LAPORAN TUGAS KECIL 2
IF2211 STRATEGI ALGORITMA

KOMPRESI GAMBAR DENGAN METODE QUADTREE

Oleh:

Darrel Adinarya Sunanda
13523061

PROGRAM STUDI TEKNIK INFORMATIKA

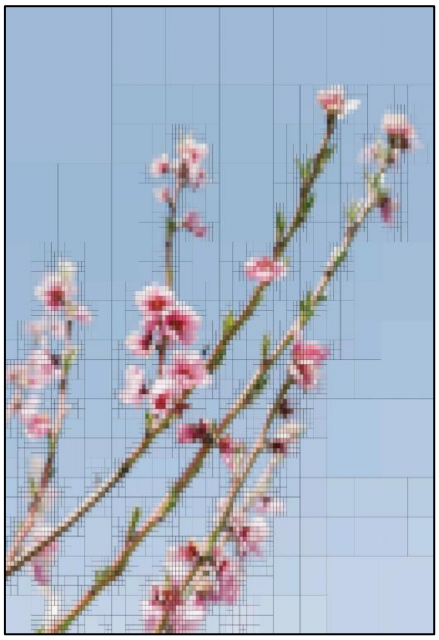
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

BAB I
Deskripsi Program



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

Quadtree adalah struktur data hierarkis yang digunakan untuk membagi ruang atau data menjadi bagian yang lebih kecil, yang sering digunakan dalam pengolahan gambar. Dalam konteks kompresi gam bar, 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.

(Dikutip dari: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/Tucil2-Stima-2025.pdf)

BAB II
Algoritma Divide and Conquer

Dalam program ini, algoritma **divide and conquer** diterapkan melalui kelas Quadtree untuk memproses citra (gambar). Citra dibagi menjadi blok-blok kuadran lebih kecil secara rekursif hingga memenuhi kriteria tertentu (berdasarkan variasi piksel).

Adapun langkah-langkah yang digunakan dalam algoritma ini adalah sebagai berikut:

1. Inisialisasi akar quadtree:
    - Di dalam Quadtree::Compress(), citra diwakili sebagai blok persegi besar (dari (0,0) hingga (width-1,height-1)).

2. Pengecekan kondisi berhenti:
    - Di konstruktor Quadtree::Quadtree, dilakukan pemeriksaan apakah blok terlalu bervariasi menggunakan fungsi getError() dan dibandingkan dengan nilai ambang threshold.

3. Jika error > threshold dan ukurannya > minBlockSize, maka:
    - Blok dibagi menjadi empat sub-blok (kuadran).
    - Fungsi memanggil dirinya sendiri secara rekursif untuk masing-masing kuadran **(inilah bagian "divide and conquer"-nya**).
    - Jika tidak, blok akan distandarkan (dinormalisasi) ke nilai rata-rata warnanya dengan fungsi image->normalize(...).

4. Pemilihan metode error:
    - Pemilihan metode error (Variance, Mean Absolute Deviation, Max Pixel Difference, Entropy) ditentukan oleh Quadtree::method dan dipanggil di getError().

Misalkan kita punya citra 256x256 piksel. Algoritma akan:

1. Periksa seluruh blok.
2. Jika terlalu beragam, blok akan dibagi menjadi 4 blok 128x128.
3. Setiap blok diuji lagi. Jika masih terlalu beragam, aka dibagi lagi menjadi 4 blok 64x64.
4. Begitu seterusnya, hingga blok cukup "homogen" atau terlalu kecil untuk dibagi lebih lanjut.

Dengan membagi gambar menjadi kuadran dan hanya memproses bagian yang kompleks, algoritma ini mampu mengurangi ukuran gambar secara signifikan tanpa mengorbankan detail penting. Pendekatan ini sangat cocok untuk gambar dengan area homogen yang luas, dan tetap mempertahankan detail pada bagian yang kompleks.

BAB III
Source Code

Image.h

```
#ifndef IMAGE_H
#define IMAGE_H

#include <string>
#include <fstream>

class Image {
    int width, height, channels;
    unsigned char*** data;
    long long*** sumTable;

public:
    static std::streamsize sizeBefore;
    static std::streamsize sizeAfter;

    explicit Image(const std::string& pathString); // Constructor
    Image(const Image&); // Copy constructor
    ~Image(); // Destructor
    Image& operator=(const Image&); // Dirty assignment operator

    bool isLoaded() const; // Checks if the image is successfully loaded
    void save(const std::string& pathString) const; // Saves the image to a file
    static std::streamsize getFileSize(const std::string& filename); // Returns the size of
the file

    int getWidth() const; // Returns the width of the image
    int getHeight() const; // Returns the height of the image

    double* getMean(int x1, int y1, int x2, int y2) const; // Returns the mean of pixel values
in a region
    double getVariance(int x1, int y1, int x2, int y2) const; // Returns the variance of pixel
values in a region
    double getMAD(int x1, int y1, int x2, int y2) const; // Returns the mean absolute
deviation of pixel values in a region
    double getMPD(int x1, int y1, int x2, int y2) const; // Returns the max pixel difference
of pixel values in a region
    double getEntropy(int x1, int y1, int x2, int y2) const; // Returns the entropy of pixel
values in a region

    void normalize(int x1, int y1, int x2, int y2) const; // Normalizes pixel values in a
region
};

#endif // IMAGE_H
```

