

LAPORAN TUGAS KECIL 2

IF2211 STRATEGI ALGORITMA

Kompresi Gambar Dengan Metode Quadtree



Dipersiapkan oleh:
Shanice Feodora Tjahjono – 13523097

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

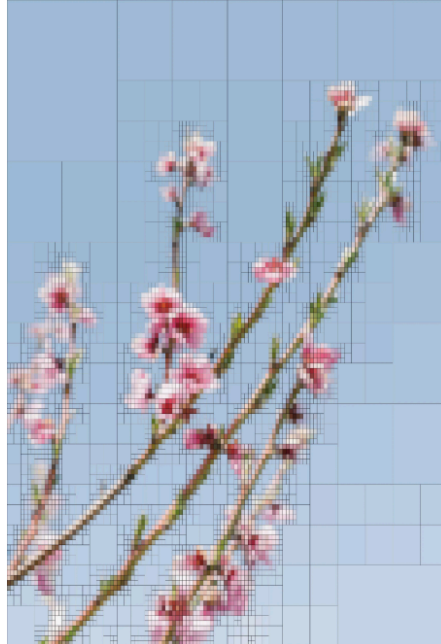
2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB I.....	2
Deskripsi Tugas.....	2
BAB II.....	6
Algoritma Divide And Conquer.....	6
2.1 Algoritma Divide and Conquer.....	6
2.2 Algoritma Divide and Conquer yang Digunakan.....	6
BAB III.....	8
Source Program.....	8
BAB IV.....	17
Pengujian Dan Analisis Implementasi.....	17
Lampiran.....	22
Daftar Pustaka.....	23

BAB I

Deskripsi Tugas



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.



Gambar 2. Proses Pembentukan Quadtree dalam Kompresi Gambar

(Sumber: https://miro.medium.com/v2/resize:fit:640/format:webp/1*LHD7PsbmbgNBFrYkxyG5dA.gif)

Ilustrasi kasus :

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.
- 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. 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.

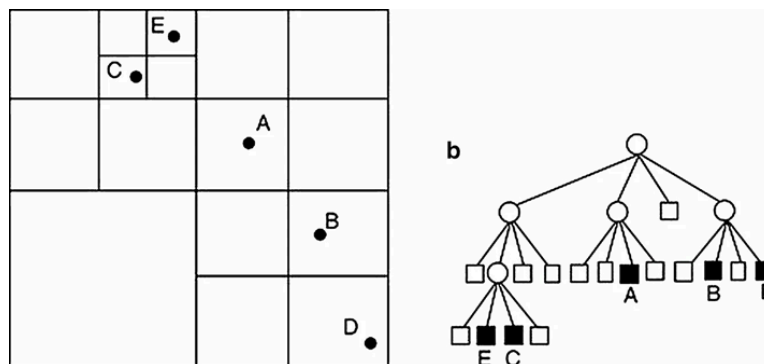
5. Rekursi dan Penghentian

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>)

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

2.1 Algoritma Divide and Conquer

Algoritma Divide and Conquer merupakan salah satu algoritma dasar yang bekerja dengan membagi masalah menjadi submasalah yang lebih kecil, menyelesaikan setiap sub-masalah secara rekursif, dan kemudian menggabungkan hasil-hasil tersebut untuk mendapatkan solusi dari masalah awal. Adapun beberapa prinsip dasar Divide and Conquer sebagai berikut,

1. **Divide (Membagi):** Memecah masalah utama menjadi beberapa sub-masalah yang serupa dengan ukuran yang lebih kecil.
2. **Conquer (Menaklukkan):** Menyelesaikan setiap sub-masalah secara rekursif. Jika submasalah cukup kecil, maka diselesaikan secara langsung.
3. **Combine (Menggabungkan):** Menggabungkan solusi dari submasalah-submasalah tersebut untuk membentuk solusi dari masalah utama.

Adapun beberapa keunggulan yang dimiliki algoritma ini. Pertama, algoritma Divide and Conquer umumnya memiliki kompleksitas waktu yang lebih baik dibandingkan algoritma lain seperti brute-force. Contohnya, Merge Sort memiliki kompleksitas $O(n \log n)$, yang lebih efisien daripada algoritma pengurutan sederhana seperti Bubble Sort yang memiliki kompleksitas $O(n^2)$. Kedua, pendekatan rekursif dalam Divide and Conquer seringkali menghasilkan algoritma yang lebih sederhana, memudahkan pemahaman dan implementasinya dalam pemecahan masalah. Ketiga, dikarenakan sub-masalah dapat diselesaikan secara independen, algoritma Divide and Conquer cocok untuk diimplementasikan dalam lingkungan komputasi paralel.

2.2 Algoritma Divide and Conquer yang Digunakan

Dalam program kompresi gambar ini, algoritma Divide and Conquer diterapkan melalui struktur Quadtree untuk membagi gambar menjadi blok-blok yang lebih kecil secara rekursif. Pendekatan ini memungkinkan pengolahan gambar yang efisien dengan mengurangi *redundancy* pada area yang seragam. Algoritma Divide and Conquer diimplementasikan melalui fungsi `buildQuadTree` dalam kelas `ImageCompressor`. Fungsi ini bekerja dengan cara membagi gambar menjadi empat sub-blok secara rekursif berdasarkan kriteria tertentu, yaitu error (misalnya variansi) pada blok tersebut dan ukuran minimum blok.

