagePath() {
        return inputImagePath;
    }

    public String getOutputImagePath() {
        return outputImagePath;
    }

    public String getOutputGifPath() {
        return outputGifPath;
    }
}

public class Entropy implements ErrorCalculator {
    @Override
    public double calculateError(BufferedImage image, Node node) {
        // Initialize frequency maps for each channel
        Map<Integer, Integer> redFreq = new HashMap<>();
        Map<Integer, Integer> greenFreq = new HashMap<>();
        Map<Integer, Integer> blueFreq = new HashMap<>();
```

```
        int totalPixels = 0;

        // Count frequencies
        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));

                    redFreq.put(pixelColor.getRed(), redFreq.getOrDefault(pixelColor.getRed(), 0) + 1);
                    greenFreq.put(pixelColor.getGreen(), greenFreq.getOrDefault(pixelColor.getGreen(),
0) + 1);
                    blueFreq.put(pixelColor.getBlue(), blueFreq.getOrDefault(pixelColor.getBlue(), 0) +
1);

                    totalPixels++;
                }
            }
        }

        if (totalPixels == 0) return 0;

        // Calculate entropy for each channel
        double entropyR = calculateChannelEntropy(redFreq, totalPixels);
        double entropyG = calculateChannelEntropy(greenFreq, totalPixels);
        double entropyB = calculateChannelEntropy(blueFreq, totalPixels);

        // Average entropy across all channels
        return (entropyR + entropyG + entropyB) / 3.0;
    }

    private double calculateChannelEntropy(Map<Integer, Integer> frequencies, int totalPixels) {
        double entropy = 0.0;

        for (Map.Entry<Integer, Integer> entry : frequencies.entrySet()) {
            double probability = (double) entry.getValue() / totalPixels;
            if (probability > 0) {
                entropy -= probability * (Math.log(probability) / Math.log(2));
            }
        }

        return entropy;
    }

    @Override
```

```
    public String getMethodName() {
        return "Entropy";
    }
}

public interface ErrorCalculator {
    double calculateError(BufferedImage image, Node node);
    String getMethodName();
}

public class ImageLoader {
    public static BufferedImage loadImage(String filePath) throws IOException {
        File file = new File(filePath);
        if (!file.exists()) {
            throw new IOException("File not found: " + filePath);
        }
        return ImageIO.read(file);
    }
}

public class ImageSaver {
    public static long saveImage(BufferedImage image, String filePath) throws IOException {
        File outputFile = new File(filePath);

        // Buat direktori jika belum ada
        if (!outputFile.getParentFile().exists()) {
            outputFile.getParentFile().mkdirs();
        }

        // Mendapatkan format file berdasarkan ekstensi (misalnya jpg, png)
        String format = filePath.substring(filePath.lastIndexOf('.') + 1).toLowerCase();

        // Menyimpan gambar dengan format yang telah ditentukan
        ImageIO.write(image, format, outputFile);

        // Mengembalikan ukuran file dalam bytes
        return outputFile.length();
    }
}

public class MAD implements ErrorCalculator {
    @Override
    public double calculateError(BufferedImage image, Node node) {
        double madR = 0;
        double madG = 0;
```

```
        double madB = 0;
        int totalPixels = 0;

        // Calculate average color
        int avgR = 0, avgG = 0, avgB = 0;
        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));
                    avgR += pixelColor.getRed();
                    avgG += pixelColor.getGreen();
                    avgB += pixelColor.getBlue();
                    totalPixels++;
                }
            }
        }

        if (totalPixels == 0) return 0;

        avgR /= totalPixels;
        avgG /= totalPixels;
        avgB /= totalPixels;

        // Calculate MAD
        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));
                    madR += Math.abs(pixelColor.getRed() - avgR);
                    madG += Math.abs(pixelColor.getGreen() - avgG);
                    madB += Math.abs(pixelColor.getBlue() - avgB);
                }
            }
        }

        madR /= totalPixels;
        madG /= totalPixels;
        madB /= totalPixels;

        // Average MAD across all channels
        double avgMAD = (madR + madG + madB) / 3.0;
        return avgMAD;
    }

    @Override
```

```
    public String getMethodName() {
        return "Mean Absolute Deviation (MAD)";
    }
}

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

        try {
            // Ambil parameter input dari pengguna
            CompressionParameter params = getUserInputs(scanner);

            // Muat gambar
            BufferedImage originalImage = ImageLoader.loadImage(params.getInputImagePath());