Image.cpp

```cpp
#include "Image.h"
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

using namespace std;

streamsize Image::sizeBefore = 0;
streamsize Image::sizeAfter = 0;

Image::Image(const string &pathString): width(0), height(0), channels(0)
// Parameterized constructor
{
    // Convert string to char* (stbi_load requires char*)
    char* path = new char [pathString.length() + 1];
    strcpy(path, pathString.c_str());

    // Load the image (returns an unsigned char* to pixel data)
    unsigned char* rawImg = stbi_load(path, &width, &height, &channels, 0);
    delete[] path;
    // Check if the image was loaded successfully
    if (rawImg == nullptr) {
        return;
    } // If loading fails, return

    // Allocate memory for the image data
    data = new unsigned char**[height];
    for (int i = 0; i < height; ++i) {
        data[i] = new unsigned char*[width];
        for (int j = 0; j < width; ++j) {
            data[i][j] = new unsigned char[channels];
        }
    }
    // Allocate memory for the sum table
    sumTable = new long long**[height];
    for (int i = 0; i < height; ++i) {
        sumTable[i] = new long long*[width];
        for (int j = 0; j < width; ++j) {
            sumTable[i][j] = new long long[channels];
            for (int k = 0; k < channels; ++k) {
                sumTable[i][j][k] = 0;
            }
        }
    }
    // Fill the image data
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            for (int k = 0; k < channels; ++k) {
                // Fill out the image data
                data[i][j][k] = rawImg[(i * width + j) * channels + k];

                // Fill out the Sum Table
                sumTable[i][j][k] = data[i][j][k];
                if (i > 0) { sumTable[i][j][k] += sumTable[i - 1][j][k]; }
                if (j > 0) { sumTable[i][j][k] += sumTable[i][j - 1][k]; }
                if (i > 0 && j > 0) { sumTable[i][j][k] -= sumTable[i - 1][j - 1][k]; }
            }
        }
    }

    // Free the raw image data
    stbi_image_free(rawImg);
    // Note the size of the image before compression
    sizeBefore = getFileSize(pathString);
}

...
```

Image.cpp

```
Image::Image(const Image &img) : width(img.width), height(img.height), channels(img.channels)
// Copy constructor
{
    // Allocate memory for the image data
    data = new unsigned char**[height];
    for (int i = 0; i < height; ++i) {
        data[i] = new unsigned char*[width];
        for (int j = 0; j < width; ++j) {
            data[i][j] = new unsigned char[channels];
        }
    }

    // Allocate memory for the sum table
    sumTable = new long long**[height];
    for (int i = 0; i < height; ++i) {
        sumTable[i] = new long long*[width];
        for (int j = 0; j < width; ++j) {
            sumTable[i][j] = new long long[channels];
            for (int k = 0; k < channels; ++k) {
                sumTable[i][j][k] = 0;
            }
        }
    }

    // Copy the Image data
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            for (int k = 0; k < channels; ++k) {
                data[i][j][k] = img.data[i][j][k];
                sumTable[i][j][k] = img.sumTable[i][j][k];
            }
        }
    }
}

Image::~Image() { // Destructor
    if (data) {
        for (int i = 0; i < height; ++i) {
            if (data[i]) {
                for (int j = 0; j < width; ++j) {
                    delete[] data[i][j];
                }
                delete[] data[i];
            }
        }
        delete[] data;
    }
}

Image& Image::operator=(const Image &img) { // Dirty assignment operator
    if (this != &img & this->width == img.width && this->height == img.height && this-
>channels == img.channels) {
        // Copy pixel data
        for (int i = 0; i < height; ++i) {
            for (int j = 0; j < width; ++j) {
                for (int k = 0; k < channels; ++k) {
                    data[i][j][k] = img.data[i][j][k];
                }
            }
        }
    }
    return *this;
}

bool Image::isLoaded() const {
    return data != nullptr && width > 0 && height > 0 && channels > 0;
}

...
```

Image.cpp

```cpp
void Image::save(const string &pathString) const {
    // Convert string to char* (stbi_write_png requires char*)
    char* path = new char[pathString.length() + 1];
    strcpy(path, pathString.c_str());

    // Revert the image data to a single array for saving
    auto* rawImg = new unsigned char[width * height * channels];
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            for (int k = 0; k < channels; ++k) {
                rawImg[(i * width + j) * channels + k] = data[i][j][k];
            }
        }
    }

    stbi_write_png(path, width, height, channels, rawImg, width * channels);

    delete[] path;
    delete[] rawImg;

    // Note the size of the image after compression
    sizeAfter = getFileSize(pathString);
}

streamsize Image::getFileSize(const string& filename) {
    ifstream file(filename, ios::binary | ios::ate);
    return file.tellg();
}

int Image::getWidth() const {
    return width;
}

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

double* Image::getMean(const int x1, const int y1, const int x2, const int y2) const {
    auto* mean = new double[channels];
    const int area = (x2 - x1 + 1) * (y2 - y1 + 1);

    for (int k = 0; k < channels; k++) {
        long long sum = sumTable[y2][x2][k];
        if (x1 > 0) sum -= sumTable[y2][x1 - 1][k];
        if (y1 > 0) sum -= sumTable[y1 - 1][x2][k];
        if (x1 > 0 && y1 > 0) sum += sumTable[y1 - 1][x1 - 1][k];

        mean[k] = static_cast<double>(sum) / area;
    }
    return mean;
}

...
```

