

**LAPORAN TUGAS KECIL 2**  
**IF2211 STRATEGI ALGORITMA 2024/2025**

**KOMPRESI GAMBAR DENGAN METODE QUADTREE**

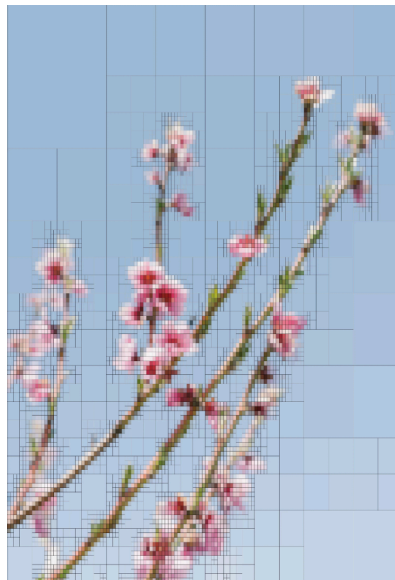


Oleh:  
Muhammad Edo Raduputu Aprima  
13523096

PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
JL. GANESA 10, BANDUNG 40132  
2025

## I. DESKRIPSI MASALAH

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.



Gambar 1. Quadtree dalam Kompresi Gambar

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

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.

Untuk dapat menkompresi gambar, kita dapat memanfaatkan algoritma Divide and Conquer. Idennya adalah menghitung nilai error pada seluruh blok gambar, artinya kita

menentukan seberapa besar perbedaan dalam satu blok gambar. Jika error blok melebihi threshold, maka blok akan dibagi menjadi 4 bagian yang lebih kecil. Kita dapat menggunakan 5 metode untuk menghitung nilai error:

- Variance

Metode	Formula
<a href="#">Variance</a>	$\sigma_c^2 = \frac{1}{N} \sum_{i=1}^N (P_{i,c} - \mu_c)^2$
	$\sigma_{RGB}^2 = \frac{\sigma_R^2 + \sigma_G^2 + \sigma_B^2}{3}$
	$\sigma_c^2$ = Variansi tiap kanal warna c (R, G, B) dalam satu blok
	$P_{i,c}$ = Nilai piksel pada posisi $i$ untuk kanal warna c
	$\mu_c$ = Nilai rata-rata tiap piksel dalam satu blok
	N = Banyaknya piksel dalam satu blok

- Mean Absolute Deviation (MAD)

Mean Absolute Deviation (MAD)	$MAD_c = \frac{1}{N} \sum_{i=1}^N  P_{i,c} - \mu_c $
	$MAD_{RGB} = \frac{MAD_R + MAD_G + MAD_B}{3}$
	$MAD_c$ = 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
	$\mu_c$ = Nilai rata-rata tiap piksel dalam satu blok
	N = Banyaknya piksel dalam satu blok

- Max Pixel Difference

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

<a href="#">Entropy</a>	$H_c = - \sum_{i=1}^N P_c(i) \log_2(P_c(i))$
	$H_{RGB} = \frac{H_R + H_G + H_B}{3}$
	$H_c$ = 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)

- Structural Similarity Index (SSIM)

<b>[Bonus]</b>  <a href="#">Structural Similarity Index (SSIM)</a>  <a href="#">(Referensi tambahan)</a>	$SSIM_c(x, y) = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(2\sigma_{xy,c} + C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + \sigma_{y,c}^2 + C_2)}$
	$SSIM_{RGB} = w_R \cdot SSIM_R + w_G \cdot SSIM_G + w_B \cdot SSIM_B$
	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.

## II. IMPLEMENTASI ALGORITMA

Algoritma kompresi gambar menggunakan QuadTree memanfaatkan strategi divide and conquer untuk merepresentasikan area gambar yang homogen secara efisien. Pendekatan ini secara rekursif membagi gambar menjadi empat bagian (kuadran) yang sama besar hingga suatu kriteria terpenuhi. Secara umum, algoritma dapat kita pecah menjadi 4 tahap, yaitu

- Tahap inisialisasi  
 Pada tahap ini algoritma dimulai dengan mempertimbangkan seluruh area gambar sebagai satu blok (node akar QuadTree). Selain itu, user juga menentukan 2 parameter yaitu threshold dan minBlockSize, serta juga memilih metode error yang akan digunakan.
- Tahap Divide  
 Pada tahap ini, jika kondisi blok gambar saat ini perlu dibagi maka blok akan dipecah menjadi 4 kuadran: Kuadran Kiri-Atas (North-West), Kanan-Atas (North-East), Kiri-Bawah (South-West), dan Kanan-Bawah (South-East).
- Tahap Conquer

Pada tahap ini akan dilihat apakah suatu blok perlu dibagi lebih lanjut atau sudah cukup homogen untuk dijadikan *leaf node* (daun). Jika ukuran blok kurang dari sama dengan `minBlockSize`, maka blok akan menjadi *leaf node*. Jika blok menjadi *leaf node*, Hitung warna rata-rata (average color) dari semua piksel di dalam blok tersebut. Simpul daun ini akan menyimpan informasi lokasi, dimensi, dan warna rata-rata tersebut. Ini adalah solusi untuk subproblem terkecil. Jika tidak, lanjutkan ke tahap *Divide* untuk blok ini, yaitu membaginya menjadi empat kuadran, dan kemudian secara rekursif terapkan algoritma lagi pada keempat kuadran tersebut.

