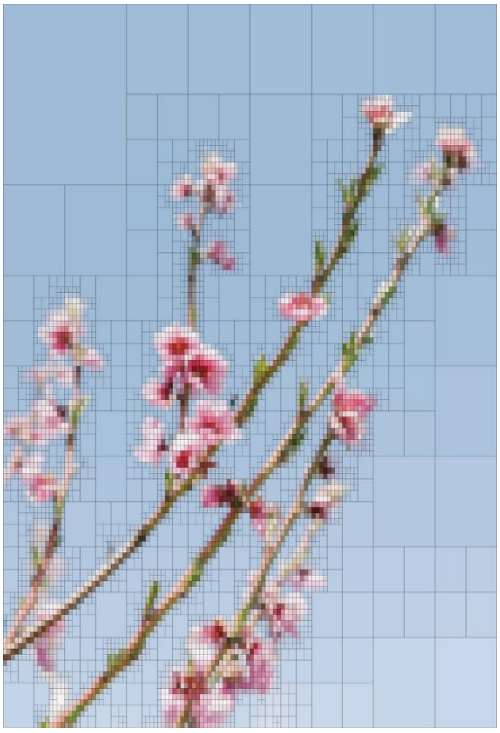
LAPORAN TUGAS KECIL 2

IF2211 Strategi Algoritma

Kompresi Gambar Dengan Metode Quadtree



Oleh :
Reza Ahmad Syarif (13523119)

PROGRAM STUDI TEKNIK INFORMATIKA SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA INSTITUT TEKNOLOGI BANDUNG JL. GANESA 10, BANDUNG 40132 2024

DAFTAR ISI

1 DESKRIPSI MASALAH

1.1 Kompresi Gambar Dengan Metode Quadtree

Quadtree adalah struktur data hierarkis yang digunakan untuk membagi ruang atau data menjadi bagian yang lebih kecil, yang sering digunakan dalam pengolahan gambar. Dalam konteks kompresi gambar, Quadtree membagi gambar menjadi blok-blok kecil berdasarkan keseragaman warna atau intensitas piksel. Prosesnya dimulai dengan membagi gambar menjadi empat bagian, lalu memeriksa apakah setiap bagian memiliki nilai yang seragam berdasarkan analisis sistem warna RGB, yaitu dengan membandingkan komposisi nilai merah (R), hijau (G), dan biru (B) pada piksel-piksel di dalamnya. 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.

Dalam program kompresi gambar dengan metode Quadtree ini terdapat beberapa prosedur sebagai berikut.

1. Inisialisasi dan Persiapan Data

   Gambar yang akan dikompresi dimasukkan dalam program, kemudian akan diolah dalam format matriks piksel dengan nilai intensitas berdasarkan sistem warna RGB. Beberapa parameter juga dimasukkan saat ingin melakukan kompresi gambar sebagai berikut.

   - Metode perhitungan variansi

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

   - Threshold variansi

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

   - Minimum block size

     Minimum block size (luas piksel) adalah ukuran terkecil dari sebuah blok yang diizinkan dalam proses kompresi. Jika ukuran blok yang akan dibagi menjadi empat sub-blok berada di bawah ukuran minimum yang

telah dikonfigurasi, maka blok tersebut tidak akan dibagi lebih lanjut, meskipun error masih di atas threshold.

2. Perhitungan Error

   Untuk setiap blok gambar yang sedang diproses, nilai variansi akan dihitung menggunakan metode variansi yang dimasukkan. Pilihan metode tersebut sebagai berikut.

   - *Variance*

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

   - *Mean absolute deviation* (MAD)

$$MAD_c = \frac{1}{N}\sum_{i=1}^{N}|P_{i,c} - \mu_c| \quad MAD_{RGB} = \frac{MAD_R + MAD_G + MAD_B}{3}$$

   - *Max pixel difference*

$$D_c = max(P_{i,c}) - min(P_{i,c}) \quad D_{RGB} = \frac{D_R + D_G + D_B}{3}$$

   - *Entropy*

$$H_c = -\sum_{i=1}^{N} P_c(i)log_2(P_c(i)) \quad H_{RGB} = \frac{H_R + H_G + H_B}{3}$$

3. Pembagian Blok

   Nilai variansi blok akan dibandingkan dengan threshold. Jika variansi di atas threshold, 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 kondisi tidak terpenuhi, proses pembagian dihentikan untuk blok tersebut.

4. Normalisai Warna

   Untuk blok yang tidak lagi dibagi, dilakukan 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.

2 ALGORITMA DIVIDE AND CONQUER

Kompresi gambar dengan Quadtree ini melibatkan algoritma Divide and Conquer. Pendekatan ini dipilih karena sangat sesuai dengan sifat hierarkis dari struktur pohon Quadtree yang membagi gambar secara rekursif ke dalam sub-blok hingga ditemukan bagian-bagian gambar yang homogen.

Prinsip Devide and Conquer dalam kompresi gambar ini dimulai dengan membagi gambar ke dalam blok besar yang mencakup keseluruhan piksel gambar. Setiap blok kemudian dianalisis untuk menentukan apakah blok tersebut homogen atau tidak. Jika tidak homogen, blok dibagi menjadi empat sub-blok kuadran. Ini adalah penerapan prinsip devide pada algoritma ini untuk membagi masalah ke persoalan-persoalan yang lebih kecil.

Prinsip Conquer diterapkan ketika setiap sub-blok diproses secara rekursif dengan cara yang sama. Jika masih ditemukan ketidakhomogenan dan ukuran blok masih di atas ukuran minimum (*minBlockSize*), maka proses pembagian akan dilanjutkan. Proses rekursif dihentikan ketika blok sudah cukup homogen atau ukurannya terlalu kecil untuk dibagi lebih lanjut.

Tahap akhir dari algoritma ini yaitu melakukan tahap Combine dari proses-proses yang telah dibagi sebelumnya. Blok-blok homogen kemudian direpresentasikan dalam gambar hasil kompresi sebagai satu warna rata-rata dan digabung kembali untuk merekonstruksi gambar akhir.