Image.cpp

```cpp
double Image::getVariance(const int x1, const int y1, const int x2, const int y2) const {
    const double* mean = getMean(x1, y1, x2, y2);

    auto* variances = new double[channels];
    for (int k = 0; k < channels; ++k) {
        variances[k] = 0;
    }

    for (int i = y1; i <= y2; ++i) {
        for (int j = x1; j <= x2; ++j) {
            for (int k = 0; k < channels; ++k) {
                const double diff = data[i][j][k] - mean[k];
                variances[k] += diff * diff;
            }
        }
    }

    for (int k = 0; k < channels; ++k) {
        variances[k] /= (x2 - x1 + 1) * (y2 - y1 + 1);
    }

    double variance = 0;
    for (int k = 0; k < channels; ++k) {
        variance += variances[k];
    }

    delete[] mean;
    delete[] variances;

    return variance / channels;
}

double Image::getMAD(const int x1, const int y1, const int x2, const int y2) const {
    const double* mean = getMean(x1, y1, x2, y2);

    auto* deviations = new double[channels];
    for (int k = 0; k < channels; ++k) {
        deviations[k] = 0;
    }

    for (int i = y1; i <= y2; ++i) {
        for (int j = x1; j <= x2; ++j) {
            for (int k = 0; k < channels; ++k) {
                deviations[k] += abs(data[i][j][k] - mean[k]) ;
            }
        }
    }

    for (int k = 0; k < channels; ++k) {
        deviations[k] /= (x2 - x1 + 1) * (y2 - y1 + 1);
    }

    double deviation = 0;
    for (int k = 0; k < channels; ++k) {
        deviation += deviations[k];
    }

    delete[] mean;
    delete[] deviations;

    return deviation / channels;
}

...
```

Image.cpp

```cpp
double Image::getMPD(const int x1, const int y1, const int x2, const int y2) const {
    int* maxes = new int[channels];
    int* mins = new int[channels];

    for (int k = 0; k < channels; ++k) {
        maxes[k] = 0;
        mins[k] = 255;
    }

    for (int i = y1; i <= y2; ++i) {
        for (int j = x1; j <= x2; ++j) {
            for (int k = 0; k < channels; ++k) {
                if (data[i][j][k] > maxes[k]) {
                    maxes[k] = data[i][j][k];
                }
                if (data[i][j][k] < mins[k]) {
                    mins[k] = data[i][j][k];
                }
            }
        }
    }

    double diff = 0;
    for (int k = 0; k < channels; ++k) {
        diff += maxes[k] - mins[k];
    }

    delete[] maxes;
    delete[] mins;

    return diff / channels;
}

double Image::getEntropy(const int x1, const int y1, const int x2, const int y2) const {
    int **histograms = new int*[channels];
    for (int k = 0; k < channels; ++k) {
        histograms[k] = new int[256]{0};
    }

    for (int i = y1; i <= y2; ++i) {
        for (int j = x1; j <= x2; ++j) {
            for (int k = 0; k < channels; ++k) {
                histograms[k][data[i][j][k]]++;
            }
        }
    }

    auto* entropies = new double[channels];
    const int width = x2 - x1 + 1;
    const int height = y2 - y1 + 1;
    for (int k = 0; k < channels; ++k) {
        entropies[k] = 0;
        for (int i = 0; i < 256; ++i) {
            if (histograms[k][i] > 0) {
                const double p = static_cast<double>(histograms[k][i]) / (width * height);
                entropies[k] -= p * log2(p);
            }
        }
    }

    double entropy = 0;
    for (int k = 0; k < channels; ++k) {
        entropy += entropies[k];
    }

    delete[] entropies;
    for (int k = 0; k < channels; ++k) {
        delete[] histograms[k];
    }
    delete[] histograms;

    return entropy / channels;
}

void Image::normalize(const int x1, const int y1, const int x2, const int y2) const {
    const double* mean = getMean(x1, y1, x2, y2);

    for (int i = y1; i <= y2; ++i) {
        for (int j = x1; j <= x2; ++j) {
            for (int k = 0; k < channels; ++k) {
                data[i][j][k] = static_cast<unsigned char>(mean[k]);
            }
        }
    }
    delete[] mean;
}
```

Quadtree.h

```h
#ifndef QUADTREE_H
#define QUADTREE_H

#include "Image.h"

class Quadtree {
    int x1, y1, x2, y2;
    Quadtree* children[4] = {};

    static int method;
    static int threshold;
    static int minBlockSize;
    static Image* image;

public:
    static int maxDepth;
    static int nodes;

    Quadtree(int x1, int y1, int x2, int y2, int depth); // Constructor
    ~Quadtree(); // Destructor

    double getError() const; // Calculates the error for the current node with the given
method
    static void setMethod(int m); // Sets the method for error calculation
    static void setImage(Image& img); // Sets the image for the quadtree
    static void setThreshold(int value); // Sets the threshold value
    static void setMinBlockSize(int size); // Sets the minimum block size
    static void Compress(); // Starts the compression process
};

#endif // QUADTREE_H
```

