

LAPORAN TUGAS KECIL 2

IF2211 - STRATEGI ALGORITMA



Disusun oleh :

Dzaky Aurelia Fawwaz 13523065

Ferdin Arsenarendra Purtadi 13523117

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

Bab Deskripsi Masalah

1.1 Algoritma Divide and Conqueror

Strategi algoritma *Divide and Conquer* merupakan metode penyelesaian masalah dengan cara membagi permasalahan besar menjadi beberapa submasalah yang lebih kecil dan lebih mudah ditangani. Pendekatan ini terdiri dari tiga tahap utama: *divide*, *conquer*, dan *combine*. Pada tahap *divide*, masalah dipecah menjadi submasalah yang serupa dengan masalah awal namun dengan skala yang lebih kecil. Selanjutnya, tahap *conquer* menyelesaikan submasalah tersebut, baik secara langsung jika ukurannya sudah cukup kecil, maupun secara rekursif jika masih cukup besar. Terakhir, tahap *combine* menggabungkan solusi dari masing-masing submasalah menjadi solusi akhir untuk permasalahan semula.

1.2 Quad Tree dalam Kompresi Gambar

Quadtree adalah struktur data berbentuk hierarki yang dirancang untuk membagi ruang atau data menjadi bagian-bagian yang lebih kecil, dan sering digunakan dalam bidang komputasi spasial, khususnya dalam pengolahan gambar digital. Dalam konteks kompresi citra, Quadtree berfungsi dengan memecah gambar menjadi blok-blok berukuran variatif, tergantung pada tingkat keseragaman warna atau intensitas piksel. Proses ini diawali dengan memotong gambar menjadi empat bagian kuadran yang sama besar. Setiap kuadran kemudian dianalisis untuk mengevaluasi keseragamannya berdasarkan nilai warna RGB, yaitu dengan membandingkan komposisi nilai merah (R), hijau (G), dan biru (B) pada piksel-piksel yang termasuk di dalamnya. Jika suatu area dianggap tidak homogen, maka area tersebut akan terus dibagi lagi hingga memenuhi kriteria keseragaman atau mencapai batas ukuran minimum yang ditentukan.

Secara struktural, Quadtree digambarkan sebagai pohon di mana setiap simpul dapat memiliki maksimal empat anak, yang masing-masing mewakili bagian hasil pembagian. Simpul-simpul ini terdiri dari dua jenis: simpul daun (leaf node) yang menunjukkan area yang sudah seragam, dan simpul internal (internal node) yang menunjukkan bahwa area tersebut masih perlu dibagi lebih lanjut. Setiap simpul menyimpan informasi penting seperti posisi koordinat (x, y), ukuran area (lebar dan tinggi), serta rata-rata nilai warna atau intensitas piksel di dalam area tersebut.

Penggunaan Quadtree dalam kompresi gambar membawa banyak keuntungan, salah satunya adalah kemampuannya untuk menyesuaikan diri dengan tingkat detail lokal gambar. Bagian gambar yang kompleks akan diberi representasi lebih detail, sedangkan bagian yang seragam akan disederhanakan. Hal ini membuat Quadtree sangat efektif untuk kompresi *lossy*, karena dapat mengurangi ukuran file dengan tetap mempertahankan elemen visual penting pada gambar.

1.3 Rancangan Algoritma Divide and Conquer dalam Kompresi Gambar pada Quad Tree

Hal ini dibagi menjadi 3 bagian sesuai dengan tahap-tahap algoritma divide and conquer yang telah dijelaskan pada poin sebelumnya.

1.3.1 Divide

Pada tahap divide dari algoritma kompresi gambar berbasis QuadTree, masalah utama yaitu kompresi seluruh gambar, dipecah menjadi submasalah yang lebih kecil dan lebih mudah ditangani. Proses ini dimulai dengan memandang seluruh gambar sebagai satu blok tunggal yang diwakili oleh node akar (root node) dalam struktur QuadTree. Setelah menghitung nilai error pada blok tersebut menggunakan metode pengukuran yang dipilih (Variance, MAD, MPD, atau Entropy), algoritma kemudian melakukan evaluasi untuk menentukan apakah blok perlu dibagi lebih lanjut. Jika nilai error melebihi threshold yang ditentukan, blok tersebut dibagi menjadi empat kuadran yang identik dalam ukuran: top-left, top-right, bottom-left, dan bottom-right. Setiap kuadran akan menjadi submasalah baru yang akan diproses secara independen. Strategi pembagian ini secara efektif mengurangi kompleksitas masalah pada setiap langkah rekursif, dengan memfokuskan analisis pada area yang lebih kecil dan lebih spesifik dari gambar asli.

1.3.2 Conqueror

Setelah masalah dibagi menjadi submasalah yang lebih kecil, tahap conquer bertugas untuk menyelesaikan masing-masing submasalah tersebut secara independen. Dalam algoritma kompresi QuadTree, tahap ini diimplementasikan melalui dua pendekatan: penyelesaian langsung atau rekursi lebih lanjut. Pendekatan pertama merupakan basis rekursi, yang terjadi ketika suatu blok dianggap sudah cukup homogen (nilai error \leq threshold) atau ketika ukuran blok telah mencapai batas minimum yang ditentukan. Dalam kondisi ini, blok tersebut tidak dibagi lagi dan ditetapkan sebagai leaf node dengan warna rataan (avgColor) yang mewakili seluruh piksel di dalam blok tersebut. Pendekatan kedua terjadi ketika kondisi basis belum tercapai, dimana tahap divide dijalankan lagi dan proses conquer dilanjutkan secara rekursif untuk masing-masing dari empat subblok yang dihasilkan. Proses rekursif ini akan terus berlanjut hingga seluruh blok dalam gambar telah mencapai basis rekursi. Tahap conquer ini sangat efisien karena memungkinkan area gambar yang homogen direpresentasikan dengan sedikit node, sementara area dengan detail tinggi diberi lebih banyak node untuk mempertahankan akurasi visual.

1.3.3. Combine

Tahap combine dalam algoritma kompresi QuadTree memiliki karakteristik unik dibandingkan dengan beberapa algoritma divide and conquer lainnya, karena penggabungan terjadi secara implisit melalui struktur pohon yang terbentuk. Setelah seluruh proses rekursif subdivisi selesai, struktur QuadTree dengan node-node leaf-nya sudah merupakan representasi terkompresi dari gambar asli. Untuk menghasilkan gambar hasil kompresi yang dapat dilihat, implementasi menjalankan fungsi buildCompressedImage() yang melakukan traversal pada pohon untuk merekonstruksi gambar. Proses ini dimulai dari root node dan secara rekursif mengunjungi setiap node. Jika node tersebut adalah leaf node, maka seluruh region/blok yang diwakilinya akan diwarnai dengan warna rataan yang sudah dihitung sebelumnya. Jika bukan leaf node, maka fungsi akan memanggil dirinya sendiri secara rekursif untuk keempat anak node. Hasil akhir dari proses penggabungan ini adalah sebuah gambar terkompresi dimana area yang homogen direpresentasikan dengan blok-blok besar, sementara area dengan banyak detail direpresentasikan dengan blok-blok yang lebih kecil, sehingga menghasilkan kompresi yang efisien namun tetap mempertahankan kualitas visual yang baik.

Bab Implementasi

2.1 Quad Tree

Dalam konteks kompresi gambar, setiap node dari Quadtree mewakili sebuah blok persegi pada gambar. Blok tersebut diperiksa apakah memiliki tingkat keseragaman warna yang cukup menggunakan suatu metode perhitungan error (seperti Variance, MAD, dsb). Jika nilai error-nya melebihi ambang batas (threshold), maka blok tersebut akan dibagi menjadi empat bagian yang sama besar (top-left, top-right, bottom-left, bottom-right). Proses ini dilakukan secara rekursif hingga blok-blok yang dihasilkan cukup homogen, atau ukuran blok sudah mencapai batas minimum yang ditentukan.

1. Inisialisasi Root Node
 - Fungsi `buildTree()` di kelas `QuadTree` memulai proses dengan mengambil seluruh gambar sebagai satu blok penuh (`Block fullImageBlock`).
 - Rata-rata warna dari seluruh gambar dihitung menggunakan `calculateAverageColor(...)`.
 - Sebuah simpul akar (root) dibuat dengan blok penuh dan warna rata-rata tersebut.
2. Pemanggilan Fungsi Rekursif `subdivide()`
 - *Fungsi subdivide(Node node, int depth)* dipanggil pada node akar.
 - Dilanjutkan proses rekursif pembelahan blok
3. Menghitung Nilai Error
 - Untuk setiap node, nilai error blok dihitung menggunakan `calculateError(...)`.
 - Fungsi ini akan memilih metode perhitungan error sesuai dengan pilihan pengguna (Variance, MAD, Entropy, dll).
 - Nilai error dibandingkan dengan threshold yang ditentukan.
4. Menentukan Perlu Dibagi atau Tidak
 - Jika nilai error melebihi threshold dan ukuran blok masih lebih besar dari batas minimum, maka:
 - Node ditandai sebagai bukan leaf (`isLeaf = false`).
 - Node akan dibagi menjadi 4 kuadran (Top-Left, Top-Right, Bottom-Left, Bottom-Right).
5. Membuat Child Node
 - Setiap kuadran dihitung rata-rata warnanya masing-masing.
 - Dibuat 4 buah Node baru untuk masing-masing blok kuadran.
 - Proses ini dilakukan dengan memanggil `Block::getQuadrant(i)` untuk tiap $i = 0$ sampai 3.
6. Pemanggilan Rekursif
 - Untuk setiap child node, fungsi `subdivide()` dipanggil kembali.
 - Proses rekursif berlanjut hingga tidak ada lagi blok yang perlu dibagi, baik karena:
 - Nilai error sudah cukup kecil (blended) atau ukuran blok sudah terlalu kecil (tidak bisa dibagi lagi).
7. Visualisasi
 - Setiap N node (misalnya 100), callback `compressionRegionCallback` dipanggil.
 - Callback ini digunakan untuk menghasilkan visualisasi proses pembentukan pohon ke dalam animasi GIF.
8. Leaf Node Menyimpan Warna Rata-Rata

- Jika node tidak dibagi (baik karena sudah homogen atau karena ukurannya terlalu kecil), node dianggap sebagai leaf.
- Leaf node akan digunakan untuk merekonstruksi gambar kompresi akhir.

9. Pembangunan Gambar Hasil

- Setelah proses subdivisi selesai, fungsi buildCompressedImage(...) akan menyusun gambar hasil kompresi dengan mewarnai setiap region (leaf node) menggunakan warna rata-ratanya (avgColor).

10. Perhitungan Statistik

- Fungsi countNodes() menghitung jumlah total node pada Quadtree.
- Fungsi calculateMaxDepth() menghitung kedalaman maksimum dari pohon.

2.2 Block

Block adalah kelas yang merepresentasikan sebuah wilayah persegi pada gambar. Setiap simpul (Node) dalam pohon Quadtree menyimpan satu Block, yang menyatakan posisi (x, y) dan ukuran (width, height) dari area gambar yang sedang diproses.

Peran utama Block adalah sebagai wadah data spasial (koordinat & ukuran) dan sebagai dasar untuk melakukan subdivisi. Saat pembelahan dilakukan dalam algoritma divide and conquer, blok-blok ini akan dibagi menjadi empat kuadran yang lebih kecil menggunakan fungsi getQuadrant(i).

1. Pembuatan Objek Block

- Saat root Quadtree dibuat, Block pertama mewakili seluruh gambar, dengan:

$x = 0$

$y = 0$

width = lebar gambar

height = tinggi gambar

- Disimpan di simpul akar Node dan digunakan sebagai titik awal pembagian.

2. Fungsi getX(), getY(), getWidth(), dan getHeight()

- Fungsi-fungsi getter ini menyediakan informasi dasar koordinat dan ukuran blok.
- Penting digunakan untuk menentukan apakah suatu blok bisa dibagi lagi dan menyusun ulang gambar hasil kompresi pada posisi yang tepat.

