LAPORAN TUGAS KECIL 2
IF2211 STRATEGI ALGORITMA

Kompresi Gambar Dengan Metode Quadtree

Disusun oleh:
Henry Filberto Shenelo 13523108

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

Daftar Isi

BAB I
ALGORITMA DIVIDE AND CONQUER

A. Penjelasan Dasar

Algoritma Divide and Conquer, merupakan pendekatan algoritmik yang memecahkan suatu persoalan besar dengan cara membaginya menjadi beberapa sub-persoalan yang lebih kecil dan mirip dengan persoalan asal. Teknik ini bekerja dalam tiga tahap utama: Divide (membagi), Conquer (menyelesaikan), dan Combine (menggabungkan). Pada tahap pertama, masalah dibagi menjadi bagian-bagian yang lebih kecil. Selanjutnya, masing-masing sub-persoalan diselesaikan secara langsung jika cukup sederhana, atau secara rekursif jika masih kompleks. Setelah semua sub-persoalan terselesaikan, hasilnya digabungkan untuk membentuk solusi akhir dari persoalan semula. Pendekatan ini lebih efisien untuk berbagai jenis persoalan dan sering kali lebih cepat dibandingkan dengan Brute Force karena dapat mengurangi kompleksitas waktu secara signifikan. Beberapa contoh permasalahan yang dapat diselesaikan menggunakan algoritma Divide and Conquer antara lain:

1. Merge Sort, yaitu mengurutkan data dengan membagi list menjadi dua bagian, mengurutkan masing-masing bagian, dan kemudian menggabungkannya kembali.
2. Binary Search, yaitu mencari elemen dalam list terurut dengan cara membagi ruang pencarian menjadi dua bagian di setiap langkah pencarian.
3. Strassen's Matrix Multiplication, yaitu mengalikan dua matriks besar dengan membaginya menjadi sub-matriks yang lebih kecil untuk mengurangi jumlah operasi.

B. Aplikasi *Divide and Conquer* pada Kompresi Gambar dengan Quad Tree

Pada metode kompresi gambar menggunakan Quad Tree, sebuah gambar akan dibagi menjadi beberapa bagian persegi. Objektif dari metode ini adalah merepresentasikan gambar dengan struktur data pohon yang dapat mengurangi ukuran penyimpanan tanpa kehilangan terlalu banyak informasi visual. Metode ini dapat disimulasikan menggunakan algoritma Divide and Conquer. Ini berarti gambar akan dibagi secara rekursif ke dalam empat bagian kuadran yang lebih kecil hingga ditemukan bagian-bagian yang homogen (nilai intensitas serupa) atau ukuran bagian tersebut sudah cukup kecil.

Berikut adalah penjelasan singkat akan aplikasi algoritma Divide and Conquer untuk kompresi gambar dengan Quad Tree.

1. Program menerima alamat absolut dari gambar yang akan dikompresi.
2. Program menerima metode untuk menghitung error, *threshold*, dan minimum *block size*.
3. Gambar dipecah menjadi 4 blok-blok persegi (quadrants) secara rekursif menggunakan struktur data QuadTree. Proses kompresi dilakukan dengan prinsip Divide and Conquer sebagai berikut:
    a. Divide: Gambar awal dianggap sebagai blok besar (root node). Blok ini akan dibagi menjadi 4 sub-blok (children nodes) jika variansi di atas threshold (belum homogen), ukuran blok lebih besar dari minimum *block size*, dan

ukuran blok setelah dibagi menjadi empat tidak kurang dari minimum *block size*. Setiap *children nodes* akan melalui proses yang sama secara rekursif hingga kondisi pembagian tidak lagi terpenuhi.

b. Conquer: Jika sub-blok tidak dapat lagi dibagi (tidak memenuhi kriteria), blok tersebut akan dihitung nilai rata-rata pixelnya untuk setiap channel. Hasil nilai rata-rata pixel akan disimpan dalam list(vector) pada setiap node Quad Tree.
c. Combine: Setiap pixel pada node Quad Tree akan diisi dengan nilai rata-rata yang tersimpan pada list.

4. Jika terdapat input target kompresi pada program (tidak 0), program akan melakukan *binary search* dengan mengompresi gambar berulang kali dengan *threshold* yang berbeda (minimum *block size* tidak berpengaruh) hingga ditemukan nilai yang paling mendekati target kompresi.
5. Program menyimpan gambar kompresi pada alamat output yang ditentukan

Berikut adalah pseudocode dari algoritma Divide and Conquer yang digunakan.

```
Function compress(node, image, method, threshold, minBlockSize):
    // Base Case: Block is too small to split
    IF (node.width * node.height <= minBlockSize) OR (node.width *
node.height == 1):
        node.calculateAverageColor(image)  // Conquer: Store avg
color
        node.isLeaf = TRUE
        RETURN

    // Calculate error
    error = CALCULATE_ERROR(image, node.x, node.y, node.width,
node.height, method)

    // Base Case: Block is homogeneous
    IF error <= threshold:
        node.calculateAverageColor(image)  // Conquer: Store avg
color
        node.isLeaf = TRUE
        RETURN

    // Divide: Split into 4 sub-blocks
    halfWidth = node.width / 2
    halfHeight = node.height / 2

//Divide: Validate child sizes
    halfWidth = node.width // 2
    halfHeight = node.height // 2
    IF (halfWidth * halfHeight < minBlockSize) OR
       (halfWidth * remainingHeight < minBlockSize) OR
       (remainingWidth * halfHeight < minBlockSize) OR
       (remainingWidth * remainingHeight < minBlockSize):
        node.calculateAverageColor(image)  //Conquer
        node.isLeaf = TRUE
        RETURN
```

```
    node.children[0] = QuadTreeNode(node.x, node.y, halfWidth,
halfHeight) //top left

    node.children[1] = QuadTreeNode(node.x + halfWidth, node.y,
remainingWidth, halfHeight)    //top right

    node.children[2] = QuadTreeNode(node.x, node.y + halfHeight,
halfWidth, remainingHeight)   //bottom left

    node.children[3] = QuadTreeNode(node.x + halfWidth, node.y +
halfHeight, remainingWidth, remainingHeight) //bottom right

    // recursively divide and compress each child

    FOR i = 0 TO 3:

        compress(node.children[i], image, method, threshold,
minBlockSize)
```

```
Function fillImage(node, outputImage):
    // Base Case:if leaf node then fill its region with avgColor
    IF node.isLeaf://Combine: Apply stored color
        FOR y FROM node.y TO node.y + node.height - 1:
            FOR x FROM node.x TO node.x + node.width - 1:
                outputImage.setPixel(x, y, node.avgColor)
        RETURN

    // Recurse on non-leaf children
    FOR i FROM 0 TO 3:
        IF node.children[i] != NULL:
            fillImage(node.children[i], outputImage)
```

C. Implementasi SSIM dan Target Compression

Pada implementasi SSIM, menggunakan rumus

$$SSIM_c(x,y) = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(2\sigma_{xy,c} + C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + \sigma_{y,c}^2 + C_2)}$$

Rumus ini dapat dipecah menjadi 3 bagian, yaitu *luminance(*l), *contrast*(c), dan *structure*(s) yang diperoleh melalui rumus

$$l(x,y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}$$

$$s(x,y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3}$$

Pada rumus ini, x dan y adalah 2 gambar yang dibandingkan. Semakin tinggi (mendekati 1), artinya semakin mirip. Untuk nilai c1,c2, dan c3, c1 didapatkan dari $c_1 = (k_1 L)^2$ dan $c_2 = (k_2 L)^2$ dengan k1 = 0.01 dan k2 = 0.03, sementara $L = 2^{8-1} = 255$. c3 adalah c2/2. Sehingga nilai c1,c2, dan c3 adalah 6.5025, 58.5225, dan 29.26125.