Quadtree.cpp

```cpp
#include "Quadtree.h"

using namespace std;

int Quadtree::method = 0;
int Quadtree::minBlockSize = 2;
int Quadtree::threshold = 0;
Image* Quadtree::image = nullptr;

int Quadtree::maxDepth = 0;
int Quadtree::nodes = 0;

Quadtree::Quadtree(const int x1, const int y1, const int x2, const int y2, const int depth) :
x1(x1), y1(y1), x2(x2), y2(y2) { // NOLINT(*-no-recursion)
    const double error = getError();

    if (error > threshold && (x2 - x1) * (y2 - y1) > minBlockSize) {
        const int midX = (x1 + x2) / 2;
        const int midY = (y1 + y2) / 2;

        children[0] = new Quadtree(x1, y1, midX, midY, depth + 1);
        children[1] = new Quadtree(midX, y1, x2, midY, depth + 1);
        children[2] = new Quadtree(x1, midY, midX, y2, depth + 1);
        children[3] = new Quadtree(midX, midY, x2, y2, depth + 1);
    }
    else {
        image->normalize(x1, y1, x2, y2);
        maxDepth = max(maxDepth, depth);
    }
    nodes++;
}

Quadtree::~Quadtree() {
    for (auto & i : children) {
        delete i;
    }
}

double Quadtree::getError() const {
    switch (method) {
        case 1:
            return image->getVariance(x1, y1, x2, y2);
        case 2:
            return image->getMAD(x1, y1, x2, y2);
        case 3:
            return image->getMPD(x1, y1, x2, y2);
        case 4:
            return image->getEntropy(x1, y1, x2, y2);
        default:
            return 0;
    }
}

void Quadtree::setMethod(const int m) {
    method = m;
}

void Quadtree::setImage(Image& img) {
    image = &img;
}

void Quadtree::setThreshold(const int t) {
    threshold = t;
}

void Quadtree::setMinBlockSize(const int m) {
    minBlockSize = m;
}

void Quadtree::Compress() {
    Quadtree compressor(0, 0, image->getWidth() - 1, image->getHeight() - 1, 1);
}
```

Main.cpp

```cpp
#include <iostream>
#include <windows.h>
#include <chrono>

#include "Image.h"
#include "Quadtree.h"

using namespace std;

void miku()
{
    SetConsoleOutputCP(CP_UTF8);
    cout <<u8R"(
```

```
)";
}

...
```

Main.cpp

```cpp
double compress(const string& outputPath, int &threshold, const double targetRatio, Image
&image,
    chrono::steady_clock::time_point *start, chrono::steady_clock::time_point *end) {
    double ratio;
    if (targetRatio > 0) {
        int left = 1, right = 255;
        const Image original(image);
        Quadtree::setThreshold(threshold);

        // Binary search for the threshold
        *start = chrono::steady_clock::now();
        // First iteration
        image = original;
        Quadtree::Compress();
        image.save(outputPath);
        ratio = static_cast<double>
        (Image::sizeAfter) / static_cast<double>(Image::sizeBefore);
        cout << "/";

        while (ratio > targetRatio) { // Get the initial range
            right += 100;
            left = threshold;
            Quadtree::setThreshold(threshold = right);
            image = original;
            Quadtree::Compress();
            image.save(outputPath);

            ratio = static_cast<double>
            (Image::sizeAfter) / static_cast<double>(Image::sizeBefore);
            cout << "/";
        }
        while (left != right) { // Binary search
            threshold = (left + right) / 2;
            Quadtree::setThreshold(threshold);
            image = original;
            Quadtree::Compress();
            image.save(outputPath);

            ratio = static_cast<double>
            (Image::sizeAfter) / static_cast<double>(Image::sizeBefore);
            cout << "/";

            if (ratio > targetRatio) { left = left == threshold ? right : threshold; }
            else { right = threshold; }
        }
        *end = chrono::steady_clock::now();
        cout << endl << "Final threshold: " << threshold << endl;
    } else {
        *start = chrono::steady_clock::now();
        Quadtree::Compress();
        *end = chrono::steady_clock::now();
        image.save(outputPath);
        ratio = static_cast<double>(Image::sizeAfter) / static_cast<double>
(Image::sizeBefore);
    }
    return ratio;
}

void openOut(const string& outputPath) {
    // Open the output image using the default image viewer
    char fullPath[MAX_PATH];
    if (GetFullPathNameA(outputPath.c_str(), MAX_PATH, fullPath, nullptr) == 0) {
        cerr << "Failed to get the full path of the output file." << endl;
    }
    ShellExecute(nullptr, "Open", fullPath, nullptr, nullptr, SW_SHOW);
}

...
```