Secara sistematis, berikut merupakan langkah-langkah kompresi gambar dengan metode Quadtree berbasis algortima Devide and Conquer.

1. Inisialisasi gambar sebagai blok persegi panjang yang mencakup seluruh piksel.
2. Kondisi dasar yaitu ketika blok terlalu kecil untuk dibagi lanjut.
3. Hitung nilai error dari blok tersebut berdasarkan metode error yang dipilih.
4. Jika nilai error dibawah ambang batas (threshold) dan ukuran blok tidak lebih kecil dari minimum block size, maka blok akan dianggap homogen.
5. Jika tidak homogen, blok dibagi menjadi empat sub-blok yaitu kiri atas, kanan atas, kiri bawah, dan kanan bawah.
6. Lakukan rekursi untuk setiap sub-blok dengan langkah yang sama.\
7. Setelah semua blok homogen terdeteksi, gambar direkontruksi kembali berdasarkan warna rata-rata dari setiap blok homogen.

Dengan menggunakan pseudocode, proses algoritma Devide and Conquer tersebut berjalan sebagai berikut.

```
procedure Subdivide(
    input imageData : MatriksPixel,
    input x, y, width, height : integer,
    input imageWidth : integer,
    input threshold : real,
```

```
    input minBlockSize : integer,
    input method : integer,
    input channels : integer,
    output node : QuadtreeNode
)

Algoritma:
    // Kondisi dasar: blok terlalu kecil untuk dibagi lebih lanjut
    if width ≤ minBlockSize or height ≤ minBlockSize or
       width ≤ 1 or height ≤ 1 or
       floor(width / 2) < minBlockSize or floor(height / 2) < minBlockSize then
        node ← LeafNode(averageColor(imageData, x, y, width, height, channels))
        return

    // Cek apakah blok homogen berdasarkan metode error tertentu
    error ← HitungError(imageData, x, y, width, height, method, channels)
    if error ≤ threshold then
        node ← LeafNode(averageColor(imageData, x, y, width, height, channels))
        return

    // Jika tidak homogen dan masih bisa dibagi, lakukan pembagian (Divide)
    halfWidth ← floor(width / 2)
    halfHeight ← floor(height / 2)

    // Buat 4 node anak dengan koordinat sub-blok
    buat node.child[0] untuk blok (x, y, halfWidth, halfHeight) // kiri atas
    buat node.child[1] untuk blok (x + halfWidth, y, width - halfWidth, halfHeight) // kanan atas
```

```
    buat node.child[2] untuk blok (x, y + halfHeight, halfWidth,
height - halfHeight) // kiri bawah

    buat node.child[3] untuk blok (x + halfWidth, y + halfHeight,
width - halfWidth, height - halfHeight) // kanan bawah



    // Conquer: lakukan subdivide secara rekursif pada setiap anak

    for i ← 0 to 3 do

        Subdivide(imageData, node.child[i].x, node.child[i].y,

                  node.child[i].width, node.child[i].height,

                  imageWidth, threshold, minBlockSize, method,
channels, node.child[i])



    node ← InternalNode(node.child[0], ..., node.child[3])

end
```

3 SOURCE CODE PROGRAM DAN TEST CASE

3.1 Source Code

3.1.1 Quadtree

```
// Compressor.hpp
#ifndef QUADTREE_NODE_HPP
#define QUADTREE_NODE_HPP
#include "../utils/RGB.hpp"


class QuadtreeNode {
private:
    int x, y, width, height;
    RGB color;
    bool isLeaf;
    QuadtreeNode* children[4];


public:
    QuadtreeNode(int x, int y, int width, int height);


    ~QuadtreeNode();


    void setColor(const RGB& c) { color = c; }
    RGB getColor() const { return color; }
    bool isLeafNode() const { return isLeaf; }
    void setLeaf(bool leaf) { isLeaf = leaf; }


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


    QuadtreeNode* getChild(int index) const { return children[index]; }
      void setChild(int index, QuadtreeNode* child) { children[index] = child; }


    void calculateAverageColor(const unsigned char* imageData, int imageWidth, int
channels);
    bool isHomogeneous(const unsigned char* imageData, int imageWidth, int method, double
threshold, int channels);
```

```
     void subdivide(const unsigned char* imageData, int imageWidth, int imageHeight,
double threshold, int minBlockSize, int method, int channels);


    int countNodes() const;

    int getDepth() const;

};

#endif


// Compressor.cpp

#include "QuadtreeNode.hpp"

#include "../utils/metrics.hpp"

#include <cstring>

#include <vector>

using namespace std;


QuadtreeNode::QuadtreeNode(int x, int y, int width, int height)

    : x(x), y(y), width(width), height(height), isLeaf(true) {

    memset(children, 0, sizeof(children));

}


QuadtreeNode::~QuadtreeNode() {

    for (auto& child : children) {

        delete child;

        child = nullptr;

    }

}


void QuadtreeNode::calculateAverageColor(const unsigned char* imageData, int imageWidth,
int channels) {

    long long sumR = 0, sumG = 0, sumB = 0;

    int count = 0;


    for (int j = y; j < y + height; ++j) {

        for (int i = x; i < x + width; ++i) {

            int idx = (j * imageWidth + i) * channels;

            sumR += imageData[idx];

            sumG += imageData[idx + 1];

            sumB += imageData[idx + 2];
```

```
            ++count;
        }
    }

    if (count > 0) {
        setColor(RGB(
            static_cast<uint8_t>(sumR / count),
            static_cast<uint8_t>(sumG / count),
            static_cast<uint8_t>(sumB / count)
        ));
    }
}

bool QuadtreeNode::isHomogeneous(const unsigned char* imageData, int imageWidth, int
method, double threshold, int channels) {
    double error = 0.0;
    switch (method) {
        case 1: error = computeVariance(imageData, x, y, width, height, imageWidth,
channels); break;
        case 2: error = computeMAD(imageData, x, y, width, height, imageWidth, channels);
break;
        case 3: error = computeMaxDiff(imageData, x, y, width, height, imageWidth,
channels); break;
        case 4: error = computeEntropy(imageData, x, y, width, height, imageWidth,
channels); break;
        default: throw std::invalid_argument("Invalid method");
    }

    return error <= threshold;
}

void QuadtreeNode::subdivide(const unsigned char* imageData, int imageWidth, int
imageHeight,double threshold, int minBlockSize, int method, int channels) {
    if (width <= minBlockSize || height <= minBlockSize ||
        width <= 1 || height <= 1 ||
        width / 2 < minBlockSize || height / 2 < minBlockSize) {
        setLeaf(true);
           calculateAverageColor(imageData, imageWidth, channels);
        return;
```

```
    }


    if (isHomogeneous(imageData, imageWidth, method, threshold, channels)) {
        setLeaf(true);
        calculateAverageColor(imageData, imageWidth, channels);
        return;
    }


    setLeaf(false);


    int halfWidth = std::max(1, width / 2);
    int halfHeight = std::max(1, height / 2);


    children[0] = new QuadtreeNode(x, y, halfWidth, halfHeight);
    children[1] = new QuadtreeNode(x + halfWidth, y, width - halfWidth, halfHeight);
    children[2] = new QuadtreeNode(x, y + halfHeight, halfWidth, height - halfHeight);
    children[3] = new QuadtreeNode(x + halfWidth, y + halfHeight, width - halfWidth,
height - halfHeight);


    for (auto& child : children) {
        child->subdivide(imageData, imageWidth, imageHeight, threshold, minBlockSize,
method, channels);
    }
}


int QuadtreeNode::countNodes() const {
    if (isLeaf) {
        return 1;
    }


    int count = 1;
    for (const auto& child : children) {
        count += child->countNodes();
    }
    return count;
}


int QuadtreeNode::getDepth() const {
    if (isLeaf) {
```

```
        return 0;
    }

    int maxDepth = 0;
    for (const auto& child : children) {
        int childDepth = child->getDepth();
        if (childDepth > maxDepth) {
            maxDepth = childDeth;
        }
    }
    return maxDepth + 1;
}
```