Pada implementasi Target Compression, dilakukan dengan cara melakukan proses kompresi dan penghitungan rasio kompresi antara gambar original dengan yang sudah terkompresi. Proses pencarian *threshold* yang sesuai menggunakan *binary search* dengan batas bawah 0 dan batas atas adalah nilai maksimum yang mungkin dari tiap metode yang digunakan (contohnya 255 untuk Max Pixel Difference). Proses iterasi ini dilakukan hingga rasio kompresi berada pada rentang toleransi 0.01 atau sudah mencapai iterasi maksimum 15 kali. *Threshold* yang didapatkan kemudian digunakan untuk mengkompresi gambar yang akan disimpan.

BAB II
SOURCE CODE

Tugas kecil ini diimplementasikan dengan menggunakan Bahasa pemrograman c++. Berikut adalah daftar *class* yang digunakan:

a. Image
b. QuadTreeNode
c. QuadTree
d. main

Program ini menggunakan library stb_image_write dan stb_image. Selain itu, daftar functions dan procedures yang digunakan adalah sebagai berikut.

a. Image.cpp
    1. Image(const string& filename)
    2. ab. Image(int width, int height)
    3. ~Image()
    4. save(const string& filename)
    5. getPixel(int x, int y, int channel)
    6. setPixel(int x, int y, int channel, int value)
    7. getWidth()
    8. getHeight()

b. QuadTreeNode
    1. QuadTreeNode(int x, int y, int width, int height)
    2. ~QuadTreeNode()
    3. getChild(int index)
    4. getWidth()
    5. getHeight()
    6. isLeafNode()
    7. countLeafNodes()
    8. countTotalNodes()
    9. depth()
    10. compress(const Image& img, const int method, double threshold, int minBlockSize, bool targetOn)
    11. fillImage(Image& img)
    12. calculateAverageColor(const Image& img)
    13. calculateVariance(const Image& img)
    14. calculateMAD(const Image& img)
    15. calculateMaxDifference(const Image& img)
    16. calculateEntropy(const Image& img)
    17. calculateSSIM(const Image& img1, const Image& img2, int x1, int y1, int x2, int y2, int width, int height)

18. compressWithSSIM(const Image& img, double threshold, int minBlockSize, bool targetOn)

c. QuadTree
    1. QuadTree(const Image& img, const int method, double threshold, int minSize, bool targetOn)
    2. compressImage(const Image& img)
    3. decompressImage(Image& img)
    4. saveImage(const string& filename)
    5. getCompressionRatio(const string& inputFilename, const string& outputFilename)
    6. getRoot()
    7. getBestThreshold(const string& inputFilename, int method, double targetRatio)
    8. getMaxThresholdForMethod(int method)

d. main()

Berikut adalah *source code* dari program yang telah dibuat

1. File: Image.hpp

```
#ifndef IMAGE_HPP
#define IMAGE_HPP

#include <vector>
#include <string>
#include <stdexcept>
using namespace std;

class Image {
public:
    Image(const string& filename);
    Image(int width, int height);
    ~Image();

    bool save(const string& filename) const;
    int getPixel(int x, int y, int channel) const;
    void setPixel(int x, int y, int channel, int value);
    int getWidth() const { return width; }
    int getHeight() const { return height; }

private:
    int width, height;
    vector<unsigned char> pixels;
};

#endif // IMAGE_HPP
```

2. File: Image.cpp

```cpp
#include "Image.hpp"
#define STB_IMAGE_IMPLEMENTATION
#include "include/stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "include/stb_image_write.h"
#include <iostream>

Image::Image(const string& filename) {
    int channels;

    FILE* testFile = fopen(filename.c_str(), "rb");
    if (!testFile) {
        cerr << "Error: File is not found.\n";
    } else {
        fclose(testFile);
    }
    unsigned char* data = stbi_load(filename.c_str(), &width, &height, &channels, 3);

    try {
        pixels.resize(width * height * 3);
        copy(data, data + width * height * 3, pixels.begin());
    } catch (...) {
        stbi_image_free(data);
        throw;
    }

    stbi_image_free(data);
}

Image::Image(int width, int height) : width(width), height(height) {
    pixels.resize(width * height * 3, 0); // Initialize to black
}

Image::~Image() {
    // Automatic cleanup by vector
}
```

```cpp
bool Image::save(const string& filename) const {
    string ext = filename.substr(filename.find_last_of(".") + 1);
    int success = 0;

    if (ext == "png") {
        success = stbi_write_png(filename.c_str(), width, height, 3, pixels.data(), width * 3);
    } else if (ext == "jpg" || ext == "jpeg") {
        success = stbi_write_jpg(filename.c_str(), width, height, 3, pixels.data(), 90);
    } else if (ext == "bmp") {
        success = stbi_write_bmp(filename.c_str(), width, height, 3, pixels.data());
    } else {
        throw runtime_error("Cannot save image. Please check your path and extension.");
    }

    return success != 0;
}

int Image::getPixel(int x, int y, int channel) const {
    if (x < 0 || x >= width || y < 0 || y >= height || channel < 0 || channel > 2) {
        throw out_of_range("Pixel coordinates or channel out of range");
    }
    return pixels[(y * width + x) * 3 + channel];
}

void Image::setPixel(int x, int y, int channel, int value) {
    if (x < 0 || x >= width || y < 0 || y >= height || channel < 0 || channel > 2) {
        throw out_of_range("Pixel coordinates or channel out of range");
    }
    if (value < 0 || value > 255) {
        throw invalid_argument("Pixel value must be between 0 and 255");
    }
    pixels[(y * width + x) * 3 + channel] = static_cast<unsigned char>(value);
}
```