3. Pembagian Blok: Fungsi getQuadrant(int index)

- Fungsi ini adalah inti dari pembagian blok (divide).
- Diberikan nilai index dari 0 hingga 3, fungsi ini mengembalikan Block baru yang mewakili salah satu dari empat kuadran:

0 = Top-Left

1 = Top-Right

2 = Bottom-Left

3 = Bottom-Right

- Perhitungan posisi kuadran dilakukan berdasarkan pembagian width dan height menjadi setengahnya (midW dan midH), lalu menentukan x dan y yang sesuai untuk tiap bagian.
- Penggunaan dalam QuadTree::subdivide()
 - Di QuadTree, saat nilai error melebihi threshold:
 - Fungsi getQuadrant(i) dipanggil 4 kali
 - Masing-masing kuadran dijadikan blok baru untuk Node anak
 - Hal ini adalah implementasi konkret dari Divide and Conquer: satu blok dibagi → masing-masing diproses ulang secara rekursif
 - Penggunaan Saat Rekonstruksi Gambar
 - Dalam fungsi buildCompressedImage(), setiap leaf node (yang berisi blok homogen) akan diwarnai ulang menggunakan warna rata-rata (avgColor) di region Block.
 - Fungsi getX(), getY(), getWidth(), dan getHeight() digunakan untuk menentukan posisi dan area yang akan diwarnai.

2.3 Node

Kelas Node merepresentasikan satu simpul (node) dalam struktur pohon Quadtree. Setiap Node menyimpan informasi tentang blok gambar yang diwakilinya, warna rata-rata blok tersebut, dan pointer ke empat anak node-nya jika dibagi lebih lanjut.

Dalam pendekatan divide and conquer, Node adalah unit rekursif utama yang akan menyimpan data satu blok, memeriksa apakah perlu dibagi lagi, dan menyimpan pointer ke hasil pembagian jika dibagi.

- Pembuatan Root Node
 - QuadTree::buildTree() membuat node root yang mewakili seluruh gambar.
- Pembelahan
 - Di QuadTree::subdivide(), jika blok tidak cukup homogen:
 - Empat Block baru dibentuk dari kuadran
 - Empat node baru dibuat untuk tiap blok
 - Pointer anak diisi sesuai posisi
- Rekursi
 - Setiap anak node diproses kembali lewat subdivide()
 - Proses berlanjut sampai blok tidak dibagi lagi

Setelah Quadtree selesai dibangun, program melakukan traversal terhadap semua Node dan setiap node yang merupakan leaf digunakan untuk mewarnai area pada gambar output (compressedImage). Koordinatnya diperoleh dari region dan warnanya diambil dari avgColor

2.4 Image Processor

Kelas ImageProcessor berfungsi sebagai modul pengendali (controller) dari seluruh alur program. Ia mengelola semua langkah mulai dari pembacaan gambar, proses kompresi Quadtree, pencarian threshold optimal, penyimpanan hasil kompresi, hingga pembuatan animasi visualisasi proses ke dalam file GIF.

Kelas ini tidak melakukan subdivisi secara langsung, namun mengatur dan memanfaatkan kelas-kelas lain seperti QuadTree, Node, Block, dan ErrorMetrics untuk menjalankan langkah-langkah pemrosesan gambar secara keseluruhan.

1. `loadImage(std::string path)`
 - Membaca gambar masukan dan menyimpannya dalam bentuk matriks 2D dari objek RGB.
 - Gambar disimpan sebagai originalImage, dan dimensinya disimpan dalam atribut width, height.
 - Setelah gambar berhasil dimuat, ia siap diproses lebih lanjut oleh Quadtree.
2. `setThreshold(int threshold)` dan `setMinBlockSize(int size)`
 - Digunakan untuk mengatur parameter kompresi:
 - threshold → batas error agar suatu blok dianggap cukup homogen
 - minBlockSize → ukuran minimum blok yang boleh dibagi
3. `setGifPath(std::string path)`
 - Menyimpan path untuk hasil GIF animasi visualisasi.
 - Digunakan hanya jika pengguna ingin menyimpan visualisasi proses kompresi Quadtree.
4. `compress()`
 - Fungsi inti yang memulai proses kompresi:
 - Membuat objek QuadTree, memberikan parameter gambar asli, metode error, threshold, dan ukuran blok minimum.
 - Memanggil buildTree() untuk membentuk struktur Quadtree (proses divide and conquer).
 - Setelah pohon terbentuk, memanggil getCompressedImage() untuk menghasilkan gambar hasil kompresi dari node-node leaf.
 - Hasilnya disimpan dalam compressedImage.
5. `findThresholdForTargetCompression(double target)`
 - Fungsi ini adalah fitur bonus yang memungkinkan pengguna menentukan target persentase kompresi (misalnya 70%).
 - Fungsi ini akan melakukan binary search terhadap nilai threshold:
 - Mencoba berbagai threshold antara batas minimum dan maksimum
 - Mengukur ukuran hasil kompresi (compressedImage)
 - Menyesuaikan threshold agar hasil kompresi mendekati target yang diminta
6. `saveCompressedImage(std::string outputPath)`
 - Menyimpan compressedImage ke file .png (atau format lain) menggunakan library seperti stb_image_write.
 - Melibatkan konversi gambar dari struktur matriks RGB menjadi array 1D dari unsigned char berisi urutan R, G, B.
7. `generateCompressionGif()`
 - Digunakan untuk membuat file GIF visualisasi proses pembentukan Quadtree.
 - Menyediakan callback (compressionRegionCallback) ke QuadTree agar setiap N node baru ditambahkan, frame bisa diambil.
 - Menggunakan GifWriter dari gif.h untuk menulis frame demi frame secara streaming.
 - Proses ini mencerminkan pembentukan visual Quadtree secara bertahap.
8. Statistik Kompresi

- Setelah proses selesai, ImageProcessor menyediakan berbagai statistik melalui fungsi getter `getOriginalSize()`, `getCompressedSize()`, `getCompressionRatio()`, `getNodeCount()`, `getMaxDepth()`

2.5 Perhitungan Error

2.5.1 Variance

Nilai variance dihitung sebagai rata-rata dari kuadrat selisih antara setiap nilai piksel dan nilai rata-ratanya. Dalam program ini, variance dihitung untuk masing-masing channel warna (Red, Green, Blue), lalu dirata-ratakan menjadi satu nilai error yang mewakili seluruh blok.

1. Conquer (Basis Rekursi)
 - Untuk setiap blok gambar, hitung nilai rata-rata piksel per channel (R, G, B)
 - Hitung nilai variance per channel, lalu ambil rata-ratanya sebagai nilai error total
 - Jika nilai error \leq threshold, maka blok dianggap cukup homogen → tidak dibagi lagi (menjadi leaf node)
2. Divide (Langkah Rekursif)
 - Jika error $>$ threshold, dan ukuran blok masih di atas `minBlockSize`, maka:
 - Blok dibagi menjadi empat kuadran (Top-Left, Top-Right, Bottom-Left, Bottom-Right)
 - Masing-masing kuadran diwakili oleh Block baru, dan dibuat node anak (Node*)
 - Fungsi `subdivide()` dipanggil secara rekursif pada keempat kuadran tersebut
3. Penggabungan Hasil (Conquer akhir)
 - Setelah pembagian selesai, hasil dari setiap node leaf digunakan untuk menyusun ulang gambar hasil kompresi
 - Setiap node leaf akan mewarnai blok-nya dengan warna rata-rata (`avgColor`)

2.5.2 MAD (Mean Absolute Deviation)

Tidak seperti variance yang menggunakan kuadrat dari selisih, MAD menghitung rata-rata dari nilai absolut selisih antara piksel dan nilai rata-ratanya sehingga lebih ringan secara komputasi dan lebih tahan terhadap outlier.

1. Conquer (Basis Rekursi):
 - Hitung nilai MAD dari blok untuk masing-masing channel (R, G, B).
 - Jika nilai error (MAD rata-rata) \leq threshold, maka blok dianggap cukup homogen → menjadi leaf node dan tidak dibagi lagi.
2. Divide (Langkah Rekursif):
 - Jika nilai MAD $>$ threshold dan ukuran blok masih lebih besar dari `minBlockSize`, maka:
 - Blok dibagi menjadi empat kuadran
 - Dibuat node anak untuk masing-masing kuadran
 - Fungsi `subdivide()` dipanggil kembali secara rekursif untuk masing-masing kuadran
3. Penggabungan Hasil (Conquer akhir):

Setelah semua node leaf diperoleh, blok-blok tersebut digunakan untuk menyusun kembali gambar hasil kompresi, dengan mewarnai tiap blok menggunakan warna rata-ratanya.

2.5.3 MPD (Maximum Pixel Difference)

Metode ini berfokus pada selisih antara nilai piksel terbesar dan terkecil dalam suatu blok untuk masing-masing channel warna (R, G, B). Jika perbedaan ini cukup besar, maka blok dianggap tidak homogen dan perlu dibagi lebih lanjut.

1. Conquer (Basis Rekursi):

- Hitung nilai maksimum dan minimum dari tiap channel dalam blok.
- Jika nilai error (rata-rata MPD dari R, G, B) \leq threshold, maka blok dianggap cukup seragam \rightarrow tidak dibagi lagi (leaf node).

2. Divide (Langkah Rekursif):

- Jika nilai error $>$ threshold, dan ukuran blok masih lebih besar dari minBlockSize, maka:
 - Blok dibagi menjadi empat kuadran
 - Dibuat empat node anak untuk masing-masing kuadran
 - Fungsi subdivide() dipanggil secara rekursif untuk tiap anak

3. Penggabungan (Conquer akhir):

- Setelah tidak ada lagi blok yang perlu dibagi, node-node leaf digunakan untuk menyusun ulang gambar hasil kompresi.

2.5.4 Entropy

Berasal dari teori informasi (Shannon entropy), metode ini memandang setiap blok gambar sebagai sumber data, dan mengevaluasi seberapa “acak” atau “beragam” data intensitas warnanya. Semakin tinggi nilai entropy, semakin kompleks blok tersebut, sehingga semakin layak untuk dibagi lebih lanjut.

1. Conquer (Basis Rekursi):

- Hitung histogram intensitas warna untuk setiap channel.
- Hitung entropy berdasarkan frekuensi kemunculan nilai intensitas tersebut.
- Jika nilai rata-rata entropy \leq threshold, maka blok dianggap cukup “teratur” \rightarrow tidak dibagi lagi (leaf node).

2. Divide (Langkah Rekursif):

- Jika nilai error $>$ threshold, dan ukuran blok masih di atas minBlockSize, maka:
 - Blok dibagi menjadi empat kuadran
 - Dibuat node-node anak untuk tiap kuadran
 - Fungsi subdivide() dipanggil kembali secara rekursif

3. Penggabungan (Conquer akhir):

- Setelah semua leaf node diperoleh, blok-blok tersebut digunakan untuk membentuk kembali gambar hasil kompresi.