- Tahap Combine

Dalam konteks kompresi QuadTree ini, tahap "Combine" tidak secara eksplisit menggabungkan hasil warna dari sub-blok. Sebaliknya, penggabungan terjadi secara struktural. Simpul internal (yang bukan leaf) secara implisit merepresentasikan gabungan dari keempat anaknya dalam struktur pohon. Hasil akhir dari keseluruhan proses *divide and conquer* adalah struktur data QuadTree itu sendiri, di mana setiap bagian gambar direpresentasikan oleh simpul daun (dengan warna rata-rata) atau simpul internal yang menunjuk ke pembagian lebih lanjut. Gambar terkompresi kemudian dapat direkonstruksi dengan menelusuri pohon ini dan mengisi area sesuai dengan warna pada simpul daun.

Algoritma ini berhenti ketika semua cabang pohon mencapai kondisi *base case* (ukuran minimum atau tingkat homogenitas sesuai threshold).

### III. SOURCE PROGRAM

```
// main.cpp

#include <iostream>
#include <chrono>
#include <iomanip>
#include <string>
#include "Image.hpp"
#include "QuadTree.hpp"
#include <filesystem>

using namespace std;
using namespace std::chrono;
namespace fs = filesystem;

void printHeader() {
    cout << "\n";
    cout <<
```

```

"-----
-----\n";
    cout << "|    Q U A D T R E E    I M A G E    C O M P R E S S
I O N    T O O L    |\n";
    cout <<
"-----
-----\n";
    cout << "\n";
}

bool validateImage(const string& path) {
    if (!fs::exists(path)) {
        cerr << "Error: File does not exist\n";
        return false;
    }

    ifstream file(path, ios::binary);
    if (!file) {
        cerr << "Error: Cannot open file\n";
        return false;
    }
    char header[8];
    file.read(header, 8);
    return (header[0] == -1 && header[1] == -40) || (header[0]
== -119 && header[1] == 'P' && header[2] == 'N' && header[3] ==
'G');
}

bool validateThreshold(int method, double threshold) {
    const pair<double, double> ranges[] = {
        {0, 65025},
        {0, 255},
        {0, 255},
        {0, 8},
        {0, 1}
    };
    return threshold >= ranges[method-1].first && threshold <=
ranges[method-1].second;
}

int main() {
    printHeader();

    string inputPath;
    cout << "Input image path:\n>> ";

```

```

cin >> inputPath;

if (!validateImage(inputPath)) return 1;

Image img;
if (!img.loadImg(inputPath)) {
    cerr << "Error: Failed to load image\n";
    return 1;
}

int method;
cout << "\nSelect Error measurement method (1-4):" << endl;
cout << "1. Variance (0-65025)" << endl;
cout << "2. MAD (0-255)" << endl;
cout << "3. Max Pixel Difference (0-255)" << endl;
cout << "4. Entropy (0-8)" << endl;
cout << "5. SSIM (0-1)" << endl;
cout << ">> ";
cin >> method;
if (method < 1 || method > 5){
    cerr << "Error: Invalid method selected\n";
    return 1;
}

double threshold;
cout << "\nEnter threshold:\n>> ";
cin >> threshold;
if (!validateThreshold(method, threshold)) {
    cerr << "Error: Invalid threshold for selected
method\n";
    return 1;
}

int minBlock;
cout << "\nEnter minimum block size (pixels):\n>> ";
cin >> minBlock;
if (minBlock < 0){
    cerr << "Error: Invalid minimum block size\n";
    return 1;
}

string outputPath;
cout << "\nOutput image path:\n>> ";
cin >> outputPath;
fs::create_directories(fs::path(outputPath).parent_path());

```

```

string gifPath;
cout << "\nOutput GIF path (leave empty to skip):\n>> ";
cin.ignore();
getline(cin, gifPath);

auto start = high_resolution_clock::now();

QuadTree quadtree(threshold, minBlock, method);

quadtree.compress(img.getPixels());
auto compressedImg = quadtree.reconstructImage();
img.saveImg(compressedImg, outputPath);

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

if (!gifPath.empty()) {
    cout << "\nSaving compression process GIF..." << endl;
    try {
        fs::create_directories(fs::path(gifPath).parent_path());
    } catch (const fs::filesystem_error& e) {
        cerr << "Warning: Could not create directories for
GIF: " << e.what() << endl;
    }
    int gifDelay = 400;
    if (!quadtree.saveGIF(gifPath, gifDelay)) {
        cerr << "Error: Failed to save compression GIF to "
<< gifPath << endl;
    }
}

size_t originalSize = img.getFileSize(inputPath);
size_t compressedSize = img.getFileSize(outputPath);
double ratio = 100.0 * (1.0 -
(double)compressedSize/originalSize);

cout << fixed << setprecision(2);
cout << "\n";
cout <<
"-----\n";
    cout << "|    C O M P R E S S I O N    R E S U L T S
|\n";
    cout <<

```



```

"-----\n";
    cout << "\n";
    cout << "Original size      : " << originalSize << " bytes"
<< endl;
    cout << "Compressed size    : " << compressedSize << "
bytes" << endl;
    cout << "Compression ratio  : " << ratio << "%" << endl;
    cout << "Error threshold    : " << threshold << endl;
    cout << "Min Block size      : " << minBlock << " pixels" <<
endl;
    cout << "Processing time    : " << duration.count() << "
ms" << endl;
    cout << "Quadtree nodes     : " << quadtree.countNodes() <<
endl;
    cout << "Leaf nodes        : " << quadtree.countLeaves()
<< endl;
    cout << "Tree depth        : " << quadtree.getDepth() <<
endl << endl;

    return 0;
}