3. **File: QuadTreeNode.hpp**

```hpp
#ifndef QUADTREE_HPP
#define QUADTREE_HPP

#include "Image.hpp"
#include "QuadTreeNode.hpp"

class QuadTree {
public:
    QuadTree(const Image& img, const int method, double threshold, int minSize, bool targetOn);

    void compressImage(const Image& img);
    void decompressImage(Image& img) const;
    bool saveImage(const string& filename) const;
    double getCompressionRatio(const string& inputFilename, const string& outputFilename) const;
    QuadTreeNode* getRoot() const { return root; }
    double getBestThreshold(const string& inputFilename, int method, double targetRatio);
    double getMaxThresholdForMethod(int method) const;

private:
    QuadTreeNode* root;
    int errorMethod;
    double threshold;
    int minBlockSize;
    int originalWidth;
    int originalHeight;
    bool targetOn;
    bool compressNow;
};

#endif // QUADTREE_HPP
```

4. **File: QuadTreeNode.cpp**

```cpp
#include "QuadTreeNode.hpp"
#include <cmath>
#include <algorithm>
#include <numeric>
#include <cstring>
#include <limits>
#include <iostream>

QuadTreeNode::QuadTreeNode(int x, int y, int width, int height)
    : x(x), y(y), width(width), height(height), isLeaf(false), avgColor{0, 0, 0}, children{nullptr, nullptr, nullptr, nullptr} {
}

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

QuadTreeNode* QuadTreeNode::getChild(int index) const {
    if (index < 0 || index > 3) return nullptr;
    return children[index];
}
```

```cpp
int QuadTreeNode::getWidth() const {
    return width;
}

int QuadTreeNode::getHeight() const {
    return height;
}

int QuadTreeNode::countLeafNodes() const {
    if (isLeaf) return 1;
    int count = 0;
    for (int i = 0; i < 4; i++) {
        if (children[i]) count += children[i]->countLeafNodes();
    }
    return count;
}

int QuadTreeNode::countTotalNodes() const {
    int count = 1; // Count this node
    for (int i = 0; i < 4; i++) {
        if (children[i]) count += children[i]->countTotalNodes();
    }
    return count;
}

int QuadTreeNode::depth() const {
    if (isLeaf) return 0;
    int maxDepth = -1;
    for (int i = 0; i < 4; i++) {
        if (children[i]) {
            int childDepth = children[i]->depth();
            if (childDepth > maxDepth) {
                maxDepth = childDepth;
            }
        }
    }
    return maxDepth + 1;
}
```

```cpp
void QuadTreeNode::compress(const Image& img, const int method, double threshold, int minBlockSize, bool targetOn) {

    if (method == 5) {
        compressWithSSIM(img, threshold, minBlockSize, targetOn);
        return;
    }

    // Calculate sub-block areas
    int subWidth1 = width / 2;
    int subWidth2 = width - subWidth1;
    int subHeight1 = height / 2;
    int subHeight2 = height - subHeight1;

    if (!targetOn) {
        // Stop if current block area is too small
        if (width * height <= minBlockSize) {
            calculateAverageColor(img);
            isLeaf = true;
            return;
        }

        // Stop if ANY sub-block would be too small
        if (subWidth1 * subHeight1 < minBlockSize ||
            subWidth1 * subHeight2 < minBlockSize ||
            subWidth2 * subHeight1 < minBlockSize ||
            subWidth2 * subHeight2 < minBlockSize) {
            calculateAverageColor(img);
            isLeaf = true;
            return;
        }
    }

    if (width * height <= 1) {
        calculateAverageColor(img);
        isLeaf = true;
        return;
    }

    // Stop if ANY sub-block would be too small
    if (subWidth1 * subHeight1 <= 1 ||
        subWidth1 * subHeight2 <= 1 ||
        subWidth2 * subHeight1 <= 1 ||
        subWidth2 * subHeight2 <= 1) {
        calculateAverageColor(img);
        isLeaf = true;
        return;
    }
```

```
double error = 0.0;
if (method == 1) {
    error = calculateVariance(img);
} else if (method == 2) {
    error = calculateMAD(img);
} else if (method == 3) {
    error = calculateMaxDifference(img);
} else if (method == 4) {
    error = calculateEntropy(img);
}

if (error <= threshold) {
    calculateAverageColor(img);
    isLeaf = true;
} else {
    int halfWidth = width / 2;
    int halfHeight = height / 2;
    int remainingWidth = width - halfWidth;
    int remainingHeight = height - halfHeight;

    children[0] = new QuadTreeNode(x, y, halfWidth, halfHeight);
    children[1] = new QuadTreeNode(x + halfWidth, y, remainingWidth, halfHeight);
    children[2] = new QuadTreeNode(x, y + halfHeight, halfWidth, remainingHeight);
    children[3] = new QuadTreeNode(x + halfWidth, y + halfHeight, remainingWidth, remainingHeight);

    for (int i = 0; i < 4; i++) {
        if (children[i]) {
            children[i]->compress(img, method, threshold, minBlockSize, targetOn);
        }
    }
}
```

```cpp
double QuadTreeNode::calculateVariance(const Image& img) const {
    double mean[3] = {0};
    double variance[3] = {0};
    int pixelCount = 0;
    int imgWidth = img.getWidth();
    int imgHeight = img.getHeight();

    // Calculate mean
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
            mean[0] += img.getPixel(x_pos, y_pos, 0);
            mean[1] += img.getPixel(x_pos, y_pos, 1);
            mean[2] += img.getPixel(x_pos, y_pos, 2);
            pixelCount++;
        }
    }

    if (pixelCount == 0) return 0.0;

    for (int k = 0; k < 3; k++) {
        mean[k] /= pixelCount;
    }

    // Calculate variance ()
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
            variance[0] += pow(img.getPixel(x_pos, y_pos, 0) - mean[0], 2);
            variance[1] += pow(img.getPixel(x_pos, y_pos, 1) - mean[1], 2);
            variance[2] += pow(img.getPixel(x_pos, y_pos, 2) - mean[2], 2);
        }
    }

    return (variance[0] + variance[1] + variance[2]) / (3 * pixelCount);
}
```

```cpp
double QuadTreeNode::calculateMAD(const Image& img) const {
    Click to collapse the range.
    int pixelCount = 0;
    int imgWidth = img.getWidth();
    int imgHeight = img.getHeight();

    // Calculate mean
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
            mean[0] += img.getPixel(x_pos, y_pos, 0);
            mean[1] += img.getPixel(x_pos, y_pos, 1);
            mean[2] += img.getPixel(x_pos, y_pos, 2);
            pixelCount++;
        }
    }

    if (pixelCount == 0) return 0.0;

    for (int k = 0; k < 3; k++) {
        mean[k] /= pixelCount;
    }

    // absolute of (pixel - mean)
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
            mad[0] += abs(img.getPixel(x_pos, y_pos, 0) - mean[0]);
            mad[1] += abs(img.getPixel(x_pos, y_pos, 1) - mean[1]);
            mad[2] += abs(img.getPixel(x_pos, y_pos, 2) - mean[2]);
        }
    }

    return (mad[0] + mad[1] + mad[2]) / (3 * pixelCount);
}
```

```cpp
double QuadTreeNode::calculateMaxDifference(const Image& img) const {
    int minVal[3] = {numeric_limits<int>::max(),
                     numeric_limits<int>::max(),
                     numeric_limits<int>::max()};
    int maxVal[3] = {numeric_limits<int>::min(),
                     numeric_limits<int>::min(),
                     numeric_limits<int>::min()};
    int imgWidth = img.getWidth();
    int imgHeight = img.getHeight();

    // Calculate min and max values for each channel
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++)
            for (int c = 0; c < 3; c++) {
                int val = img.getPixel(x_pos, y_pos, c);
                minVal[c] = min(minVal[c], val);