3.1.2 Compressor

```
#ifndef COMPRESSOR_HPP

#define COMPRESSOR_HPP


#include "QuadtreeNode.hpp"

#include "RGB.hpp"

#include <string>

using namespace std;


class Compressor {

    private:

        QuadtreeNode* root;

        unsigned char* imageData;

        const char* inputPath;

        const char* outputPath;

        int height, width, channels;

        int method;

        int minBlockSize;

        double threshold;

        double targetCompressionRatio;


        size_t originalSizeBytes;

        size_t compressedSizeBytes;

        double compressionRatio;

        double executionTime;


        bool compressionSuccess;

    public:

        Compressor(const char* inputPath, const char* outputPath, int method, double
threshold, int minBlockSize, double targetCompressionRatio);

        ~Compressor();

        void compressImage();

        void analyzeImage();

        void saveCompressedImage();


        int getChannels() const { return channels; }

        void fillBlock(unsigned char* data, int imageWidth, const QuadtreeNode* node)
const;
```

```
        void renderQuadtree(unsigned char* data, int imageWidth, const QuadtreeNode* node)
const;


    private:

        void buildQuadtree();

        void freeImageData();

};


#endif


#include "Compressor.hpp"

#include "../utils/image_io.hpp"

#include "../utils/stb_image.h"

#include "../utils/stb_image_write.h"

#include <iostream>

#include <chrono>

#include <thread>


#include <filesystem>

namespace fs = std::filesystem;


using namespace std;


struct ThresholdRange {

    double low;

    double high;

    int maxAttempt;

};


ThresholdRange getThresholdRange(int method) {

    switch (method) {

        case 1: return {0.0, 65025.0, 16};

        case 2: return {0.0, 127.5, 8};

        case 3: return {0.0, 255.0, 9};

        case 4: return {0.0, 8.0, 10};

        default: return {1.0, 10000.0, 12};

    }
```

```
}


Compressor::Compressor(const char* inputPath, const char* outputPath, int method, double
threshold, int minBlockSize, double targetCompressionRatio) :
    root(nullptr),
    imageData(nullptr),
    inputPath(inputPath),
    outputPath(outputPath),
    height(0),
    width(0),
    channels(0),
    method(method),
    minBlockSize(minBlockSize),
    threshold(threshold),
    targetCompressionRatio(targetCompressionRatio),
    compressedSizeBytes(0),
    compressionSuccess(false) {}


Compressor::~Compressor() {
    if (root) {
        delete root;
    }
    freeImageData();
}


void Compressor::compressImage() {
    cout << "[LOG] Starting compression..." << endl;
    cout << "[LOG] Reading image: " << inputPath << endl;
    imageData = read_image(inputPath, &width, &height, &channels);
    if (!imageData) {
        cerr << "Failed to load image: " << inputPath << endl;
        return;
    }


    cout << "[LOG] Image loaded: " << width << "x" << height << " channels: " << channels
<< endl;
    originalSizeBytes = fs::file_size(inputPath);
```

```
    auto start = std::chrono::high_resolution_clock::now();
    if (targetCompressionRatio > 0.0) {
        minBlockSize = 4;
        ThresholdRange range = getThresholdRange(method);
        double low = range.low;
        double high = range.high;
        int maxAttempt = range.maxAttempt;
        const double epsilon = 1e-3;

        double bestThreshold = -1;
        double bestRatio = 1.0;

        int attempts = 1;
        while (low <= high && attempts < maxAttempt + 1) {
            double progress = static_cast<double>(attempts) / maxAttempt;
            cout << "[LOG] Loading: " << static_cast<int>(progress * 100) << "%\r" <<
flush;
            attempts++;
            threshold = (low + high) / 2.0;
            if (root) delete root;

            buildQuadtree();
            saveCompressedImage();
            compressionRatio = static_cast<double>(compressedSizeBytes) /
originalSizeBytes;
            double compressionPercent = 1.0 - compressionRatio;

            // cout << "[DEBUG] threshold=" << threshold
            //      << ", ratio=" << compressionRatio
            //      << ", penyusutan=" << compressionPercent * 100 << "%" << endl;

            if (compressionPercent >= targetCompressionRatio) {
                bestThreshold = threshold;
                bestRatio = compressionRatio;
                high = threshold - epsilon;
            } else {
                low = threshold + epsilon;
            }
        }
```

```
        if (bestThreshold >= 0.0) {
            threshold = bestThreshold;
            if (root) delete root;
            buildQuadtree();
            saveCompressedImage();
            compressionRatio = bestRatio;
        } else {
            cerr << "[WARNING] Tidak ditemukan threshold yang memenuhi target penyusutan."
<< endl;
        }
    } else {
        buildQuadtree();
        saveCompressedImage();
        compressionRatio = static_cast<double>(compressedSizeBytes) / originalSizeBytes;
    }

    auto end = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double>(end - start).count();
    analyzeImage();
}

void Compressor::analyzeImage() {
    if (!root) return;
    cout << fixed << setprecision(4);
    cout << "\n=== Statistik Kompresi ===" << endl;
    cout << "Waktu eksekusi:           " << executionTime << " detik" << endl;
    cout << "Ukuran gambar (asli):     " << originalSizeBytes << " bytes" << endl;
    cout << "Ukuran gambar (kompresi): " << compressedSizeBytes << " bytes" << endl;
    cout << "Persentase kompresi:      " << (100.0 * (1.0 - compressionRatio)) << "%" <<
endl;
    cout << "Kedalaman pohon:          " << root->getDepth() << endl;
    cout << "Jumlah simpul pohon:      " << root->countNodes() << endl;

    if (compressionSuccess) {
        cout << "Gambar disimpan di:       " << outputPath << endl;
    }

    cout << "==========================" << endl;
```

```
}


void Compressor::fillBlock(unsigned char* data, int imageWidth, const QuadtreeNode* node)
const {
    RGB color = node->getColor();
    for (int j = node->getY(); j < node->getY() + node->getHeight(); ++j) {
        for (int i = node->getX(); i < node->getX() + node->getWidth(); ++i) {
            int idx = (j * imageWidth + i) * channels;
            data[idx]     = color.getR();
            data[idx + 1] = color.getG();
            data[idx + 2] = color.getB();
            if (channels == 4) data[idx + 3] = 255;
        }
    }
}


void Compressor::renderQuadtree(unsigned char* data, int imageWidth, const QuadtreeNode*
node) const {
    if (node->isLeafNode()) {
        fillBlock(data, imageWidth, node);
    } else {
        for (int i = 0; i < 4; ++i) {
            if (node->getChild(i)) {
                renderQuadtree(data, imageWidth, node->getChild(i));
            }
        }
    }
}


void Compressor::saveCompressedImage() {
    if (!root) return;


    int size = width * height * channels;
    unsigned char* outputData = new unsigned char[size];
    std::fill(outputData, outputData + size, 0);


    renderQuadtree(outputData, width, root);


    write_image(outputPath, outputData, width, height, channels);
```

```
    if (fs::exists(outputPath)) {
        size_t tempSize = fs::file_size(outputPath);
        compressedSizeBytes = tempSize;
        if (tempSize <= originalSizeBytes) {
            compressionSuccess = true;
        } else {
            fs::remove(outputPath);
            cerr << "[WARNING] Kompresi tidak efisien. File tidak disimpan.\n";
            cout << "[INFO] Kompresi gambar tidak disimpan.\n";
            cout << "[SUGGESTION] Gunakan threshold atau minBlockSize yang lebih
besar.\n";
            return;
        }
    } else {
        cerr << "[ERROR] File output tidak ditemukan setelah penyimpanan!\n";
        return;
    }
    delete[] outputData;
}

void Compressor::buildQuadtree() {
    root = new QuadtreeNode(0, 0, width, height);
    root->subdivide(imageData, width, height, threshold, minBlockSize, method, channels);
}

void Compressor::freeImageData() {
    if (imageData) {
        stbi_image_free(imageData);
        imageData = nullptr;
    }
}
```

