

Laporan Tugas Kecil 2 IF2211 Strategi Algoritma: Kompresi Gambar dengan Metode Quadtree

Dave Daniell Yanni – 13523003¹, Nathaniel Jonathan Rusli – 13523013²

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

¹13523003@std.stei.itb.ac.id, ²13523013@std.stei.itb.ac.id

Daftar Isi

<u>Deskripsi Masalah</u>	<u>2</u>
<u>Perancangan Algoritma Divide and Conquer</u>	<u>3</u>
<u>Source Code Program</u>	<u>6</u>
<u>Test Case: Input & Output</u>	<u>18</u>
1. Test Case 1	18
2. Test Case 2	21
3. Test Case 3	24
<u>Analisis Algoritma Divide & Conquer Quadtree</u>	<u>28</u>
<u>Implementasi Bonus</u>	<u>30</u>
1. Bonus 1: Compression Percentage	30
2. Bonus 2: Structural Similarity Index (SSIM)	33
3. Bonus 3: GIF Generation	36
<u>Lampiran</u>	<u>39</u>

Deskripsi Masalah



Gambar 1. Quadtree dalam Kompresi Gambar

(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 gambar, Quadtree membagi gambar menjadi blok-blok kecil berdasarkan keseragaman warna atau intensitas piksel. Prosesnya dimulai dengan membagi gambar menjadi empat bagian, lalu memeriksa apakah setiap bagian memiliki nilai yang seragam berdasarkan analisis sistem warna RGB, yaitu dengan membandingkan komposisi nilai merah (R), hijau (G), dan biru (B) pada piksel-piksel di dalamnya. Jika bagian tersebut tidak seragam, maka bagian tersebut akan terus dibagi hingga mencapai tingkat keseragaman tertentu atau ukuran minimum yang ditentukan.

Dalam implementasi teknis, sebuah Quadtree direpresentasikan sebagai simpul (node) dengan maksimal empat anak (children). Simpul daun (leaf) merepresentasikan area gambar yang seragam, sementara simpul internal menunjukkan area yang masih membutuhkan pembagian lebih lanjut. Setiap simpul menyimpan informasi seperti posisi (x, y), ukuran (width, height), dan nilai rata-rata warna atau intensitas piksel dalam area tersebut. Struktur ini memungkinkan pengkodean data gambar yang lebih efisien dengan menghilangkan redundansi pada area yang seragam. QuadTree sering digunakan dalam algoritma kompresi lossy karena mampu mengurangi ukuran file secara signifikan tanpa mengorbankan detail penting pada gambar.

Perancangan Algoritma *Divide and Conquer*

Algoritma *divide and conquer* yang diterapkan dalam struktur Quadtree bertujuan untuk menyederhanakan representasi gambar dengan membaginya menjadi blok-blok piksel yang lebih seragam. Dengan menentukan area gambar yang homogen menjadi satu blok warna rata-rata, ukuran file gambar dapat dikompresi secara signifikan tanpa mengorbankan terlalu banyak detail visual. Berikut adalah langkah-langkah utama dari algoritma ini:

1. Divide (Pemisahan Wilayah):

Pada tahap awal, gambar dibagi menjadi **empat kuadran**: kiri atas (northwest), kanan atas (northeast), kiri bawah (southwest), dan kanan bawah (southeast). Pembagian ini dilakukan secara **rekursif** dengan menentukan titik tengah dari wilayah yang sedang diproses, kemudian membentuk 4 subregion berdasarkan titik tersebut.

2. Kondisi Berhenti (Base Case):

Proses pembagian yang dilakukan secara rekursi akan berhenti jika ukuran blok sudah terlalu kecil (di bawah `minBlockSize`) atau blok hanya terdiri dari 1 piksel (baik lebar maupun tinggi). Dalam kasus tersebut, algoritma akan berhenti melanjutkan pembagian, dan sebagai gantinya, rata-rata warna blok akan dihitung dan disimpan sebagai representasi warna dari blok tersebut.

3. Conquer (Pembangunan Subtree):

Setiap subregion yang telah terbentuk akan diproses kembali secara rekursif menggunakan metode yang sama. Hasil dari pemrosesan ini akan membentuk subtree dari node induknya, dan secara bertahap membentuk struktur pohon Quadtree penuh.