1. Divide (Membagi)

Gambar awal dianggap sebagai satu blok besar yang direpresentasikan oleh simpul akar (root) pada Quadtree. Fungsi `buildQuadTree` menghitung error pada blok tersebut menggunakan metode yang dipilih (misalnya variansi). Jika error melebihi threshold yang ditentukan dan ukuran blok lebih besar dari `minBlockSize`, blok tersebut dibagi menjadi empat sub-blok yang sama besar (kiri atas, kanan atas, kiri bawah, kanan bawah). Proses ini dilakukan dengan membuat empat simpul anak pada Quadtree untuk masing-masing sub-blok.

2. **Conquer (Menaklukkan)**
Setiap sub-blok diproses secara rekursif dengan memanggil fungsi `buildQuadTree` pada masing-masing simpul anak. Jika sebuah sub-blok memiliki error di bawah `threshold` atau ukurannya lebih kecil dari `minBlockSize`, proses pembagian dihentikan, dan sub-blok tersebut ditandai sebagai simpul daun (`leaf node`). Pada simpul daun, warna rata-rata RGB dihitung dan disimpan untuk digunakan dalam rekonstruksi gambar.
3. **Combine (Menggabungkan)**
Setelah semua sub-blok diproses, gambar direkonstruksi menggunakan fungsi `reconstructImage`. Simpul daun pada Quadtree merepresentasikan blok yang seragam, di mana semua pixel dalam blok tersebut diberi warna rata-rata RGB dari blok tersebut. Proses ini dilakukan dengan traversi pohon Quadtree dan mengisi nilai pixel pada gambar keluaran berdasarkan warna rata-rata pada simpul daun.

Adapun langkah-langkah algoritma Divide and Conquer yang digunakan dalam program ini,

1. Gambar dimuat dan direpresentasikan sebagai matriks piksel dengan nilai RGB. Simpul akar Quadtree dibuat untuk mewakili seluruh gambar dengan koordinat (0, 0) dan ukuran `imgWidth x imgHeight`.
2. Pada tahap 'divide' dalam algoritma Quadtree, setiap simpul diperiksa untuk menentukan apakah perlu dibagi lagi atau tidak. Pemeriksaan ini dilakukan dengan menghitung error pada blok tersebut, misalnya menggunakan nilai variansi, melalui fungsi `calculateError`. Jika nilai error melebihi ambang batas tertentu (`varianceThreshold`) dan ukuran blok masih lebih besar dari ukuran minimum yang diizinkan (`minBlockSize`), maka blok tersebut akan dibagi menjadi empat sub-blok. Keempat sub-blok tersebut memiliki koordinat sebagai berikut: kiri atas pada posisi (x, y, width/2, height/2), kanan atas pada (x, y + width/2, width/2, height/2), kiri bawah pada (x + width/2, y, width/2, height/2), dan kanan bawah pada (x + width/2, y + width/2, width/2, height/2). Untuk masing-masing sub-blok, akan dibuat simpul anak yang untuk masing-masing sub-blok, dan simpul induk akan ditandai sebagai simpul bukan-daun (`isLeaf = false`).
3. Ulangi langkah pembagian untuk setiap sub-blok secara rekursif hingga kondisi penghentian terpenuhi ($\text{error} \leq \text{varianceThreshold}$ atau $\text{ukuran blok} \leq \text{minBlockSize}$). Jika kondisi terpenuhi, simpul ditandai sebagai daun (`isLeaf = true`), dan warna rata-rata RGB dihitung menggunakan `calculateAverageColor`.
4. Setelah pohon Quadtree selesai dibangun, rekonstruksi gambar dilakukan dengan fungsi `reconstructImage`, di mana untuk setiap simpul daun, diisi semua piksel dalam blok tersebut dengan warna rata-rata yang telah dihitung. Proses ini dilakukan secara rekursif dengan traversi pohon Quadtree.
5. Gambar terkompresi disimpan ke file menggunakan `saveCompressedImage`. Jika opsi GIF diaktifkan, simpan frame-frame selama proses pembagian untuk membuat visualisasi GIF menggunakan `generateGIF`.

BAB III

Source Program

3.1 RGB.h

```
#ifndef RGB_H
#define RGB_H

struct RGB {
    unsigned char r, g, b;
    RGB(unsigned char r_ = 0, unsigned char g_ = 0, unsigned char b_ = 0) : r(r_), g(g_), b(b_) {}
};

#endif
```

Gambar 3.1. RGB.h

3.2 QuadTreeNode.h

```
#ifndef QUADREENODE_H
#define QUADREENODE_H

#include "RGB.h"

class QuadTreeNode {
public:
    int x, y;
    int width, height;
    RGB averageColor;
    QuadTreeNode* child[4];
    bool isLeaf;

    QuadTreeNode(int x_, int y_, int w_, int h_);
    ~QuadTreeNode();
};

#endif
```

Gambar 3.2. QuadTreeNode.h

3.3 QuadTreeNode.cpp

```
#include "QuadTreeNode.h"

QuadTreeNode::QuadTreeNode(int x_, int y_, int w_, int h_) : x(x_), y(y_), width(w_), height(h_), isLeaf(true) {
    for (int i = 0; i < 4; ++i) {
        child[i] = nullptr;
    }
}

QuadTreeNode::~~QuadTreeNode() {
    for (int i = 0; i < 4; ++i) {
        delete child[i];
    }
}
```