3.1.3 Utils

```
#ifndef IMAGE_IO_HPP

#define IMAGE_IO_HPP


#include <string>

using namespace std;


bool isValidExtension(const string& filename);

string getFileExtension(const std::string& filename);

unsigned char* read_image(const char* filename, int* width, int* height, int* channels);

void write_image(const string& filename, unsigned char* data, int width, int height, int
channels);


#endif


#define STB_IMAGE_IMPLEMENTATION

#include "stb_image.h"


#define STB_IMAGE_WRITE_IMPLEMENTATION

#include "stb_image_write.h"


#include "image_io.hpp"

#include <algorithm>


bool isValidExtension(const std::string& filename) {

    std::string extension = filename.substr(filename.find_last_of('.') + 1);

    std::transform(extension.begin(), extension.end(), extension.begin(), ::tolower);

    return (extension == "png" || extension == "jpg" || extension == "jpeg");

}


std::string getFileExtension(const std::string& filename) {

    size_t dot = filename.find_last_of('.');

    if (dot == std::string::npos) return "";

    std::string ext = filename.substr(dot + 1);

    std::transform(ext.begin(), ext.end(), ext.begin(), ::tolower);

    return ext;

}
```

```
unsigned char* read_image(const char* filename, int* width, int* height, int* channels) {
    unsigned char* data = stbi_load(filename, width, height, channels, 0);
    if (data == nullptr) {
        fprintf(stderr, "Error loading image: %s\n", stbi_failure_reason());
    }
    return data;
}


void write_image(const std::string& filename, unsigned char* data, int width, int height,
int channels) {
    std::string ext = getFileExtension(filename);
    bool success = false;


    if (ext == "png") {
        success = stbi_write_png(filename.c_str(), width, height, channels, data, width *
channels);
    } else if (ext == "jpg" || ext == "jpeg") {
        success = stbi_write_jpg(filename.c_str(), width, height, channels, data, 100);
    } else {
        fprintf(stderr, "[ERROR] Format file output tidak didukung: %s\n", ext.c_str());
        return;
    }


    if (!success) {
        fprintf(stderr, "Error writing image: %s\n", stbi_failure_reason());
    }
}


#ifndef METRICS_HPP
#define METRICS_HPP


double computeVariance(const unsigned char* imageData, int x, int y, int width, int
height, int imagewidth, int channels);
double computeMAD(const unsigned char* imageData, int x, int y, int width, int height, int
imagewidth, int channels);
double computeMaxDiff(const unsigned char* imageData, int x, int y, int width, int height,
int imagewidth, int channels);
double computeEntropy(const unsigned char* imageData, int x, int y, int width, int height,
int imagewidth, int channels);


#endif
```

```
#include "metrics.hpp"
#include <cmath>
#include <iostream>
#include <array>


double computeVariance(const unsigned char* imageData, int x, int y, int width, int
height, int imageWidth, int channels) {
    long long sumR = 0, sumG = 0, sumB = 0;
    int N = width * height;


    for (int j = y; j < y + height; ++j) {
        for (int i = x; i < x + width; ++i) {
            int idx = (j * imageWidth + i) * channels;
            sumR += imageData[idx];
            sumG += imageData[idx + 1];
            sumB += imageData[idx + 2];
        }
    }


    double meanR = static_cast<double>(sumR) / N;
    double meanG = static_cast<double>(sumG) / N;
    double meanB = static_cast<double>(sumB) / N;


    double varR = 0.0, varG = 0.0, varB = 0.0;


    for (int j = y; j < y + height; ++j) {
        for (int i = x; i < x + width; ++i) {
            int idx = (j * imageWidth + i) * channels;
            varR += (imageData[idx] - meanR) * (imageData[idx] - meanR);
            varG += (imageData[idx + 1] - meanG) * (imageData[idx + 1] - meanG);
            varB += (imageData[idx + 2] - meanB) * (imageData[idx + 2] - meanB);
        }
    }


    varR /= N;
    varG /= N;
    varB /= N;
```

