

LAPORAN TUGAS KECIL #2
KOMPRESI GAMBAR DENGAN METODE QUADTREE

IF2211– Strategi Algoritma



Dosen:

Dr. Ir. Rinaldi, M.T.

Anggota Kelompok:

Raudhah Yahya Kuddah 13122003

Muhammad Adli Arindra 18222089

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

2024

Daftar Isi

Daftar Isi.....	2
Rumusan Masalah.....	3
Tujuan.....	4
Penjelasan Algoritma.....	5
1. Penjelasan Teori Kompresi.....	5
2. Penjelasan Teori Algoritma Divide and Conquer.....	5
3. Penjelasan DFS dan BFS.....	6
4. Penjelasan Algoritma pada Program.....	7
Isi Program.....	9
1. shhdshd.....	9
Pengetesan Program.....	10
Lampiran.....	11

Rumusan Masalah

Diberikan suatu gambar digital berwarna dalam format RGB yang terdiri dari sekumpulan *pixel* dua dimensi dengan nilai intensitas warna baik warna merah (Red), hijau (Green), dan biru (Blue). Penyimpanan gambar dengan menyimpan nilai RGB untuk setiap *pixel* secara langsung dapat menyebabkan *file size* sangat besar. Sehingga diperlukan suatu metode kompresi untuk mengurangi *file size* gambar tersebut tanpa kehilangan banyak informasi visual penting atau *detail* yang terdapat di dalamnya.

Komponen utama dari masalah ini adalah gambar sebagai *input* yang direpresentasikan sebagai matriks dua dimensi dari *pixel* warna. Masing-masing blok *pixel* kemudian akan dianalisis untuk mengetahui apakah blok tersebut memiliki nilai warna yang seragam atau tidak. Jika tidak seragam, maka blok tersebut perlu dibagi lagi menjadi blok-blok yang lebih kecil untuk dilakukan analisis serupa. Untuk menyelesaikan permasalahan ini, digunakan pendekatan berbasis algoritma *divide and conquer* menggunakan struktur data *Quadtree*, yang secara rekursif membagi gambar menjadi empat bagian untuk mendeteksi area dengan warna yang seragam.

Tujuan

1. Memahami konsep algoritma divide and conquer dalam menyelesaikan permasalahan pemrosesan dan kompresi data visual.
2. Menganalisis performa algoritma Quadtree dalam hal efisiensi kompresi dan kompleksitas pemrosesan terhadap berbagai jenis gambar uji.
3. Mengembangkan program kompresi gambar yang efisien dengan parameter yang dapat disesuaikan, seperti metode pengukuran error, threshold, dan ukuran blok minimum.

Penjelasan Algoritma

1. Penjelasan Teori Kompresi

Kompresi adalah proses untuk mengurangi ukuran data dengan tujuan menghemat ruang penyimpanan atau mempercepat proses pengiriman data. Dalam konteks gambar digital, kompresi bertujuan untuk menyimpan representasi visual suatu gambar dalam *file size* yang lebih kecil tanpa mengorbankan terlalu banyak kualitas visual. Kompresi bisa dilakukan secara *lossless* (tanpa kehilangan informasi) atau *lossy* (dengan sedikit kehilangan informasi untuk efisiensi yang lebih besar).

Untuk melakukan kompresi, komputer terlebih dahulu membaca gambar dalam bentuk matriks dua dimensi yang terdiri dari *pixel-pixel*. Setiap *pixel* memiliki tiga nilai warna utama yaitu merah (Red), hijau (Green), dan biru (Blue), yang dikenal sebagai sistem warna RGB. Hasil pembacaan ini kemudian direpresentasikan sebagai array dua dimensi, di mana setiap elemen menyimpan nilai RGB dari satu *pixel*.

Selanjutnya, algoritma kompresi akan menganalisis blok-blok gambar untuk mencari bagian-bagian yang memiliki keseragaman warna atau intensitas yang tinggi. Bagian yang dianggap seragam dapat direpresentasikan dalam bentuk sederhana tanpa menyimpan data tiap *pixel*-nya. Salah satu metode yang digunakan dalam tugas ini adalah dengan membagi gambar menggunakan struktur data Quadtree, yang akan membagi gambar menjadi empat bagian secara rekursif hingga diperoleh blok-blok yang cukup seragam. Dengan menghilangkan data berulang dan hanya menyimpan informasi penting dari tiap blok, ukuran gambar dapat dikurangi secara signifikan, menghasilkan proses kompresi yang efisien.

2. Penjelasan Teori Algoritma *Divide and Conquer*

Divide and conquer adalah strategi algoritma yang menyelesaikan masalah dengan cara membaginya menjadi sub-masalah yang lebih kecil, menyelesaikan masing-masing secara rekursif, lalu menggabungkan hasilnya. Dalam kompresi gambar menggunakan *Quadtree*, pendekatan ini digunakan untuk membagi gambar menjadi blok-blok kecil berdasarkan keseragaman warna. Gambar awal dibagi menjadi empat bagian, lalu setiap bagian dianalisis: jika warnanya tidak seragam, maka blok tersebut dibagi lagi menjadi empat sub-blok, dan proses ini terus diulang secara rekursif.