Source Code

3.1 Main.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "ImageProcessor.hpp"
#include "Utils.hpp"

#define RESET "\033[0m"
#define RED "\033[31m"
#define GREEN "\033[32m"
#define YELLOW "\033[33m"
#define BLUE "\033[34m"
#define MAGENTA "\033[35m"
#define CYAN "\033[36m"

void printColoredText(const std::string& text, const std::string& color) {
    std::cout << color << text << RESET;
}

void displayUsage(const char* programName) {
    std::cout << "Usage: " << programName << " <options>\n";
    std::cout << "Options:\n";
    std::cout << " -i, --input <file>           Input image file path\n";
    std::cout << " -o, --output <file>          Output compressed image file path\n";
    std::cout << " -m, --method <n>            Error metric method (1=Variance, 2=MAD, 3=MaxDiff, 4=Entropy, 5=SSIM)\n";
    std::cout << " -t, --threshold <n>        Error threshold\n";
    std::cout << " -b, --blocksize <n>       Minimum block area in square pixels\n";
    std::cout << " -c, --compression <percent> Target compression percentage (0.0-1.0, 0 to disable)\n";
    std::cout << " -g, --gif <file>           Output GIF visualization file path (optional)\n";
    std::cout << " -h, --help                 Display this help message\n";
    std::cout << "nOr run without arguments to use interactive mode.\n";
}

QuadTree::ErrorMetricType getErrorMetricFromString(const std::string& input) {
    if (input == "1") return QuadTree::VARIANCE;
    if (input == "2") return QuadTree::MEAN_ABSOLUTE_DEVIATION;
    if (input == "3") return QuadTree::MAX_PIXEL_DIFFERENCE;
    if (input == "4") return QuadTree::ENTROPY;
    if (input == "5") return QuadTree::SSIM;

    return QuadTree::VARIANCE;
}

int main(int argc, char* argv[]) {
    std::string inputPath, outputPath, gifPath;
    QuadTree::ErrorMetricType errorMethod = QuadTree::VARIANCE;
    double threshold = -1.0;
    int minBlockSize = Utils::getDefaultMinBlockArea();
    double targetCompression = 0.0;
    bool interactiveMode = (argc <= 1);

    if (!interactiveMode) {
        for (int i = 1; i < argc; i++) {
            std::string arg = argv[i];

            if (arg == "-h" || arg == "--help") {
                displayUsage(argv[0]);
                return 0;
            } else if (arg == "-i" || arg == "--input") {
                if (i + 1 < argc) inputPath = argv[i + 1];
            } else if (arg == "-o" || arg == "--output") {
                if (i + 1 < argc) outputPath = argv[i + 1];
            } else if (arg == "-m" || arg == "--method") {
                if (i + 1 < argc) errorMethod = getErrorMetricFromString(argv[i + 1]);
            } else if (arg == "-t" || arg == "--threshold") {
                if (i + 1 < argc) threshold = std::stod(argv[i + 1]);
            } else if (arg == "-b" || arg == "--blocksize") {
                if (i + 1 < argc) minBlockSize = std::stoi(argv[i + 1]);
            } else if (arg == "-c" || arg == "--compression") {
                if (i + 1 < argc) targetCompression = std::stod(argv[i + 1]);
            } else if (arg == "-g" || arg == "--gif") {
                if (i + 1 < argc) gifPath = argv[i + 1];
            }
        }
    }

    if (inputPath.empty()) {
        std::cerr << "Error: Input path is required.\n";
        displayUsage(argv[0]);
        return 1;
    }

    if (!Utils::fileExists(inputPath)) {
        std::cerr << "Error: Input file " << inputPath << " does not exist.\n";
        return 1;
    }

    if (outputPath.empty()) {
        outputPath = Utils::getDefaultOutputPath(inputPath, errorMethod,
                                                threshold, minBlockSize, targetCompression);
        std::cout << "Using default output path: " << outputPath << std::endl;
    }

    if (gifPath.empty() && targetCompression > 0.0) {
        gifPath = Utils::getDefaultGifPath(inputPath, errorMethod,
                                           threshold, minBlockSize, targetCompression);
        std::cout << "Using default GIF path: " << gifPath << std::endl;
    }
}
```

```

if (threshold < 0) {
    threshold = Utils::getDefaultThreshold(errorMethod);
    std::cout << "Using default threshold: " << threshold << std::endl;
}
} else {
printColoredText("\n==== QuadTree Image Compression ====\n\n", BLUE);

printColoredText("Enter input image path: ", GREEN);
std::getline(std::cin, inputPath);

if (!Utils::fileExists(inputPath)) {
    printColoredText("Error: Input file does not exist. Please check the path and try again.\n", RED);
    return 1;
}

printColoredText("Enter output image path (leave empty for default): ", GREEN);
std::getline(std::cin, outputPath);

printColoredText("\nSelect error metric method:\n", YELLOW);
std::cout << " 1. Variance\n";
std::cout << " 2. Mean Absolute Deviation (MAD)\n";
std::cout << " 3. Max Pixel Difference\n";
std::cout << " 4. Entropy\n";
std::cout << " 5. SSIM (Structural Similarity Index)\n";
printColoredText("Enter your choice (1-5) [default: 1]: ", GREEN);
std::string methodStr;
bool validMethod = false;
do {
    std::getline(std::cin, methodStr);
    if (methodStr.empty()) {
        errorMethod = QuadTree::VARIANCE;
        validMethod = true;
    } else {
        try {
            int methodNum = std::stoi(methodStr);
            if (methodNum >= 1 && methodNum <= 5) {
                errorMethod = getErrorMetricFromString(methodStr);
                validMethod = true;
            } else {
                printColoredText("Invalid method number. Please enter a number between 1 and 5: ", RED);
            }
        } catch (const std::exception&) {
            printColoredText("Invalid input. Please enter a number between 1 and 5: ", RED);
        }
    }
} while (!validMethod);

double defaultThreshold = Utils::getDefaultThreshold(errorMethod);
double minThreshold, maxThreshold;
Utils::getThresholdLimits(errorMethod, minThreshold, maxThreshold);

printColoredText("\nEnter error threshold (" + std::to_string(minThreshold) +
    " to " + std::to_string(maxThreshold) +
    ") [default: " + std::to_string(defaultThreshold) + "]: ", GREEN);
std::string thresholdStr;
bool validThreshold = false;
do {
    std::getline(std::cin, thresholdStr);
    if (thresholdStr.empty()) {
        threshold = defaultThreshold;
        validThreshold = true;
    } else {
        try {
            threshold = std::stod(thresholdStr);
            if (threshold >= minThreshold && threshold <= maxThreshold) {
                validThreshold = true;
            } else {
                printColoredText("Threshold must be between " + std::to_string(minThreshold) +
                    " and " + std::to_string(maxThreshold) + ":", RED);
            }
        } catch (const std::exception&) {
            printColoredText("Invalid input. Please enter a valid number: ", RED);
        }
    }
} while (!validThreshold);

printColoredText("Enter minimum block area in square pixels [default: " +
    std::to_string(Utils::getDefaultMinBlockArea()) + "]: ", GREEN);
std::string blockSizeStr;
std::getline(std::cin, blockSizeStr);
if (!blockSizeStr.empty()) minBlockSize = std::stoi(blockSizeStr);

printColoredText("\nEnter target compression percentage (0.0-1.0, 0 to disable): ", GREEN);
std::string compressionStr;
std::getline(std::cin, compressionStr);
if (!compressionStr.empty()) targetCompression = std::stod(compressionStr);

if (outputPath.empty()) {
    outputPath = Utils::getDefaultOutputPath(inputPath, errorMethod,
                                             minBlockSize, threshold, targetCompression);
    printColoredText("Using default output path: " + outputPath + "\n", CYAN);
}
}

```

```

printColoredText("Enter GIF visualization path (leave empty for default or 'n' to skip): ", GREEN);
std::getline(std::cin, gifPath);

if (gifPath.empty() && targetCompression > 0.0) {
    gifPath = Utils::getDefaultGifPath(inputPath, errorMethod,
                                      threshold, minBlockSize, targetCompression);
    printColoredText("Using default GIF path: " + gifPath + "\n", CYAN);
} else if (gifPath == "n" || gifPath == "N") {
    gifPath = "";
}

printColoredText("\nStarting compression with the following parameters:\n", CYAN);
std::cout << " Input: " << outputPath << "\n";
std::cout << " Output: " << outputPath << "\n";
std::cout << " Error Method: " << Utils::errorMetricToString(errorMethod) << "\n";
std::cout << " Threshold: " << threshold << " (range: " << minThreshold << " to " << maxThreshold << ")\n";
std::cout << " Min Block Area: " << minBlockSize << " square pixels\n";
std::cout << " Target Compression: " << std::fixed << std::setprecision(1)
      << (targetCompression * 100) << "%\n";
if (!gifPath.empty()) {
    std::cout << " GIF Output: " << gifPath << "\n";
}
std::cout << "\n";

ImageProcessor processor(inputPath, outputPath, minBlockSize, threshold,
                        errorMethod, targetCompression, gifPath);

printColoredText("Loading image...\n", YELLOW);
bool success = processor.loadImage();
if (!success) {
    printColoredText("Failed to load the input image.\n", RED);
    return 1;
}

printColoredText("Compressing image...\n", YELLOW);
success = processor.compressImage();
if (!success) {
    printColoredText("Failed to compress the image.\n", RED);
    return 1;
}

printColoredText("Saving compressed image...\n", YELLOW);
success = processor.saveCompressedImage();
if (!success) {
    printColoredText("Failed to save the compressed image.\n", RED);
    return 1;
}

processor.displayMetrics();

if (interactiveMode) {
    printColoredText("\nCompression completed successfully!\n", GREEN);
    std::cout << "Press Enter to exit...";
    std::cin.get();
}

return 0;
}

```

3.2 Block.cpp

```
#include "Block.hpp"

Block::Block() : x(0), y(0), width(0), height(0) {}

Block::Block(int _x, int _y, int _width, int _height)
: x(_x), y(_y), width(_width), height(_height) {}

int Block::getX() const {
    return x;
}

int Block::getY() const {
    return y;
}

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

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

void Block::setX(int _x) {
    x = _x;
}

void Block::setY(int _y) {
    y = _y;
}

void Block::setWidth(int _width) {
    width = _width;
}

void Block::setHeight(int _height) {
    height = _height;
}

bool Block::operator==(const Block& other) const {
    return x == other.x && y == other.y &&
           width == other.width && height == other.height;
}

Block Block::getQuadrant(int quadrant) const {
    int halfWidth = width / 2;
    int halfHeight = height / 2;

    switch (quadrant) {
        case 0: // Top-left
            return Block(x, y, halfWidth, halfHeight);
        case 1: // Top-right
            return Block(x + halfWidth, y, halfWidth, halfHeight);
        case 2: // Bottom-left
            return Block(x, y + halfHeight, halfWidth, halfHeight);
        case 3: // Bottom-right
            return Block(x + halfWidth, y + halfHeight, halfWidth, halfHeight);
        default:
            return Block(); // Default empty block
    }
}

bool Block::contains(int pointX, int pointY) const {
    return pointX >= x && pointX < x + width &&
           pointY >= y && pointY < y + height;
}
```

3.3 ErrorMetrics.cpp

```
#include "ErrorMetrics.hpp"
#include <cmath>
#include <map>
#include <algorithm>
#include <numeric>

// Menghitung variance RGB dalam sebuah region
double ErrorMetrics::calculateVariance(const vector<vector<RGB>>& image,
                                         const Block& region,
                                         const RGB& avgColor) {
    // Menghitung variance untuk setiap channel dan mengambil rata-ratanya
    double varR = calculateChannelVariance(image, region, avgColor, 0);
    double varG = calculateChannelVariance(image, region, avgColor, 1);
    double varB = calculateChannelVariance(image, region, avgColor, 2);

    // Rata-rata variance dari ketiga channel
    return (varR + varG + varB) / 3.0;
}

// Menghitung variance untuk satu channel warna
double ErrorMetrics::calculateChannelVariance(const vector<vector<RGB>>& image,
                                               const Block& region,
                                               const RGB& avgColor,
                                               int channelOffset) {
    int width = region.getWidth();
    int height = region.getHeight();
    int startX = region.getX();
    int startY = region.getY();

    if (width <= 0 || height <= 0) {
        return 0.0;
    }

    unsigned char avgVal;
    if (channelOffset == 0) avgVal = avgColor.getRed();
    else if (channelOffset == 1) avgVal = avgColor.getGreen();
    else avgVal = avgColor.getBlue();

    double sumSquaredDiff = 0.0;
    int count = 0;

    // Untuk setiap piksel dalam region
    for (int y = startY; y < startY + height && y < image.size(); y++) {
        for (int x = startX; x < startX + width && x < image[y].size(); x++) {
            unsigned char pixelVal;
            if (channelOffset == 0) pixelVal = image[y][x].getRed();
            else if (channelOffset == 1) pixelVal = image[y][x].getGreen();
            else pixelVal = image[y][x].getBlue();