Gambar 3.3. QuadTreeNode.cpp

3.4 ImageCompressor.h

```
// ImageCompressor.h
#ifndef IMAGE_COMPRESSOR_H
#define IMAGE_COMPRESSOR_H

#include <string>
#include <vector>
#include "QuadTreeNode.h"
#include "RGB.h"

class ImageCompressor {
private:
    unsigned char* imageData;
    unsigned char* originalData;
    unsigned char* workingData;
    int imgWidth, imgHeight, channels;
    int minBlockSize;
    double varianceThreshold;
    int errorMethod;
    int treeDepth;
    int nodeCount;
    double targetCompression;
    QuadTreeNode* root;
    bool generateGif;

    RGB calculateAverageColor(int x, int y, int w, int h);
    double calculateError(int x, int y, int w, int h);
    void buildQuadTree(QuadTreeNode* node);
    void reconstructImage(unsigned char* outputData, QuadTreeNode* node);
    void adjustThresholdForTarget();
    int calculateDepth(QuadTreeNode* node);
    int calculateNodeCount(QuadTreeNode* node);
    void saveFrame();
};
```

Gambar 3.4.1 ImageCompressor.h

```
public:
    ImageCompressor(const std::string& inputPath, int minSize, double threshold, int method, double targetComp, bool genGif = true);
    ~ImageCompressor();
    void compress();
    void saveCompressedImage(const std::string& outputPath);
    void generateGIF(const std::string& gifOutputPath);
    int getOriginalSize();
    int getCompressedSize();
    double getCompressionPercentage();
    int getTreeDepth();
    int getNodeCount();
};

#endif
```

Gambar 3.4.2 ImageCompressor.h

3.5 ImageCompressor.cpp

```
#include "ImageCompressor.h"
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

int frameCount = 0;

ImageCompressor::ImageCompressor(const string& inputPath, int minSize, double threshold, int method, double targetComp, bool genGif) {
    minBlockSize = minSize;
    varianceThreshold = threshold;
    errorMethod = method;
    treeDepth = 0;
    nodeCount = 0;
    targetCompression = targetComp;
    generateGif = genGif;

    if (generateGif) {
#ifdef WIN32
        system("mkdir src\\frames >nul 2>&1");
    #else
        system("mkdir -p src/frames >/dev/null 2>&1");
    #endif
    }

    imageData = stbi_load(inputPath.c_str(), &imgWidth, &imgHeight, &channels, 3);
    if (!imageData) {
        cerr << "Error: Could not load image " << inputPath << endl;
        exit(1);
    }
}
```

Gambar 3.5.1. ImageCompressor.cpp

```
originalData = new unsigned char[imgWidth * imgHeight * 3];
memcpy(originalData, imageData, imgWidth * imgHeight * 3);

workingData = new unsigned char[imgWidth * imgHeight * 3];
memcpy(workingData, originalData, imgWidth * imgHeight * 3);

root = new QuadTreeNode(0, 0, imgWidth, imgHeight);
}

ImageCompressor::~ImageCompressor() {
    stbi_image_free(imageData);
    delete[] originalData;
    delete[] workingData;
    delete root;
}

RGB ImageCompressor::calculateAverageColor(int x, int y, int w, int h) {
    long long sumR = 0, sumG = 0, sumB = 0;
    int count = 0;

    for (int i = x; i < x + h && i < imgHeight; i++) {
        for (int j = y; j < y + w && j < imgWidth; j++) {
            int idx = (i * imgWidth + j) * 3;
            sumR += imageData[idx];
            sumG += imageData[idx + 1];
            sumB += imageData[idx + 2];
            count++;
        }
    }

    if (count == 0) return RGB(0, 0, 0);
    return RGB(sumR / count, sumG / count, sumB / count);
}
```

Gambar 3.5.2. ImageCompressor.cpp

```
double ImageCompressor::calculateError(int x, int y, int w, int h) {
    int count = 0;
    vector<RGB> pixels;

    for (int i = x; i < x + h && i < imgHeight; i++) {
        for (int j = y; j < y + w && j < imgWidth; j++) {
            int idx = (i * imgWidth + j) * 3;
            pixels.push_back(RGB(imageData[idx], imageData[idx + 1], imageData[idx + 2]));
            count++;
        }
    }

    if (count == 0) return 0;

    if (errorMethod == 1) { // variance using Welford's method
        double meanR = 0, meanG = 0, meanB = 0;
        double m2R = 0, m2G = 0, m2B = 0;

        for (int i = 0; i < pixels.size(); i++) {
            double r = pixels[i].r;
            double g = pixels[i].g;
            double b = pixels[i].b;

            double deltaR = r - meanR;
            double deltaG = g - meanG;
            double deltaB = b - meanB;
            meanR += deltaR / (i + 1);
            meanG += deltaG / (i + 1);
            meanB += deltaB / (i + 1);

            m2R += deltaR * (r - meanR);
            m2G += deltaG * (g - meanG);
            m2B += deltaB * (b - meanB);
        }
    }
```

Gambar 3.5.3. ImageCompressor.cpp

```
        double varR = m2R / count;
        double varG = m2G / count;
        double varB = m2B / count;
        return (varR + varG + varB) / 3.0;
    }
    else if (errorMethod == 2) { // mean absolute deviation (MAD)
        double meanR = 0, meanG = 0, meanB = 0;
        for (const auto& p : pixels) {
            meanR += p.r;
            meanG += p.g;
            meanB += p.b;
        }
        meanR /= count;
        meanG /= count;
        meanB /= count;

        double madR = 0, madG = 0, madB = 0;
        for (const auto& p : pixels) {
            madR += abs(p.r - meanR);
            madG += abs(p.g - meanG);
            madB += abs(p.b - meanB);
        }
        madR /= count;
        madG /= count;
        madB /= count;
        return (madR + madG + madB) / 3.0;
    }
}
```