                maxVal[c] = max(maxVal[c], val);
            }
        }
    }

    //calculate max difference (max - min) for each channel and divide by 3
    double maxDiff = 0;
    for (int c = 0; c < 3; c++) {
        maxDiff += (maxVal[c] - minVal[c]);
    }

    return maxDiff / 3.0;
}
```

```cpp
double QuadTreeNode::calculateEntropy(const Image& img) const {
    int occ[3][256] = {0};
    int pixelCount = 0;
    int imgWidth = img.getWidth();
    int imgHeight = img.getHeight();

    // calculate occurences of each pixel value for each channel
    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++)
            for (int c = 0; c < 3; c++) {
                int val = img.getPixel(x_pos, y_pos, c);
                occ[c][val]++;
            }
            pixelCount++;
        }
    }

    if (pixelCount == 0) return 0.0;

    // calculate entropy for each channel
    double entropy[3] = {0};
    for (int c = 0; c < 3; c++) {
        for (int val = 0; val < 256; val++) {
            if (occ[c][val] > 0) {
                double p = static_cast<double>(occ[c][val]) / pixelCount;
                entropy[c] -= p * log2(p);
            }
        }
    }

    return (entropy[0] + entropy[1] + entropy[2]) / 3.0;
}
```

```
double QuadTreeNode::calculateSSIM(const Image& img1, const Image& img2, int x1, int y1, int x2, int y2, int width, int height) {
    const double C1 = 6.5025;  // (0.01*255)^2
    const double C2 = 58.5225; // (0.03*255)^2
    const double C3 = C2 / 2;
    double sumX = 0.0, sumY = 0.0;
    double sumX2 = 0.0, sumY2 = 0.0;
    double sumXY = 0.0;
    int pixelCount = 0;

    int img1Width = img1.getWidth();
    int img1Height = img1.getHeight();
    int img2Width = img2.getWidth();
    int img2Height = img2.getHeight();

    for (int dy = 0; dy < height; dy++) {
        for (int dx = 0; dx < width; dx++) {
            int px1 = x1 + dx;
            int py1 = y1 + dy;
            int px2 = x2 + dx;
            int py2 = y2 + dy;


            if (px1 >= img1Width || py1 >= img1Height || px2 >= img2Width || py2 >= img2Height) {
                continue;
            }

            // Convert to luminance
            double l1 = 0.299 * img1.getPixel(px1, py1, 0) +
                        0.587 * img1.getPixel(px1, py1, 1) +
                        0.114 * img1.getPixel(px1, py1, 2);

            double l2 = 0.299 * img2.getPixel(px2, py2, 0) +
                        0.587 * img2.getPixel(px2, py2, 1) +
                        0.114 * img2.getPixel(px2, py2, 2);

            sumX += l1;
            sumY += l2;
            sumX2 += l1 * l1;
            sumY2 += l2 * l2;
            sumXY += l1 * l2;
            pixelCount++;
        }
    }

    if (pixelCount == 0) return 0.0;
```

```
// Calculate means
double muX = sumX / pixelCount;
double muY = sumY / pixelCount;

// Calculate variance and covariance
double sigmaX2 = (sumX2 / pixelCount) - (muX * muX);
double sigmaY2 = (sumY2 / pixelCount) - (muY * muY);
double sigmaXY = (sumXY / pixelCount) - (muX * muY);

// Calculate SSIM components
double luminance = (2 * muX * muY + C1) / (muX * muX + muY * muY + C1);
double contrast = (2 * sqrt(sigmaX2) * sqrt(sigmaY2) + C2) / (sigmaX2 + sigmaY2 + C2);
double structure = (sigmaXY + C3) / (sqrt(sigmaX2) * sqrt(sigmaY2) + C3);

return luminance * contrast * structure;
```

```cpp
void QuadTreeNode::compressWithSSIM(const Image& img, double threshold, int minBlockSize, bool targetOn) {

    // Calculate sub-block areas
    int subWidth1 = width / 2;
    int subWidth2 = width - subWidth1;
    int subHeight1 = height / 2;
    int subHeight2 = height - subHeight1;

    if (!targetOn) {
        // Stop if current block area is too small
        if (width * height <= minBlockSize) {
            calculateAverageColor(img);
            isLeaf = true;
            return;
        }

        // Stop if ANY sub-block would be too small
        if (subWidth1 * subHeight1 < minBlockSize ||
            subWidth1 * subHeight2 < minBlockSize ||
            subWidth2 * subHeight1 < minBlockSize ||
            subWidth2 * subHeight2 < minBlockSize) {
            calculateAverageColor(img);
            isLeaf = true;
            return;
        }
    }

    if (width * height <= 1) {
        calculateAverageColor(img);
        isLeaf = true;
        return;
    }

    // Stop if ANY sub-block would be too small
    if (subWidth1 * subHeight1 <= 1 ||
        subWidth1 * subHeight2 <= 1 ||
        subWidth2 * subHeight1 <= 1 ||
        subWidth2 * subHeight2 <= 1) {
        calculateAverageColor(img);
        isLeaf = true;
        return;
    }


    // Calculate the average color for the current node's region
    calculateAverageColor(img);
```

```
// Create a temporary image filled with the average color
Image temp(width, height);
for (int y_pos = 0; y_pos < height; y_pos++) {
    for (int x_pos = 0; x_pos < width; x_pos++) {
        temp.setPixel(x_pos, y_pos, 0, static_cast<int>(avgColor[0]));
        temp.setPixel(x_pos, y_pos, 1, static_cast<int>(avgColor[1]));
        temp.setPixel(x_pos, y_pos, 2, static_cast<int>(avgColor[2]));
    }
}

// Calculate SSIM between the original region and the temp image
double ssim = calculateSSIM(img, temp, x, y, 0, 0, width, height);

if (ssim >= threshold) {
    isLeaf = true;
} else {
    // Split into 4 children and compress them
    int halfWidth = width / 2;
    int halfHeight = height / 2;
    int remainingWidth = width - halfWidth;
    int remainingHeight = height - halfHeight;

    children[0] = new QuadTreeNode(x, y, halfWidth, halfHeight);
    children[1] = new QuadTreeNode(x + halfWidth, y, remainingWidth, halfHeight);
    children[2] = new QuadTreeNode(x, y + halfHeight, halfWidth, remainingHeight);
    children[3] = new QuadTreeNode(x + halfWidth, y + halfHeight, remainingWidth, remainingHeight);

    for (int i = 0; i < 4; i++) {
        if (children[i]) {
            children[i]->compressWithSSIM(img, threshold, minBlockSize, targetOn);
        }
    }
}
```

```cpp
void QuadTreeNode::calculateAverageColor(const Image& img) {
    avgColor = {0, 0, 0};
    int pixelCount = 0;
    int imgWidth = img.getWidth();
    int imgHeight = img.getHeight();

    for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
        for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
            avgColor[0] += img.getPixel(x_pos, y_pos, 0);
            avgColor[1] += img.getPixel(x_pos, y_pos, 1);
            avgColor[2] += img.getPixel(x_pos, y_pos, 2);
            pixelCount++;
        }
    }

    if (pixelCount > 0) {
        for (int k = 0; k < 3; k++) {
            avgColor[k] /= pixelCount;
        }
    }
}
```

```cpp
void QuadTreeNode::fillImage(Image& img) const { // Fill the image with the average color of this node
    if (isLeaf) {
        int imgWidth = img.getWidth();
        int imgHeight = img.getHeight();
        for (int y_pos = y; y_pos < y + height && y_pos < imgHeight; y_pos++) {
            for (int x_pos = x; x_pos < x + width && x_pos < imgWidth; x_pos++) {
                img.setPixel(x_pos, y_pos, 0, avgColor[0]);
                img.setPixel(x_pos, y_pos, 1, avgColor[1]);
                img.setPixel(x_pos, y_pos, 2, avgColor[2]);
            }
        }
    } else {
        for (int i = 0; i < 4; i++) {
            if (children[i]) {
                children[i]->fillImage(img);
            }
        }
    }
}
```