Proses ini memungkinkan kompresi yang efisien karena bagian-bagian gambar yang seragam tidak perlu disimpan pixel demi pixel, cukup disimpan sebagai satu *node* dalam struktur *Quadtree*. Dengan cara ini, algoritma dapat mengurangi jumlah data yang disimpan tanpa kehilangan terlalu banyak informasi visual, sehingga menghasilkan gambar terkompresi yang tetap representatif.

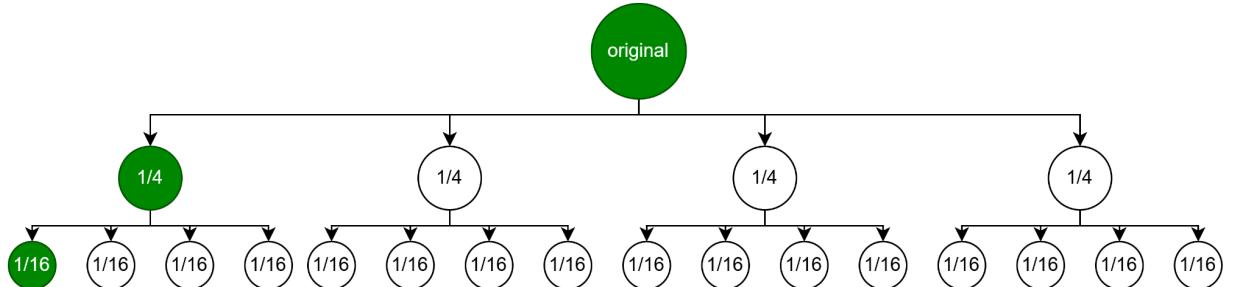
3. Penjelasan DFS dan BFS

Dalam kompresi gambar menggunakan struktur data *Quadtree*, proses pembagian blok mengikuti prinsip algoritma *divide and conquer*, yaitu dengan membagi gambar menjadi bagian-bagian yang lebih kecil secara rekursif sampai ditemukan blok yang cukup seragam atau mencapai ukuran minimum. Strategi ini secara alami berjalan sejalan dengan algoritma pencarian seperti DFS (*Depth-First Search*) atau BFS (*Breadth-First Search*), tergantung pada parameter dan tujuan kompresi yang digunakan.

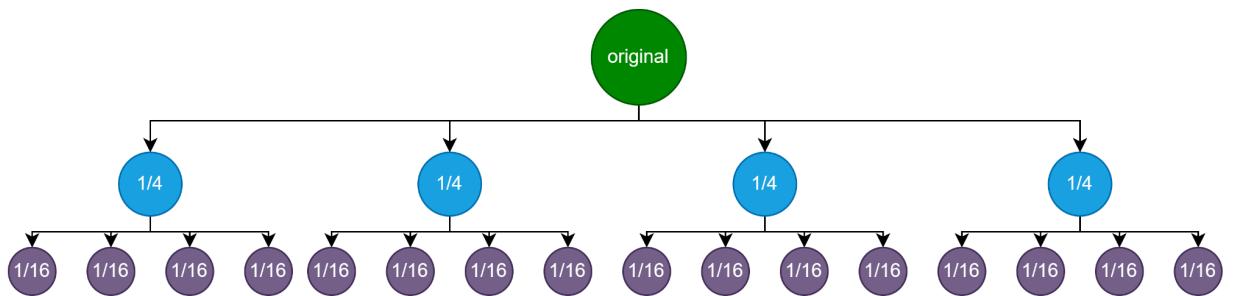
Jika kompresi dilakukan berdasarkan *threshold error* dan *minimum block size*, maka traversal pohon yang digunakan adalah DFS. Dalam pendekatan ini, program akan langsung membagi dan menyelesaikan satu blok hingga tuntas ke dalam sub-blok terkecil sebelum melanjutkan ke blok lainnya. Ini sangat cocok dengan pola *divide and conquer* yang menyelesaikan setiap sub-masalah sepenuhnya sebelum kembali ke masalah yang lebih besar. Namun, pendekatan DFS ini memiliki kelemahan karena sifatnya yang fokus ke satu jalur secara mendalam, hasil kompresi bisa tidak merata. Misalnya, hanya satu bagian gambar (yang lebih kompleks atau bervariasi warnanya) yang terkompresi sangat dalam, sementara bagian lainnya tidak dibagi sama sekali karena belum dieksplorasi. Hal ini menyebabkan ketidakseimbangan dalam struktur pohon dan distribusi blok pada gambar hasil kompresi.

Sebaliknya, ketika menggunakan target persentase kompresi, pendekatan yang lebih tepat adalah BFS. BFS bekerja dengan memproses semua node pada suatu level pohon sebelum berpindah ke level berikutnya, sehingga pembagian blok dilakukan secara merata di seluruh area gambar pada kedalaman yang sama. Pendekatan ini memungkinkan kontrol yang lebih baik terhadap jumlah node dan ukuran blok secara global, yang penting untuk mencapai target kompresi yang ditentukan. Dengan BFS, struktur *Quadtree* yang dihasilkan cenderung lebih seimbang dan kompresi menjadi lebih merata di seluruh gambar.