Gambar 5.3.4. ImageCompressor.cpp

```

else if (errorMethod == 3) { // max pixel difference
    double maxDiffR = 0, maxDiffG = 0, maxDiffB = 0;
    for (size_t i = 0; i < pixels.size(); i++) {
        for (size_t j = i + 1; j < pixels.size(); j++) {
            maxDiffR = max(maxDiffR, (double)abs(pixels[i].r - pixels[j].r));
            maxDiffG = max(maxDiffG, (double)abs(pixels[i].g - pixels[j].g));
            maxDiffB = max(maxDiffB, (double)abs(pixels[i].b - pixels[j].b));
        }
    }
    return (maxDiffR + maxDiffG + maxDiffB) / 3.0;
}
else if (errorMethod == 4) { // entropy
    vector<int> histR(256, 0), histG(256, 0), histB(256, 0);
    for (const auto& p : pixels) {
        histR[p.r]++;
        histG[p.g]++;
        histB[p.b]++;
    }

    double entropyR = 0, entropyG = 0, entropyB = 0;
    for (int i = 0; i < 256; i++) {
        if (histR[i] > 0) {
            double p = (double)histR[i] / count;
            entropyR -= p * log2(p);
        }
        if (histG[i] > 0) {
            double p = (double)histG[i] / count;
            entropyG -= p * log2(p);
        }
        if (histB[i] > 0) {
            double p = (double)histB[i] / count;
            entropyB -= p * log2(p);
        }
    }
    return (entropyR + entropyG + entropyB) / 3.0;
}
return 0;
}

```

Gambar 5.3.5. ImageCompressor.cpp

```

void ImageCompressor::buildQuadTree(QuadTreeNode* node) {
    double error = calculateError(node->x, node->y, node->width, node->height);

    bool shouldSplit = error > varianceThreshold &&
        node->width > minBlockSize &&
        node->height > minBlockSize &&
        node->width / 2 >= minBlockSize &&
        node->height / 2 >= minBlockSize;

    if (shouldSplit) {
        if (generateGif) saveFrame();
        node->isLeaf = false;
        int newW = node->width / 2;
        int newH = node->height / 2;

        node->child[0] = new QuadTreeNode(node->x, node->y, newW, newH);
        node->child[1] = new QuadTreeNode(node->x, node->y + newW, newW, newH);
        node->child[2] = new QuadTreeNode(node->x + newH, node->y, newW, newH);
        node->child[3] = new QuadTreeNode(node->x + newH, node->y + newW, newW, newH);

        for (int i = 0; i < 4; i++) {
            buildQuadTree(node->child[i]);
        }
    } else {
        node->averageColor = calculateAverageColor(node->x, node->y, node->width, node->height);
        for (int i = node->x; i < node->x + node->width && i < imgWidth; i++) {
            for (int j = node->y; j < node->y + node->height && j < imgHeight; j++) {
                int idx = (i * imgWidth + j) * 3;
                workingData[idx] = node->averageColor.r;
                workingData[idx + 1] = node->averageColor.g;
                workingData[idx + 2] = node->averageColor.b;
            }
        }
        if (generateGif) saveFrame();
    }
}

```

Gambar 5.3.6. ImageCompressor.cpp

```
void ImageCompressor::reconstructImage(unsigned char* outputData, QuadTreeNode* node) {
    if (node->isLeaf) {
        for (int i = node->x; i < node->x + node->height; i++) {
            for (int j = node->y; j < node->y + node->width; j++) {
                int idx = (i * imgWidth + j) * 3;
                outputData[idx] = node->averageColor.r;
                outputData[idx + 1] = node->averageColor.g;
                outputData[idx + 2] = node->averageColor.b;
            }
        }
    } else {
        for (int i = 0; i < 4; i++) {
            if (node->child[i]) {
                reconstructImage(outputData, node->child[i]);
            }
        }
    }
}

void ImageCompressor::adjustThresholdForTarget() {
    if (targetCompression <= 0.0) {
        return;
    }

    double low = 0.0;
    double high = 100000.0;
    const double tolerance = 0.01;
    const int maxIterations = 50;

    if (targetCompression >= 0.99) {
        varianceThreshold = 0.0;
        delete root;
        root = new QuadTreeNode(0, 0, imgWidth, imgHeight);
        frameCount = 0;
        buildQuadTree(root);
        treeDepth = calculateDepth(root);
        nodeCount = calculateNodeCount(root);
        return;
    }
}
```

Gambar 3.5.7. ImageCompressor.cpp

```
for (int i = 0; i < maxIterations; i++) {
    delete root;
    root = new QuadTreeNode(0, 0, imgWidth, imgHeight);
    frameCount = 0;
    varianceThreshold = (low + high) / 2;
    buildQuadTree(root);
    treeDepth = calculateDepth(root);
    nodeCount = calculateNodeCount(root);

    double currentCompression = getCompressionPercentage() / 100.0;
    if (abs(currentCompression - targetCompression) <= tolerance) {
        break;
    }
    if (currentCompression > targetCompression) {
        high = varianceThreshold;
    } else {
        low = varianceThreshold;
    }
}

void ImageCompressor::compress() {
    if (generateGif) {
#ifdef _WIN32
        system("del /Q src\\frames\\frame_*.png >nul 2>&1");
    #else
        system("rm -f src/frames/frame_*.png >/dev/null 2>&1");
    #endif
    }
    frameCount = 0;

    buildQuadTree(root);
    treeDepth = calculateDepth(root);
    nodeCount = calculateNodeCount(root);
    adjustThresholdForTarget();
}
```