            // Akumulasi perbedaan kuadrat
            double diff = static_cast<double>(pixelVal) - static_cast<double>(avgVal);
            sumSquaredDiff += diff * diff;
            count++;
        }
    }

    // Jika tidak ada piksel yang valid
    if (count == 0) {
        return 0.0;
    }

    // Hitung variance (rata-rata perbedaan kuadrat)
    return sumSquaredDiff / count;
}

// Menghitung Mean Absolute Deviation (MAD) dalam sebuah region
double ErrorMetrics::calculateMAD(const vector<vector<RGB>>& image,
                                   const Block& region,
                                   const RGB& avgColor) {
    // Menghitung MAD untuk setiap channel dan mengambil rata-ratanya
    double madR = calculateChannelMAD(image, region, avgColor, 0);
    double madG = calculateChannelMAD(image, region, avgColor, 1);
    double madB = calculateChannelMAD(image, region, avgColor, 2);

    // Rata-rata MAD dari ketiga channel
    return (madR + madG + madB) / 3.0;
}

// Menghitung MAD untuk satu channel warna
double ErrorMetrics::calculateChannelMAD(const vector<vector<RGB>>& image,
                                         const Block& region,
                                         const RGB& avgColor,
                                         int channelOffset) {
    int width = region.getWidth();
    int height = region.getHeight();
    int startX = region.getX();
    int startY = region.getY();

    if (width <= 0 || height <= 0) {
        return 0.0;
    }

    unsigned char avgVal;
    if (channelOffset == 0) avgVal = avgColor.getRed();
    else if (channelOffset == 1) avgVal = avgColor.getGreen();
    else avgVal = avgColor.getBlue();

    double sumAbsDiff = 0.0;
    int count = 0;

    // Untuk setiap piksel dalam region
    for (int y = startY; y < startY + height && y < image.size(); y++) {
        for (int x = startX; x < startX + width && x < image[y].size(); x++) {
```

```

        if (channelOffset == 0) pixelVal = image[y][x].getRed();
        else if (channelOffset == 1) pixelVal = image[y][x].getGreen();
        else pixelVal = image[y][x].getBlue();

        // Akumulasi absolute difference
        double diff = abs(static_cast<double>(pixelVal) - static_cast<double>(avgVal));
        sumAbsDiff += diff;
        count++;
    }

}

// Jika tidak ada piksel yang valid
if (count == 0) {
    return 0.0;
}

// Hitung MAD (rate-rata absolute difference)
return sumAbsDiff / count;
}

// Menghitung perbedaan piksel maksimum dalam sebuah region
double ErrorMetrics::calculateMaxDifference(const vector<vector<RGB>>& image,
                                             const Block& region,
                                             const RGB& avgColor) {
    // Menghitung max difference untuk setiap channel dan mengambil rata-ratanya
    double maxDiffR = calculateChannelMaxDifference(image, region, avgColor, 0);
    double maxDiffG = calculateChannelMaxDifference(image, region, avgColor, 1);
    double maxDiffB = calculateChannelMaxDifference(image, region, avgColor, 2);

    // Rata-rata max difference dari ketiga channel
    return (maxDiffR + maxDiffG + maxDiffB) / 3.0;
}

// Menghitung perbedaan maksimum untuk satu channel warna
double ErrorMetrics::calculateChannelMaxDifference(const vector<vector<RGB>>& image,
                                                    const Block& region,
                                                    const RGB& avgColor,
                                                    int channelOffset) {
    int width = region.getWidth();
    int height = region.getHeight();
    int startX = region.getX();
    int startY = region.getY();

    if (width <= 0 || height <= 0) {
        return 0.0;
    }

    unsigned char avgVal;
    if (channelOffset == 0) avgVal = avgColor.getRed();
    else if (channelOffset == 1) avgVal = avgColor.getGreen();
    else avgVal = avgColor.getBlue();

    unsigned char minValue = 255;
    unsigned char maxValue = 0;

    // Cari nilai minimum dan maksimum dalam region
    for (int y = startY; y < startY + height && y < image.size(); y++) {
        for (int x = startX; x < startX + width && x < image[y].size(); x++) {
            unsigned char pixelVal;
            if (channelOffset == 0) pixelVal = image[y][x].getRed();
            else if (channelOffset == 1) pixelVal = image[y][x].getGreen();
            else pixelVal = image[y][x].getBlue();

            minValue = min(minValue, pixelVal);
            maxValue = max(maxValue, pixelVal);
        }
    }

    // Hitung perbedaan maksimum
    return static_cast<double>(maxValue - minValue);
}

// Menghitung entropy dalam sebuah region
double ErrorMetrics::calculateEntropy(const vector<vector<RGB>>& image,
                                      const Block& region) {
    // Menghitung entropy untuk setiap channel dan mengambil rata-ratanya
    double entropyR = calculateChannelEntropy(image, region, 0);
    double entropyG = calculateChannelEntropy(image, region, 1);
    double entropyB = calculateChannelEntropy(image, region, 2);

    // Rata-rata entropy dari ketiga channel
    return (entropyR + entropyG + entropyB) / 3.0;
}

// Menghitung entropy untuk satu channel warna
double ErrorMetrics::calculateChannelEntropy(const vector<vector<RGB>>& image,
                                              const Block& region,
                                              int channelOffset) {
    int width = region.getWidth();
    int height = region.getHeight();
    int startX = region.getX();
    int startY = region.getY();

    if (width <= 0 || height <= 0) {
        return 0.0;
    }

    // Hitung histogram (frekuensi) nilai piksel (0-255)
    map<unsigned char, int> histogram;
    int totalPixels = 0;

```

```

        for (int y = startY; y < startY + height && y < image.size(); y++) {
            for (int x = startX; x < startX + width && x < image[y].size(); x++) {
                unsigned char pixelVal;
                if (channelOffset == 0) pixelVal = image[y][x].getRed();
                else if (channelOffset == 1) pixelVal = image[y][x].getGreen();
                else pixelVal = image[y][x].getBlue();

                histogram[pixelVal]++;
                totalPixels++;
            }
        }

        // Jika tidak ada piksel yang valid
        if (totalPixels == 0) {
            return 0.0;
        }

        // Hitung entropy menggunakan rumus: -sum(p * log2(p))
        double entropy = 0.0;
        for (const auto& pair : histogram) {
            double probability = static_cast<double>(pair.second) / totalPixels;
            if (probability > 0.0) {
                entropy -= probability * log2(probability);
            }
        }

        return entropy;
    }

    // Menghitung Structural Similarity Index (SSIM) dalam sebuah region (bonus)
    double ErrorMetrics::calculateSSIM(const vector<vector<RGB>>& image,
                                         const Block& region,
                                         const RGB& avgColor) {
        // Konstanta untuk SSIM
        const double C1 = 0.01 * 255 * 0.01 * 255; // (k1*L)^2, L=255, k1=0.01
        const double C2 = 0.03 * 255 * 0.03 * 255; // (k2*L)^2, L=255, k2=0.03

        // Menghitung SSIM untuk setiap channel dan mengambil weighted average
        double ssimR = calculateChannelSSIM(image, region, avgColor, 0);
        double ssimG = calculateChannelSSIM(image, region, avgColor, 1);
        double ssimB = calculateChannelSSIM(image, region, avgColor, 2);

        // Weighted average (human eye is more sensitive to green)
        return 0.299 * ssimR + 0.587 * ssimG + 0.114 * ssimB;
    }

    // Menghitung SSIM untuk satu channel warna
    double ErrorMetrics::calculateChannelSSIM(const vector<vector<RGB>>& image,
                                              const Block& region,
                                              const RGB& avgColor,
                                              int channelOffset) {
        int width = region.getWidth();
        int height = region.getHeight();
        int startX = region.getX();
        int startY = region.getY();

        if (width <= 0 || height <= 0) {
            return 1.0; // SSIM = 1 untuk region kosong (sama persis)
        }

        unsigned char avgVal;
        if (channelOffset == 0) avgVal = avgColor.getRed();
        else if (channelOffset == 1) avgVal = avgColor.getGreen();
        else avgVal = avgColor.getBlue();

        // Kita sudah tahu nilai rata-rata dari original region (avgVal)
        double mu1 = static_cast<double>(avgVal);
        double mu2 = mu1; // Untuk region yang sudah dikompresi, nilai semua piksel = avgVal

        // Hitung variance untuk region asli
        double sigma1_sq = 0.0;
        int count = 0;

        for (int y = startY; y < startY + height && y < image.size(); y++) {
            for (int x = startX; x < startX + width && x < image[y].size(); x++) {
                unsigned char pixelVal;
                if (channelOffset == 0) pixelVal = image[y][x].getRed();
                else if (channelOffset == 1) pixelVal = image[y][x].getGreen();
                else pixelVal = image[y][x].getBlue();

                double diff = static_cast<double>(pixelVal) - mu1;
                sigma1_sq += diff * diff;
                count++;
            }
        }

        if (count > 0) {
            sigma1_sq /= count;
        }

        // Untuk region terkompresi, semua piksel sama, jadi variance = 0
        double sigma2_sq = 0.0;

        // Covariance juga 0 karena region terkompresi tidak memiliki variasi
        double sigma12 = 0.0;

        // Konstanta untuk stabilitas
        const double C1 = 0.01 * 255 * 0.01 * 255;
        const double C2 = 0.03 * 255 * 0.03 * 255;

        // Hitung SSIM
        double ssim = ((2 * mu1 * mu2 + C1) * (2 * sigma12 + C2)) /
                    ((mu1 * mu1 + mu2 * mu2 + C1) * (sigma1_sq + sigma2_sq + C2));

        // SSIM seharusnya berada di range [-1, 1], nilai lebih tinggi = lebih mirip
        // Kita ingin mengubahnya menjadi error metric (0 = mirip, nilai tinggi = beda)
        return 1.0 - ssim;
    }
}

```

3.4 ImageProcessor.cpp

```
ImageProcessor::ImageProcessor(const string& _inputPath, const string& _outputPath,
                               int _minBlockSize, double _threshold,
                               QuadTree::ErrorMetricType _errorMetricType,
                               double _targetCompressionPercentage,
                               const string& _gifPath)
: inputPath(_inputPath), outputPath(_outputPath), gifPath(_gifPath),
  width(0), height(0), channels(0), minBlockSize(_minBlockSize),
  threshold(_threshold), errorMetricType(_errorMetricType),
  targetCompressionPercentage(_targetCompressionPercentage),
  quadTree(nullptr), compressionPercentage(0.0),
  nodeCount(0), maxDepth(0), executionTime(0),
  originalSize(0), compressedSize(0) {
}

// Memuat gambar dari file
bool ImageProcessor::loadImage() {
    // Cek apakah file ada
    if (!Utils::fileExists(inputPath)) {
        std::cerr << "Error: Input file does not exist - " << inputPath << std::endl;
        return false;
    }

    // Load gambar menggunakan stbi_image
    stbi_set_flip_vertically_on_load(false);
    unsigned char* data = stbi_load(inputPath.c_str(), &width, &height, &channels, 0);

    if (!data) {
        std::cerr << "Error: Failed to load image - " << inputPath << std::endl;
        return false;
    }

    if (minBlockSize > (width * height)) {
        std::cerr << "Error: Minimum block area (" << minBlockSize << " square pixels) is larger than the image area (" 
        << (width * height) << " square pixels)." << std::endl;
        stbi_image_free(data);
        return false;
    }
    // Konversi data gambar ke format yang kita gunakan
    originalImage.resize(height);
    for (int y = 0; y < height; ++y) {
        originalImage[y].resize(width);
        for (int x = 0; x < width; ++x) {
            int idx = (y * width + x) * channels;
            unsigned char r = channels >= 1 ? data[idx] : 0;
            unsigned char g = channels >= 2 ? data[idx + 1] : r;
            unsigned char b = channels >= 3 ? data[idx + 2] : r;
            originalImage[y][x] = RGB(r, g, b);
        }
    }
    // Hitung ukuran file original
    originalSize = Utils::getFileSize(inputPath);