4. Combine (Penggabungan Kembali Bila Memungkinkan):

Setelah keempat subregion selesai dibuat, algoritma melakukan evaluasi terhadap rata-rata warna dari setiap blok dan menghitung nilai penyimpangan atau error dari piksel di blok terhadap rata-ratanya. Perhitungan ini menggunakan metode yang ditentukan, seperti:

- *Variance*
- *MAD*
- *Max Pixel Difference*
- *Entropy*
- *Structural Similarity Index Measure (SSIM)*

5. Thresholding (Keputusan Penggabungan):

Jika nilai *error* yang dihitung lebih kecil dari *threshold* yang telah ditetapkan, artinya blok dianggap cukup seragam, maka empat *subtree* yang sebelumnya terbentuk akan dihapus atau diabaikan dan blok digantikan oleh satu *node* tunggal yang menyimpan rata-rata warna. Langkah ini penting untuk mengurangi kompleksitas pohon dan menghasilkan kompresi yang lebih efektif, terutama pada area gambar yang memiliki warna seragam.

Langkah-langkah tersebut dapat direalisasikan melalui potongan *pseudocode*:

```
ALGORITHM QuadtreeCompression(image, minBlockSize, errorThreshold, errorMethod)
    // Inisialisasi quadtree dengan dimensi gambar masukan
    quadtree = new Quadtree(image.width, image.height, image.rgbMatrix, errorMethod,
errorThreshold, minBlockSize)

    // Kompresi gambar
    quadtree.compressImage()

    // Render gambar yang telah dikompresi
    outputImage = quadtree.renderCompressedImage()

    RETURN outputImage
END ALGORITHM

FUNCTION compressImage()
    IF root is null THEN
        root = new TreeNode(0, 0, width, height, rgbMatrix)
    END IF
END FUNCTION

CLASS TreeNode
    CONSTRUCTOR(x, y, width, height, rgbMatrix)
        block = new Block(x, y, width, height, minBlockSize, threshold, method,
rgbMatrix)
        isLeaf = true
        child[0..3] = null
        subdivide(rgbMatrix)
    END CONSTRUCTOR

    FUNCTION subdivide(rgbMatrix)
        IF NOT block.calcIsValid() THEN
            isLeaf = false

            // Kalkulasi dimensi untuk blok children
            childWidth = block.width / 2
            childHeight = block.height / 2

            // Handle odd dimensions
```

```

        remainderWidth = block.width % 2
        remainderHeight = block.height % 2

        // Membuat empat children nodes
        child[0] = new TreeNode(block.x, block.y,
                                childWidth, childHeight, rgbMatrix)
        child[1] = new TreeNode(block.x + childWidth, block.y,
                                childWidth + remainderWidth, childHeight,
rgbMatrix)
        child[2] = new TreeNode(block.x, block.y + childHeight,
                                childWidth, childHeight + remainderHeight,
rgbMatrix)
        child[3] = new TreeNode(block.x + childWidth, block.y + childHeight,
                                childWidth + remainderWidth, childHeight +
remainderHeight, rgbMatrix)
    ELSE
        isLeaf = true
    END IF
END FUNCTION
END CLASS

FUNCTION calcIsValid()
    // Cek apakah blok terlalu kecil untuk dibagi lagi
    IF (area / 4) < minBlockSize THEN
        RETURN true
    END IF

    // Kalkulasi error method berdasarkan pilihan
    SWITCH methodNum
        CASE 1: error = variance
        CASE 2: error = meanAbsoluteDeviation
        CASE 3: error = maxPixelDiff
        CASE 4: error = entropy
        CASE 5: error = structSimIdx
    END SWITCH

    // Cek apakah error di bawah batasan
    RETURN error < threshold
END FUNCTION