Singkatnya, DFS cocok untuk kompresi berbasis lokal yang adaptif terhadap variasi warna, tetapi bisa menghasilkan hasil yang tidak merata. BFS lebih baik untuk pendekatan kompresi global yang membutuhkan distribusi blok yang seimbang dan efisiensi yang terukur terutama saat bekerja dengan target ukuran file atau persentase kompresi tertentu.



Gambar 1. Penggunaan *Depth-First Search*



Gambar 2. Penggunaan *Breadth-First Search*

4. Penjelasan Algoritma pada Program

Untuk menyelesaikan proses kompresi gambar, digunakan algoritma *divide and conquer* berbasis struktur data *Quadtree*. Algoritma ini bekerja dengan cara membagi gambar menjadi blok-blok yang lebih kecil secara rekursif, berdasarkan keseragaman nilai warna *pixel* dalam setiap blok. Tujuan dari pendekatan ini adalah untuk menyederhanakan representasi gambar dengan hanya menyimpan informasi penting dari bagian-bagian yang seragam.

Langkah pertama yang dilakukan oleh program adalah membaca *input* berupa *file* gambar, yang kemudian dikonversi ke dalam bentuk matriks *pixel* dengan sistem warna RGB. Setiap *pixel* direpresentasikan sebagai *array* dari tiga nilai warna: merah, hijau, dan biru. Nilai-nilai ini kemudian disimpan dalam struktur matriks dua dimensi untuk memudahkan proses analisis blok.

Selanjutnya, algoritma akan memulai proses kompresi dari seluruh gambar sebagai satu blok utuh. Kemudian, blok ini akan diperiksa apakah nilai warnanya cukup seragam berdasarkan metode pengukuran *error* yang dipilih (seperti variansi, MAD, dan lainnya) serta nilai *threshold* yang ditentukan pengguna. Jika blok tidak memenuhi kriteria keseragaman dan masih dapat dibagi (ukurannya lebih besar dari nilai *minimum block size*), maka blok akan dibagi menjadi empat bagian: kiri atas, kanan atas, kiri bawah, dan kanan bawah. Proses ini dilakukan secara rekursif untuk setiap sub-blok.

Struktur data *Quadtree* digunakan untuk menyimpan hubungan antara blok-blok ini. Setiap *node* dalam *Quadtree* mewakili satu blok gambar, dan jika blok tersebut dibagi, maka *node* tersebut akan memiliki empat anak. *Node* yang tidak dibagi lagi (karena sudah seragam atau terlalu kecil untuk dibagi) disebut sebagai *leaf* dan menyimpan nilai warna rata-rata dari blok tersebut.

Program juga menghitung informasi penting seperti jumlah *node*, kedalaman maksimum pohon, ukuran gambar sebelum dan sesudah dikompresi, serta persentase kompresi. Jika pengguna mengaktifkan fitur target kompresi, maka algoritma akan menyesuaikan nilai *threshold* secara dinamis dan traversal pohon dilakukan secara per-depth (BFS) agar distribusi kompresi merata di seluruh gambar. Namun, jika pengguna hanya memberikan threshold dan ukuran blok minimum, maka traversal dilakukan secara rekursif dan mendalam (DFS), yang fokus pada penyelesaian blok demi blok secara lokal. Dengan ini, program dapat menghasilkan gambar hasil kompresi yang jauh lebih ringan namun tetap mempertahankan bentuk dan informasi visual utama dari gambar asli.