            // Pilih metode perhitungan error
            ErrorCalculator errorCalculator = selectErrorCalculator(params.getErrorMetricChoice());

            System.out.println("Memulai kompresi menggunakan " + errorCalculator.getMethodName() +
"...");

            // Ukur waktu eksekusi
            long startTime = System.currentTimeMillis();

            // Buat dan jalankan quadtree
            QuadTree quadTree = new QuadTree(originalImage, errorCalculator, params.getMinBlockSize(),
params.getThreshold());
            quadTree.compress();

            // Ambil gambar hasil kompresi
            BufferedImage compressedImage = quadTree.getCompressedImage();

            // Hitung waktu eksekusi
            long endTime = System.currentTimeMillis();
            long waktuEksekusi = endTime - startTime;

            // Simpan gambar hasil kompresi
            System.out.println("Menyimpan gambar hasil kompresi...");
            long ukuranAsli = new File(params.getInputImagePath()).length();
            long ukuranHasil = ImageSaver.saveImage(compressedImage, params.getOutputImagePath());

            // Hitung persentase kompresi
            double persentaseKompresi = quadTree.calculateCompressionPercentage(ukuranAsli,
ukuranHasil);
```

```
            // Tampilkan hasil output
            System.out.println("\n=== Hasil Kompresi ===");
            System.out.println("Waktu eksekusi         : " + waktuEksekusi + " ms");
            System.out.println("Ukuran gambar sebelum   : " + ukuranAsli + " bytes");
            System.out.println("Ukuran gambar setelah   : " + ukuranHasil + " bytes");
            System.out.println("Persentase kompresi     : " + String.format("%.2f%%",
persentaseKompresi));
            System.out.println("Kedalaman pohon         : " + quadTree.getTreeDepth());
            System.out.println("Jumlah simpul pada pohon: " + quadTree.getNodeCount());
            System.out.println("Gambar hasil kompresi disimpan di: " + params.getOutputImagePath());

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

    private static CompressionParameter getUserInputs(Scanner scanner) {
        // Jalur gambar input
        System.out.print("Masukkan alamat absolut gambar yang akan dikompresi: ");
        String inputImagePath = scanner.nextLine();

        // Metode perhitungan error
        System.out.println("\nPilih metode perhitungan error:");
        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): ");
        int errorMetricChoice = Integer.parseInt(scanner.nextLine());

        // Threshold
        System.out.print("\nMasukkan nilai threshold: ");
        double threshold = Double.parseDouble(scanner.nextLine());

        // Ukuran blok minimum
        System.out.print("\nMasukkan ukuran blok minimum: ");
        int minBlockSize = Integer.parseInt(scanner.nextLine());
```

```
        // Persentase kompresi target (fitur bonus)
        System.out.print("\nMasukkan persentase kompresi target (0 untuk menonaktifkan, misalnya 0.5
untuk 50%) [not implemented]: ");
        double targetCompressionPercentage = Double.parseDouble(scanner.nextLine());

        // Jalur output gambar hasil kompresi
        System.out.print("\nMasukkan alamat absolut untuk gambar hasil kompresi: ");
        String outputImagePath = scanner.nextLine();

        // Jalur output GIF proses kompresi (fitur bonus)
        System.out.print("\nMasukkan alamat absolut untuk GIF proses kompresi (kosongkan untuk
melewati) [not implemented]: ");
        String outputGifPath = scanner.nextLine();

        return new CompressionParameter(
            errorMetricChoice,
            threshold,
            minBlockSize,
            targetCompressionPercentage,
            inputImagePath,
            outputImagePath,
            outputGifPath
        );
    }

    private static ErrorCalculator selectErrorCalculator(int choice) {
        switch (choice) {
            case 1:
                return new Variance();
            case 2:
                return new MAD();
            case 3:
                return new MaxPixelDiff();
            case 4:
                return new Entropy();
            case 5:
                return new SSIM();
            default:
                System.out.println("Pilihan tidak valid, menggunakan Variance sebagai default.");
                return new Variance();
        }
    }
}

public class MaxPixelDiff implements ErrorCalculator {
```

```java
    @Override
    public double calculateError(BufferedImage image, Node node) {
        int maxR = 0, minR = 255;
        int maxG = 0, minG = 255;
        int maxB = 0, minB = 255;

        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));