```

Source Code Program

Main.cpp

```
#include <opencv2/opencv.hpp>
#include <Magick++.h>
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>

#include "ImageProcessor.hpp"
#include "Quadtree.hpp"
#include "Utils.hpp"
#include "GIF.hpp"

using namespace cv;
using namespace std;

int main()
{
    string inputImagePath;
    string outputImagePath;
    string outputGIFPath;
    int varianceMethod;
    double varianceThreshold;
    int minBlockSize;
    vector<vector<vector<int>>> rgbMatrix;

    int gifFrameDelay = 1000;

    printTitle();

    inputImagePath = ImageProcessor::inputImagePath();
    if (inputImagePath == "exit")
    {
        return -1;
    }
    rgbMatrix = ImageProcessor::loadImage(inputImagePath);

    int width = rgbMatrix[0].size();
    int height = rgbMatrix.size();

    tie(varianceMethod, varianceThreshold, minBlockSize) =
validateInputConstraints(inputImagePath, rgbMatrix, width, height);
;
    outputImagePath = askValidPath("image output", {".jpg", ".jpeg", ".png"});

    outputGIFPath = askValidPath("GIF output", {".gif"});

    long long originalFileSize = getFileSize(inputImagePath);
```

```

    auto start = chrono::high_resolution_clock::now();

    Quadtree quadtree(width, height, &rgbMatrix, varianceMethod, varianceThreshold,
minBlockSize);
    cout << "Compressing . . . " << endl << endl;
    quadtree.compressImage();

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

    // Compression process
    if (quadtree.saveCompressedImage(outputImagePath))
    {
        long long compressedFileSize = getFileSize(outputImagePath);

        double compressionPercentage = 0.0;
        if (originalFileSize > 0 && compressedFileSize > 0)
        {
            compressionPercentage = (1.0 - (static_cast<double>(compressedFileSize)
/ static_cast<double>(originalFileSize))) * 100.0;
        }

        cout << "Waktu eksekusi: " << duration.count() << " ms" << endl;
        cout << "Ukuran original: " << originalFileSize << " bytes" << endl;
        cout << "Ukuran compressed: " << compressedFileSize << " bytes" << endl;
        cout << "Persentase kompresi: " << compressionPercentage << "%" << endl;
        cout << "Kedalaman pohon: " << quadtree.getTreeDepth() << endl;
        cout << "Banyak simpul: " << quadtree.getTotalNodes() << endl << endl;

        cout << "Gambar berhasil disimpan di: " << outputImagePath << endl << endl;

        cout << "Membuat output GIF . . ." << endl;
        GIF gifGenerator(width, height, gifFrameDelay);
        gifGenerator.generateFramesFromQuadtree(quadtree);

        if (gifGenerator.saveGif(outputGIFPath))
        {
            cout << "GIF berhasil disimpan di: " << outputGIFPath << endl;
        }
        else
        {
            cout << "Gagal menyimpan GIF!" << endl;
        }
    }
    else
    {
        cout << "Gagal menyimpan gambar!" << endl;
    }

    return 0;
}

```

Block.cpp

```
#include "Block.hpp"
#include <bits/stdc++.h>
#include <vector>

Block::Block(int x, int y, int width, int height, int minBlockSize, double
threshold, int methodNum, const vector<vector<vector<int>>> *rgbMatrix) : x(x),
y(y), width(width), height(height), area(width * height),
minBlockSize(minBlockSize), methodNum(methodNum), threshold(threshold),
rgbMatrix(rgbMatrix)
{
    averageRGB = getAverageRGB();
}

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

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

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

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

int Block::getArea() const
{
    return area;
}

vector<double> Block::getAverageRGB() const
{
    double sumR = 0, sumG = 0, sumB = 0;
    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j++)
        {
            sumR += (*rgbMatrix)[i][j][0];
            sumG += (*rgbMatrix)[i][j][1];
            sumB += (*rgbMatrix)[i][j][2];
        }
    }
}
```



```

    }
}

vector<double> averageRGB(3);
averageRGB[0] = sumR / area;
averageRGB[1] = sumG / area;
averageRGB[2] = sumB / area;

return averageRGB;
}

double Block::getMaxPixelDiff() const
{
    double maxR = (*rgbMatrix)[y][x][0];
    double maxG = (*rgbMatrix)[y][x][1];
    double maxB = (*rgbMatrix)[y][x][2];
    double minR = (*rgbMatrix)[y][x][0];
    double minG = (*rgbMatrix)[y][x][1];
    double minB = (*rgbMatrix)[y][x][2];

    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j++)
        {
            if ((double)(*rgbMatrix)[i][j][0] > maxR)
            {
                maxR = (*rgbMatrix)[i][j][0];
            }
            if ((double)(*rgbMatrix)[i][j][1] > maxG)
            {
                maxG = (*rgbMatrix)[i][j][1];
            }
            if ((double)(*rgbMatrix)[i][j][2] > maxB)
            {
                maxB = (*rgbMatrix)[i][j][2];
            }
            if ((double)(*rgbMatrix)[i][j][0] < minR)
            {
                minR = (*rgbMatrix)[i][j][0];
            }
            if ((double)(*rgbMatrix)[i][j][1] < minG)
            {
                minG = (*rgbMatrix)[i][j][1];
            }
            if ((double)(*rgbMatrix)[i][j][2] < minB)
            {
                minB = (*rgbMatrix)[i][j][2];
            }
        }
    }

    return ((maxR - minR) + (maxG - minG) + (maxB - minB)) / 3.0;
}