Gambar 3.5.8. ImageCompressor.cpp

```
void ImageCompressor::saveCompressedImage(const string& outputPath) {
    unsigned char* outputData = new unsigned char[imgWidth * imgHeight * 3];
    reconstructImage(outputData, root);
    stbi_write_png(outputPath.c_str(), imgWidth, imgHeight, 3, outputData, imgWidth * 3);
    delete[] outputData;
}

int ImageCompressor::calculateDepth(QuadTreeNode* node) {
    if (node->isLeaf) return 1;
    int maxChildDepth = 0;
    for (int i = 0; i < 4; i++) {
        if (node->child[i]) {
            int childDepth = calculateDepth(node->child[i]);
            maxChildDepth = std::max(maxChildDepth, childDepth);
        }
    }
    return 1 + maxChildDepth;
}

int ImageCompressor::calculateNodeCount(QuadTreeNode* node) {
    int count = 1;
    if (!node->isLeaf) {
        for (int i = 0; i < 4; i++) {
            if (node->child[i]) {
                count += calculateNodeCount(node->child[i]);
            }
        }
    }
    return count;
}
```

Gambar 3.5.9. ImageCompressor.cpp

```
int ImageCompressor::getOriginalSize() {
    return imgWidth * imgHeight * 3; // approximate
}

int ImageCompressor::getCompressedSize() {
    return nodeCount * (sizeof(int) * 4 + sizeof(RGB)); // approximate
}

double ImageCompressor::getCompressionPercentage() {
    return (1.0 - (double)getCompressedSize() / getOriginalSize()) * 100.0;
}

int ImageCompressor::getTreeDepth() {
    return treeDepth;
}

int ImageCompressor::getNodeCount() {
    return nodeCount;
}

void ImageCompressor::saveFrame() {
    char filename[100];
    sprintf(filename, "src/frames/frame_%03d.png", frameCount++);
    stbi_write_png(filename, imgWidth, imgHeight, 3, workingData, imgWidth * 3);
}

void ImageCompressor::generateGIF(const string& gifOutputPath) {
    if (!generateGif) {
        cout << "Skipping GIF generation..." << endl;
        return;
    }

    string command = "convert -delay 10 -loop 0 src/frames/frame_*.png " + gifOutputPath;
    int result = system(command.c_str());

    if (result != 0) {
        cerr << "Failed to create GIF." << endl;
    } else {
        cout << "GIF saved to: " << gifOutputPath << endl;
    }
}
```

Gambar 3.5.10. ImageCompressor.cpp

3.6 main.cpp

```
// main.cpp
#include <iostream>
#include <string>
#include <chrono>
#include "ImageCompressor.h"
using namespace std;

int main() {
    string inputPath, outputPath, gifPath;
    int errorMethod, minBlockSize;
    double threshold, targetCompression;
    char continueChoice, generateGifChoice;
    bool generateGif;

    cout << "WELCOME TO" << endl;
    cout << "QUANTREE \t COMPRESSION" << endl;
    cout << "QUANTREE \t COMPRESSION" << endl;
    cout << "QUANTREE \t COMPRESSION" << endl;
    cout << endl;

    do {
        #ifdef _WIN32
            system("cls");
        #else
            system("clear");
        #endif

        cout << "WELCOME TO" << endl;
        cout << "QUANTREE \t COMPRESSION" << endl;
        cout << "QUANTREE \t COMPRESSION" << endl;
        cout << "QUANTREE \t COMPRESSION" << endl;
        cout << endl;
    }
```

Gambar 3.6.1. main.cpp

```
    cout << "Enter input image path: ";
    getline(cin, inputPath);
    cout << "Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): ";
    cin >> errorMethod;
    cout << "Enter threshold: ";
    cin >> threshold;
    cout << "Enter minimum block size: ";
    cin >> minBlockSize;
    cout << "Enter target compression percentage (0 to disable): ";
    cin >> targetCompression;
    cout << "Do you want to generate a GIF? (y/n): ";
    cin >> generateGifChoice;
    generateGif = (generateGifChoice == 'y' || generateGifChoice == 'Y');
    cin.ignore();

    cout << "Enter output image path: ";
    getline(cin, outputPath);

    if (generateGif) {
        cout << "Enter output GIF path: ";
        getline(cin, gifPath);
    } else {
        gifPath = "";
    }

    cout << "Compressing image..." << endl;

    auto start = chrono::high_resolution_clock::now();

    ImageCompressor compressor(inputPath, minBlockSize, threshold, errorMethod, targetCompression, generateGif);
    compressor.compress();
    compressor.saveCompressedImage(outputPath);
    compressor.generateGIF(gifPath);

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
```

Gambar 3.6.2. main.cpp


```
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

cout << endl;
cout << "Execution time: " << duration.count() << " ms" << endl;
cout << "Previous image size: " << compressor.getOriginalSize() << " bytes" << endl;
cout << "Current image size after compression: " << compressor.getCompressedSize() << " bytes" << endl;
cout << "Compression percentage: " << compressor.getCompressionPercentage() << "%" << endl;
cout << "Depth of tree: " << compressor.getTreeDepth() << endl;
cout << "Number of nodes: " << compressor.getNodeCount() << endl;

cout << endl;
cout << "Do you want to compress another image? (y/n): ";
cin >> continueChoice;
cin.ignore();

} while (continueChoice == 'y' || continueChoice == 'Y');

cout << endl;
cout << "Byeeee" << endl;

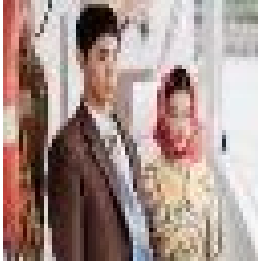
return 0;
}
```