                    // Update max and min for each channel
                    maxR = Math.max(maxR, pixelColor.getRed());
                    minR = Math.min(minR, pixelColor.getRed());

                    maxG = Math.max(maxG, pixelColor.getGreen());
                    minG = Math.min(minG, pixelColor.getGreen());

                    maxB = Math.max(maxB, pixelColor.getBlue());
                    minB = Math.min(minB, pixelColor.getBlue());
                }
            }
        }

        // Calculate differences
        int diffR = maxR - minR;
        int diffG = maxG - minG;
        int diffB = maxB - minB;

        // Average difference across all channels
        double avgDiff = (diffR + diffG + diffB) / 3.0;
        return avgDiff;
    }

    @Override
    public String getMethodName() {
        return "Max Pixel Difference";
    }
}

public class SSIM implements ErrorCalculator {
    // Konstanta perhitungan SSIM
    private static final double C1 = Math.pow(0.01 * 255, 2);
    private static final double C2 = Math.pow(0.03 * 255, 2);
```

```
    // Bobot untuk setiap channel warna
    private static final double WEIGHT_R = 0.33;
    private static final double WEIGHT_G = 0.33;
    private static final double WEIGHT_B = 0.34;

    @Override
    public double calculateError(BufferedImage image, Node node) {
        // Membuat gambar sementara untuk menyimpan warna rata-rata
        BufferedImage avgImage = new BufferedImage(node.getWidth(), node.getHeight(),
BufferedImage.TYPE_INT_RGB);
        Color avgColor = node.getAverageColor();

        if (avgColor == null) {
            node.calculateAverageColor(image);
            avgColor = node.getAverageColor();
        }

        // Isi gambar sementara dengan warna rata-rata
        int avgRGB = avgColor.getRGB();
        for (int j = 0; j < node.getHeight(); j++) {
            for (int i = 0; i < node.getWidth(); i++) {
                avgImage.setRGB(i, j, avgRGB);
            }
        }

        // Ekstrak pixel values dari gambar asli
        List<Integer> redX = new ArrayList<>();
        List<Integer> greenX = new ArrayList<>();
        List<Integer> blueX = new ArrayList<>();

        // Ekstrak pixel values dari warna rata-rata
        List<Integer> redY = new ArrayList<>();
        List<Integer> greenY = new ArrayList<>();
        List<Integer> blueY = new ArrayList<>();

        for (int j = 0; j < node.getHeight(); j++) {
            for (int i = 0; i < node.getWidth(); i++) {
                int x_i = node.getX() + i;
                int y_j = node.getY() + j;

                if (x_i < image.getWidth() && y_j < image.getHeight()) {
                    Color pixelColorX = new Color(image.getRGB(x_i, y_j));
                    redX.add(pixelColorX.getRed());
                    greenX.add(pixelColorX.getGreen());
                    blueX.add(pixelColorX.getBlue());
```

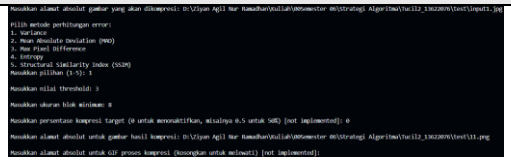
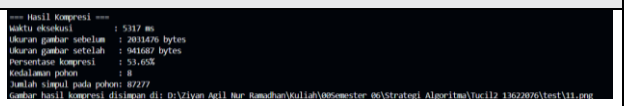
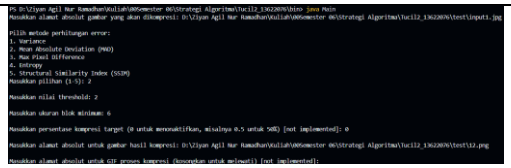
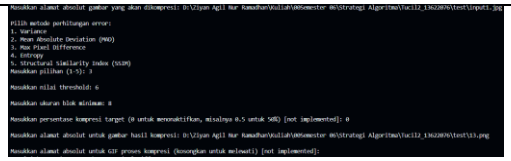
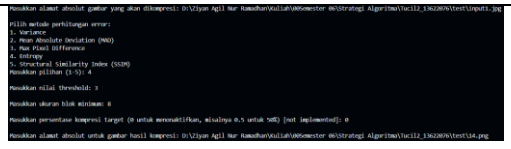
```
                redY.add(avgColor.getRed());
                greenY.add(avgColor.getGreen());
                blueY.add(avgColor.getBlue());
            }
        }
    }