```

```

double Block::getVariance() const
{
    double varianceR = 0, varianceG = 0, varianceB = 0;
    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j++)
        {
            varianceR += pow((double)(*rgbMatrix)[i][j][0] - averageRGB[0], 2);
            varianceG += pow((double)(*rgbMatrix)[i][j][1] - averageRGB[1], 2);
            varianceB += pow((double)(*rgbMatrix)[i][j][2] - averageRGB[2], 2);
        }
    }
    varianceR = varianceR / area;
    varianceG = varianceG / area;
    varianceB = varianceB / area;

    return (varianceR + varianceG + varianceB) / 3.0;
}

double Block::getMeanAbsoluteDeviation() const
{
    double madR = 0, madG = 0, madB = 0;
    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j++)
        {
            madR += abs((double)(*rgbMatrix)[i][j][0] - averageRGB[0]);
            madG += abs((double)(*rgbMatrix)[i][j][1] - averageRGB[1]);
            madB += abs((double)(*rgbMatrix)[i][j][2] - averageRGB[2]);
        }
    }
    madR = madR / area;
    madG = madG / area;
    madB = madB / area;

    return (madR + madG + madB) / 3.0;
}

double Block::getEntropy() const
{
    vector<int> histR(256, 0);
    vector<int> histG(256, 0);
    vector<int> histB(256, 0);

    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j++)
        {
            int r = (*rgbMatrix)[i][j][0];
            int g = (*rgbMatrix)[i][j][1];
            int b = (*rgbMatrix)[i][j][2];

            histR[r]++;

```

```

        histG[g]++;
        histB[b]++;
    }
}

double entropyR = 0, entropyG = 0, entropyB = 0;

for (int i = 0; i < 256; i++)
{
    double probR = (double)histR[i] / area;
    double probG = (double)histG[i] / area;
    double probB = (double)histB[i] / area;

    if (probR > 0)
        entropyR += (probR * log2(probR));
    if (probG > 0)
        entropyG += (probG * log2(probG));
    if (probB > 0)
        entropyB += (probB * log2(probB));
}

entropyR = -entropyR;
entropyG = -entropyG;
entropyB = -entropyB;

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

double Block::getStructSimIdx() const
{
    const double K1 = 0.01, K2 = 0.03;
    const double L = 255.0;
    const double C1 = (K1 * L) * (K1 * L);
    const double C2 = (K2 * L) * (K2 * L);

    double ssimSum = 0.0;

    for (int channel = 0; channel < 3; channel++)
    {
        double muX = 0.0;
        double sigmaX = 0.0;

        for (int i = y; i < y + height; ++i)
        {
            for (int j = x; j < x + width; ++j)
            {
                muX += (*rgbMatrix)[i][j][channel];
            }
        }
        muX /= area;

        double muY = averageRGB[channel];
    }
}

```

```

        for (int i = y; i < y + height; ++i)
        {
            for (int j = x; j < x + width; ++j)
            {
                double diff = (*rgbMatrix)[i][j][channel] - muX;
                sigmaX += diff * diff;
            }
        }
        sigmaX /= area;

        double sigmaXY = 0.0;
        for (int i = y; i < y + height; ++i)
        {
            for (int j = x; j < x + width; ++j)
            {
                sigmaXY += ((*rgbMatrix)[i][j][channel] - muX) * (muY - muY);
            }
        }
        sigmaXY /= area;

        double numerator = (2 * muX * muY + C1);
        double denominator = (muX * muX + muY * muY + C1) * (sigmaX + C2);
        double ssim = (denominator > 0) ? numerator / denominator : 1.0;

        if (channel == 0)
        {
            ssimSum += 0.3 * ssim;
        }
        else if (channel == 1)
        {
            ssimSum += 0.59 * ssim;
        }
        else
        {
            ssimSum += 0.11 * ssim;
        }
    }

    return 1.0 - ssimSum;
}

bool Block::calcIsValid() const
{
    return ((area / 4) < minBlockSize) ||
        (methodNum == 1 && getVariance() < threshold) ||
        (methodNum == 2 && getMeanAbsoluteDeviation() < threshold) ||
        (methodNum == 3 && getMaxPixelDiff() < threshold) ||
        (methodNum == 4 && getEntropy() < threshold) ||
        (methodNum == 5 && getStructSimIdx() < threshold);
}