5. **File: QuadTree.hpp**

```hpp
#ifndef QUADTREE_HPP
#define QUADTREE_HPP

#include "Image.hpp"
#include "QuadTreeNode.hpp"

class QuadTree {
public:
    QuadTree(const Image& img, const int method, double threshold, int minSize, bool targetOn);

    void compressImage(const Image& img);
    void decompressImage(Image& img) const;
    bool saveImage(const string& filename) const;
    double getCompressionRatio(const string& inputFilename, const string& outputFilename) const;
    QuadTreeNode* getRoot() const { return root; }
    double getBestThreshold(const string& inputFilename, int method, double targetRatio);
    double getMaxThresholdForMethod(int method) const;

private:
    QuadTreeNode* root;
    int errorMethod;
    double threshold;
    int minBlockSize;
    int originalWidth;
    int originalHeight;
    bool targetOn;
    bool compressNow;
};

#endif // QUADTREE_HPP
```

6. **File: QuadTree.cpp**

```cpp
#include "QuadTree.hpp"
#include <fstream>
#include <filesystem>
#include <iostream>

QuadTree::QuadTree(const Image& img, int method, double threshold, int minSize, bool targetOn)
    : root(nullptr), errorMethod(method), threshold(threshold), minBlockSize(minSize),
      originalWidth(img.getWidth()), originalHeight(img.getHeight()), targetOn(targetOn) {
    if (minSize < 1) {
        throw invalid_argument("Minimum block size must be at least 1");
    }
    if (threshold < 0) {
        throw invalid_argument("Threshold must be non-negative");
    }

    root = new QuadTreeNode(0, 0, originalWidth, originalHeight);

    compressImage(img);
}

void QuadTree::compressImage(const Image& img) {
    if (root) {
        root->compress(img, errorMethod, threshold, minBlockSize, targetOn);
    }
}

void QuadTree::decompressImage(Image& img) const { // Fill the image with the average color of each node
    if (root) {
        root->fillImage(img);
    }
}

bool QuadTree::saveImage(const string& filename) const {
    if (!root) return false;

    Image decompressedImage(originalWidth, originalHeight);
    decompressImage(decompressedImage);

    return decompressedImage.save(filename);
}

double QuadTree::getCompressionRatio(const string& originalFile, const string& compressedFile) const {
    // Get the size of the original image and the compressed image
    size_t originalSize = 0;
    size_t compressedSize = 0;
```

```
    try {
        originalSize = filesystem::file_size(originalFile);
        compressedSize = filesystem::file_size(compressedFile);
    } catch (const filesystem::filesystem_error& e) {
        cerr << e.what() << endl;
        return 1.0;
    }

    if (compressedSize == 0) {
        cerr << "Error: Compressed file size is zero. Invalid compression result." << endl;
        return 1.0;
    }

    cout << "Original size: " << originalSize << " bytes" << endl;
    cout << "Compressed size: " << compressedSize << " bytes" << endl;
    cout << "Compression percentage: " << (1.0 - static_cast<double>(compressedSize)/originalSize) * 100.0 << "%" << endl;

    return (1.0 - static_cast<double>(compressedSize)/originalSize) * 100.0;
}

double QuadTree::getBestThreshold(const string& inputFilename, int method, double targetRatio) {
    if (targetRatio < 0 || targetRatio > 1) {
        throw invalid_argument("Target ratio must be between 0 and 1");
    }

    double low = 0.0;
    double high = getMaxThresholdForMethod(method);
    double bestThreshold = 0.0;
    double bestError = numeric_limits<double>::max();
    const double tolerance = 0.01;
    const int maxIterations = 15;
    int iteration = 0;


    while (iteration < maxIterations && (high - low) > tolerance) {
        try {
            Image img(inputFilename);
            size_t originalSize = filesystem::file_size(inputFilename);
            double mid = (low + high) / 2.0;
            string tempOutput = "temp_" + filesystem::path(inputFilename).filename().string();

            QuadTree quadTree(img, method, mid, 1, targetOn);
            quadTree.compressImage(img); // Compress the image with the current threshold

            if (!quadTree.saveImage(tempOutput)) {
                throw runtime_error("Failed to save temporary image.");
```

```
        }

        size_t compressedSize = filesystem::file_size(tempOutput);
        double currentRatio = static_cast<double>(compressedSize) / originalSize;
        double currentError = abs(currentRatio - targetRatio);

        if (currentError < bestError) {
            bestError = currentError;
            bestThreshold = mid;
        }

        if (method == 5) { // SSIM case
            if (currentRatio < targetRatio) {
                low = mid;
            } else {
                high = mid;
            }
        } else { // Other methods
            if (currentRatio < targetRatio) {
                high = mid;
            } else {
                low = mid;
            }
        }

        iteration++;
        filesystem::remove(tempOutput); // Clean up temporary file
    }
    catch (const exception& e) {
        cerr << "Error in iteration " << iteration << ": " << e.what() << std::endl;
    }
 }
return bestThreshold;
```

```
double QuadTree::getMaxThresholdForMethod(int method) const {
    switch(method) {
        case 1: return 16256.25;   // Variance
        case 2: return 127.5;     // MAD
        case 3: return 255.0;     // Max Difference
        case 4: return 8.0;       // Entropy
        case 5: return 1.0;       // SSIM
        default: throw invalid_argument("Invalid method for threshold calculation");
    }
}
```