Gambar 3.6.3. main.cpp

BAB IV

Pengujian Dan Analisis Implementasi

4.1 Pengujian



test1_64x64.png

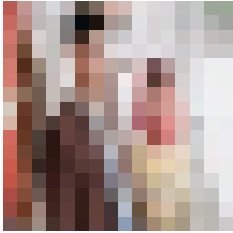
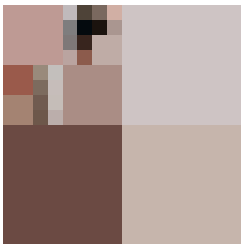
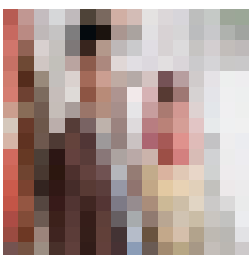
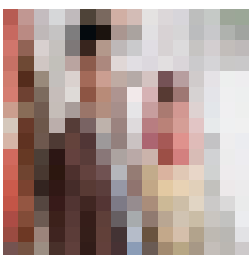



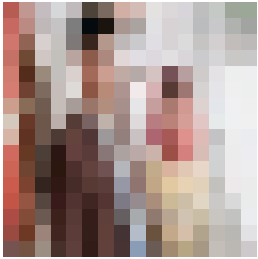

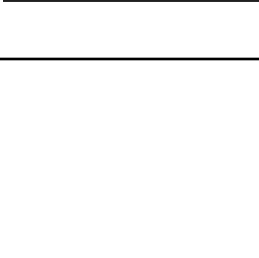
test2_128x128.png

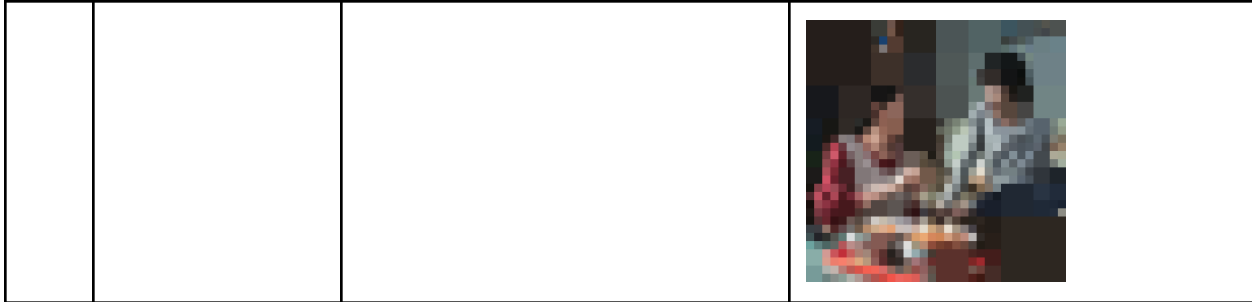


test3_256x256.png

No.	Test Case	Masukan (Input)	Keluaran (Output)
1.	Uji metode Variance pada gambar terkecil (64x64) tanpa persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_64x64.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 1 Enter threshold: 1000 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0 Enter output image path: test/output1.png Enter output GIF path: test/gif1a.gif </pre>	<pre> Compressing image... GIF saved to: test/gif1a.gif Execution time: 9862 ms Previous image size: 12288 bytes Current image size after compression: 4123 bytes Compression percentage: 66.4469% Depth of tree: 5 Number of nodes: 217 Do you want to compress another image? (y/n): </pre> 
2.	Uji metode Variance pada gambar sedang (128x128) dengan persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_64x64.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 1 Enter threshold: 1000 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0.1 Enter output image path: test/output2.png Enter output GIF path: test/gif2.gif </pre>	<pre> Compressing image... GIF saved to: test/gif2.gif Execution time: 102113 ms Previous image size: 12288 bytes Current image size after compression: 6479 bytes Compression percentage: 47.2738% Depth of tree: 5 Number of nodes: 341 Do you want to compress another image? (y/n): </pre> 

			
3.	Uji metode MAD pada gambar terkecil (64x64) tanpa persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_64x64.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 2 Enter threshold: 20.0 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0 Enter output image path: test/output3.png Enter output GIF path: test/gif3.gif </pre>	<pre> Compressing image... GIF saved to: test/gif3.gif Execution time: 3133 ms Previous image size: 12288 bytes Current image size after compression: 703 bytes Compression percentage: 94.279% Depth of tree: 5 Number of nodes: 37 Do you want to compress another image? (y/n): </pre> 
4.	Uji metode MAD pada gambar kecil (64x64) dengan persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_64x64.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 2 Enter threshold: 20.0 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0.15 Enter output image path: test/output4.png Enter output GIF path: test/gif4.gif </pre>	<pre> Compressing image... GIF saved to: test/gif4.gif Execution time: 98979 ms Previous image size: 12288 bytes Current image size after compression: 6479 bytes Compression percentage: 47.2738% Depth of tree: 5 Number of nodes: 341 Do you want to compress another image? (y/n): </pre> 
5.	Uji metode Max Pixel Difference pada gambar berukuran sedang (128x128) tanpa persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test2_128x128.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 3 Enter threshold: 20.0 Enter minimum block size: 8 Enter target compression percentage (0 to disable): 0 Enter output image path: test/output5.png Enter output GIF path: test/gif5.gif </pre>	<pre> Compressing image... GIF saved to: test/gif5.gif Execution time: 13639 ms Previous image size: 49152 bytes Current image size after compression: 2527 bytes Compression percentage: 94.8588% Depth of tree: 5 Number of nodes: 133 Do you want to compress another image? (y/n): </pre> 