    // Hitung SSIM untuk setiap channel warna
    double ssimR = calculateChannelSSIM(redX, redY);
    double ssimG = calculateChannelSSIM(greenX, greenY);
    double ssimB = calculateChannelSSIM(blueX, blueY);

    // Hitung total SSIM dengan bobot
    double ssimTotal = WEIGHT_R * ssimR + WEIGHT_G * ssimG + WEIGHT_B * ssimB;

    return (1.0 - ssimTotal) / 2.0 * 255.0;
}

private double calculateChannelSSIM(List<Integer> x, List<Integer> y) {
    if (x.isEmpty() || y.isEmpty()) return 1.0;

    // Hitung rata-rata
    double muX = mean(x);
    double muY = mean(y);

    // Hitung varians dan kovarians
    double sigmaX2 = variance(x, muX);
    double sigmaY2 = variance(y, muY);
    double sigmaXY = covariance(x, y, muX, muY);

    // Hitung SSIM
    double numerator = (2 * muX * muY + C1) * (2 * sigmaXY + C2);
    double denominator = (muX * muX + muY * muY + C1) * (sigmaX2 + sigmaY2 + C2);

    if (denominator == 0) return 1.0;

    return numerator / denominator;
}

private double mean(List<Integer> values) {
    double sum = 0;
    for (int value : values) {
        sum += value;
    }
```

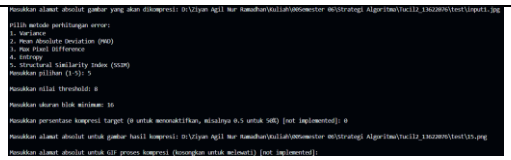
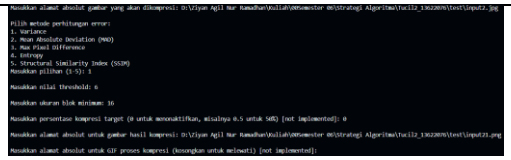
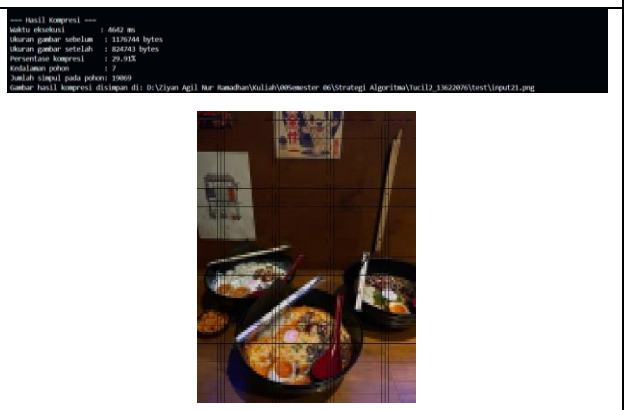
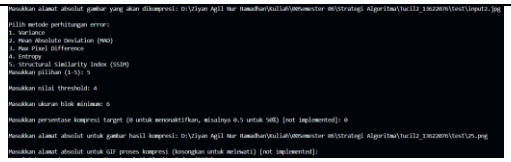
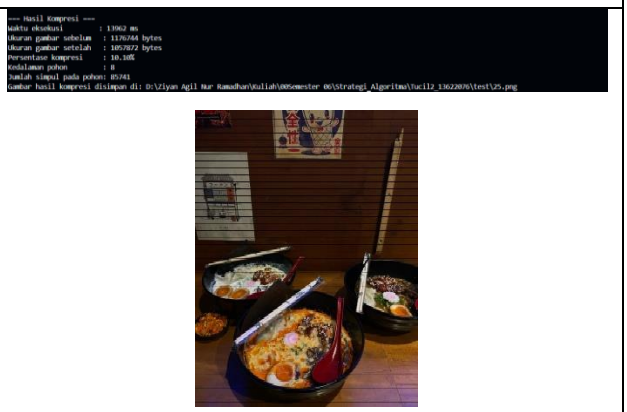
```
        return sum / values.size();
    }

    private double variance(List<Integer> values, double mean) {
        double sum = 0;
        for (int value : values) {
            sum += Math.pow(value - mean, 2);
        }
        return sum / values.size();
    }

    private double covariance(List<Integer> x, List<Integer> y, double meanX, double meanY) {
        double sum = 0;
        for (int i = 0; i < x.size(); i++) {
            sum += (x.get(i) - meanX) * (y.get(i) - meanY);
        }
        return sum / x.size();
    }