Main.cpp

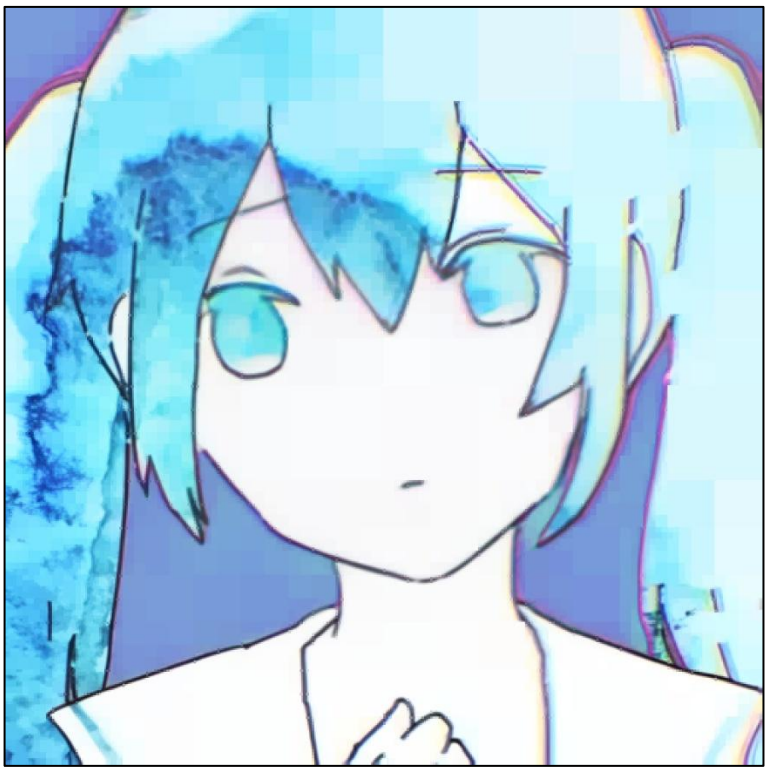
```cpp
int main() {
    miku(); // Splash Art :3

    string inputPath, outputPath;
    int method, threshold, minBlockSize;
    double targetRatio;

    cout << "Enter the path to your image:" << endl;
    cin >> inputPath;

    Image image = Image(inputPath);
    if (!image.isLoaded()) {
        cerr << "Failed to load image. Please check the file path." << endl;
        return 1;
    }
    Quadtree::setImage(image);

    cout<<"1.Variance\n2.Mean Abosolute Deviation\n3.Max Pixel Difference\n4.Entropy"<<endl;
    cout << "Enter the method of compression (1 to 4):" << endl;
    cin >> method;
    if (method < 1 || method > 4) {
        cerr << "Invalid method. Please enter a number between 1 and 4." << endl;
        return 1;
    }
    Quadtree::setMethod(method);

    cout << "Enter the threshold value (e.g., 63):" << endl;
    cin >> threshold;
    if (threshold < 0) {
        cerr << "Invalid threshold. Please enter a non-negative number." << endl;
        return 1;
    }
    Quadtree::setThreshold(threshold);

    cout << "Enter the minimum block size (e.g., 15):" << endl;
    cin >> minBlockSize;
    if (minBlockSize < 1) {
        cerr << "Invalid block size. Please enter a non-negative number." << endl;
        return 1;
    }
    Quadtree::setMinBlockSize(minBlockSize);

    cout << "Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:" << endl;
    cin >> targetRatio;
    if (targetRatio < 0) {
        cerr << "Invalid ratio. Please enter a non-negative number." << endl;
        return 1;
    }

    cout << "Enter the output path for the processed image:" << endl;
    cin >> outputPath;

    cout << "Processing image..." << endl;

    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    chrono::steady_clock::time_point end;

    const double ratio = compress(outputPath, threshold, targetRatio, image, &begin, &end);

    const auto elapsedTime = chrono::duration_cast<chrono::milliseconds>(end - begin).count();
    cout << endl << "Processing completed in " << elapsedTime << " ms." << endl;

    cout << "Size before compression: " << Image::sizeBefore << " bytes" << endl;
    cout << "Size after compression: " <<  Image::sizeAfter << " bytes" << endl;
    cout << "Compression ratio: " << ratio * 100 << "%" << endl;

    cout << "Tree depth: " << Quadtree::maxDepth << endl;
    cout << "Number of nodes: " << Quadtree::nodes << endl;

    openOut(outputPath);

    cout << endl << "(≥∇≤)/ Thank you for using this program!" << endl;
    return 0;
}
```

BAB IV
Eksperimen

Berikut adalah beberapa contoh keluaran program pada berbagai uji kasus:

1. Variance

```
 _____                 _____                                  _
|  _  |               |_   _|                                | |
| | | |_   _  __ _  __| | | |_ _ __ ___  ___  ___ | |_ ___ _ __ ___  ___ ___ _ __ ___  _ __
| | | | | | |/ _` |/ _` | | __| '__/ _ \/ _ \ | |   / _ \| '_ ` _ \| '_ ` _ \/ __/ _ \ '__/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | | |_| | |  __/  __/ | \__/\ (_) | | | | | | | | | | (_) | | | | (__| | | | | |
 \_/\_\\__,_|\__,_|\__,_| \__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                   |_|
Enter the path to your image:
../test/LoveBluePotion.png

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
1

Enter the threshold value (e.g., 63):
63

Enter the minimum block size (e.g., 15):
15

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0

Enter the output path for the processed image:
../test/LoveBluePotion_compressed.png

Processing image...

Processing completed in 42 ms.
Size before compression: 1202799 bytes
Size after compression: 272156 bytes
Compression ratio: 22.6269%
Tree depth: 10
Number of nodes: 73129

(≥V≤)/ Thank you for using this program!

Process finished with exit code 0
```