```

```
// Image.hpp
```

```

#ifndef IMAGE_HPP
#define IMAGE_HPP

```

```

#include <vector>
#include <string>
#include <fstream>
#include "stb_image.h"
#include "stb_image_write.h"

```

```

struct RGB {
    uint8_t r, g, b;
    RGB() : r(0), g(0), b(0) {}
    RGB(uint8_t r, uint8_t g, uint8_t b) : r(r), g(g), b(b) {}

    bool operator==(const RGB& other) const {
        return r == other.r && g == other.g && b == other.b;
    }

    bool operator!=(const RGB& other) const {
        return !(*this == other);
    }
}

```

```

};

class Image {
private:
    std::vector<std::vector<RGB>> pixels;
    int width = 0;
    int height = 0;

public:
    bool loadImg(const std::string& filename) {
        int channels;
        unsigned char* data = stbi_load(filename.c_str(),
&width, &height, &channels, 3);
        if (!data) return false;

        pixels.resize(height, std::vector<RGB>(width));
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int idx = (y * width + x) * 3;
                pixels[y][x] = RGB(data[idx], data[idx+1],
data[idx+2]);
            }
        }
        stbi_image_free(data);
        return true;
    }

    bool saveImg(const std::vector<std::vector<RGB>>& imgData,
const std::string& filename) {
        std::vector<uint8_t> data(width * height * 3);
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int idx = (y * width + x) * 3;
                data[idx] = imgData[y][x].r;
                data[idx+1] = imgData[y][x].g;
                data[idx+2] = imgData[y][x].b;
            }
        }
        return stbi_write_jpg(filename.c_str(), width, height,
3, data.data(), 90);
    }

    std::vector<std::vector<RGB>> getPixels() const { return
pixels; }
    int getWidth() const { return width; }

```

```

        int getHeight() const { return height; }

        size_t getFileSize(const std::string& filename) {
            std::ifstream file(filename, std::ios::binary |
std::ios::ate);
            return file.tellg();
        }
    };
#endif

```

```

// Quadtree.hpp

```

```

#ifndef QUADTREE_HPP
#define QUADTREE_HPP

```

```

#include <memory>
#include <vector>
#include <cmath>
#include <map>
#include <string>
#include "QuadNode.hpp"
#include "Image.hpp"

```

```

class QuadTree {
private:
    std::shared_ptr<QuadNode> root;
    std::vector<std::vector<RGB>> pixels;
    double threshold;
    int minBlockSize;
    int errorMethod;

    RGB calculateAverage(int x, int y, int width, int height);
    double calculateError(int x, int y, int width, int height);
    std::shared_ptr<QuadNode> buildTree(int x, int y, int
width, int height);

    void findNodesPerLevel(std::shared_ptr<QuadNode> node, int
level, std::map<int, std::vector<std::shared_ptr<QuadNode>>>&
nodesMap, int& maxLevelFound);
    void drawNodeArea(std::vector<std::vector<RGB>>& canvas,
std::shared_ptr<QuadNode> node);

public:
    QuadTree(double threshold, int minSize, int method):

```

```

threshold(threshold), minBlockSize(minSize),
errorMethod(method) {}

    void compress(const std::vector<std::vector<RGB>>&
imagePixels);

    bool saveGIF(const std::string& filename, int delay = 100,
bool dither = false);

    std::vector<std::vector<RGB>> reconstructImage();

    int countNodes() const;
    int countLeaves() const;
    int getDepth() const;
};

#endif

```

```

// QuadTree.cpp

```

```

#include "QuadTree.hpp"
#include <iostream>
#include <algorithm>
#include <cmath>
#include <map>
#include <unordered_map>
#include <functional>
#include <vector>
#include "gif.h"

using namespace std;

RGB QuadTree::calculateAverage(int x, int y, int width, int
height) {
    double r = 0, g = 0, b = 0;
    int count = 0;

    for (int i = y; i < y + height && i < pixels.size(); i++) {
        for (int j = x; j < x + width && j < pixels[i].size();
j++) {
            r += pixels[i][j].r;
            g += pixels[i][j].g;
            b += pixels[i][j].b;
            count++;
        }
    }
}

```

```

    }

    if (count == 0) return RGB();
    return RGB(static_cast<uint8_t>(r/count),
               static_cast<uint8_t>(g/count),
               static_cast<uint8_t>(b/count));
}

double QuadTree::calculateError(int x, int y, int width, int
height) {
    RGB avg = calculateAverage(x, y, width, height);
    int count = 0;
    double error = 0;

    switch(errorMethod) {
        case 1: {
            for (int i = y; i < y + height && i <
pixels.size(); i++) {
                for (int j = x; j < x + width && j <
pixels[i].size(); j++) {
                    error += pow(pixels[i][j].r - avg.r, 2) +
pow(pixels[i][j].g - avg.g, 2) + pow(pixels[i][j].b - avg.b,
2);
                    count++;
                }
            }
            return count > 0 ? error / (count * 3) : 0;
        }

        case 2: {
            for (int i = y; i < y + height && i <
pixels.size(); i++) {
                for (int j = x; j < x + width && j <
pixels[i].size(); j++) {
                    error += abs(pixels[i][j].r - avg.r) +
abs(pixels[i][j].g - avg.g) + abs(pixels[i][j].b - avg.b);
                    count++;
                }
            }
            return count > 0 ? error / (count * 3) : 0;
        }

        case 3: {
            uint8_t minR = 255, maxR = 0, minG = 255, maxG = 0,
minB = 255, maxB = 0;