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


double computeMAD(const unsigned char* imageData, int x, int y, int width, int height,
int imageWidth, int channels) {
    long long sumR = 0, sumG = 0, sumB = 0;
    int N = width * height;


    for (int j = y; j < y + height; ++j) {
        for (int i = x; i < x + width; ++i) {
            int idx = (j * imageWidth + i) * channels;
            sumR += imageData[idx];
            sumG += imageData[idx + 1];
            sumB += imageData[idx + 2];
        }
    }


    double meanR = static_cast<double>(sumR) / N;
    double meanG = static_cast<double>(sumG) / N;
    double meanB = static_cast<double>(sumB) / N;


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


    for (int j = y; j < y + height; ++j) {
        for (int i = x; i < x + width; ++i) {
            int idx = (j * imageWidth + i) * channels;
            madR += std::abs(imageData[idx] - meanR);
            madG += std::abs(imageData[idx + 1] - meanG);
            madB += std::abs(imageData[idx + 2] - meanB);
        }
    }


    madR /= N;
    madG /= N;
    madB /= N;
```

```
    double mad = (madR + madG + madB) / 3.0;
    return mad;
}

double computeMaxDiff(const unsigned char* imageData, int x, int y, int width, int height,
int imageWidth, int channels) {
    int minR = 255, minG = 255, minB = 255;
    int maxR = 0, maxG = 0, maxB = 0;

    for (int j = y; j < y + height; j++) {
        for (int i = x; i < x + width; i++) {
            int idx = (j * imageWidth + i) * channels;
            unsigned char r = imageData[idx];
            unsigned char g = imageData[idx + 1];
            unsigned char b = imageData[idx + 2];

            if (r < minR) minR = r;
            if (g < minG) minG = g;
            if (b < minB) minB = b;
            if (r > maxR) maxR = r;
            if (g > maxG) maxG = g;
            if (b > maxB) maxB = b;
        }
    }

    double diffR = static_cast<double>(maxR - minR);
    double diffG = static_cast<double>(maxG - minG);
    double diffB = static_cast<double>(maxB - minB);