2. Mean Abosolute Deviation

```
|  _  |              | | |              /  __ \                              (_)
| | | |_   _  __ _  __| | |_ _ __ ___  ___  | /  \/ ___  _ __ ___  _ __  _ __ ___  ___ ___ _  ___  _ __
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                  |_|
Enter the path to your image:
../test/MikuConcert.jpg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
2

Enter the threshold value (e.g., 63):
15

Enter the minimum block size (e.g., 15):
7

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0

Enter the output path for the processed image:
../test/MikuConcert_compressed.png

Processing image...

Processing completed in 57 ms.
Size before compression: 540480 bytes
Size after compression: 408725 bytes
Compression ratio: 75.6226%
Tree depth: 11
Number of nodes: 93521

(≧∇≦)/ Thank you for using this program!

Process finished with exit code 0
```

3. Max Pixel Difference

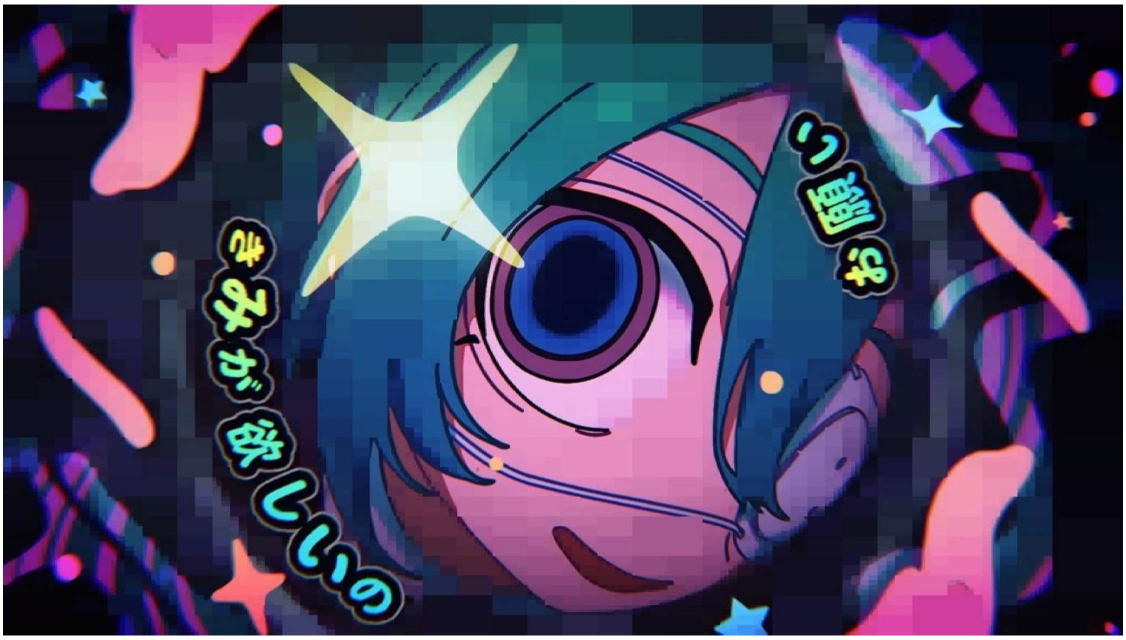
```
|  _  |             | | |               /  __ \                                  (_)
| | | |_   _  __ _  __| | |_ _ __ ___  ___  | /  \/ ___  _ __ ___  _ __  _ __ ___  ___ ___ _  ___  _ __
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                  |_|
Enter the path to your image:
../test/Mikuloid.jpeg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
3

Enter the threshold value (e.g., 63):
127

Enter the minimum block size (e.g., 15):
7

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0

Enter the output path for the processed image:
../test/Mikuloid_compressed.jpeg

Processing image...

Processing completed in 18 ms.
Size before compression: 105085 bytes
Size after compression: 92697 bytes
Compression ratio: 88.2114%
Tree depth: 10
Number of nodes: 22049

(≧∇≦)/ Thank you for using this program!

Process finished with exit code 0
```

4. Entropy

```
|  _  |              | | |               /  __ \                                 (_)
| | | |_   _  __ _  __| | |_ _ __ ___  ___  | /  \/ ___  _ __ ___  _ __  _ __ ___  ___ ___ _  ___  _ __
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                  |_|
Enter the path to your image:
../test/Teto.jpg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
4

Enter the threshold value (e.g., 63):
5

Enter the minimum block size (e.g., 15):
3

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0

Enter the output path for the processed image:
../test/Teto_compressed.png

Processing image...

Processing completed in 38 ms.
Size before compression: 89689 bytes
Size after compression: 47260 bytes
Compression ratio: 52.6932%
Tree depth: 8
Number of nodes: 7341

(≥∇≤)/ Thank you for using this program!

Process finished with exit code 0
```