    @Override
    public String getMethodName() {
        return "Structural Similarity Index (SSIM)";
    }
}

public class Variance implements ErrorCalculator {
    @Override
    public double calculateError(BufferedImage image, Node node) {
        double varianceR = 0;
        double varianceG = 0;
        double varianceB = 0;
        int totalPixels = 0;

        // Hitung rata-rata warna
        int avgR = 0, avgG = 0, avgB = 0;
        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));
                    avgR += pixelColor.getRed();
                    avgG += pixelColor.getGreen();
                    avgB += pixelColor.getBlue();
                    totalPixels++;
                }
            }
```

```
        }

        if (totalPixels == 0) return 0;

        avgR /= totalPixels;
        avgG /= totalPixels;
        avgB /= totalPixels;

        // Hitung varians untuk setiap channel
        for (int j = node.getY(); j < node.getY() + node.getHeight(); j++) {
            for (int i = node.getX(); i < node.getX() + node.getWidth(); i++) {
                if (i < image.getWidth() && j < image.getHeight()) {
                    Color pixelColor = new Color(image.getRGB(i, j));
                    varianceR += Math.pow(pixelColor.getRed() - avgR, 2);
                    varianceG += Math.pow(pixelColor.getGreen() - avgG, 2);
                    varianceB += Math.pow(pixelColor.getBlue() - avgB, 2);
                }
            }
        }

        varianceR /= totalPixels;
        varianceG /= totalPixels;
        varianceB /= totalPixels;

        // Rata-rata varians untuk semua channel
        double avgVariance = (varianceR + varianceG + varianceB) / 3.0;
        return avgVariance;
    }

    @Override
    public String getMethodName() {
        return "Variance";
    }
}
```

BAB IV

HASIL PENGUJIAN

No	Test Case	Solution
1		 
2		 
3		 
4		 

5		
6		
7		

Penjelasan:

1. Analisis Kompleksitas Waktu

   Asumsikan bahwa N = w × h dengan w dan h adalah lebar dan tinggi gambar. Kompleksitas waktu akan ditentukan oleh tiga metode utama.

   a. Pembangunan Quadtree (buildQuadTree)

Pada tiap node, terdapat pengecekan apakah node perlu dibagi lebih lanjut atau tidak. Jika memenuhi, node akan dibagi menjadi 4 node anak. Proses ini berlangsung secara rekursif hingga mencapai kondisi berhenti. Kedalaman pohon maksimum yang mungkin adalah $\log_4(N) = \log(N)/\log(4) \approx \log(N)/2$ sehingga Kompleksitasnya O(log N).

b. Perhitungan Warna Rata-rata (calculateAverageColor)
   Untuk tiap node, algoritma mengakses setiap piksel dalam area node. Kompleksitas fungsi ini adalah O(w × h) atau O(N).
c. Penerapan Warna ke Gambar (applyColorToImage)
   Untuk setiap node daun, algoritma mengakses seluruh piksel dalam area node. Kompleksitas akan menjadi O(w × h) atau O(N) untuk tiap node daun.

Quadtree membagi gambar menjadi 4 bagian pada setiap level. Pada setiap level, seluruh piksel gambar diproses sekali sehingga total kompleksitas adalah O(N × log N). Sehingga jika seluruh gambar dibagi hingga mencapai ukuran minimum (minBlockSize), kompleksitas waktu totalnya adalah O(N log N).

2. Analisis Kompleksitas Ruang
   Gambar original dan gambar terkompresi: O(N)
   Struktur quadtree: O(k) di mana k adalah jumlah node
   Dalam kasus terburuk k dapat mencapai O(N) jika setiap piksel menjadi node tersendiri
   Sehingga kompleksitas ruang total adalah O(N).

Secara keseluruhan, algoritma kompresi gambar menggunakan quadtree ini memiliki kompleksitas waktu O(N log N) dan kompleksitas ruang O(N), di mana N adalah jumlah piksel dalam gambar (width × height).

LAMPIRAN

No	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	✓	

Pranala kode program:

https://github.com/ziyanagil/Tucil2_13622076.git

REFERENSI

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/07-Algoritma-Divide-and-Conquer-(2025)-Bagian1.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/08-Algoritma-Divide-and-Conquer-(2025)-Bagian2.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/09-Algoritma-Divide-and-Conquer-(2025)-Bagian3.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/10-Algoritma-Divide-and-Conquer-(2025)-Bagian4.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/Tucil2-Stima-2025.pdf