Isi Program

1. Program

```
using Emgu.CV.CvEnum;
using Emgu.CV;
using System.Drawing;
using Emgu.CV.Util;
using System.Diagnostics;
using System.IO;
using Emgu.CV.Structure;

namespace QuadtreeCompression
{
    internal class Program
    {
        static string defaultInputPath = "D:\\semester
6\\stima\\Tucil2_13122003_18222089\\src\\QuadtreeCompression\\images\\rose.webp";
        static string defaultOutputPath = "D:\\semester
6\\stima\\Tucil2_13122003_18222089\\src\\QuadtreeCompression\\output.png";

        static string
            inputPath = defaultInputPath,
            outputPath = defaultOutputPath;
        static int
            methodIdx = 0,
            threshold = 20,
            minBlockSize = 10,
            nodecount = 0,
            depth = 0;
        static double compressionTarget;
        static List<List<Pixel>> image;
        static long compressedPixels = 1,
            currentWidth = -1;
        static double initialError,
            totalPixels;

        static void Main(string[] args)
        {
            GetInput();

            Mat mat = CvInvoke.Imread(inputPath, ImreadModes.Color);
            image = Image.MatToPixels(mat);

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
```

```

        Loading.Start();
        if (compressionTarget <= 0)
        {
            Console.WriteLine("Menggunakan threshold dan minBlockSize untuk kompresi");
            Solve(0, 0, image[0].Count, image.Count);
        }
        else
        {
            Console.WriteLine("Menggunakan target kompresi untuk kompresi");
            SolveTarget();
        }
        Loading.Stop();

        stopwatch.Stop();
        Mat output = Image.PixelsToMat(image);
        Image.ExportMat(output, outputPath);
        Console.WriteLine($"Waktu eksekusi: {stopwatch.ElapsedMilliseconds} ms");
        Console.WriteLine($"Ukuran gambar sebelum: {Utils.GetFileSize(inputPath)} bytes");
        Console.WriteLine($"Ukuran gambar setelah: {Utils.GetFileSize(outputPath)} bytes");
        Console.WriteLine($"Presentase gambar terkompresi: {(100 - (Utils.GetFileSize(outputPath) * 100 / Utils.GetFileSize(inputPath)))}%");
        Console.WriteLine($"Kedalaman pohon: {depth}");
        Console.WriteLine($"Banyak simpul: {nodecount}");
        Image.Show(image);
    }

    static void GetInput()
    {
        Console.Write("Masukkan absolute path dari gambar\n> ");
        inputPath = Console.ReadLine();
        Console.Write("Masukkan nomor dari metode perhitungan error:\n" +
                    "1. Variansi\n2. Mean Absolute Difference\n3. Maximum Pixel Difference\n" +
                    "4. Entropy\n> ");
        methodIdx = int.Parse(Console.ReadLine());
        Console.Write("Masukkan ambang batas\n> ");
        threshold = int.Parse(Console.ReadLine());
        Console.Write("Masukkan ukuran blok minimum\n> ");
        minBlockSize = int.Parse(Console.ReadLine());
        Console.Write("Masukkan target kompresi (masukkan 0 untuk tidak memakai target, masukkan persentase (0..1) untuk memakai target dan mengabaikan threshold & minblocksize: \n> ");
        compressionTarget = double.Parse(Console.ReadLine());
        Console.Write("Masukkan absolute path untuk menyimpan gambar\n> ");
        outputPath = Console.ReadLine();
    }

    static void Solve(int startX, int startY, int width, int height)

```

```

{
    double error = GetError(image, startX, startY, width, height);
    double area = width * height;
    compressedPixels += 4;

    if (width != currentWidth)
    {
        currentWidth = width;
        depth++;
    }

    if (area < minBlockSize || error < threshold)
    {
        Image.NormalizePixels(image, startX, startY, width, height);
        nodecount++;
        return;
    }

    int halfWidth = width / 2;
    int halfHeight = height / 2;

    Solve(startX, startY, halfWidth, halfHeight);
    Solve(startX + halfWidth, startY, halfWidth, halfHeight);
    Solve(startX, startY + halfHeight, halfWidth, halfHeight);
    Solve(startX + halfWidth, startY + halfHeight, halfWidth, halfHeight);
}

static void SolveTarget()
{
    Queue<(int startX, int startY, int width, int height)> queue = new();
    queue.Enqueue((0, 0, image[0].Count, image.Count));
    methodIdx = 4;
    initialError = GetError(image, 0, 0, image[0].Count, image.Count);
    totalPixels = image.Count * image[0].Count * initialError;
    int currentCompressedPixels = 1;
    bool merge = false;

    while (queue.Count > 0)
    {
        var (startX, startY, width, height) = queue.Dequeue();
        double error = GetError(image, startX, startY, width, height);
        double area = width * height;

        if (currentWidth != width)
        {
            currentWidth = width;
            depth++;
            if (!merge)
            {
                currentCompressedPixels *= 4;
                compressedPixels = 0;
            }
        }
    }
}

```

```

        }

    }

    double currentPixelsEstimated = compressedPixels *
Math.Pow(compressionTarget, 8);

    if ((1 - currentPixelsEstimated / totalPixels) < compressionTarget)
        merge = true;

    if (merge)
    {
        Image.NormalizePixels(image, startX, startY, width, height);
        nodecount++;
        continue;
    }

    int halfWidth = width / 2;
    int halfHeight = height / 2;

    if (halfWidth == 0 || halfHeight == 0)
    {
        Image.NormalizePixels(image, startX, startY, width, height);
        nodecount++;
        continue;
    }

    compressedPixels += currentCompressedPixels;
    queue.Enqueue((startX, startY, halfWidth, halfHeight));
    queue.Enqueue((startX + halfWidth, startY, halfWidth, halfHeight));
    queue.Enqueue((startX, startY + halfHeight, halfWidth, halfHeight));
    queue.Enqueue((startX + halfWidth, startY + halfHeight, halfWidth,
halfHeight));
}
}

static double GetError(List<List<Pixel>> image, int startX, int startY, int
width, int height)
{
    List<List<Pixel>> subImage = Image.ExtractSubImage(image, startX, startY,
width, height);

    switch (methodIdx)
    {
        case 1:
            return Measurer.Variance(subImage);
        case 2:
            return Measurer.MeanAbsoluteDeviation(subImage);
        case 3:
            return Measurer.MaxPixelDifference(subImage);
        case 4:
            return Measurer.Entropy(subImage);
        default:
    }
}