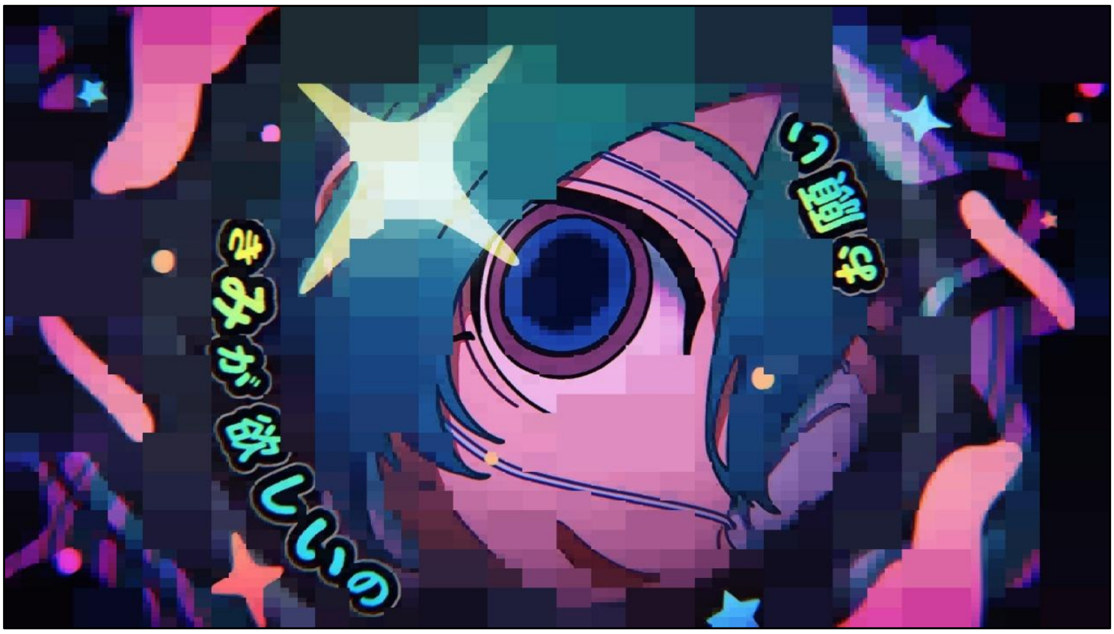
5. Ratio w/ Variance

```
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                   |_|
Enter the path to your image:
../test/Monitoring.jpeg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
1

Enter the threshold value (e.g., 63):
63

Enter the minimum block size (e.g., 15):
3

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0.5

Enter the output path for the processed image:
../test/Monitoring_compressedRatVar.png

Processing image...
////////////
Final threshold: 317

Processing completed in 2420 ms.
Size before compression: 1322954 bytes
Size after compression: 661701 bytes
Compression ratio: 50.0169%
Tree depth: 12
Number of nodes: 2341756

(≧∇≦)ﾉ Thank you for using this program!

Process finished with exit code 0
```



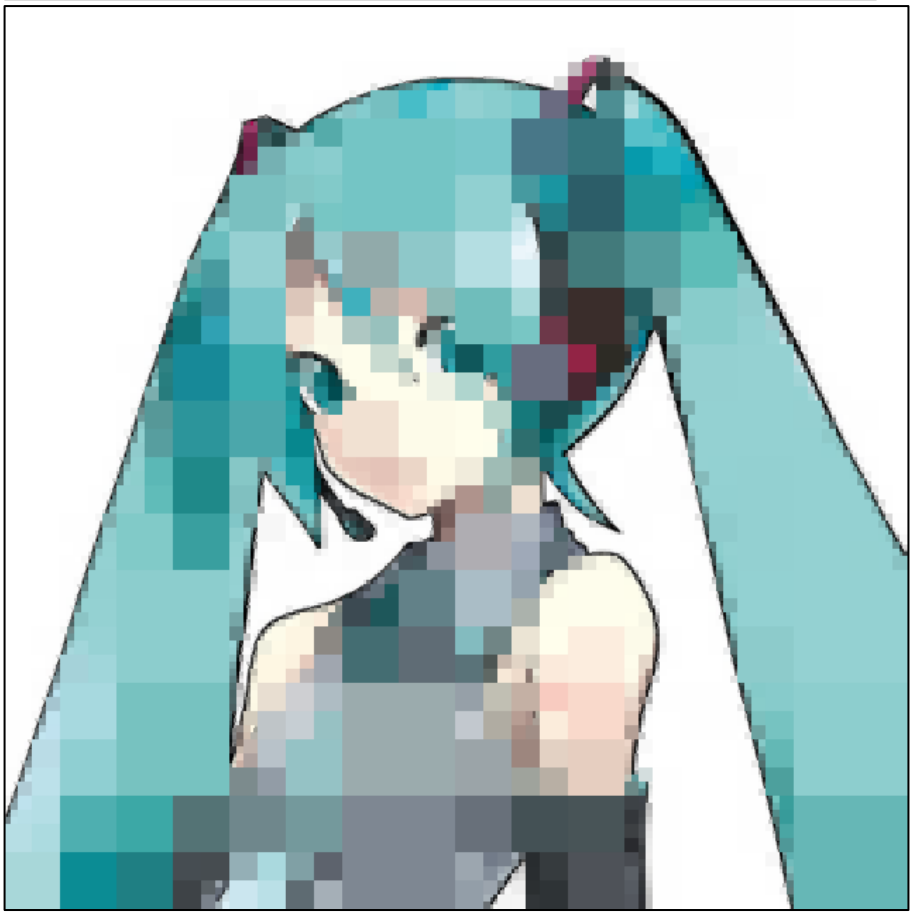
6. Ratio w/ MAD

```
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                   |_|
Enter the path to your image:
../test/Monitoring.jpeg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
2

Enter the threshold value (e.g., 63):
63

Enter the minimum block size (e.g., 15):
3

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0.3

Enter the output path for the processed image:
../test/Monitoring_compressedRatMAD.png

Processing image...
//////////
Final threshold: 19

Processing completed in 1360 ms.
Size before compression: 1322954 bytes
Size after compression: 410575 bytes
Compression ratio: 31.0347%
Tree depth: 12
Number of nodes: 585610

(≧∇≦)/ Thank you for using this program!

Process finished with exit code 0
```