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

double computeEntropy(const unsigned char* imageData, int x, int y, int width, int height,
int imageWidth, int channels) {
    std::array<int, 256> freqR = {0};
    std::array<int, 256> freqG = {0};
    std::array<int, 256> freqB = {0};
    int N = width * height;
```

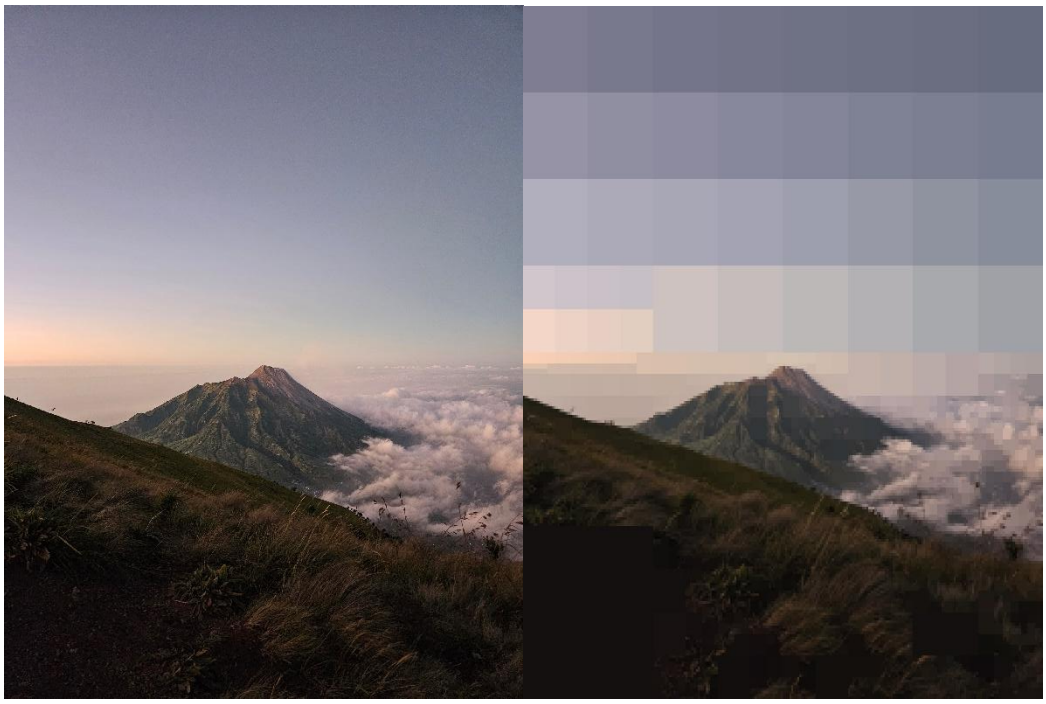
```
    for (int j = y; j < y + height; ++j) {
        for (int i = x; i < x + width; ++i) {
            int idx = (j * imageWidth + i) * channels;
            freqR[imageData[idx]]++;
            freqG[imageData[idx + 1]]++;
            freqB[imageData[idx + 2]]++;
        }
    }

    auto calcEntropy = [N](const std::array<int, 256>& freq) {
        double entropy = 0.0;
        for (int i = 0; i < 256; ++i) {
            double p = static_cast<double>(freq[i]) / N;
            if (p > 0) {
                entropy -= p * std::log2(p);
            }
        }
        return entropy;
    };

    double entropyR = calcEntropy(freqR);
    double entropyG = calcEntropy(freqG);
    double entropyB = calcEntropy(freqB);

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

#ifndef RGB_HPP
#define RGB_HPP

#include <cstdint>
#include <cmath>
#include <iostream>

class RGB {
private:
    uint8_t r, g, b;
```

```
public:
    RGB() : r(0), g(0), b(0) {}
    RGB(uint8_t r, uint8_t g, uint8_t b) : r(r), g(g), b(b) {}

    uint8_t getR() const { return r; }
    uint8_t getG() const { return g; }
    uint8_t getB() const { return b; }
    void setR(uint8_t value) { r = value; }
    void setG(uint8_t value) { g = value; }
    void setB(uint8_t value) { b = value; }

    double distanceTo(const RGB& other) const {
        int dr = int(r) - int(other.r);
        int dg = int(g) - int(other.g);
        int db = int(b) - int(other.b);
        return std::sqrt(dr * dr + dg * dg + db * db);
    }

    RGB operator+(const RGB& other) const {
        return RGB(r + other.r, g + other.g, b + other.b);
    }

    RGB operator/(int scalar) const {
        return RGB(r / scalar, g / scalar, b / scalar);
    }

    friend std::ostream& operator<<(std::ostream& os, const RGB& color) {
        os << "(" << int(color.r) << ", " << int(color.g) << ", " << int(color.b) << ")";
        return os;
    }
};

#endif
```

3.2 Test Case File Besar

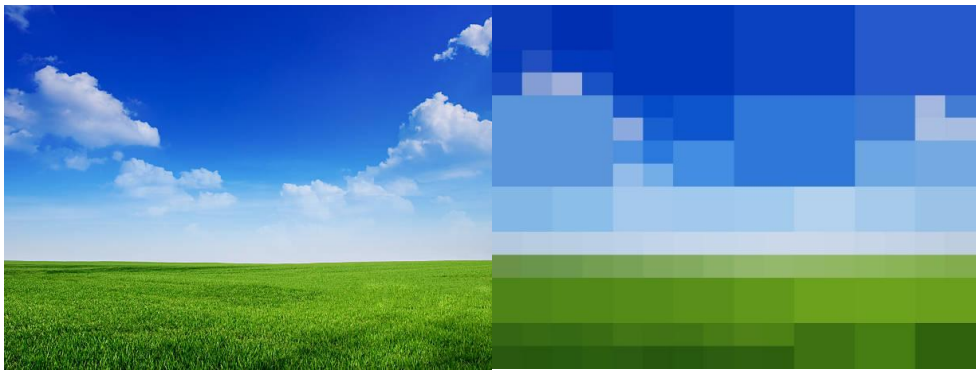
3.2.1 Test Case File JPG

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg1.jpg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 1
[INFO] Threshold Variance: 0 - 65025
Masukkan nilai threshold: 100
Masukkan ukuran blok minimum (1 - 64): 8
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg1.jpg

=========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg1.jpg
[LOG] Image loaded: 4000x3000 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           1.3590 detik
Ukuran gambar (asli):     3767684 bytes
Ukuran gambar (kompresi): 1345585 bytes
Persentase kompresi:      64.2862%
Kedalaman pohon:          8
Jumlah simpul pohon:      20965
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg1.jpg
=========================
```



```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg2.jpg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 2
[INFO] Threshold MAD: 0 - 127.5
Masukkan nilai threshold: 25
Masukkan ukuran blok minimum (1 - 64): 4
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg2.jpg

=========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg2.jpg
[LOG] Image loaded: 4000x2252 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           1.4907 detik
Ukuran gambar (asli):     9548045 bytes
Ukuran gambar (kompresi): 4774246 bytes
Persentase kompresi:      49.9977%
Kedalaman pohon:          9
Jumlah simpul pohon:      100729
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg2.jpg
=========================
```

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg3.jpg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 1
[INFO] Threshold Variance: 0 - 65025
Masukkan nilai threshold: 1000
Masukkan ukuran blok minimum (1 - 64): 16
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg3.jpg