7. **main.cpp**

```cpp
#include <iostream>
#include <string>
#include <chrono>
#include <fstream>
#include "Image.hpp"
#include "QuadTree.hpp"

using namespace std;
using namespace chrono;

int main() {

    cout << "\n"
    << "========================================\n"
    << "       QuadTree Image Compression      \n"
    << "========================================\n"
    << endl;
    try {
        string filename;
        double threshold;
        int method;
        int minBlockSize;

        cout << "Enter the absolute path of the image: ";
        cin >> filename;

        // Check if the file exists
        ifstream fileCheck(filename);
        if (!fileCheck) {
            cerr << "Error: File not found. Please check the path and try again.\n";
            return 1;
        }
        fileCheck.close();
```

```
do {
    cout << "Select the error calculation method:" << endl;
    cout << "1. Variance" << endl;
    cout << "2. MAD (Mean Absolute Deviation)" << endl;
    cout << "3. Max Difference" << endl;
    cout << "4. Entropy" << endl;
    cout << "5. SSIM (Structural Similarity Index)" << endl;
    cout << "Enter your choice (1-5): ";

    double tempMethod;
    cin >> tempMethod;

    if (cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Invalid input. Please enter a number.\n";
    } else if (tempMethod < 1 || tempMethod > 5) {
        cout << "Invalid input. Please enter a number between 1 and 5.\n";
    } else if (tempMethod != floor(tempMethod)) {
        cout << "Invalid input. Please enter an integer.\n";
    } else {
        method = static_cast<int>(tempMethod);
        break;
    }
} while (true);

double targetCompression;
bool targetOn;
```

```
do {
    cout << "Enter the target compression ratio (0 for no target): ";
    cin >> targetCompression;

    if (cin.fail() || targetCompression < 0 || targetCompression > 1) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "Invalid input. Please enter a value between 0 and 1.\n";
    } else {
    break;
    }
} while (true);

targetOn = (targetCompression != 0);

if (!targetOn) {
    bool validThreshold = false;
    do {
    cout << "Enter the compression threshold: ";
    cin >> threshold;

    if (cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Invalid input. Please enter a numeric value.\n";
        continue;
    }

    switch (method) {
        case 1: // Variance
        if (threshold >= 0 && threshold <= 16256.25) validThreshold = true;
        else cout << "Threshold must be between 0 and 16256.25.\n";
        case 2: // MAD
        if (threshold >= 0 && threshold <= 127.5) validThreshold = true;
        else cout << "Threshold must be between 0 and 127.5.\n";
        break;

        case 3: // Max Difference
        if (threshold >= 0 && threshold <= 255) validThreshold = true;
        else cout << "Max Difference threshold must be between 0 and 255.\n";
        break;
        case 4: // Entropy
        if (threshold >= 0 && threshold <= 8) validThreshold = true;
        else cout << "Entropy threshold must be between 0 and 8.\n";
        break;
```

```
        case 5: // SSIM
        if (threshold >= 0 && threshold <= 1) validThreshold = true;
        else cout << "SSIM threshold must be between 0 and 1.\n";
        break;
    }

    } while (!validThreshold);

    do {
        cout << "Enter the minimum block size (integer): ";
        double tempBlockSize;
        cin >> tempBlockSize;

        if (cin.fail()) {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout << "Invalid input. Please enter a number.\n";
        } else if (tempBlockSize < 1) {
            cout << "Block size must be at least 1.\n";
        } else if (tempBlockSize != floor(tempBlockSize)) {
            cout << "Block size must be an integer.\n";
        } else {
            minBlockSize = static_cast<int>(tempBlockSize);
            break;
        }
    } while (true);
    cout << endl;
}


// Start
auto start = high_resolution_clock::now();

// Load the image
Image img(filename);

// Create and compress the image using QuadTree
QuadTree quadTree(img, method, 0, 1, targetOn);
if (targetOn) { // if target compression is on
    QuadTree quadTreeTemp(img, method, 0, 1, targetOn);
    double bestThreshold = quadTreeTemp.getBestThreshold(filename, method, 1-targetCompression);
    quadTree = QuadTree(img, method, bestThreshold, 1, targetOn);
```

```
}
else {
    quadTree = QuadTree(img, method, threshold, minBlockSize, targetOn);
    quadTree.compressImage(img);
}
```

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

    // Save the compressed image
    string outputFilename;
    bool validPath = false;

    do {
        cout << "Enter the absolute path to save the compressed image: ";
        cin >> outputFilename;

        // Validate the path
        if (outputFilename.empty()) {
        cout << "Invalid path. Please enter a valid absolute path.\n";
        } else {
        validPath = true;
        }
    } while (!validPath);

    if (quadTree.saveImage(outputFilename)) {
        cout << "Compressed image saved in: " << outputFilename << endl;
    } else {
        cerr << "Failed to save the compressed image." << endl;
    }

    // Display compression ratio
    quadTree.getCompressionRatio(filename, outputFilename);

    cout << "Total nodes: " << quadTree.getRoot()->countTotalNodes() << endl;
    cout << "Depth of the QuadTree: " << quadTree.getRoot()->depth() << endl;

    cout << "Execution time: " << duration.count() << " ms" << endl;

} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
    return 1;
}

return 0;
```

BAB III
SCREENSHOT HASIL TEST

Pada program ini, output terletak pada folder test.

1. Input: flower.jpg, threshold = 50, minBlockSize = 8, method = variance

   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\flower.jpg
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 1
Enter the target compression ratio (0 for no target): 0
Enter the compression threshold: 50
Enter the minimum block size: 8

Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compressed.jpg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compressed.jpg
Original size: 14195 bytes
Compressed size: 17566 bytes
Compression percentage: -23.7478%
Total nodes: 3597
Depth of the QuadTree: 6
Execution time: 133 ms
```

Original

Terkompresi

2. Input: cewek.jpg, threshold = 5, minBlockSize = 4, method = MAD

   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\cewek.jpg
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 2
Enter the target compression ratio (0 for no target): 0
Enter the compression threshold: 150
Threshold must be between 0 and 127.5.
Enter the compression threshold: 5
Enter the minimum block size: 0
Invalid input. Please enter block size larger or equal to 1.
Enter the minimum block size: 4

Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\cewek_compressed.jpg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\cewek_compressed.jpg
Original size: 34564 bytes
Compressed size: 32064 bytes
Compression percentage: 7.23296%
Total nodes: 9757
Depth of the QuadTree: 7
Execution time: 161 ms
```

Original

Terkompresi

3. Input: castle.jpeg, threshold: 0, minBlockSize = 1, method: Max Difference

   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\castle.jpeg
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 3
Enter the target compression ratio (0 for no target): 0
Enter the compression threshold: a
Invalid input. Please enter a numeric value.
Enter the compression threshold: 256
Max Difference threshold must be between 0 and 255.
Enter the compression threshold: 0
Enter the minimum block size: 1

Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle_compressed.jpeg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle_compressed.jpeg
Original size: 2949469 bytes
Compressed size: 3215463 bytes
Compression percentage: -9.01837%
Total nodes: 5436269
Depth of the QuadTree: 11
Execution time: 23946 ms
```

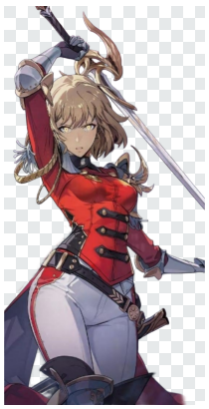
Original

Compressed

4. Input: haein.png, threshold = 3, minBlockSize = 16, method: entropy
   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\haein_og.png
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 4
Enter the target compression ratio (0 for no target): 0
Enter the compression threshold: 9
Entropy threshold must be between 0 and 8.
Enter the compression threshold: 3
Enter the minimum block size: f
Invalid input. Please enter block size larger or equal to 1.
Enter the minimum block size: 16

Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\haein_og_compressed.png
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\haein_og_compressed.png
Original size: 107030 bytes
Compressed size: 13426 bytes
Compression percentage: 87.4559%
Total nodes: 3005
Depth of the QuadTree: 6
Execution time: 104 ms
```



5. Input: linux.png, threshold: 0.81, minBlockSize = 5, method: ssim

Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\linux.png
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 5
Enter the target compression ratio (0 for no target): 0
Enter the compression threshold: 0.81
Enter the minimum block size (integer): 5

Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\linux_compressed.png
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\linux_compressed.png
Original size: 41236 bytes
Compressed size: 19574 bytes
Compression percentage: 52.5318%
Total nodes: 6409
Depth of the QuadTree: 7
Execution time: 237 ms
```

Original

Terkompresi

6. Input: castle.jpeg, dynamic threshold (35% compression), method = variance
   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\castle.jpeg
Select the error calculation method:
1. Variance
2. MAD (Mean Absolute Deviation)
3. Max Difference
4. Entropy
5. SSIM (Structural Similarity Index)
Enter your choice (1-5): 1
Enter the target compression ratio (0 for no target): 0.35
Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle_compressed_dynamic.jpeg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle_compressed_dynamic.jpeg
Original size: 2949469 bytes
Compressed size: 1917080 bytes
Compression percentage: 35.0025%
Total nodes: 443337
Depth of the QuadTree: 11
Execution time: 515577 ms
```

Original

Terkompresi (35%)

7. Input: path input/output salah
   Output:

```
Enter the absolute path of the image: C:\Users\YOGA\Downloads\cewekuhuy.jpg
Error: File not found. Please check the path and try again.
```

```
Enter the absolute path to save the compressed image: C:\Users\YOGA\Downloads\cewek
Error saving image: Please check your path and extension.
Failed to save the compressed image.
filesystem error: cannot get file size: No such file or directory [C:\Users\YOGA\Downloads\cewek]
Total nodes: 21289
Depth of the QuadTree: 7
Execution time: 429 ms
```

8. Input: perbandingan 5 cara untuk tingkat kompresi 20%

   Output:
   Metode variance

```
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_1.jpg
Original size: 14195 bytes
Compressed size: 11366 bytes
Compression percentage: 19.9296%
Total nodes: 2829
Depth of the QuadTree: 7
Execution time: 978 ms
```

   Metode MAD

```
Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_2.jpg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_2.jpg
Original size: 14195 bytes
Compressed size: 11482 bytes
Compression percentage: 19.1124%
Total nodes: 3165
Depth of the QuadTree: 7
Execution time: 398 ms
```

   Metode Max Difference

```
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_3.jpg
Original size: 14195 bytes
Compressed size: 11322 bytes
Compression percentage: 20.2395%
Total nodes: 929
Depth of the QuadTree: 7
Execution time: 563 ms
```

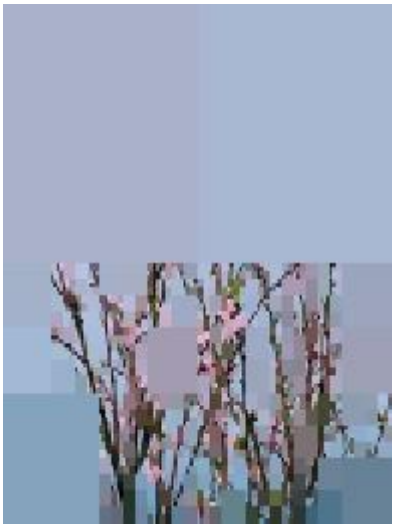
Metode entropy

```
Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_4.jpg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_4.jpg
Original size: 14195 bytes
Compressed size: 11346 bytes
Compression percentage: 20.0704%
Total nodes: 897
Depth of the QuadTree: 6
Execution time: 481 ms
```

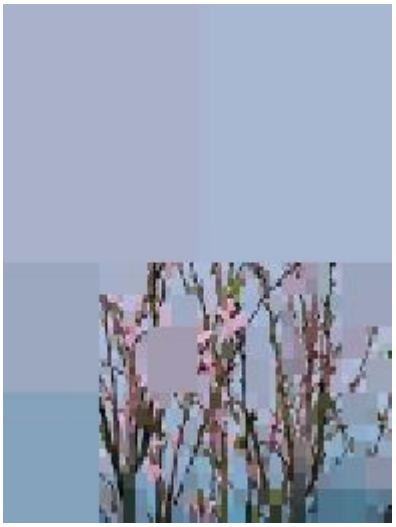
Metode SSIM