```

```

        bool pixelFound = false;
        for (int i = y; i < y + height && i <
pixels.size(); i++) {
            for (int j = x; j < x + width && j <
pixels[i].size(); j++) {
                minR = min(minR, pixels[i][j].r);
                maxR = max(maxR, pixels[i][j].r);
                minG = min(minG, pixels[i][j].g);
                maxG = max(maxG, pixels[i][j].g);
                minB = min(minB, pixels[i][j].b);
                maxB = max(maxB, pixels[i][j].b);
                pixelFound = true;
            }
        }
        if (!pixelFound) return 0.0;
        return ((maxR - minR) + (maxG - minG) + (maxB -
minB)) / 3.0;
    }

    case 4: {
        unordered_map<uint8_t, int> histR, histG, histB;
        int totalPixels = 0;

        for (int i = y; i < y + height; i++) {
            for (int j = x; j < x + width; j++) {
                histR[pixels[i][j].r]++;
                histG[pixels[i][j].g]++;
                histB[pixels[i][j].b]++;
                totalPixels++;
            }
        }

        auto calcEntropy = [totalPixels](const auto& hist)
{
            double entropy = 0;
            for (const auto& pair : hist) {
                if (pair.second > 0) {
                    double p = pair.second /
static_cast<double>(totalPixels);
                    entropy -= p * log2(p);
                }
            }
            return entropy;
        };
    };

```

```

        return (calcEntropy(histR) + calcEntropy(histG) +
calcEntropy(histB)) / 3.0;
    }

    case 5: {
        avg = calculateAverage(x, y, width, height);
        long long totalPixels = 0;
        double meanOrigR = 0, meanOrigG = 0, meanOrigB = 0;
        double varOrigR = 0, varOrigG = 0, varOrigB = 0;

        for (int i = y; i < y + height && i <
pixels.size(); ++i) {
            for (int j = x; j < x + width && j <
pixels[i].size(); ++j) {
                meanOrigR += pixels[i][j].r;
                meanOrigG += pixels[i][j].g;
                meanOrigB += pixels[i][j].b;
                totalPixels++;
            }
        }

        if (totalPixels == 0) return 0.0;

        meanOrigR /= totalPixels;
        meanOrigG /= totalPixels;
        meanOrigB /= totalPixels;

        for (int i = y; i < y + height && i <
pixels.size(); ++i) {
            for (int j = x; j < x + width && j <
pixels[i].size(); ++j){
                double diffR = pixels[i][j].r - meanOrigR;
                double diffG = pixels[i][j].g - meanOrigG;
                double diffB = pixels[i][j].b - meanOrigB;
                varOrigR += diffR * diffR;
                varOrigG += diffG * diffG;
                varOrigB += diffB * diffB;
            }
        }

        if (totalPixels > 1) {
            varOrigR /= (totalPixels - 1);
            varOrigG /= (totalPixels - 1);
            varOrigB /= (totalPixels - 1);
        } else {

```

```

        varOrigR = varOrigG = varOrigB = 0;
    }

    double meanCompR = avg.r;
    double meanCompG = avg.g;
    double meanCompB = avg.b;
    const double varCompR = 0.0, varCompG = 0.0,
varCompB = 0.0;
    const double covR = 0.0, covG = 0.0, covB = 0.0;

    const double K1 = 0.01;
    const double K2 = 0.03;
    const double L = 255.0;
    const double C1 = (K1 * L) * (K1 * L);
    const double C2 = (K2 * L) * (K2 * L);

    double ssimR = ((2.0 * meanOrigR * meanCompR + C1)
* C2) /
                ((meanOrigR * meanOrigR + meanCompR
* meanCompR + C1) * (varOrigR + varCompR + C2));
    double ssimG = ((2.0 * meanOrigG * meanCompG + C1)
* C2) /
                ((meanOrigG * meanOrigG + meanCompG
* meanCompG + C1) * (varOrigG + varCompG + C2));
    double ssimB = ((2.0 * meanOrigB * meanCompB + C1)
* C2) /
                ((meanOrigB * meanOrigB + meanCompB
* meanCompB + C1) * (varOrigB + varCompB + C2));

    double avgSSIM = (ssimR + ssimG + ssimB) / 3.0;

    avgSSIM = std::max(-1.0, std::min(1.0, avgSSIM));

    return 1.0 - avgSSIM;
}

default:
    return 0;
}
}

std::shared_ptr<QuadNode> QuadTree::buildTree(int x, int y, int
width, int height) {
    if (width <= 0 || height <= 0) {
        return nullptr;
    }
}

```



```

    }
    bool makeLeaf = false;
    if (width * height <= minBlockSize) {
        makeLeaf = true;
    } else {
        double error = calculateError(x, y, width, height);
        if (errorMethod == 5) {
            if (1.0 - error <= threshold) {
                makeLeaf = true;
            }
        } else {
            if (error <= threshold) {
                makeLeaf = true;
            }
        }
    }

    if (!makeLeaf) {
        int halfW = width / 2;
        int halfH = height / 2;
        if (halfW == 0 || halfH == 0) {
            makeLeaf = true;
        } else {
            int remW = width - halfW;
            int remH = height - halfH;
            if ((halfW * halfH < minBlockSize) ||
                (remW * halfH < minBlockSize) ||
                (halfW * remH < minBlockSize) ||
                (remW * remH < minBlockSize))
            {
                makeLeaf = true;
            }
        }
    }

    if (makeLeaf) {
        return std::make_shared<QuadNode>(x, y, width, height,
            calculateAverage(x, y, width, height), true);
    } else {
        int halfW = width / 2;
        int halfH = height / 2;
        int remW = width - halfW;
        int remH = height - halfH;

        if (halfW <= 0 || halfH <= 0 || remW <= 0 || remH <= 0)
    {