=========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg3.jpg
[LOG] Image loaded: 612x459 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           0.0458 detik
Ukuran gambar (asli):     46400 bytes
Ukuran gambar (kompresi): 42605 bytes
Persentase kompresi:      8.1789%
Kedalaman pohon:          4
Jumlah simpul pohon:      137
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg3.jpg
=========================
```



3.2.2 Test Case File JPEG

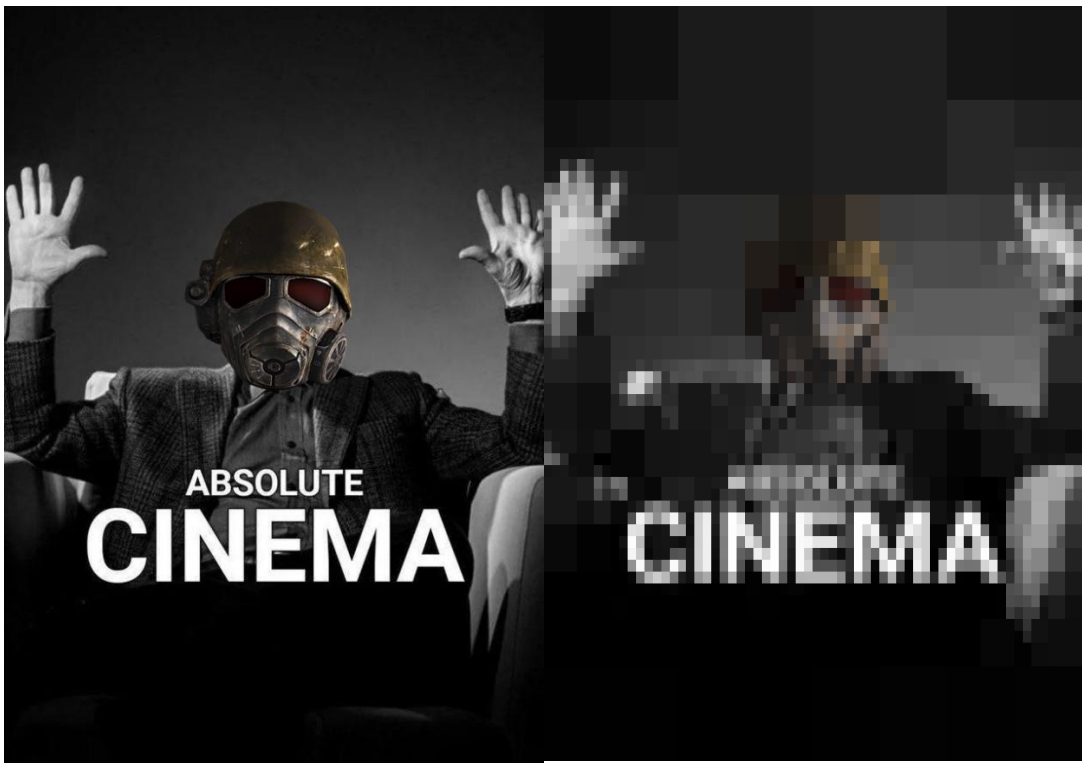
```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpeg1.jpeg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 2
[INFO] Threshold MAD: 0 - 127.5
Masukkan nilai threshold: 20
Masukkan ukuran blok minimum (1 - 64): 8
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpeg1.jpeg

========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpeg1.jpeg
[LOG] Image loaded: 6120x8160 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           6.8109 detik
Ukuran gambar (asli):     19235318 bytes
Ukuran gambar (kompresi): 14902521 bytes
Persentase kompresi:      22.5252%
Kedalaman pohon:          9
Jumlah simpul pohon:      153289
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpeg1.jpeg
========================
```

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpeg2.jpeg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 3
[INFO] Threshold MaxDiff: 0 - 255
Masukkan nilai threshold: 200
Masukkan ukuran blok minimum (1 - 64): 8
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpeg2.jpeg

========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpeg2.jpeg
[LOG] Image loaded: 3000x4000 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           0.8750 detik
Ukuran gambar (asli):     2169561 bytes
Ukuran gambar (kompresi): 917196 bytes
Persentase kompresi:      57.7244%
Kedalaman pohon:          8
Jumlah simpul pohon:      3149
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpeg2.jpeg
========================
```

3.2.3 Test Case File PNG

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\png1.png
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 4
[INFO] Threshold Entropy: 0 - 8
Masukkan nilai threshold: 2
Masukkan ukuran blok minimum (1 - 64): 8
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_png1.png

=========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\png1.png
[LOG] Image loaded: 1536x1024 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           0.3629 detik
Ukuran gambar (asli):     2890027 bytes
Ukuran gambar (kompresi): 100549 bytes
Persentase kompresi:      96.5208%
Kedalaman pohon:          7
Jumlah simpul pohon:      21845
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_png1.png
=========================
```

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\png2.png
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 3
[INFO] Threshold MaxDiff: 0 - 255
Masukkan nilai threshold: 100
Masukkan ukuran blok minimum (1 - 64): 8
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_png2.png

========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\png2.png
[LOG] Image loaded: 637x893 channels: 3

=== Statistik Kompresi ===
Waktu eksekusi:           0.1224 detik
Ukuran gambar (asli):     311115 bytes
Ukuran gambar (kompresi): 23372 bytes
Persentase kompresi:      92.4877%
Kedalaman pohon:          6
Jumlah simpul pohon:      1733
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_png2.png
========================
```



3.2.4 Test Case Target Compression

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\target1.jpg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 1
[INFO] Threshold Variance: 0 - 65025
Masukkan nilai threshold: 212
Masukkan ukuran blok minimum (1 - 64): 6
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0.5
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\target1.jpg