```
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\flower_compare_5.jpg
Original size: 14195 bytes
Compressed size: 11532 bytes
Compression percentage: 18.7601%
Total nodes: 3869
Depth of the QuadTree: 7
Execution time: 533 ms
```

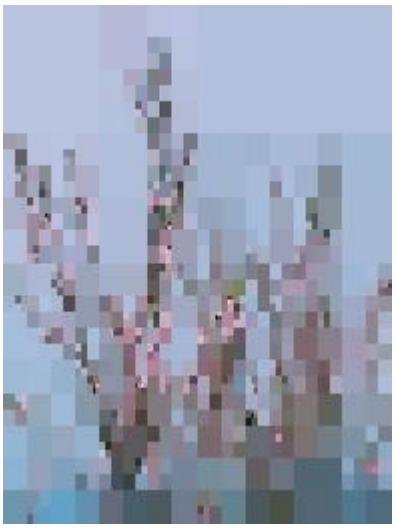
Metode variance



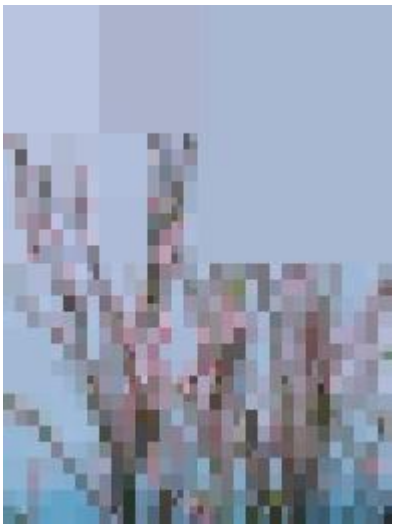
Metode MAD



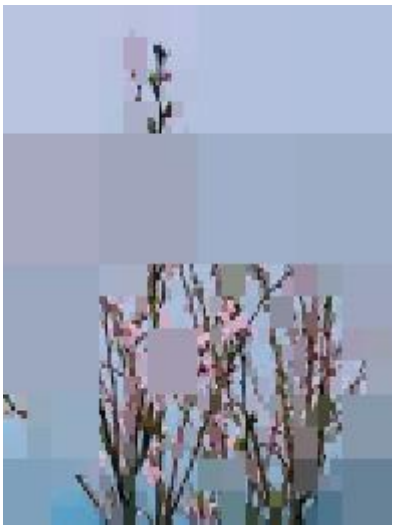
Metode Max Difference



Metode entropy



Metode SSIM



9. Input: castle_5000.jpeg(5000x5000), dynamic threshold (40% compression), method: variance

```
Enter the target compression ratio (0 for no target): 0.4
Enter the absolute path to save the compressed image: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle5000_compressed.jpeg
Compressed image saved in: C:\Users\YOGA\Documents\Tucil2_13523108\test\castle5000_compressed.jpeg
Original size: 3939983 bytes
Compressed size: 2364167 bytes
Compression percentage: 39.9955%
Total nodes: 341729
Depth of the QuadTree: 11
Execution time: 433916 ms
```

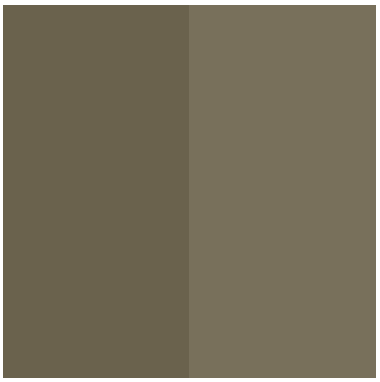
Original

Terkompresi (40%)

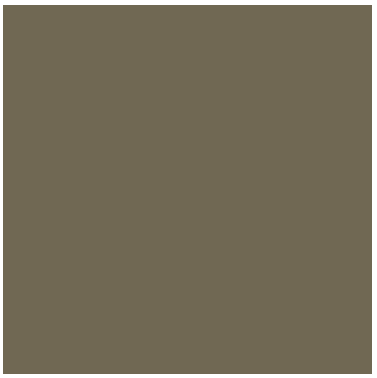
10. Input: small.jpg (300 bytes)

    Output:

```
Enter the absolute path to save the compressed image: C:\Users\YOGA\Downloads\Tucil2_13523108\test\small_compressed.jpg
Compressed image saved in: C:\Users\YOGA\Downloads\Tucil2_13523108\test\small_compressed.jpg
Original size: 300 bytes
Compressed size: 617 bytes
Compression percentage: -105.667%
Total nodes: 1
Depth of the QuadTree: 0
Execution time: 0 ms
```



Original



Terkompresi

BAB IV
ANALISIS KOMPRESI QUAD TREE

A. Kompleksitas waktu

a. Worst case

Ukuran gambar N = w x h. Misalkan w = h = 2^L. Maka kedalaman maksimum adalah $L = log_2(w)$. Jumlah node pada pohon dengan asumsi pohon lengkap yaitu $S = 4^0 + 4^1 + \ldots + 4^L$. Didapatkan $S = \frac{4^{L+1}-1}{4-1} = \frac{4^{L+1}-1}{3} = \frac{4N-1}{3} = O(N)$.

Dalam kasus terburuk, setiap pixel akan menjadi daun. Pada setiap level l, setiap level l, blok akan berukuran $\frac{N}{4^l}$ dan ada 4^l blok. Setiap pixel harus dioperasikan, sehingga akan ada sebanyak $\frac{N}{4^l}$ operasi untuk menghitung *average color* dan error. Terdapat $\frac{2N}{4^l} x 4^l = 2N$ operasi untuk tiap level. Total operasi pada semua level adalah $T(N) = 2N x log_2(w) = O(N\ log\ N)$.

b. Best case

Pada kasus ini, kedalaman = 0 dan hanya ada 1 node sehingga $T(N) = 2N = O(N)$.

c. Average case

Asumsi sebagian besar blok berhenti membelah lebih awal, sebagian kecil area membelah hingga ukuran minimum, dan N = w x h dengan w = h.

Pada area seragam, Terdapat $\frac{\alpha N}{w^2}$ blok dan setiap blok operasi yang dilakukan sebesar $O(N)$. Kompleksitas waktu yang dilakukan adalah $O(\alpha N)$.

Pada area tidak seragam,kompleksitas waktu adalah $O((1 - \alpha)N))$. Sehingga kompleksitas waktu secara total adalah $O(N)$.

B. Kompleksitas ruang

a. Worst case

Kompleksitas ruang terburuk dengan asumsi setiap node berukuran 1 yaitu $O(N)$.

b. Best case

Kompleksitas ruang terbaik yaitu dibangkitkan 1 node $O(1)$

c. Average case

Pada kasus rata-rata (average case), diasumsikan bahwa sebagian besar area gambar bersifat relatif seragam, sehingga pembelahan pada pohon quadtree tidak dilakukan hingga ke ukuran piksel. Hanya sebagian kecil area gambar (misalnya, tepi objek atau detail tinggi) yang memerlukan pembelahan hingga ukuran minimum. Total jumlah node pada quadtree dilambangkan dengan K, dan karena tidak semua area dibelah hingga tingkat maksimal, maka K$\ll$N.

Jika rata-rata ukuran blok daun (leaf block) adalah $m \times m$, maka jumlah daun $\frac{N}{m^2}$. Kompleksitas ruangnya adalah $O(K)$ dengan K<<N dan $K \sim O(\frac{N}{m^2})$.

Jika rata-rata blok besar (nilai $m$xm besar), maka jumlah node sedikit, dan ruang yang dibutuhkan semakin kecil. Sebaliknya, jika gambar memiliki banyak detail (nilai $m$xm kecil), maka jumlah daun meningkat, dan kompleksitas ruang mendekati $O(N)$ sebagai kasus terburuk.

LINK REPOSITORY

https://github.com/henry204xx/Tucil2_13523108

CHECKLIST

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	✓	