```

```

        return std::make_shared<QuadNode>(x, y, width,
height, calculateAverage(x, y, width, height), true);
    }

    auto nw = buildTree(x, y, halfW, halfH);
    auto ne = buildTree(x + halfW, y, remW, halfH);
    auto sw = buildTree(x, y + halfH, halfW, remH);
    auto se = buildTree(x + halfW, y + halfH, remW, remH);
    RGB avgColorForGif = calculateAverage(x, y, width,
height);
    auto node = std::make_shared<QuadNode>(x, y, width,
height, avgColorForGif, false);
    node->setChildren(nw, ne, sw, se);
    return node;
}
}

void QuadTree::compress(const std::vector<std::vector<RGB>>&
imagePixels) {
    if (imagePixels.empty() || imagePixels[0].empty()) {
        root = nullptr;
        return;
    }
    pixels = imagePixels;
    int height = pixels.size();
    int width = pixels[0].size();

    root = buildTree(0, 0, width, height);
}

void QuadTree::findNodesPerLevel(std::shared_ptr<QuadNode>
node, int level,
                                std::map<int,
std::vector<std::shared_ptr<QuadNode>>>& nodesMap,
                                int& maxLevelFound) {
    if (!node) {
        return;
    }
    nodesMap[level].push_back(node);
    maxLevelFound = std::max(maxLevelFound, level);

    if (!node->isLeafNode()) {
        for (int i = 0; i < 4; ++i) {
            findNodesPerLevel(node->getChild(i), level + 1,
nodesMap, maxLevelFound);

```

```

    }
}

void QuadTree::drawNodeArea(std::vector<std::vector<RGB>>&
canvas, std::shared_ptr<QuadNode> node) {
    if (!node) return;

    RGB color = node->getColor();
    int startX = node->getX();
    int startY = node->getY();
    int nodeWidth = node->getWidth();
    int nodeHeight = node->getHeight();

    int canvasHeight = canvas.size();
    if (canvasHeight == 0) return;
    int canvasWidth = canvas[0].size();
    if (canvasWidth == 0) return;

    int endY = std::min(startY + nodeHeight, canvasHeight);
    int endX = std::min(startX + nodeWidth, canvasWidth);
    startY = std::max(0, startY);
    startX = std::max(0, startX);

    for (int y = startY; y < endY; ++y) {
        for (int x = startX; x < endX; ++x) {
            canvas[y][x] = color;
        }
    }
}

bool QuadTree::saveGIF(const std::string& filename, int delay,
bool dither) {
    int imgHeight = pixels.size();
    int imgWidth = pixels[0].size();

    std::map<int, std::vector<std::shared_ptr<QuadNode>>>
nodesByLevel;
    int maxDepth = -1;
    findNodesPerLevel(this->root, 0, nodesByLevel, maxDepth);

    if (maxDepth < 0) {
        cerr << "Error: Pohon kosong." << endl;
        return false;
    }
}

```

```

GifWriter writer = {};
int gifDelay = delay / 10;
if (gifDelay < 1) gifDelay = 1;

if (!GifBegin(&writer, filename.c_str(), imgWidth,
imgHeight, gifDelay)) {
    cerr << "Error: Gagal memulai pembuatan GIF ke " <<
filename << endl;
    return false;
}

vector<vector<RGB>> frameCanvas(imgHeight,
vector<RGB>(imgWidth));
vector<uint8_t> imageBuffer(imgWidth * imgHeight * 4);

if (nodesByLevel.count(0) && !nodesByLevel[0].empty()) {
    drawNodeArea(frameCanvas, nodesByLevel[0][0]);
} else {
    for (auto& row : frameCanvas) std::fill(row.begin(),
row.end(), RGB(0, 0, 0));
}

for (int y = 0; y < imgHeight; ++y) {
    for (int x = 0; x < imgWidth; ++x) {
        size_t idx = (y * imgWidth + x) * 4;
        const RGB& pixel = frameCanvas[y][x];
        imageBuffer[idx + 0] = pixel.r;
        imageBuffer[idx + 1] = pixel.g;
        imageBuffer[idx + 2] = pixel.b;
        imageBuffer[idx + 3] = 255;
    }
}

if (!GifWriteFrame(&writer, imageBuffer.data(), imgWidth,
imgHeight, gifDelay, 8, dither)) {
    cerr << "Error: Gagal membuat frame awal GIF." << endl;
    GifEnd(&writer);
    return false;
}
for (int level = 1; level <= maxDepth; ++level) {
    if (nodesByLevel.count(level)) {
        for (const auto& node : nodesByLevel[level]) {
            drawNodeArea(frameCanvas, node);
        }
    }
}