7. Ratio w/ MPD

```
 | | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
 \ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
  \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                   |_|

Enter the path to your image:
../test/Mikuloid.jpeg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
3

Enter the threshold value (e.g., 63):
63

Enter the minimum block size (e.g., 15):
3

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0.4

Enter the output path for the processed image:
../test/Mikuloid_compressedRatMPD.png

Processing image...
//////////
Final threshold: 246

Processing completed in 312 ms.
Size before compression: 105085 bytes
Size after compression: 42243 bytes
Compression ratio: 40.1989%
Tree depth: 11
Number of nodes: 130083

(≥∇≤)/ Thank you for using this program!

Process finished with exit code 0
```



8. Ratio w/ Entropy

```
| | | | | | |/ _` |/ _` | __| '__/ _ \/ _ \ | |    / _ \| '_ ` _ \| '_ \| '__/ _ \/ __/ __| |/ _ \| '_ \
\ \/' / |_| | (_| | (_| | |_| | |  __/  __/ | \__/\ (_) | | | | | | |_) | | |  __/\__ \__ \ | (_) | | | |
 \_/\_\\__,_|\__,_|\__,_|\__|_|  \___|\___|  \____/\___/|_| |_| |_| .__/|_|  \___||___/___/_|\___/|_| |_|
                                                                  |_|
Enter the path to your image:
../test/Mikuloid.jpeg

1. Variance
2. Mean Abosolute Deviation
3. Max Pixel Difference
4. Entropy
Enter the method of compression (1 to 4):
4

Enter the threshold value (e.g., 63):
1

Enter the minimum block size (e.g., 15):
1

Enter the target compression ratio (e.g., 0.5). Enter 0 to ignore:
0.5

Enter the output path for the processed image:
../test/Mikuloid_compressedRatEnt.png

Processing image...
///////////
Final threshold: 4

Processing completed in 801 ms.
Size before compression: 105085 bytes
Size after compression: 67083 bytes
Compression ratio: 63.8369%
Tree depth: 11
Number of nodes: 504792

(≥∇≤)/ Thank you for using this program!

Process finished with exit code 0
```

BAB V
Analisis dan Pembahasan

Algoritma menyediakan empat metode error (Variansi, Mean Absolute Deviation, Max Pixel Difference, dan Entropi). Masing-masing menghasilkan struktur Quadtree dan hasil kompresi yang berbeda. Misalnya, Variansi lebih sensitif terhadap perubahan piksel kecil, sementara Entropy lebih memperhatikan sebaran nilai warna secara statistik.

Nilai *threshold* sangat berpengaruh terhadap jumlah pembagian blok. Semakin rendah threshold, maka semakin banyak pembagian (karena lebih sulit mencapai homogenitas), yang berarti struktur Quadtree menjadi lebih dalam dan kompleks. Ukuran *minBlockSize* juga membatasi seberapa kecil sebuah blok boleh dibagi sehingga menjadi parameter penting dalam mengontrol performa dan kualitas hasil.

Berdasarkan hasil uji, waktu pemrosesan bergantung pada ukuran gambar, nilai threshold, serta metode error. Gambar berukuran besar dengan threshold kecil cenderung menghasilkan waktu proses lebih lama, akibat pembagian yang lebih dalam. Namun, penggunaan tabel jumlah terintegrasi (sumTable) untuk mempercepat perhitungan mean/error merupakan optimisasi penting yang mengurangi waktu kalkulasi.

Percobaan menunjukkan bahwa algoritma divide and conquer berbasis Quadtree memberikan pendekatan yang efisien dan fleksibel untuk kompresi gambar. Meskipun kompleksitas terburuknya bisa tinggi, penerapannya secara cerdas dengan threshold dan ukuran blok minimum mampu menyeimbangkan antara performa dan kualitas kompresi. Struktur pohon yang dinamis dan adaptif menjadikannya cocok untuk berbagai jenis citra, terutama yang memiliki pola visual yang konsisten.

BAB VI
Pembahasan Bonus

Pengerjaan bonus Ratio Targetting dilakukan dengan memanfaatkan algoritma Binary Search untuk mencari threshold yang tepat untuk mendapatkan ratio yang diinginkan. Dengan awalnya mencari batas kanan threshold dengan menambahkan ‘right’ sekian kali hingga sudah di bawah ratio yang diinginkan, algoritma kemudian mencari threshold paling tepat dengan memilih titik tengah batas kiri dan kanan lalu melanjutkan selayaknya Binary Search umumnya.

BAB VII
Pranala Repositori

Berikut adalah tautan menuju repositori yang digunakan dalam pembuatan tugas ini:
https://github.com/Darsua/Tucil2_13523061

BAB VIII
Referensi

Berikut adalah referensi laporan yang digunakan dalam pembuatan laporan ini:
https://github.com/mrsyaban/24-Solver/blob/main/doc/Laporan_Tucil1.pdf

BAB IX
Lampiran

No.	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	√	
2	Program berhasil dijalankan	√	
3	Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	√	
4	Mengimplementasi seluruh metode perhitungan error wajib	√	
5	**[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	√	