========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\target1.jpg
[LOG] Image loaded: 4000x2252 channels: 3
[WARNING] Kompresi tidak efisien. File tidak disimpan.
[INFO] Kompresi gambar tidak disimpan.
[SUGGESTION] Gunakan threshold atau minBlockSize yang lebih besar.
[LOG] Loading: 100%
=== Statistik Kompresi ===
Waktu eksekusi:           18.4797 detik
Ukuran gambar (asli):     5031881 bytes
Ukuran gambar (kompresi): 2512239 bytes
Persentase kompresi:      50.0736%
Kedalaman pohon:          9
Jumlah simpul pohon:      47293
Gambar disimpan di:       D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\target1.jpg
========================
```

3.2.5 Test Case Persentase Kompresi Negatif

```
Masukkan alamat absolut gambar input: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg3.jpg
Pilih metode perhitungan error (1: Variance, 2: MAD, 3: MaxDiff, 4: Entropy): 3
[INFO] Threshold MaxDiff: 0 - 255
Masukkan nilai threshold: 10
Masukkan ukuran blok minimum (1 - 64): 2
Masukkan target persentase kompresi (1.0 = 100%, 0 = nonaktif): 0
Masukkan alamat absolut gambar output: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\output\output_jpg3.jpg

========================
[INFO] Jumlah channel gambar: 0
[LOG] Starting compression...
[LOG] Reading image: D:\Kuliah\Tubes\Strategi Algoritma\Tucil2\Tucil2_13523119\test\input\jpg3.jpg
[LOG] Image loaded: 612x459 channels: 3
[WARNING] Kompresi tidak efisien. File tidak disimpan.
[INFO] Kompresi gambar tidak disimpan.
[SUGGESTION] Gunakan threshold atau minBlockSize yang lebih besar.

=== Statistik Kompresi ===
Waktu eksekusi:          0.0816 detik
Ukuran gambar (asli):    46400 bytes
Ukuran gambar (kompresi): 228001 bytes
Persentase kompresi:     -391.3815%
Kedalaman pohon:         8
Jumlah simpul pohon:     28889
========================
```

4 ANALISIS KOMPLEKSITAS WAKTU

Algoritma devide and Conquer pada kompresi gambar dengan metode Quadtree bekerja dengan cara memeriksa homogenitas warna pada setiap blok gambar. Jika suatu blok dianggap tidak homogen, maka blok tersebut dibagi menjadi empat sub-blok, dan proses yang sama dilakukan secara rekursif terhadap masing-masing sub-blok.

Misalkan ukuran gambar adalah **W × H** piksel, dan total area gambar adalah **n = W × H**. Setiap blok berukuran **w × h** akan dihitung nilai error-nya. Karena perhitungan error harus melibatkan semua piksel dalam blok tersebut, maka waktu yang dibutuhkan pada satu langkah rekursi adalah **O(w · h)**. Dalam kasus awal (seluruh gambar), biaya ini menjadi **O(W · H)**.

Quadtree akan membagi setiap blok menjadi 4 bagian berukuran setengah semula sehingga jumlah sub-masalah adalah 4. Kemudian, ukuran masalah dibagi dua pada setiap tingkat rekursi. Dengan demikian, kompleksitas waktu total dapat dimodelkan dalam bentuk relasi rekursif sebagai berikut:

$$\boldsymbol{T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)}$$

Namun karena setiap pemrosesan pada suatu blok ukuran **n** memerlukan akses ke seluruh piksel dalam blok (bukan hanya konstan), maka sebenarnya biaya per level adalah **O(n²)**, dan relasi yang benar adalah:

$$\boldsymbol{T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n^2)}$$

Dengan $a = 4,\ b = 2\ ,\ f(n) = O(n^2)$, dapat dibandingkan $f(n)$ dengan $n^{\log_2 a} = n^2$. Karena $f(n) = O(n^2)$, maka sesuai dengan Master Theorem kasus 2:

$$\boldsymbol{T(n) = O(n^2 log\ n)}$$

Maka, jika total piksel gambar adalah n = W · H, maka kompleksitas waktu total dari algoritma kompresi gambar menggunakan Quadtree dapat dinyatakan sebagai:

$$\boldsymbol{T(n) = O(W \cdot H \cdot log\ (W.H))}$$

5 LAMPIRAN

5.1 Penjelasan Bonus

5.1.1 *Compression Percentage* (Persentase Kompresi)

Compression percentage atau persentase kompresi adalah ukuran yang menunjukkan seberapa besar ukuran gambar berhasil dikurangi setelah proses kompresi dilakukan. Nilai ini memberikan gambaran seberapa efektif algoritma Quadtree dalam menyederhanakan informasi visual tanpa menyimpan setiap piksel secara eksplisit. Dalam program ini, pengukuran besar gambar sebelum dan sesudah dilakukan kompresi menggunakan library filesystem. Library ini hanya tersedia pada versi C++17 ke atas. Oleh karena itu, perlu dipastikan memiliki versi C++17 ke atas serta kompiler yang mendukung C++17 (seperti GCC 8+, Clang 7+, MSVC 2017+) apabila ingin meng-*compile* ulang. Gunakan prompt berikut saat kompilasi.

```
g++ src\main\main.cpp src\utils\image_io.cpp src\utils\metrics.cpp
src\Quadtree\QuadtreeNode.cpp src\Compressor\Compressor.cpp -I
src\utils -I src\Quadtree -I src\Compressor -std=c++17 -Wall -o
bin\quadtree_compression.exe
```

5.1.2 Target Persentase Kompresi

Bonus target persentase kompresi memungkinkan user untuk menentukan seberapa besar tingkat kompresi yang ingin dicapai. Alih-alih menentukan threshold secara manual, program akan menyesuaikan nilai threshold secara otomatis untuk mencapai target penyusutan yang diinginkan. Untuk menemukan persentase kompresi yang dicari, program akan melakukan *binary search* dengan minimal threshold sebagai pointer *low* dan maksimal threshold sebagai pointer *high*. Minimal dan maksimal threshold ini berbeda-beda tergantung metode yang digunakan.

5.2 Tautan Repository Github

https://github.com/Rejaah/Tucil2_13523119