    // Free memory
    stbi_image_free(data);

    std::cout << "Image loaded: " << width << "x" << height
           << ", channels: " << channels << std::endl;

    return true;
}

// Melakukan kompresi gambar
bool ImageProcessor::compressImage() {
    try {
        auto startTime = std::chrono::high_resolution_clock::now();

        // Jika target persentase kompresi diaktifkan
        if (targetCompressionPercentage > 0.0) {
            std::cout << "Adaptive threshold mode enabled." << std::endl;
            std::cout << "Original threshold: " << threshold << std::endl;

            // Cari threshold yang optimal untuk mencapai target persentase
            double newThreshold = findThresholdForTargetCompression();

            std::cout << "Adjusted threshold: " << newThreshold << std::endl;
            threshold = newThreshold;
        }

        // Buat dan bangun QuadTree
        quadTree = make_unique<QuadTree>(originalImage, minBlockSize, threshold, errorMetricType);

        // Set callback jika gif path diset
        if (!gifPath.empty()) {
            std::cout << "GIF recording enabled." << std::endl;
            // Capture frame untuk visualisasi
            vector<vector<vector<RGB>>> frames;

            quadTree->setCompressionCallback([&frames](const vector<vector<RGB>>& frame) {
                frames.push_back(frame);
                std::cout << "Frame captured. Total frames: " << frames.size() << "\r" << std::flush;
            });
        }

        // Bangun tree
        std::cout << "Building quadtree..." << std::endl;
        quadTree->buildTree();
    }

    // Dapatkan hasil kompresi dan metrik
}
```

```

// Dapatkan hasil kompresi dan metrik
std::cout << "Retrieving compressed image..." << std::endl;
compressedImage = quadTree->getCompressedImage();
nodeCount = quadTree->getNodeCount();
maxDepth = quadTree->getMaxDepth();
compressionPercentage = quadTree->getCompressionPercentage();

auto endTime = std::chrono::high_resolution_clock::now();
executionTime = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

// Hitung perkiraan ukuran file terkompresi
compressedSize = calculateImageSize(compressedImage);

// Jika path GIF ditentukan, buat GIF visualisasi
if (!gifPath.empty()) {
    std::cout << "Generating compression visualization GIF..." << std::endl;
    generateCompressionGif();
}

return true;
} catch (const std::exception& e) {
    std::cerr << "Error during image compression: " << e.what() << std::endl;
    return false;
}

// Menyimpan gambar hasil kompresi
bool ImageProcessor::saveCompressedImage() {
    // Validasi
    if (compressedImage.empty() || compressedImage[0].empty()) {
        std::cerr << "Error: No compressed image to save" << std::endl;
        return false;
    }

    // Prepare data untuk stb_image_write
    unsigned char* data = new unsigned char[width * height * 3]; // Always save as RGB

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int idx = (y * width + x) * 3;
            data[idx] = compressedImage[y][x].getRed();
            data[idx + 1] = compressedImage[y][x].getGreen();
            data[idx + 2] = compressedImage[y][x].getBlue();
        }
    }

    // Tentukan format output berdasarkan ekstensi file
    bool success = false;
    string ext = Utils::getFileExtension(outputPath);

    if (ext == "jpg" || ext == "jpeg") {
        success = stbi_write_jpg(outputPath.c_str(), width, height, 3, data, 90); // quality 90
    } else if (ext == "png") {
        success = stbi_write_png(outputPath.c_str(), width, height, 3, data, width * 3);
    } else if (ext == "bmp") {
        success = stbi_write_bmp(outputPath.c_str(), width, height, 3, data);
    } else {
        // Default to PNG
        success = stbi_write_png(outputPath.c_str(), width, height, 3, data, width * 3);
    }

    delete[] data;

    if (!success) {
        std::cerr << "Error: Failed to save compressed image to " << outputPath << std::endl;
        return false;
    }
}

return true;
}

// Menampilkan semua metrik hasil kompresi
void ImageProcessor::displayMetrics() const {
    std::cout << "\n==== Compression Results ===" << std::endl;

    // Informasi tentang threshold dan metode
    std::cout << "Error metric : " << Utils::errorMetricToString(errorMetricType) << std::endl;

    if (targetCompressionPercentage > 0.0) {
        std::cout << "Target compression: " << std::fixed << std::setprecision(1)
            << (targetCompressionPercentage * 100) << "%" << std::endl;
        std::cout << "Final threshold : " << threshold << std::endl;
    } else {
        std::cout << "Threshold : " << threshold << std::endl;
    }

    // Ukuran file dan persentase kompresi
    std::cout << "Original size : " << originalSize << " bytes" << std::endl;

    // Perkiraan ukuran terkompresi berdasarkan file output yang sebenarnya
    size_t actualCompressedSize = compressedSize; // Gunakan nilai saat ini sebagai default
    if (Utils::fileExists(outputPath)) {
        actualCompressedSize = Utils::getFileSize(outputPath);
    }

    std::cout << "Compressed size : " << actualCompressedSize << " bytes" << std::endl;

    double actualCompression = 100.0 * (1.0 - static_cast<double>(actualCompressedSize) / originalSize);
    std::cout << "Compression : " << std::fixed << std::setprecision(2)
        << actualCompression << "%" << std::endl;

    // Statistik QuadTree
}

```

```

        }

        if (difference <= tolerance) {
            std::cout << "Target achieved within tolerance!" << std::endl;
            return currentThreshold;
        }

        if (achievedCompression < targetCompressionPercentage) {

            lowerBound = currentThreshold;
        } else {
            upperBound = currentThreshold;
        }

        iterations++;
    }

    std::cout << "Search completed. Best threshold = " << bestThreshold
        << " (difference = " << (bestDifference * 100) << "%)" << std::endl;

    return bestThreshold;
}

bool ImageProcessor::generateCompressionGif() {
    try {

        std::string directory = gifPath.substr(0, gifPath.find_last_of("\\\\"));
        if (!directory.empty()) {
            Utils::createDirectoryIfNotExists(directory);
        }

        GifWriter gifWriter;
        if (!GifBegin(&gifWriter, gifPath.c_str(), width, height, 10)) {
            std::cerr << "Failed to initialize GIF writer\n";
            return false;
        }

        std::cout << "Generating GIF (streaming)...\\n";

        std::vector<std::vector<RGB>> gifBuffer = originalImage;

        int frameCounter = 0;
        const int frameInterval = 100;

        auto callback = [&](const Block& region, const RGB& avgColor) {

            for (int y = region.getY(); y < region.getY() + region.getHeight(); ++y) {
                for (int x = region.getX(); x < region.getX() + region.getWidth(); ++x) {
                    gifBuffer[y][x] = avgColor;
                }
            }

            if (frameCounter % frameInterval == 0) {
                std::vector<uint8_t> frameData(width * height * 4);
                for (int y = 0; y < height; ++y) {
                    for (int x = 0; x < width; ++x) {
                        int idx = (y * width + x) * 4;
                        const RGB& pixel = gifBuffer[y][x];
                        frameData[idx + 0] = pixel.getRed();
                        frameData[idx + 1] = pixel.getGreen();
                        frameData[idx + 2] = pixel.getBlue();
                        frameData[idx + 3] = 255;
                    }
                }
                std::cout << "Callback triggered at frameCounter = " << frameCounter << std::endl;
                GifWriteFrame(&gifWriter, frameData.data(), width, height, 10);
                std::cout << "Frame " << (frameCounter / frameInterval) << " written\\n";
            }
        };

        frameCounter++;

        Sleep(1);
    };
}

QuadTree treeCopy(originalImage, minBlockSize, threshold, errorMetricType);
treeCopy.setCompressionRegionCallback(callback);
treeCopy.buildTree();

std::vector<uint8_t> finalFrame(width * height * 4);

```

3.5 Node.cpp

```
Node::Node(const Block& _region, const RGB& _avgColor, bool _isLeaf)
| : region(_region), avgColor(_avgColor), isLeaf(_isLeaf),
|   topLeft(nullptr), topRight(nullptr), bottomLeft(nullptr), bottomRight(nullptr) {}

const Block& Node::getRegion() const {
    return region;
}

const RGB& Node::getAvgColor() const {
    return avgColor;
}

bool Node::getIsLeaf() const {
    return isLeaf;
}

// Mendapatkan child node pada kuadran tertentu
Node* Node::getChild(int quadrant) const {
    switch (quadrant) {
        case 0: // Top-left
        |     return topLeft.get();
        case 1: // Top-right
        |     return topRight.get();
        case 2: // Bottom-left
        |     return bottomLeft.get();
        case 3: // Bottom-right
        |     return bottomRight.get();
        default:
        |     return nullptr;
    }
}
```

3.6 QuadTree.cpp

```
#include "QuadTree.hpp"
#include "ErrorMetrics.hpp"
#include <algorithm>
#include <iostream>

// Constructor
QuadTree::QuadTree(const vector<vector<RGB>>& _image, int _minBlockSize, double _threshold,
    ErrorMetricType _errorMetric)
: image(_image), minBlockSize(_minBlockSize), threshold(_threshold),
  errorMetric(_errorMetric), nodeCount(0), maxDepth(0), root(nullptr) {
    this->maxDepth = maxDepth;
}

// Membangun QuadTree dengan pendekatan divide and conquer
void QuadTree::buildTree() {
    // Validasi gambar
    if (image.empty() || image[0].empty()) {
        std::cerr << "Error: Empty image" << std::endl;
        return;
    }

    // Buat root node dengan seluruh gambar sebagai region
    int width = image[0].size();
    int height = image.size();
    Block fullImageBlock(0, 0, width, height);

    // Hitung warna rata-rata untuk seluruh gambar
    RGB avgColor = calculateAverageColor(fullImageBlock);

    // Buat root node
    root = make_unique<Node>(fullImageBlock, avgColor);

    // Mulai proses subdivisi dari root node
    subdivide(root.get(), 0);

    // Hitung jumlah node dan kedalaman maksimum
    nodeCount = countNodes(root.get());
    maxDepth = calculateMaxDepth(root.get(), 0);
}

// Mendapatkan hasil kompresi sebagai gambar
vector<vector<RGB>> QuadTree::getCompressedImage() const {
    // Buat gambar kosong dengan ukuran yang sama
    int height = image.size();
    int width = image.empty() ? 0 : image[0].size();
    vector<vector<RGB>> result(height, vector<RGB>(width));

    // Jika tree belum dibangun
    if (!root) {
        return result;
    }

    // Rekursi untuk membangun gambar hasil kompresi
    buildCompressedImage(result, root.get());

    return result;
}

// Getter untuk jumlah node
int QuadTree::getNodeCount() const {
    return nodeCount;
}

// Getter untuk kedalaman maksimum
int QuadTree::getMaxDepth() const {
    return maxDepth;
}

// Menghitung persentase kompresi
double QuadTree::getCompressionPercentage() const {
    // Ukuran gambar asli (setiap piksel = 3 byte RGB)
    size_t originalSize = image.size() * image[0].size() * 3;

    // Ukuran terkompresi (setiap node leaf = 3 byte untuk warna + 16 byte untuk posisi & ukuran)
    size_t compressedSize = nodeCount * (3 + 16);

    // Hitung persentase kompresi
    double compressionRatio = 1.0 - (static_cast<double>(compressedSize) / originalSize);

    // Batasi ke range 0.0-1.0
    return std::max(0.0, std::min(1.0, compressionRatio));
}
```

```

vector<vector<RGB>> QuadTree::getCurrentStateImage() const {
    // Buat gambar kosong dengan ukuran yang sama
    int height = image.size();
    int width = image.empty() ? 0 : image[0].size();
    vector<vector<RGB>> result(height, vector<RGB>(width));

    // Jika tree belum dibangun
    if (!root) {
        return result;
    }

    // Rekursi untuk membangun gambar hasil kompresi saat ini
    buildCompressedImage(result, root.get());

    return result;
}

// Private methods

// Membagi node dengan pendekatan divide and conquer
void QuadTree::subdivide(Node* node, int depth) {
    if (!node) return;
    static int nodeProcessedCount = 0;