			
6.	Uji metode Max Pixel Difference pada gambar berukuran kecil (64x64) dengan persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_64x64.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 3 Enter threshold: 9000 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0.2 Enter output image path: test/output6.png Enter output GIF path: test/gif6.gif </pre>	<pre> Compressing image... GIF saved to: test/gif6.gif Execution time: 114038 ms Previous image size: 12288 bytes Current image size after compression: 6479 bytes Compression percentage: 47.2738% Depth of tree: 5 Number of nodes: 341 Do you want to compress another image? (y/n): </pre> 
7.	Uji metode Entropy pada gambar berukuran sedang (128x128) tanpa persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_128x128.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 4 Enter threshold: 4 Enter minimum block size: 4 Enter target compression percentage (0 to disable): 0 Enter output image path: test/output7b.png Enter output GIF path: test/gif7b.gif </pre>	<pre> Compressing image... GIF saved to: test/gif7b.gif Execution time: 64113 ms Previous image size: 49152 bytes Current image size after compression: 18943 bytes Compression percentage: 61.4604% Depth of tree: 6 Number of nodes: 997 Do you want to compress another image? (y/n): </pre> 
8.	Uji metode Variance pada gambar berukuran besar (256x256) tanpa persentase kompresi target.	<pre> WELCOME TO QUADTREE COMPRESSION Enter input image path: test/test1_256x256.png Enter error measurement method (1: Variance, 2: MAD, 3: Max Pixel Difference, 4: Entropy): 1 Enter threshold: 8000 Enter minimum block size: 8 Enter target compression percentage (0 to disable): 0 Enter output image path: test/output8.png Enter output GIF path: test/gif8.gif </pre>	<pre> Compressing image... GIF saved to: test/gif8.gif Execution time: 80565 ms Previous image size: 196608 bytes Current image size after compression: 13395 bytes Compression percentage: 93.187% Depth of tree: 6 Number of nodes: 705 Do you want to compress another image? (y/n): </pre> 



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

Dari segi kompleksitas algoritma, fungsi utama `buildQuadTree` yang mengimplementasikan Divide and Conquer memiliki kompleksitas waktu yang bergantung pada ukuran gambar dan parameter `minBlockSize`. Untuk gambar berukuran $N \times N$, perhitungan error pada setiap blok membutuhkan waktu $O(w \times h)$, di mana w dan h adalah lebar dan tinggi blok. Dalam kasus terburuk, gambar dibagi hingga ukuran blok mencapai `minBlockSize`, sehingga kedalaman maksimum pohon Quadtree adalah $\log_2(N/\text{minBlockSize})$. Pada setiap level pembagian, total pixel yang diproses adalah $O(N^2)$, sehingga kompleksitas waktu untuk `buildQuadTree` adalah $O(N^2 \log_2(N/\text{minBlockSize}))$. Fungsi `reconstructImage` memiliki kompleksitas $O(N^2)$ karena setiap pixel diproses tepat sekali. Namun, jika fitur GIF diaktifkan, setiap perubahan pada gambar disimpan sebagai frame, yang membutuhkan waktu $O(N^2)$ per frame. Jumlah frame maksimum adalah $O(N^2/\text{minBlockSize}^2)$ (jumlah daun pada pohon), sehingga total waktu untuk pembuatan GIF menjadi $O(N^4/\text{minBlockSize}^2)$, yang sangat tidak efisien untuk gambar besar.

Fungsi `adjustThresholdForTarget`, yang merupakan fitur bonus untuk mencapai target persentase kompresi, menggunakan pendekatan binary search dengan maksimal 50 iterasi. Setiap iterasi membangun ulang pohon Quadtree, sehingga kompleksitasnya adalah $O(50 \times N^2 \log_2(N/\text{minBlockSize}))$, atau secara sederhana tetap $O(N^2 \log_2(N/\text{minBlockSize}))$. Secara keseluruhan, kompleksitas algoritma tanpa GIF adalah $O(N^2 \log_2(N/\text{minBlockSize}))$, yang cukup efisien untuk gambar kecil hingga menengah. Namun, dengan GIF, kompleksitas didominasi oleh penyimpanan frame, menjadi $O(N^4/\text{minBlockSize}^2)$, yang membuatnya tidak praktis untuk gambar beresolusi tinggi.

Berdasarkan analisis ini, algoritma Divide and Conquer dengan metode Quadtree efektif untuk kompresi gambar pada area yang seragam, tetapi memiliki keterbatasan dalam hal efisiensi waktu ketika fitur GIF diaktifkan. Untuk gambar kecil seperti 128×128 pixel, performa tanpa GIF sangat baik, tetapi untuk gambar yang lebih besar atau dengan `minBlockSize` yang kecil, waktu eksekusi dapat meningkat signifikan, terutama jika pembuatan GIF diaktifkan. Oleh karena itu, penggunaan fitur GIF sebaiknya dibatasi pada keperluan visualisasi saja, dan optimasi lebih lanjut diperlukan untuk mengurangi jumlah frame yang disimpan.