```

```
        return 0.0;
    }

}

}
}
```

2. Measurer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace QuadtreeCompression
{
    internal static class Measurer
    {
        public static double Variance(List<List<Pixel>> image)
        {
            int rows = image.Count;
            int cols = image[0].Count;
            int totalPixels = rows * cols;

            double sumR = 0.0, sumG = 0.0, sumB = 0.0;
            double sumSqR = 0.0, sumSqG = 0.0, sumSqB = 0.0;

            for (int y = 0; y < rows; y++)
            {
                for (int x = 0; x < cols; x++)
                {
                    Pixel pixel = image[y][x];

                    sumR += pixel.R;
                    sumG += pixel.G;
                    sumB += pixel.B;

                    sumSqR += pixel.R * pixel.R;
                    sumSqG += pixel.G * pixel.G;
                    sumSqB += pixel.B * pixel.B;
                }
            }

            double meanR = sumR / totalPixels;
```

```

        double meanG = sumG / totalPixels;
        double meanB = sumB / totalPixels;

        double varR = (sumSqR / totalPixels) - (meanR * meanR);
        double varG = (sumSqG / totalPixels) - (meanG * meanG);
        double varB = (sumSqB / totalPixels) - (meanB * meanB);

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

    public static double MeanAbsoluteDeviation(List<List<Pixel>> image)
    {
        int rows = image.Count;
        int cols = image[0].Count;
        int totalPixels = rows * cols;

        double sumR = 0.0, sumG = 0.0, sumB = 0.0;

        foreach (var row in image)
        {
            foreach (var pixel in row)
            {
                sumR += pixel.R;
                sumG += pixel.G;
                sumB += pixel.B;
            }
        }

        double meanR = sumR / totalPixels;
        double meanG = sumG / totalPixels;
        double meanB = sumB / totalPixels;

        double madR = 0.0, madG = 0.0, madB = 0.0;

        foreach (var row in image)
        {
            foreach (var pixel in row)
            {
                madR += Math.Abs(pixel.R - meanR);
                madG += Math.Abs(pixel.G - meanG);
                madB += Math.Abs(pixel.B - meanB);
            }
        }

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

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

```

```

public static double MaxPixelDifference(List<List<Pixel>> image)
{
    int maxR = int.MinValue, maxG = int.MinValue, maxB = int.MinValue;
    int minR = int.MaxValue, minG = int.MaxValue, minB = int.MaxValue;

    foreach (var row in image)
    {
        foreach (var pixel in row)
        {
            maxR = Math.Max(maxR, pixel.R);
            minR = Math.Min(minR, pixel.R);

            maxG = Math.Max(maxG, pixel.G);
            minG = Math.Min(minG, pixel.G);

            maxB = Math.Max(maxB, pixel.B);
            minB = Math.Min(minB, pixel.B);
        }
    }

    double diffR = maxR - minR;
    double diffG = maxG - minG;
    double diffB = maxB - minB;

    return (diffR + diffG + diffB) / 3.0;
}

public static double Entropy(List<List<Pixel>> image)
{
    int totalPixels = image.Count * image[0].Count;

    double EntropyForChannel(Func<Pixel, int> channelSelector)
    {
        var histogram = new int[256];
        foreach (var row in image)
        {
            foreach (var pixel in row)
            {
                histogram[channelSelector(pixel)]++;
            }
        }

        double entropy = 0.0;
        foreach (var count in histogram)
        {
            if (count == 0) continue;
            double p = (double)count / totalPixels;
            entropy -= p * Math.Log2(p);
        }
    }

    return entropy;
}

```

```

        }

        double entropyR = EntropyForChannel(p => p.R);
        double entropyG = EntropyForChannel(p => p.G);
        double entropyB = EntropyForChannel(p => p.B);

        return (entropyR + entropyG + entropyB) / 3.0;
    }
}
}

```

3. Image

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Emgu.CV.CvEnum;
using Emgu.CV.Util;
using Emgu.CV;

namespace QuadtreeCompression
{
    public class Pixel
    {
        public int R { get; set; }
        public int G { get; set; }
        public int B { get; set; }

        public Pixel(int r, int g, int b)
        {
            R = r;
            G = g;
            B = b;
        }
    }

    internal static class Image
    {
        private static readonly int _width = 600;
        public static void Show(Mat image)
        {
            int newHeight = (image.Height * _width) / image.Width;