    // Hitung error untuk region saat ini
    double error = calculateError(node->region, node->avgColor);

    // if (error > 1000000 || error < 0) { // Deteksi nilai ekstrem //DEBUGGING
    //     std::cerr << "Warning: Extreme error value " << error
    //     << " for method " << errorMetric << std::endl;
    // }

    // Tentukan apakah perlu subdivisi
    bool shouldSubdivide = error > threshold;
    int subBlockWidth = node->region.getWidth() / 2;
    int subBlockHeight = node->region.getHeight() / 2;

    // Calculate area of potential sub-blocks
    int subBlockArea = subBlockWidth * subBlockHeight;

    // Periksa ukuran minimum blok
    if (subBlockArea < minBlockSize) {
        shouldSubdivide = false;
    }

    if (shouldSubdivide) {
        nodeProcessedCount++;
        // Ubah status node menjadi internal (bukan leaf)
        node->isLeaf = false;

        // Buat sub-regions untuk 4 kuadran
        Block topLeftRegion = node->region.getQuadrant(0);
        Block topRightRegion = node->region.getQuadrant(1);
        Block bottomLeftRegion = node->region.getQuadrant(2);
        Block bottomRightRegion = node->region.getQuadrant(3);

        // Hitung warna rata-rata untuk setiap kuadran
        RGB topLeftColor = calculateAverageColor(topLeftRegion);
        RGB topRightColor = calculateAverageColor(topRightRegion);
        RGB bottomLeftColor = calculateAverageColor(bottomLeftRegion);
        RGB bottomRightColor = calculateAverageColor(bottomRightRegion);

        // Buat child nodes
        node->topLeft = make_unique<Node>(topLeftRegion, topLeftColor);
        node->topRight = make_unique<Node>(topRightRegion, topRightColor);
        node->bottomLeft = make_unique<Node>(bottomLeftRegion, bottomLeftColor);
        node->bottomRight = make_unique<Node>(bottomRightRegion, bottomRightColor);

        // Panggil callback untuk visualisasi jika tersedia
        if (compressionRegionCallback && nodeProcessedCount % 100 == 0) {
            compressionRegionCallback(node->region, node->avgColor);
        }

        // Rekursif subdivisi child nodes
        subdivide(node->topLeft.get(), depth + 1);
        subdivide(node->topRight.get(), depth + 1);
        subdivide(node->bottomLeft.get(), depth + 1);
        subdivide(node->bottomRight.get(), depth + 1);
    }
}

```

```

RGB QuadTree::calculateAverageColor(const Block& region) const {
    int width = region.getWidth();
    int height = region.getHeight();
    int startX = region.getX();
    int startY = region.getY();

    if (width <= 0 || height <= 0 ||
        startX >= image[0].size() || startY >= image.size()) {
        return RGB(0, 0, 0);
    }

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

    for (int y = startY; y < startY + height && y < image.size(); y++) {
        for (int x = startX; x < startX + width && x < image[y].size(); x++) {
            sumR += image[y][x].getRed();
            sumG += image[y][x].getGreen();
            sumB += image[y][x].getBlue();
            count++;
        }
    }

    if (count == 0) {
        return RGB(0, 0, 0);
    }

    unsigned char avgR = static_cast<unsigned char>(sumR / count);
    unsigned char avgG = static_cast<unsigned char>(sumG / count);
    unsigned char avgB = static_cast<unsigned char>(sumB / count);

    return RGB(avgR, avgG, avgB);
}

// Menghitung error untuk region berdasarkan metode yang dipilih
double QuadTree::calculateError(const Block& region, const RGB& avgColor) const {
    switch (errorMetric) {
        case VARIANCE:
            return ErrorMetrics::calculateVariance(this->image, region, avgColor);
        case MEAN_ABSOLUTE_DEVIATION:
            return ErrorMetrics::calculateMAD(this->image, region, avgColor);
        case MAX_PIXEL_DIFFERENCE:
            return ErrorMetrics::calculateMaxDifference(this->image, region, avgColor);
        case ENTROPY:
            return ErrorMetrics::calculateEntropy(this->image, region);
        case SSIM:
            return ErrorMetrics::calculateSSIM(this->image, region, avgColor);
        default:
            return ErrorMetrics::calculateVariance(this->image, region, avgColor);
    }
}

// Menghitung kedalaman maksimum dari pohon
int QuadTree::calculateMaxDepth(const Node* node, int currentDepth) const {
    if (!node) {
        return currentDepth - 1;
    }

    // Jika leaf node, kembalikan kedalaman saat ini
    if (node->isLeaf) {
        return currentDepth;
    }

    // Rekursi ke semua child nodes dan ambil kedalaman maksimum
    int depthTL = calculateMaxDepth(node->topLeft.get(), currentDepth + 1);
    int depthTR = calculateMaxDepth(node->topRight.get(), currentDepth + 1);
    int depthBL = calculateMaxDepth(node->bottomLeft.get(), currentDepth + 1);
    int depthBR = calculateMaxDepth(node->bottomRight.get(), currentDepth + 1);

    return std::max({depthTL, depthTR, depthBL, depthBR});
}

// Menghitung jumlah node dalam pohon
int QuadTree::countNodes(const Node* node) const {
    if (!node) {
        return 0;
    }

    // Hitung node saat ini
    int count = 1;

    // Jika bukan leaf node, tambahkan jumlah node dari semua child
    if (!node->isLeaf) {
        count += countNodes(node->topLeft.get());
        count += countNodes(node->topRight.get());
        count += countNodes(node->bottomLeft.get());
        count += countNodes(node->bottomRight.get());
    }

    return count;
}

// Membangun gambar hasil kompresi
void QuadTree::buildCompressedImage(vector<vector<RGB>>& result, const Node* node) const {
    if (!node) {
        return;
    }

    // Jika leaf nodo, isi region dengan warna rata-rata
    if (node->isLeaf) {
        int startX = node->region.getX();
        int startY = node->region.getY();
        int width = node->region.getWidth();
        int height = node->region.getHeight();

        for (int y = startY; y < startY + height && y < result.size(); y++) {
            for (int x = startX; x < startX + width && x < result[y].size(); x++) {
                result[y][x] = node->avgColor;
            }
        }
    } else {
        // Jika bukan leaf node, rekursi ke semua child nodes
        buildCompressedImage(result, node->topLeft.get());
        buildCompressedImage(result, node->topRight.get());
        buildCompressedImage(result, node->bottomLeft.get());
        buildCompressedImage(result, node->bottomRight.get());
    }
}

void QuadTree::setCompressionRegionCallback(const std::function<void(const Block&, const RGB&)>& cb) {
    this->compressionRegionCallback = cb;
}

```

3.7 RGB.cpp

```
#include "RGB.hpp"

RGB::RGB() : r(0), g(0), b(0) {}

RGB::RGB(unsigned char red, unsigned char green, unsigned char blue)
    : r(red), g(green), b(blue) {}

unsigned char RGB::getRed() const {
    return r;
}

unsigned char RGB::getGreen() const {
    return g;
}

unsigned char RGB::getBlue() const {
    return b;
}

void RGB::setRed(unsigned char red) {
    r = red;
}

void RGB::setGreen(unsigned char green) {
    g = green;
}

void RGB::setBlue(unsigned char blue) {
    b = blue;
}

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

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

3.8 Utils.cpp

```
void Utils::getThresholdLimits(QuadTree::ErrorMetricType method, double& minThreshold, double& maxThreshold) {
    switch(method) {
        case QuadTree::VARIANCE:
            minThreshold = 0.0;
            maxThreshold = 1000;
            break;
        case QuadTree::MEAN_ABSOLUTE_DEVIATION:
            minThreshold = 2;
            maxThreshold = 50;
            break;
        case QuadTree::MAX_PIXEL_DIFFERENCE:
            minThreshold = 3;
            maxThreshold = 225;
            break;
        case QuadTree::ENTROPY:
            minThreshold = 0.0;
            maxThreshold = 6;
            break;
        case QuadTree::SSIM:
            minThreshold = 0.0;
            maxThreshold = 1;
            break;
        default:
            minThreshold = 0.0;
            maxThreshold = 1000;
    }
}

std::string Utils::getDefaultGifPath(const std::string& inputPath,
    QuadTree::ErrorMetricType method,
    double threshold,
    int minBlockSize,
    double percentageCompression) {
    size_t lastSlash = inputPath.find_last_of("//");
    std::string directory = (lastSlash != std::string::npos) ?
        inputPath.substr(0, lastSlash + 1) : "";
    std::string filename = (lastSlash != std::string::npos) ?
        inputPath.substr(lastSlash + 1) : inputPath;
    size_t lastDot = filename.find_last_of(".");
    std::string baseName = (lastDot != std::string::npos) ?
        filename.substr(0, lastDot) : filename;
    std::string gifDir;
    size_t inputPos = directory.find("input");
    if (inputPos != std::string::npos) {
        gifDir = directory.substr(0, inputPos) + "output/gif/";
    } else {
        gifDir = directory + "output/gif/";
    }
    createDirectoryIfNotExists(gifDir);
    std::stringstream ss;
    if (percentageCompression > 0.0) {
        ss << baseName << "_percentage_" << percentageCompression;
    } else {
        ss << baseName << "_" << static_cast<int>(method) << "_"
        << threshold << "_" << minBlockSize;
    }
    ss << ".gif";
    return gifDir + ss.str();
}

std::string Utils::getDefaultOutputPath(const std::string& inputPath,
    QuadTree::ErrorMetricType method,
    double threshold,
    int minBlockSize,
    double percentageCompression) {
    size_t lastSlash = inputPath.find_last_of("//");
    std::string directory = (lastSlash != std::string::npos) ?
        inputPath.substr(0, lastSlash + 1) : "";
    std::string filename = (lastSlash != std::string::npos) ?
        inputPath.substr(lastSlash + 1) : inputPath;
    size_t lastDot = filename.find_last_of(".");
    std::string baseName = (lastDot != std::string::npos) ?
        filename.substr(0, lastDot) : filename;
```

```

        std::string extension = (lastDot != std::string::npos) ?
            filename.substr(lastDot) : "";

        std::string outputDirectory = directory;
        size_t inputPos = outputDirectory.find("input");
        if (inputPos != std::string::npos) {
            outputDirectory.replace(inputPos, 5, "output");
        } else {

            size_t lastDirSlash = outputDirectory.substr(0, outputDirectory.length() - 1).find_last_of("/\\");
            if (lastDirSlash != std::string::npos)
                outputDirectory = outputDirectory.substr(0, lastDirSlash + 1) + "output/";
            } else {

                outputDirectory = directory;
            }
        }

        std::stringstream ss;
        if (percentageCompression > 0.0) {
            ss << baseName << "_percentage_" << percentageCompression;
        } else {
            ss << baseName << "_" << static_cast<int>(method) << "_"
            << threshold << "_" << minBlockSize;
        }
        ss << extension;

        return outputDirectory + ss.str();
    }

    bool Utils::fileExists(const std::string& filePath) {
        std::ifstream file(filePath);
        return file.good();
    }

    size_t Utils::getFileSize(const std::string& filePath) {
        std::ifstream file(filePath, std::ios::binary | std::ios::ate);
        if (!file.good())
            std::cerr << "Error getting file size for " << filePath << std::endl;
        return 0;
    }
    return static_cast<size_t>(file.tellg());
}

std::string Utils::getFileExtension(const std::string& filePath) {
    size_t dotPos = filePath.find_last_of('.');
    if (dotPos != std::string::npos) {
        return filePath.substr(dotPos + 1);
    }
    return "";
}

std::string Utils::errorMetricToString(QuadTree::ErrorMetricType method) {
    switch (method) {
        case QuadTree::VARIANCE: return "Variance";
        case QuadTree::MEAN_ABSOLUTE_DEVIATION: return "Mean Absolute Deviation";
        case QuadTree::MAX_PIXEL_DIFFERENCE: return "Max Pixel Difference";
        case QuadTree::ENTROPY: return "Entropy";
        case QuadTree::SSIM: return "Structural Similarity Index (SSIM)";
        default: return "Unknown";
    }
}

double Utils::getDefaultThreshold(QuadTree::ErrorMetricType method) {
    switch (method) {
        case QuadTree::VARIANCE: return 20.0;
        case QuadTree::MEAN_ABSOLUTE_DEVIATION: return 15.0;
        case QuadTree::MAX_PIXEL_DIFFERENCE: return 50.0;
        case QuadTree::ENTROPY: return 0.5;
        case QuadTree::SSIM: return 0.15;
        default: return 20.0;
    }
}

int Utils::getDefaultMinBlockArea() {
    return 16;
}