```

```

        for (int y = 0; y < imgHeight; ++y) {
            for (int x = 0; x < imgWidth; ++x) {
                size_t idx = (y * imgWidth + x) * 4;
                const RGB& pixel = frameCanvas[y][x];
                imageBuffer[idx + 0] = pixel.r;
                imageBuffer[idx + 1] = pixel.g;
                imageBuffer[idx + 2] = pixel.b;
                imageBuffer[idx + 3] = 255;
            }
        }

        if (!GifWriteFrame(&writer, imageBuffer.data(),
imgWidth, imgHeight, gifDelay, 8, dither)) {
            cerr << "Error: Gagal membuat frame GIF level "
<< level << "." << endl;
            GifEnd(&writer);
            return false;
        }
    }
}

if (!GifEnd(&writer)) {
    cerr << "Error: Gagal menyelesaikan pembuatan GIF." <<
endl;
    return false;
}

cout << "\nGIF berhasil disimpan!" << endl;
return true;
}

std::vector<std::vector<RGB>> QuadTree::reconstructImage() {
    std::vector<std::vector<RGB>> result(pixels.size(),
std::vector<RGB>(pixels[0].size()));

    std::function<void(std::shared_ptr<QuadNode>)> reconstruct;
    reconstruct = [&](std::shared_ptr<QuadNode> node) {
        if (!node) return;

        if (node->isLeafNode()) {
            for (int y = node->getY(); y < node->getY() +
node->getHeight(); y++) {
                for (int x = node->getX(); x < node->getX() +
node->getWidth(); x++) {
                    result[y][x] = node->getColor();
                }
            }
        }
    };
    reconstruct(node);
}

```

```

        }
    }
} else {
    for (int i = 0; i < 4; i++) {
        reconstruct(node->getChild(i));
    }
}
};

reconstruct(root);
return result;
}

int QuadTree::countNodes() const {
    std::function<int(std::shared_ptr<QuadNode>)> count;
    count = [&](std::shared_ptr<QuadNode> node) {
        if (!node) return 0;
        int total = 1;
        for (int i = 0; i < 4; i++) {
            total += count(node->getChild(i));
        }
        return total;
    };
    return count(root);
}

int QuadTree::countLeaves() const {
    std::function<int(std::shared_ptr<QuadNode>)> count;
    count = [&](std::shared_ptr<QuadNode> node) {
        if (!node) return 0;
        if (node->isLeafNode()) return 1;
        int total = 0;
        for (int i = 0; i < 4; i++) {
            total += count(node->getChild(i));
        }
        return total;
    };
    return count(root);
}

int QuadTree::getDepth() const {
    std::function<int(std::shared_ptr<QuadNode>)> depth;
    depth = [&](std::shared_ptr<QuadNode> node) {
        if (!node || node->isLeafNode()) return 1;
        int maxDepth = 0;

```

```

        for (int i = 0; i < 4; i++) {
            maxDepth = std::max(maxDepth,
depth(node->getChild(i)));
        }
        return 1 + maxDepth;
    };
    return depth(root);
}

```

```
// QuadNode.hpp
```

```

#ifndef QUADTREE_NODE_HPP
#define QUADTREE_NODE_HPP

```

```

#include <memory>
#include "Image.hpp"

```

```

class QuadNode {
private:
    int x, y, width, height;
    RGB averageColor;
    bool isLeaf;
    std::shared_ptr<QuadNode> children[4];

```

```

public:
    QuadNode(int x, int y, int w, int h, RGB color, bool leaf)
        : x(x), y(y), width(w), height(h), averageColor(color),
isLeaf(leaf) {}

```

```

    void setChildren(std::shared_ptr<QuadNode> nw,
std::shared_ptr<QuadNode> ne,
                    std::shared_ptr<QuadNode> sw,
std::shared_ptr<QuadNode> se) {
        children[0] = nw;
        children[1] = ne;
        children[2] = sw;
        children[3] = se;
        isLeaf = false;
    }

```

```

    int getX() const { return x; }
    int getY() const { return y; }
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    RGB getColor() const { return averageColor; }

```

```

    bool isLeafNode() const { return isLeaf; }
    std::shared_ptr<QuadNode> getChild(int index) const {
return children[index]; }
};

#endif

```

#### IV. UJI TEST CASE

##### TEST CASE 1 (Variance)



GIF:

[https://github.com/poetoeeee/Tucil2\\_13523096/blob/main/test/gif/sbm\\_1\\_50.gif](https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sbm_1_50.gif)

COMPRESSION RESULTS	
Original size	: 213139 bytes
Compressed size	: 111796 bytes
Compression ratio	: 47.55%
Error threshold	: 50.00
Min Block size	: 16 pixels
Processing time	: 555 ms
Quadtree nodes	: 15057
Leaf nodes	: 11293
Tree depth	: 8

##### TEST CASE 2 (Variance)





<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftm_1_10.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftm_1_10.gif</a></p>	<pre>    C O M P R E S S I O N   R E S U L T S     -----  Original size      : 174246 bytes Compressed size    : 104673 bytes Compression ratio  : 39.93% Error threshold    : 10.00 Min Block size     : 10 pixels Processing time    : 628 ms Quadtree nodes     : 15761 Leaf nodes         : 11821 Tree depth         : 8 </pre>
--	---

### TEST CASE 3 (MAD)

	
<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftmd_2_20.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftmd_2_20.gif</a></p>	<pre>    C O M P R E S S I O N   R E S U L T S     -----  Original size      : 248725 bytes Compressed size    : 118135 bytes Compression ratio  : 52.50% Error threshold    : 20.00 Min Block size     : 10 pixels Processing time    : 212 ms Quadtree nodes     : 10521 Leaf nodes         : 7891 Tree depth         : 8 </pre>