```

```

        Mat resizedImage = new Mat();
        CvInvoke.Resize(image, resizedImage, new Size(_width, newHeight), 0, 0,
Inter.Linear);

        CvInvoke.Imshow("Resized Image", resizedImage);
        CvInvoke.WaitKey(0);
    }

    public static void Show(List<List<Pixel>> image)
    {
        int rows = image.Count;
        int cols = image[0].Count;

        Mat ret = new Mat(rows, cols, DepthType.Cv8U, 3);

        byte[] imageData = new byte[rows * cols * 3];

        for (int y = 0; y < rows; y++)
        {
            for (int x = 0; x < cols; x++)
            {
                int index = (y * cols + x) * 3;
                Pixel pixel = image[y][x];

                imageData[index] = (byte)pixel.B;
                imageData[index + 1] = (byte)pixel.G;
                imageData[index + 2] = (byte)pixel.R;
            }
        }

        System.Runtime.InteropServices.Marshal.Copy(imageData, 0,
ret.DataPointer, imageData.Length);

        int newHeight = (ret.Height * _width) / ret.Width;
        Mat resizedImage = new Mat();
        CvInvoke.Resize(ret, resizedImage, new Size(_width, newHeight), 0, 0,
Inter.Linear);

        CvInvoke.Imshow("Resized Image", resizedImage);
        CvInvoke.WaitKey(0);
    }

    public static List<List<Pixel>> MatToPixels(Mat image)
    {
        List<List<Pixel>> result = new List<List<Pixel>>();

        int rows = image.Rows;
        int cols = image.Cols;
    }
}

```

```

byte[] imageData = new byte[rows * cols * 3];

System.Runtime.InteropServices.Marshal.Copy(image.DataPointer, imageData,
0, imageData.Length);

for (int y = 0; y < rows; y++)
{
    List<Pixel> row = new List<Pixel>();
    for (int x = 0; x < cols; x++)
    {
        int index = (y * cols + x) * 3;
        int b = imageData[index];
        int g = imageData[index + 1];
        int r = imageData[index + 2];

        row.Add(new Pixel(r, g, b));
    }
    result.Add(row);
}

return result;
}

public static Mat PixelsToMat(List<List<Pixel>> image)
{
    int rows = image.Count;
    int cols = image[0].Count;
    Mat matImage = new Mat(rows, cols, DepthType.Cv8U, 3);

    byte[] imageData = new byte[rows * cols * 3];

    for (int y = 0; y < rows; y++)
    {
        for (int x = 0; x < cols; x++)
        {
            int index = (y * cols + x) * 3;
            Pixel pixel = image[y][x];

            imageData[index] = (byte)pixel.B;
            imageData[index + 1] = (byte)pixel.G;
            imageData[index + 2] = (byte)pixel.R;
        }
    }

    System.Runtime.InteropServices.Marshal.Copy(imageData, 0,
matImage.DataPointer, imageData.Length);
    return matImage;
}

public static void ExportMat(Mat image, string outputPath)
{

```

```

        CvInvoke.Imwrite(outputPath, image);
    }

    public static void NormalizePixels(List<List<Pixel>> image, int startX, int
startY, int width, int height)
    {
        long sumR = 0, sumG = 0, sumB = 0;
        int totalPixels = width * height;

        for (int y = startY; y < startY + height; y++)
        {
            for (int x = startX; x < startX + width; x++)
            {
                sumR += image[y][x].R;
                sumG += image[y][x].G;
                sumB += image[y][x].B;
            }
        }

        int meanR = (int)(sumR / totalPixels);
        int meanG = (int)(sumG / totalPixels);
        int meanB = (int)(sumB / totalPixels);

        for (int y = startY; y < startY + height; y++)
        {
            for (int x = startX; x < startX + width; x++)
            {
                image[y][x] = new Pixel(meanR, meanG, meanB);
            }
        }
    }

    public static List<List<List<Pixel>>> SplitIntoQuarters(List<List<Pixel>>
image)
    {
        int rows = image.Count / 2;
        int cols = image[0].Count / 2;

        List<List<Pixel>> topLeft = new List<List<Pixel>>();
        List<List<Pixel>> topRight = new List<List<Pixel>>();
        List<List<Pixel>> bottomLeft = new List<List<Pixel>>();
        List<List<Pixel>> bottomRight = new List<List<Pixel>>();

        for (int y = 0; y < rows; y++)
        {
            topLeft.Add(image[y].GetRange(0, cols));
            topRight.Add(image[y].GetRange(cols, cols));
        }

        for (int y = rows; y < rows * 2; y++)
    }
}

```

```

        {
            bottomLeft.Add(image[y].GetRange(0, cols));
            bottomRight.Add(image[y].GetRange(cols, cols));
        }

        return new List<List<List<Pixel>>> { topLeft, topRight, bottomLeft,
bottomRight };
    }

    public static List<List<Pixel>> ExtractSubImage(List<List<Pixel>> image, int
startX, int startY, int width, int height)
{
    List<List<Pixel>> subImage = new List<List<Pixel>>();

    for (int y = startY; y < startY + height; y++)
    {
        List<Pixel> row = new List<Pixel>();
        for (int x = startX; x < startX + width; x++)
        {
            row.Add(image[y][x]);
        }
        subImage.Add(row);
    }

    return subImage;
}
}
}

```

4. Utils

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Emgu.CV.Dnn;

namespace QuadtreeCompression
{
    internal class Utils
    {
        public static long GetFileSize(string absolutePath)
        {
            FileInfo fileInfo = new FileInfo(absolutePath);

```