```

Bab Eksperimen

4.1 Variansi

Input

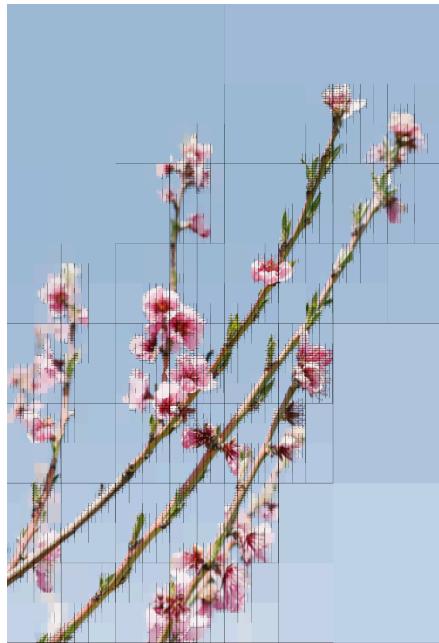


Kasus 1 (Threshold 100, Minimum block 4, Target 0)

Hasil

```
==== Compression Results ====
Error metric      : Variance
Threshold        : 100
Original size    : 5002028 bytes
Compressed size  : 743102 bytes
Compression       : 85.14%
Min block size   : 4 pixels
QuadTree nodes   : 53901
Maximum depth    : 9
Execution time   : 3631 ms
Image successfully compressed and saved to ..\test\output\exp1_1.png
Visualization GIF saved to ..\test\gif\exp1_1.gif
```

Hasil Gambar :

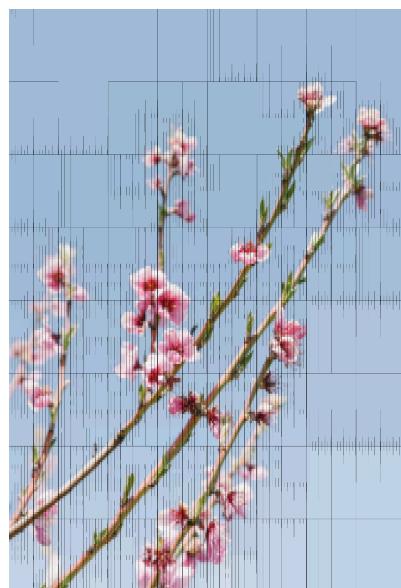


Kasus 2 (Threshold 500, Minimum block 8, Target 0.7)

Hasil

```
==== Compression Results ===
Error metric      : Variance
Target compression: 70.0%
Final threshold   : 0.1
Original size     : 5002028 bytes
Compressed size   : 590437 bytes
Compression       : 88.20%
Min block size    : 8 pixels
QuadTree nodes    : 38529
Maximum depth     : 8
Execution time    : 60571 ms
Image successfully compressed and saved to ..\test\output\exp1_2.png
Visualization GIF saved to ..\test\gif\exp1_2.gif
```

Hasil Gambar :



Kasus 3 (Threshold 1000, Minimum block 16, Target 0.9)

Hasil

```
==== Compression Results ===
Error metric      : Variance
Target compression: 90.0%
Final threshold   : 0.1
Original size     : 5002028 bytes
Compressed size   : 515190 bytes
Compression       : 89.70%
Min block size    : 16 pixels
QuadTree nodes    : 11849
Maximum depth     : 7
Execution time    : 56877 ms
Image successfully compressed and saved to ..\test\output\exp1_3.png
Visualization GIF saved to ..\test\gif\exp1_3.gif
```

Hasil Gambar :



4.2 MAD

Input

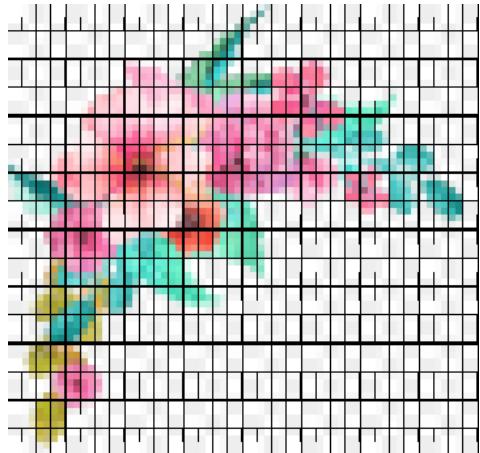


Kasus 1 (Threshold 500, Minimum block 8, Target 0.7)

Hasil

```
==== Compression Results ====
Error metric      : Mean Absolute Deviation (MAD)
Threshold        : 5
Original size    : 30288 bytes
Compressed size  : 15190 bytes
Compression      : 49.85%
Min block size   : 4 pixels
QuadTree nodes   : 4805
Maximum depth    : 6
Execution time   : 54 ms
Image successfully compressed and saved to ..\test\output\exp2_1.png
Visualization GIF saved to ..\test\gif\exp2_1.gif
```

Hasil Gambar :

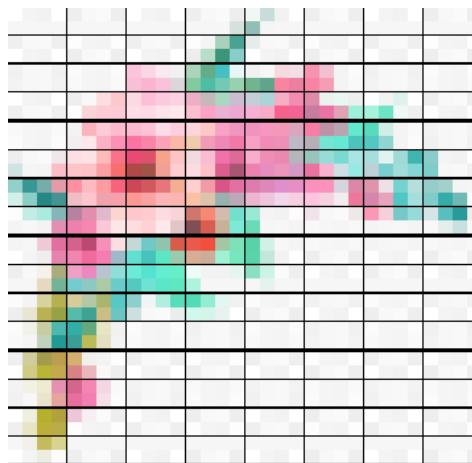


Kasus 2 (Threshold 20, Minimum block 8, Target 0.7)

Hasil

```
== Compression Results ==
Error metric      : Mean Absolute Deviation (MAD)
Target compression: 70.0%
Final threshold   : 7.9
Original size     : 30288 bytes
Compressed size   : 7754 bytes
Compression       : 74.40%
Min block size    : 8 pixels
QuadTree nodes    : 1365
Maximum depth     : 5
Execution time    : 630 ms
Image successfully compressed and saved to ..\test\output\exp2_2.png
Visualization GIF saved to ..\test\gif\exp2_2.gif
```

Hasil Gambar :

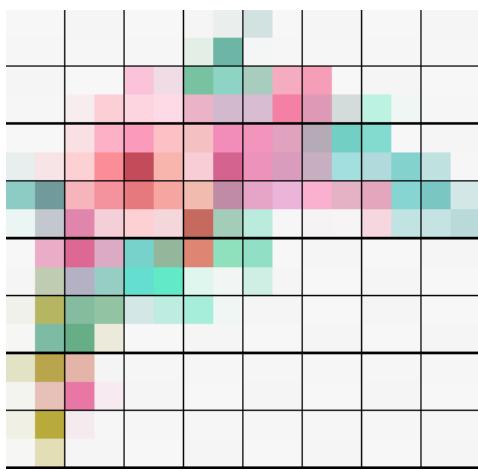


Kasus 3 (Threshold 50, Minimum block 16, Target 0.9)

Hasil

```
== Compression Results ==
Error metric      : Mean Absolute Deviation (MAD)
Target compression: 90.0%
Final threshold   : 7.9
Original size     : 30288 bytes
Compressed size   : 5190 bytes
Compression       : 82.86%
Min block size    : 16 pixels
QuadTree nodes    : 341
Maximum depth     : 4
Execution time    : 568 ms
Image successfully compressed and saved to ..\test\output\exp2_3.png
Visualization GIF saved to ..\test\gif\exp2_3.gif
```

Hasil Gambar :



4.3 MPD

Input



Kasus 1 (Threshold 10, Minimum block 4, Target 0.0)

Hasil

```
== Compression Results ==
Error metric      : Max Pixel Difference
Threshold        : 10
Original size    : 5002028 bytes
Compressed size  : 845316 bytes
Compression       : 83.10%
Min block size   : 4 pixels
QuadTree nodes   : 84181
Maximum depth    : 9
Execution time   : 4043 ms
Image successfully compressed and saved to ..\test\output\exp3_1.png
Visualization GIF saved to ..\test\gif\exp3_1.gif
```

Hasil Gambar :



Kasus 2 (Threshold 30, Minimum block 8, Target 0.7)

Hasil

```
==== Compression Results ====
Error metric      : Max Pixel Difference
Target compression: 70.0%
Final threshold   : 0.2
Original size     : 5002028 bytes
Compressed size   : 590886 bytes
Compression       : 88.19%
Min block size    : 8 pixels
QuadTree nodes    : 41393
Maximum depth     : 8
Execution time    : 78601 ms
Image successfully compressed and saved to ..\test\output\exp3_2.png
Visualization GIF saved to ..\test\gif\exp3_2.gif
```

Hasil Gambar :



Kasus 3 (Threshold 60, Minimum block 16, Target 0.9)

Hasil

```
==== Compression Results ====
Error metric      : Max Pixel Difference
Target compression: 90.0%
Final threshold   : 0.2
Original size     : 5002028 bytes
Compressed size   : 515246 bytes
Compression       : 89.70%
Min block size    : 16 pixels
QuadTree nodes    : 12829
Maximum depth     : 7
Execution time    : 64625 ms
Image successfully compressed and saved to ..\test\output\exp3_3.png
Visualization GIF saved to ..\test\gif\exp3_3.gif
```

Hasil Gambar :



4.4 Entropy

Input

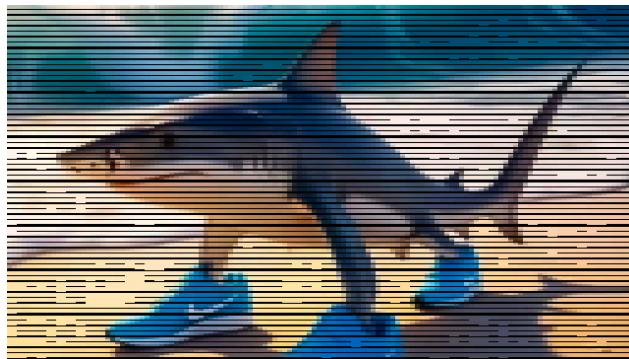


Kasus 1 (Threshold 0.5, Minimum block 4, Target 0.0)

Hasil

```
== Compression Results ==
Error metric      : Entropy
Threshold         : 0.5
Original size    : 15275 bytes
Compressed size  : 59304 bytes
Compression       : -288.24%
Min block size   : 4 pixels
QuadTree nodes   : 21117
Maximum depth    : 7
Execution time   : 864 ms
Image successfully compressed and saved to ..\test\output\exp4_1.png
Visualization GIF saved to ..\test\gif\exp4_1.gif
```

Hasil Gambar :



Kasus 2 (Threshold 1.5, Minimum block 8, Target 0.7)

Hasil

```
== Compression Results ==
Error metric      : Entropy
Target compression: 70.0%
Final threshold   : 0.2
Original size    : 15275 bytes
Compressed size  : 24221 bytes
Compression       : -58.57%
Min block size   : 8 pixels
QuadTree nodes   : 5457
Maximum depth    : 6
Execution time   : 7422 ms
Image successfully compressed and saved to ..\test\output\exp4_2.png
Visualization GIF saved to ..\test\gif\exp4_2.gif
```

Hasil Gambar :



Kasus 3 (Threshold 3, Minimum block 16, Target 0.9)