4.3 Implementasi Bonus

4.3.1. Target Persentase Kompresi

Algoritma `adjustThresholdForTarget()` berfungsi untuk menyesuaikan nilai `varianceThreshold` secara dinamis agar hasil kompresi mendekati target persentase kompresi yang ditentukan oleh pengguna.

Nilai target ini bersifat *floating point*, misalnya 1.0 berarti 100% gambar dikompresi, sedangkan nilai 0.0 akan menonaktifkan mode ini. Mekanisme penyesuaian threshold ini menggunakan pendekatan *binary search* untuk mencari nilai `varianceThreshold` yang tepat, sehingga hasil kompresi (dalam hal jumlah node quadtree yang terbentuk) sesuai dengan persentase kompresi yang diharapkan.

Prosesnya dimulai dengan rentang nilai threshold awal antara 0.0 hingga 100000.0, kemudian secara iteratif akan dicoba nilai tengah dari rentang ini. Untuk setiap iterasi, pohon quadtree akan dibangun ulang dari awal menggunakan threshold yang sedang diuji. Setelah itu, algoritma menghitung persentase kompresi saat ini menggunakan fungsi `getCompressionPercentage()`, dan membandingkannya dengan target. Jika perbedaannya cukup kecil (kurang dari 0.01 atau 1%), proses berhenti. Namun jika belum sesuai, rentang pencarian akan disesuaikan: jika kompresi terlalu tinggi, maka batas atas akan diturunkan, sedangkan jika kompresi terlalu rendah, batas bawah dinaikkan. Proses ini berulang hingga maksimal 50 iterasi atau sampai didapatkan threshold yang mendekati target.

Meskipun pendekatan ini fleksibel dan memungkinkan pengguna menentukan tingkat efisiensi yang diinginkan, prosesnya cukup berat secara komputasi karena membutuhkan rekonstruksi ulang pohon quadtree berulang kali. Hal ini memperlambat performa, terutama jika gambar yang dikompresi beresolusi besar.

4.3.2. GIF Proses Kompresi

GIF yang dihasilkan dalam program ini merepresentasikan proses kompresi gambar secara bertahap menggunakan algoritma QuadTree. Selama proses ini berlangsung, data visual dari setiap tahap perubahan disimpan dalam buffer `workingData`, dan setiap versi buffer tersebut diekspor menjadi gambar yang digunakan sebagai frame untuk membentuk GIF.

Alur penyimpanan GIF sendiri dilakukan dengan cara memanggil fungsi `saveFrame()` setiap kali terjadi perubahan pada tampilan gambar. Setiap frame yang dihasilkan merepresentasikan hasil kompresi pada titik tertentu dalam algoritma, sehingga GIF akhir menampilkan animasi proses kompresi dari gambar asli menuju gambar hasil kompresi akhir. Dengan menyusun seluruh frame ini secara berurutan menggunakan ImageMagick, pengguna dapat melihat bagaimana setiap bagian gambar dikompresi secara bertahap oleh algoritma QuadTree.

Namun, metode ini memiliki kekurangan dari segi efisiensi, terutama saat digunakan untuk gambar beresolusi tinggi atau ketika kompresi dilakukan dengan target persentase tertentu. Karena frame disimpan satu per satu setiap kali terjadi perubahan, jumlah frame yang dihasilkan sangat banyak—bahkan bisa mencapai ribuan untuk proses kompresi yang kompleks. Hal ini menyebabkan proses penyimpanan menjadi sangat lambat dan memakan banyak waktu serta ruang penyimpanan. Oleh karena itu, meskipun metode pembuatan GIF ini efektif dalam memvisualisasikan proses kompresi, pendekatan ini tidak optimal untuk penggunaan dalam skala besar.

Lampiran

Link Repository GitHub

https://github.com/feodorashanice/Tucil2_13523097.git

Tabel Penyelesaian Tugas Kecil

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	✓	

Daftar Pustaka

- J. Salonen, “Deriving Welford’s method for computing variance,” *jonisalonen.com*, 2013.
[Online]. Available:
<https://jonisalonen.com/2013/deriving-welfords-method-for-computing-variance/>.
[Accessed: Apr. 11, 2025].
- R. Munir, “Algoritma Divide and Conquer (Bagian 1),” *IF2211 Strategi Algoritma*, 2025.
[Online]. Available:
[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/2024-2025/07-Algoritma-Divide-and-Conquer-(2025)-Bagian1.pdf). [Accessed: Apr. 11, 2025].
- R. Munir, “Algoritma Divide and Conquer (Bagian 2),” *IF2211 Strategi Algoritma*, 2025.
[Online]. Available:
[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/2024-2025/08-Algoritma-Divide-and-Conquer-(2025)-Bagian2.pdf). [Accessed: Apr. 11, 2025].
- R. Munir, “Algoritma Divide and Conquer (Bagian 3),” *IF2211 Strategi Algoritma*, 2025.
[Online]. Available:
[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/2024-2025/09-Algoritma-Divide-and-Conquer-(2025)-Bagian3.pdf). [Accessed: Apr. 11, 2025].
- R. Munir, “Algoritma Divide and Conquer (Bagian 4),” *IF2211 Strategi Algoritma*, 2025.
[Online]. Available:
[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/2024-2025/10-Algoritma-Divide-and-Conquer-(2025)-Bagian4.pdf). [Accessed: Apr. 11, 2025].