```
        if ( fileInfo.Exists )
        {
            return fileInfo.Length;
        }
        else
        {
            throw new FileNotFoundException("File not found.", absolutePath);
        }
    }
}
```

Pengetesan Program

Berikut merupakan tabel dari pengujian yang dilakukan terhadap program. Pengujian dilakukan dengan menggunakan gambar yang berbeda, dan/atau konfigurasi yang berbeda.

No.	<i>threshold</i>	<i>Minimum block size</i>	<i>Compression Target</i>	<i>method</i>
1.	400	10	-	<i>Variance</i>
2.	45	10	-	MAD
3.	115	4	-	<i>Maximum Pixel Difference</i>
4.	2	4	-	Entropy
5.	-	-	0.5	<i>Compression Target</i>
6.	-	-	0.9	<i>Compression Target</i>
7.	800	10	-	<i>Variance</i>
8.	50	4	-	MAD

<i>Image name</i>	<i>Image before compressed</i>
-------------------	--------------------------------

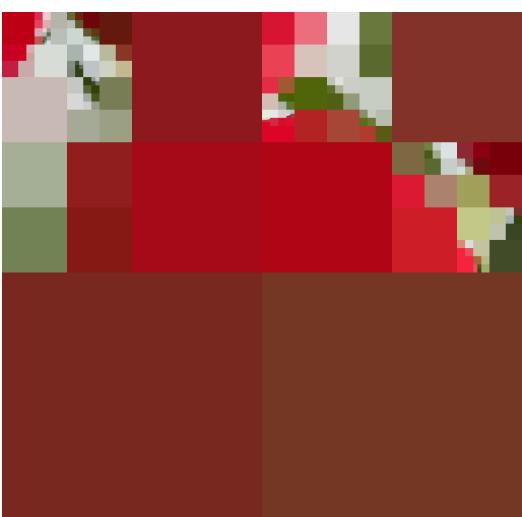
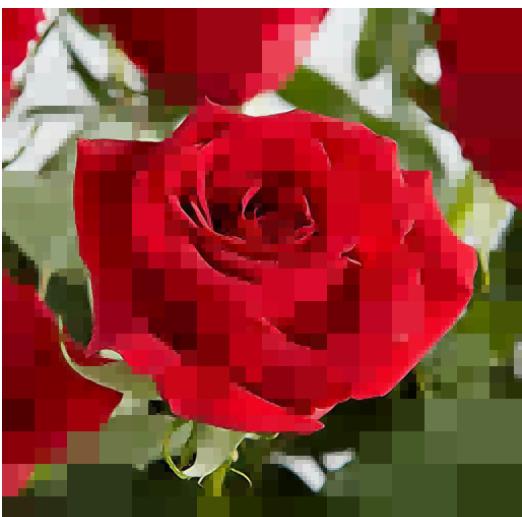
rose



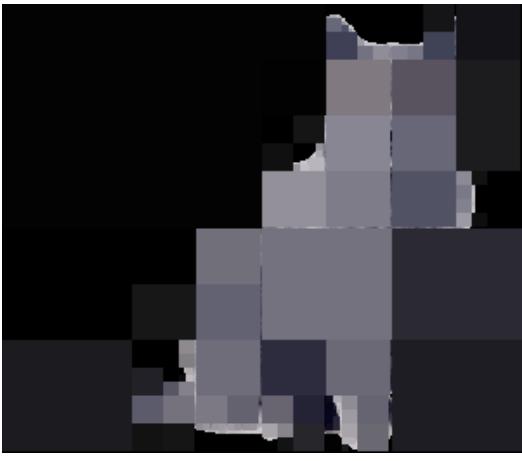
cat



No.	<i>Output</i>	<i>Compressed Image</i>
-----	---------------	-------------------------

1.	<pre>Waktu eksekusi: 8196 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 1405130 bytes Presentase gambar terkompresi: 89% Kedalaman pohon: 11399 Banyak simpul: 19573</pre>	
2.	<pre>Waktu eksekusi: 4587 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 159315 bytes Presentase gambar terkompresi: 99% Kedalaman pohon: 114 Banyak simpul: 190</pre>	
3.	<pre>Waktu eksekusi: 7682 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 967446 bytes Presentase gambar terkompresi: 92% Kedalaman pohon: 4744 Banyak simpul: 8098</pre>	

4.	<pre> Waktu eksekusi: 16867 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 10168997 bytes Presentase gambar terkompresi: 14% Kedalaman pohon: 711977 Banyak simpul: 1220464 </pre>	
5.	<pre> Waktu eksekusi: 3586 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 56514 bytes Presentase gambar terkompresi: 100% Kedalaman pohon: 1 Banyak simpul: 1 </pre>	
6.	<pre> Waktu eksekusi: 15704 ms Ukuran gambar sebelum: 11820712 bytes Ukuran gambar setelah: 6229263 bytes Presentase gambar terkompresi: 48% Kedalaman pohon: 10 Banyak simpul: 220441 </pre>	

7.	<pre> Waktu eksekusi: 523 ms Ukuran gambar sebelum: 77102 bytes Ukuran gambar setelah: 26104 bytes Presentase gambar terkompresi: 67% Kedalaman pohon: 946 Banyak simpul: 1630 </pre>	
8.	<pre> Waktu eksekusi: 511 ms Ukuran gambar sebelum: 77102 bytes Ukuran gambar setelah: 10806 bytes Presentase gambar terkompresi: 86% Kedalaman pohon: 202 Banyak simpul: 361 </pre>	

Dapat dilihat dari tabel di atas, bahwa terdapat perbedaan persentase kompresi antara persentase yang dimasukkan oleh user, dan juga hasil persentase keluaran gambar. Hal ini dikarenakan algoritma tersebut menggunakan perbandingan jumlah pixel dari gambar awal dengan gambar yang dikompresi untuk memperkirakan perbandingan besar file. Artinya, program akan mengasumsikan bahwa gambar yang dihasilkan sudah memiliki 50% dari besar file semula apabila gambar yang dihasilkan sudah memiliki 50% dari total jumlah pixel dari file semula.

Pada kenyataannya, gambar berformat png juga melakukan kompresi di dalam pembuatannya dengan cara melakukan kompresi pada bagian-bagian pixel yang memiliki pola berulang. Ini artinya perbandingan total pixel sebelum dan setelah kompresi tidak berkorelasi lurus dengan perbandingan besar file sebelum dan sesudah.