### TEST CASE 4 (MAD)

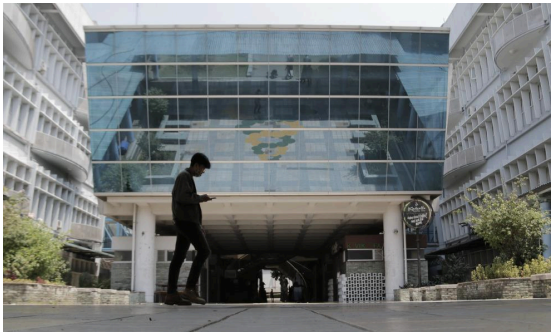
	
---	--

<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftsl_2_5.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/ftsl_2_5.gif</a></p>	<table> <tr> <th colspan="2">COMPRESSION RESULTS</th></tr> <tr> <td>Original size</td><td>: 231287 bytes</td></tr> <tr> <td>Compressed size</td><td>: 103177 bytes</td></tr> <tr> <td>Compression ratio</td><td>: 55.39%</td></tr> <tr> <td>Error threshold</td><td>: 5.00</td></tr> <tr> <td>Min Block size</td><td>: 10 pixels</td></tr> <tr> <td>Processing time</td><td>: 225 ms</td></tr> <tr> <td>Quadtree nodes</td><td>: 15197</td></tr> <tr> <td>Leaf nodes</td><td>: 11398</td></tr> <tr> <td>Tree depth</td><td>: 8</td></tr> </table>	COMPRESSION RESULTS		Original size	: 231287 bytes	Compressed size	: 103177 bytes	Compression ratio	: 55.39%	Error threshold	: 5.00	Min Block size	: 10 pixels	Processing time	: 225 ms	Quadtree nodes	: 15197	Leaf nodes	: 11398	Tree depth	: 8
COMPRESSION RESULTS																					
Original size	: 231287 bytes																				
Compressed size	: 103177 bytes																				
Compression ratio	: 55.39%																				
Error threshold	: 5.00																				
Min Block size	: 10 pixels																				
Processing time	: 225 ms																				
Quadtree nodes	: 15197																				
Leaf nodes	: 11398																				
Tree depth	: 8																				

#### TEST CASE 5 (Max Pixel Difference)

																					
<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/stei_3_25.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/stei_3_25.gif</a></p>	<table> <tr> <th colspan="2">COMPRESSION RESULTS</th></tr> <tr> <td>Original size</td><td>: 209636 bytes</td></tr> <tr> <td>Compressed size</td><td>: 119958 bytes</td></tr> <tr> <td>Compression ratio</td><td>: 42.78%</td></tr> <tr> <td>Error threshold</td><td>: 25.00</td></tr> <tr> <td>Min Block size</td><td>: 10 pixels</td></tr> <tr> <td>Processing time</td><td>: 334 ms</td></tr> <tr> <td>Quadtree nodes</td><td>: 15749</td></tr> <tr> <td>Leaf nodes</td><td>: 11812</td></tr> <tr> <td>Tree depth</td><td>: 8</td></tr> </table>	COMPRESSION RESULTS		Original size	: 209636 bytes	Compressed size	: 119958 bytes	Compression ratio	: 42.78%	Error threshold	: 25.00	Min Block size	: 10 pixels	Processing time	: 334 ms	Quadtree nodes	: 15749	Leaf nodes	: 11812	Tree depth	: 8
COMPRESSION RESULTS																					
Original size	: 209636 bytes																				
Compressed size	: 119958 bytes																				
Compression ratio	: 42.78%																				
Error threshold	: 25.00																				
Min Block size	: 10 pixels																				
Processing time	: 334 ms																				
Quadtree nodes	: 15749																				
Leaf nodes	: 11812																				
Tree depth	: 8																				

#### TEST CASE 6 (Max Pixel Difference)

	
---	--

<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sith_3_5.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sith_3_5.gif</a></p>	<table> <tr> <th colspan="2">COMPRESSION RESULTS</th></tr> <tr> <td>Original size</td><td>: 182964 bytes</td></tr> <tr> <td>Compressed size</td><td>: 115289 bytes</td></tr> <tr> <td>Compression ratio</td><td>: 36.99%</td></tr> <tr> <td>Error threshold</td><td>: 5.00</td></tr> <tr> <td>Min Block size</td><td>: 10 pixels</td></tr> <tr> <td>Processing time</td><td>: 338 ms</td></tr> <tr> <td>Quadtree nodes</td><td>: 21001</td></tr> <tr> <td>Leaf nodes</td><td>: 15751</td></tr> <tr> <td>Tree depth</td><td>: 8</td></tr> </table>	COMPRESSION RESULTS		Original size	: 182964 bytes	Compressed size	: 115289 bytes	Compression ratio	: 36.99%	Error threshold	: 5.00	Min Block size	: 10 pixels	Processing time	: 338 ms	Quadtree nodes	: 21001	Leaf nodes	: 15751	Tree depth	: 8
COMPRESSION RESULTS																					
Original size	: 182964 bytes																				
Compressed size	: 115289 bytes																				
Compression ratio	: 36.99%																				
Error threshold	: 5.00																				
Min Block size	: 10 pixels																				
Processing time	: 338 ms																				
Quadtree nodes	: 21001																				
Leaf nodes	: 15751																				
Tree depth	: 8																				