Hasil

```
== Compression Results ==
Error metric      : Entropy
Target compression: 90.0%
Final threshold   : 1.1
Original size     : 15275 bytes
Compressed size   : 11987 bytes
Compression       : 21.53%
Min block size    : 16 pixels
QuadTree nodes    : 1365
Maximum depth     : 5
Execution time    : 6556 ms
Image successfully compressed and saved to ..\test\output\exp4_3.png
Visualization GIF saved to ..\test\gif\exp4_3.gif
```

Hasil Gambar :



4.5 SSIM

Input



Kasus 1 (Threshold 0.05, Minimum block 4, Target 0.0)

Hasil

```
== Compression Results ==
Error metric : Structural Similarity Index (SSIM)
Threshold : 0.05
Original size : 7968 bytes
Compressed size : 6745 bytes
Compression : 15.35%
Min block size : 4 pixels
QuadTree nodes : 1329
Maximum depth : 5
Execution time : 14 ms
Image successfully compressed and saved to ..\test\output\exp5_1.png
Visualization GIF saved to ..\test\gif\exp5_1.gif
```

Hasil Gambar :

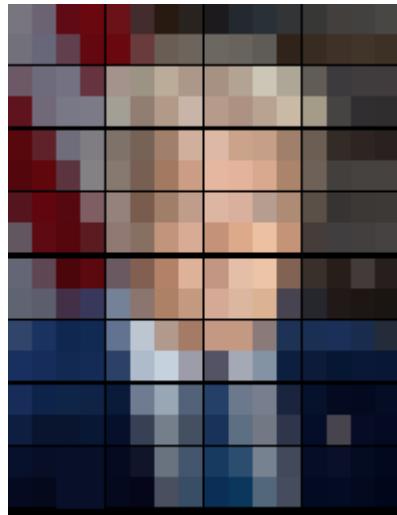


Kasus 2 (Threshold 0.1, Minimum block 8, Target 0.7)

Hasil

```
== Compression Results ==
Error metric : Structural Similarity Index (SSIM)
Target compression: 70.0%
Final threshold : 0.2
Original size : 7968 bytes
Compressed size : 3204 bytes
Compression : 59.79%
Min block size : 8 pixels
QuadTree nodes : 337
Maximum depth : 4
Execution time : 124 ms
Image successfully compressed and saved to ..\test\output\exp5_2.png
Visualization GIF saved to ..\test\gif\exp5_2.gif
```

Hasil Gambar :

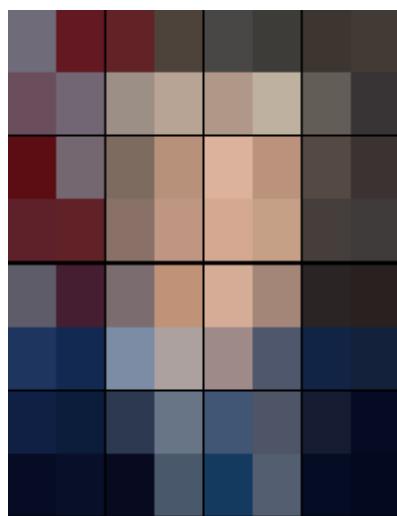


Kasus 3 (Threshold 3, Minimum block 16, Target 0.9)

Hasil

```
== Compression Results ==
Error metric      : Structural Similarity Index (SSIM)
Target compression: 90.0%
Final threshold   : 0.6
Original size     : 7968 bytes
Compressed size   : 2181 bytes
Compression       : 72.63%
Min block size    : 16 pixels
QuadTree nodes    : 85
Maximum depth     : 3
Execution time    : 118 ms
Image successfully compressed and saved to ..\test\output\exp5_3.png
Visualization GIF saved to ..\test\gif\exp5_3.gif
```

Hasil Gambar :



Bab Analisis

Algoritma divide and conquer yang digunakan dalam program ini merupakan pendekatan rekursif untuk membagi gambar menjadi blok-blok yang lebih kecil hingga memenuhi kondisi homogenitas

berdasarkan metode perhitungan error tertentu (seperti Variance, MAD, MPD, Entropy, atau SSIM). Algoritma Quadtree bekerja dengan prinsip membagi blok gambar secara rekursif menjadi empat bagian sampai ukuran minimum tercapai atau blok dianggap cukup seragam.

5.1 Analisis Kompleksitas Waktu

Kompleksitas waktu dalam kasus terburuk adalah $O(n^2)$.

di mana n adalah sisi gambar (jika gambar persegi $n \times n$). Hal ini terjadi jika setiap blok terus dibelah hingga unit terkecil (1x1), sehingga jumlah simpul (node) mendekati $O(n^2)$ karena setiap subdivisi menghasilkan 4 anak.

Pada praktiknya, kompleksitas tergantung pada:

- Metode error: metode seperti SSIM dan Entropy lebih berat daripada MPD atau MAD.
- Threshold: semakin kecil threshold, semakin banyak pembagian blok → lebih dalam rekursi.
- Karakteristik gambar: gambar dengan banyak detail (seperti foto) akan lebih kompleks diproses dibanding gambar polos (seperti logo atau grafik).

Waktu rekursi per node tergantung metode error:

Variance	$O(w \cdot h)$
MAD	$O(w \cdot h)$
MPD	$O(w \cdot h)$
Entropy	$O(w \cdot h) + O(k)$
SSIM	$O(w \cdot h)$

5.2 Analisis Kompleksitas Ruang

Kompleksitas ruang adalah $O(n^2)$

karena gambar disimpan dalam matriks dua dimensi dan setiap node menyimpan informasi blok dan anak-anaknya secara eksplisit.

Tambahan ruang digunakan untuk:

- Menyimpan struktur pohon (Node)
- Buffer gambar hasil kompresi
- Visualisasi GIF (jika diaktifkan)

Analisis Percobaan Berdasarkan Gambar

- Pada gambar sederhana seperti logo hitam-putih atau simbol, jumlah subdivisi sangat sedikit dan menghasilkan pohon yang dangkal.
- Pada gambar kompleks seperti foto atau ilustrasi berwarna, pembagian terjadi lebih sering, menghasilkan pohon yang dalam dan banyak node.
- Ketika threshold terlalu kecil, program akan membagi hampir semua blok — hasilnya akurat, tapi lambat dan besar ukurannya.
- Ketika threshold terlalu besar, gambar terlalu kasar dan kompresi tidak efektif.

Bab Implementasi Bonus

6.1 SSIM

Dalam program ini, SSIM diadaptasi untuk digunakan dalam proses kompresi gambar berbasis Quadtree, dengan tujuan menilai homogenitas blok gambar secara lebih representatif dibandingkan metode numerik seperti Variance atau Mean Absolute Deviation. SSIM digunakan untuk menentukan apakah suatu blok gambar cukup “mirip” secara struktur dengan versi penyederhanaannya (dalam hal ini, blok yang hanya diwakili oleh satu warna rata-rata). Jika nilai SSIM tinggi (mendekati 1), maka blok dianggap cukup seragam dan tidak perlu dibagi lagi. Nilai SSIM kemudian dikonversi menjadi nilai error melalui:

```
error = 1.0 - SSIM;
```

1. Inisialisasi nilai-nilai rata-rata dan statistik:
 - meanX adalah rata-rata piksel asli dari channel R, G, B
 - meanY adalah warna rata-rata (avgColor)
 - Variansi dan kovarians dihitung dari perbedaan terhadap nilai rata-rata
2. Perhitungan statistik:
 - Untuk setiap piksel (x, y):

```
double diffX = channel[x][y] - meanX;  
double diffY = avgColor.getChannel() - meanY;  
covXY += diffX * diffY;
```

```
varX += diffX * diffX;  
varY += diffY * diffY;
```

3. Konversi ke Nilai SSIM dengan Konstanta Stabilitas

```
double C1 = 6.5025, C2 = 58.5225;  
SSIM = ((2 * meanX * meanY + C1) * (2 * covXY + C2)) /  
((meanX * meanX + meanY * meanY + C1) * (varX + varY + C2));
```

4. Hasil akhir dikembalikan sebagai error 1 - SSIM

6.2 Gif

Implementasi dibuat bertujuan untuk merekam proses pembentukan gambar hasil kompresi Quadtree secara bertahap, lalu menyimpannya dalam bentuk file GIF animasi. Setiap frame dalam GIF memperlihatkan hasil sementara dari proses kompresi, saat blok-blok dalam gambar dipecah menjadi bagian yang lebih kecil. Selain itu, program memanfaatkan gif.h sebagai library dalam mengimplementasikan program ini.

1. Inisialisasi GIF Writer

- Gif menggunakan library eksternal gif.h untuk menulis GIF.
- Proses diawali dengan memanggil:

```
GifBegin(&gifWriter, gifPath.c_str(), width, height, 10);
```

Artinya file GIF mulai dengan ukuran $\text{width} \times \text{height}$ dan delay antar frame = 10 (0.1 detik).

2. Menyiapkan gifBuffer

- Buffer gifBuffer bertipe `vector<vector<RGB>>` yang merepresentasikan kondisi gambar pada frame saat ini.

3. Callback Streaming Visualisasi ke GIF

- Diatur dengan fungsi : `setCompressionRegionCallback(callback)`
- Callback dipanggil setiap N node (misalnya `frameInterval = 100`) di `QuadTree::subdivide()`
- Dalam callback:
 - Update gifBuffer berdasarkan blok (Block) dan warna rata-rata (`avgColor`)
 - Lalu mengkonversi buffer tersebut menjadi array RGBA (`uint8_t[]`)
 - Array ini ditulis ke frame GIF melalui:

```
GifWriteFrame(&gifWriter, frameData.data(), width, height, 10);
```

4. Menutup GIF Setelah GIF Terbentuk

- Setelah semua node selesai diproses, tulis frame terakhir

```
GifWriteFrame(&gifWriter, finalFrame.data(), width, height, 10);
```

- Kemudian gif ditutup :

```
GifEnd(&gifWriter);
```

Bab Kesimpulan

Dalam tugas kecil ini, telah dikembangkan sebuah program kompresi gambar menggunakan algoritma Quadtree berbasis divide and conquer. Pendekatan ini bekerja dengan cara membagi gambar secara rekursif menjadi empat bagian jika sebuah blok tidak memenuhi kriteria homogenitas yang ditentukan oleh metode perhitungan error. Proses pembagian dilakukan hingga blok dianggap cukup seragam atau telah mencapai ukuran minimum yang ditentukan. Dengan struktur rekursif ini, program dapat menyesuaikan tingkat detail kompresi secara adaptif terhadap konten lokal gambar, sehingga lebih efisien dibandingkan metode kompresi statis.

Hasil eksperimen menunjukkan bahwa pendekatan divide and conquer ini efektif digunakan bersama lima metode error yang didukung, yaitu Variance, MAD, MPD, Entropy, dan SSIM. Variance dan SSIM menghasilkan kualitas visual tinggi namun dengan biaya komputasi yang lebih besar. MAD dan MPD lebih ringan dan cepat, cocok untuk gambar yang lebih sederhana atau kontras tinggi. Entropy berada di tengah-tengah, dengan kemampuan mendeteksi kompleksitas visual yang baik, terutama pada gambar natural. Strategi pembagian blok berbasis error ini menunjukkan bahwa pemilihan metode dan threshold sangat mempengaruhi struktur pohon Quadtree, kedalaman rekursi, serta ukuran dan kualitas hasil kompresi.

Program mampu mengatur keseimbangan antara kualitas visual dan ukuran file dengan parameter yang fleksibel. Dengan pencatatan statistik seperti jumlah simpul, kedalaman pohon, waktu eksekusi, dan rasio kompresi, pengguna dapat mengevaluasi performa metode yang digunakan. Metode error yang beragam memungkinkan pengguna memilih strategi kompresi yang sesuai dengan jenis gambar dan kebutuhan praktis, baik untuk kualitas tinggi maupun ukuran file minimum.

Lampiran

Link repository

https://github.com/WwzFwz/Tucil2_13523065_13523117.git

Tabel CheckList

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	✓	