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

Algoritma kompresi gambar yang dikembangkan menggunakan pendekatan *divide and conquer* dengan memanfaatkan struktur data *Quadtree* untuk merepresentasikan blok-blok gambar yang memiliki keseragaman warna. Secara umum, algoritma membagi gambar secara rekursif menjadi empat sub-blok, lalu menghitung nilai error dari setiap blok berdasarkan metode yang dipilih, seperti variansi, mean absolute deviation (MAD), maximum pixel difference, atau entropy. Proses pembagian dilakukan hingga tercapai batas minimum blok (`minBlockSize`), nilai error sudah berada di bawah threshold tertentu, atau hingga target persentase kompresi terpenuhi.

Berikut adalah hasil analisis dari implementasi dan percobaan:

1. Kompleksitas Waktu

Kompleksitas waktu algoritma bergantung pada konten gambar dan kedalaman pohon Quadtree. Dalam kasus terburuk (gambar sangat tidak seragam), algoritma akan membagi hingga blok terkecil, menghasilkan kompleksitas waktu mendekati $O(n^2)$, di mana n adalah jumlah pixel dalam gambar. Untuk setiap blok, dilakukan perhitungan nilai error terhadap seluruh pixel dalam blok, sehingga waktu total $\approx O(b \cdot m)$, dengan b adalah jumlah blok dan m adalah jumlah pixel per blok.

2. Kompleksitas Ruang

Pohon Quadtree menyimpan simpul untuk setiap blok yang tidak seragam. Dalam kasus terburuk, setiap pixel bisa menjadi simpul sendiri, menghasilkan kompleksitas ruang mendekati $O(n^2)$. Namun, pada gambar yang cenderung seragam, jumlah simpul jauh lebih sedikit dibanding menyimpan semua pixel secara eksplisit, sehingga efisiensi ruang meningkat signifikan.

3. Efektivitas Kompresi

Efektivitas kompresi sangat dipengaruhi oleh parameter threshold, metode penghitungan error, dan ukuran minimum blok. Gambar dengan area seragam (misalnya latar polos) dapat dikompresi lebih efisien. Sebaliknya, gambar dengan detail tinggi dan warna bervariasi menghasilkan pohon yang lebih dalam dan bercabang, sehingga kompresinya lebih terbatas.

4. Performa Eksekusi

Berdasarkan pengujian terhadap gambar rose.png dan cat.png, waktu eksekusi berada di kisaran 500–15000 ms, tergantung parameter input. Kedalaman pohon dan jumlah simpul meningkat seiring penurunan nilai threshold atau kenaikan target kompresi.

5. Efektifitas kompresi berdasarkan target persentase kompresi

Program sudah dapat menerima inputan berupa target kompresi dari file yang diinginkan. Namun, karena untuk mengecek ukuran file yang sebenarnya di setiap pembagiannya memakan waktu yang lama, target file diambil dari sebuah estimasi perbandingan jumlah pixel. Oleh karena itu, meskipun prosesnya jauh lebih cepat, keakuratan hasil persentase dengan persentase yang diminta tidaklah tinggi.

Kesimpulannya, algoritma divide and conquer berbasis Quadtree yang diimplementasikan mampu melakukan kompresi gambar secara efisien dan fleksibel. Kompleksitas dapat ditekan signifikan pada gambar seragam. Pemilihan parameter yang tepat menjadi kunci dalam menjaga kualitas visual sekaligus menekan ukuran file hasil kompresi.

Lampiran

Pranala GitHub

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	Implementasi persentase kompresi sebagai parameter tambahan	✓	
6	Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error		✓
7	Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar		✓
8	Program dan laporan dibuat (kelompok) sendiri	✓	