### TEST CASE 7 (Entropy)

																					
<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sappk_4_2.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sappk_4_2.gif</a></p>	<table> <tr> <th colspan="2">COMPRESSION RESULTS</th></tr> <tr> <td>Original size</td><td>: 221416 bytes</td></tr> <tr> <td>Compressed size</td><td>: 125398 bytes</td></tr> <tr> <td>Compression ratio</td><td>: 43.37%</td></tr> <tr> <td>Error threshold</td><td>: 2.00</td></tr> <tr> <td>Min Block size</td><td>: 10 pixels</td></tr> <tr> <td>Processing time</td><td>: 1083 ms</td></tr> <tr> <td>Quadtree nodes</td><td>: 20525</td></tr> <tr> <td>Leaf nodes</td><td>: 15394</td></tr> <tr> <td>Tree depth</td><td>: 8</td></tr> </table>	COMPRESSION RESULTS		Original size	: 221416 bytes	Compressed size	: 125398 bytes	Compression ratio	: 43.37%	Error threshold	: 2.00	Min Block size	: 10 pixels	Processing time	: 1083 ms	Quadtree nodes	: 20525	Leaf nodes	: 15394	Tree depth	: 8
COMPRESSION RESULTS																					
Original size	: 221416 bytes																				
Compressed size	: 125398 bytes																				
Compression ratio	: 43.37%																				
Error threshold	: 2.00																				
Min Block size	: 10 pixels																				
Processing time	: 1083 ms																				
Quadtree nodes	: 20525																				
Leaf nodes	: 15394																				
Tree depth	: 8																				

### TEST CASE 8 (SSIM)

	
---	--



<p>GIF:  <a href="https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sf_5_05.gif">https://github.com/poetoeeee/Tucil2_13523096/blob/main/test/gif/sf_5_05.gif</a></p>	<pre> -----    C O M P R E S S I O N   R E S U L T S     ----- Original size      : 394503 bytes Compressed size    : 165055 bytes Compression ratio  : 58.16% Error threshold    : 0.50 Min Block size     : 10 pixels Processing time    : 934 ms Quadtree nodes     : 20697 Leaf nodes         : 15523 Tree depth         : 8 </pre>
--	---

## V. ANALISIS

- Mekanisme**

Algoritma melakukan kompresi dengan mengganti blok-blok piksel yang memiliki variasi warna dibawah threshold yang ditentukan menjadi satu warna rata-rata.
- Threshold**

Parameter ini berpengaruh dalam menentukan tingkat kompresi dan kualitas gambar. Semakin rendah threshold maka pohon akan semakin dalam dan lebih banyak leaf node. Artinya kualitas gambar akan lebih baik, namun rasio kompresi lebih rendah. Sebaliknya Threshold yang tinggi akan menghasilkan pohon yang lebih dangkal dan lebih sedikit leaf node. Kualitas gambar akan lebih rendah dan tingkat kompresi lebih tinggi.
- Minimum Block Size**

Parameter ini berguna untuk menetapkan batas bawah ukuran blok yang bisa menjadi leaf node. Ukuran minimum yang lebih besar akan memaksa penggabungan blok meskipun variasinya mungkin masih di atas threshold, yang dapat menurunkan kualitas visual secara signifikan pada area detail tetapi meningkatkan rasio kompresi dan mempercepat proses.
- Error Method**

Pilihan metode pengukuran error (Variance, MAD, Max Pixel Difference, Entropy, SSIM) mempengaruhi *bagaimana* homogenitas blok dinilai. Setiap metode sensitif terhadap jenis variasi warna yang berbeda. SSIM, misalnya, mencoba mengukur kemiripan struktural, sementara Variance mengukur penyebaran kuadratik dari rata-rata. Pilihan metode dapat menghasilkan struktur

pohon dan hasil kompresi yang berbeda untuk gambar yang sama, bahkan dengan threshold yang "serupa".

- Kompleksitas

Dalam kasus terburuk (worst case), yaitu ketika gambar memiliki detail tinggi di mana-mana sehingga pohon harus membelah hingga minBlockSize (atau mendekati 1 piksel), algoritma perlu memeriksa setiap piksel. Perhitungan error pada suatu node yang mencakup k piksel membutuhkan waktu  $O(k)$  (tergantung pada metode error). Ditambah dengan tahap rekonstruksi gambar, dimana setiap piksel diisi tepat satu kali, kompleksitas waktunya adalah  $O(N)$ .

Sedangkan untuk kompleksitas ruang, Struktur QuadTree itu sendiri membutuhkan ruang untuk menyimpan informasi node (posisi, dimensi, warna rata-rata, penunjuk anak, status leaf/internal). Dalam kasus terburuk (gambar sangat detail), jumlah node bisa mendekati jumlah piksel, sehingga kompleksitas ruang untuk pohon adalah  $O(N)$ . Selain itu program membutuhkan ruang untuk menyimpan data piksel asli dan data piksel hasil rekonstruksi, yang masing-masing adalah  $O(N)$ .

## VI. LAMPIRAN

Repository program ini dapat dilihat [disini](#).

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	V	
2. Program berhasil dijalankan	V	
3. Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	V	
4. Mengimplementasi seluruh metode perhitungan error wajib	V	
5. <b>[Bonus]</b> Implementasi persentase kompresi sebagai parameter tambahan		V
6. <b>[Bonus]</b> Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error	V	
7. <b>[Bonus]</b> Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar	V	
8. Program dan laporan dibuat (kelompok) sendiri	V	