```

Quadtree.cpp

```
#include "Quadtree.hpp"
#include <opencv2/opencv.hpp>
#include "Utils.hpp"

using namespace std;
using namespace cv;

int TreeNode::varianceMethod = 0;
double TreeNode::varianceThreshold = 0.0;
int TreeNode::minBlockSize = 0;

TreeNode::TreeNode(int x, int y, int width, int height, const
vector<vector<vector<int>>> *rgbMatrix)
    : block(x, y, width, height, TreeNode::minBlockSize,
TreeNode::varianceThreshold, TreeNode::varianceMethod, rgbMatrix), isLeaf(true)
{
    for (int i = 0; i < 4; i++)
    {
        child[i] = nullptr;
    }
    subdivide(rgbMatrix);
}

TreeNode::~TreeNode()
{
    for (int i = 0; i < 4; i++)
    {
        if (child[i] != nullptr)
        {
            delete child[i];
            child[i] = nullptr;
        }
    }
}

void TreeNode::subdivide(const vector<vector<vector<int>>> *rgbMatrix)
{
    if (!block.calcIsValid())
    {
        isLeaf = false;

        int childWidth = block.getWidth() / 2;
        int childHeight = block.getHeight() / 2;

        int remainderWidth = block.getWidth() % 2;
        int remainderHeight = block.getHeight() % 2;

        child[0] = new TreeNode(block.getX(), block.getY(), childWidth, childHeight,
rgbMatrix);
```

```

        child[1] = new TreeNode(block.getX() + childWidth, block.getY(), childWidth
+ remainderWidth, childHeight, rgbMatrix);
        child[2] = new TreeNode(block.getX(), block.getY() + childHeight,
childWidth, childHeight + remainderHeight, rgbMatrix);
        child[3] = new TreeNode(block.getX() + childWidth, block.getY() +
childHeight, childWidth + remainderWidth, childHeight + remainderHeight, rgbMatrix);
    }
    else
    {
        isLeaf = true;
    }
}

void TreeNode::draw(Mat &output)
{
    if (isLeaf)
    {
        vector<double> avgColor = block.getAverageRGB();
        Scalar color(avgColor[2], avgColor[1], avgColor[0]);

        Point topLeft(block.getX(), block.getY());
        Point bottomRight(block.getX() + block.getWidth(), block.getY() +
block.getHeight());

        rectangle(output, topLeft, bottomRight - Point(1, 1), color, -1);
    }
    else
    {
        for (int i = 0; i < 4; i++)
        {
            if (child[i] != nullptr)
            {
                child[i]->draw(output);
            }
        }
    }
}

int TreeNode::getMaxDepth() const
{
    return getMaxDepthRecursive();
}

int TreeNode::getMaxDepthRecursive() const
{
    if (isLeaf)
    {
        return 1;
    }

    int maxChildDepth = 0;
    for (int i = 0; i < 4; i++)
    {

```

```

        if (child[i] != nullptr)
        {
            int childDepth = child[i]->getMaxDepthRecursive();
            if (childDepth > maxChildDepth)
            {
                maxChildDepth = childDepth;
            }
        }
    }
    return maxChildDepth + 1;
}

int TreeNode::getTotalNodes() const
{
    return getTotalNodesRecursive();
}

int TreeNode::getTotalNodesRecursive() const
{
    int count = 1;

    if (!isLeaf)
    {
        for (int i = 0; i < 4; i++)
        {
            if (child[i] != nullptr)
            {
                count += child[i]->getTotalNodesRecursive();
            }
        }
    }
    return count;
}

Quadtree::Quadtree(int width, int height, const vector<vector<vector<int>>>
*rgbMatrix, int varianceMethod, double varianceThreshold, int minBlockSize) :
width(width), height(height), rgbMatrix(rgbMatrix)
{
    TreeNode::varianceMethod = varianceMethod;
    TreeNode::varianceThreshold = varianceThreshold;
    TreeNode::minBlockSize = minBlockSize;

    root = nullptr;
}

Quadtree::~~Quadtree()
{
    if (root != nullptr)
    {
        delete root;
        root = nullptr;
    }
}

```

```

void Quadtree::compressImage()
{
    if (root == nullptr)
    {
        root = new TreeNode(0, 0, width, height, rgbMatrix);
    }
}

bool Quadtree::saveCompressedImage(const string &outputPath)
{
    if (root == nullptr)
    {
        return false;
    }

    int width = root->block.getWidth();
    int height = root->block.getHeight();

    Mat output(height, width, CV_8UC3, Scalar(255, 255, 255));

    root->draw(output);

    return imwrite(convertWindowsToWslPath(outputPath), output);
}

int Quadtree::getTreeDepth()
{
    if (root == nullptr)
    {
        return 0;
    }
    return root->getMaxDepth();
}

int Quadtree::getTotalNodes()
{
    if (root == nullptr)
    {
        return 0;
    }
    return root->getTotalNodes();
}

void TreeNode::drawAtDepth(Mat &output, int targetDepth, int currentDepth)
{
    if (currentDepth == targetDepth || (isLeaf && currentDepth < targetDepth))
    {
        vector<double> avgColor = block.getAverageRGB();
        Scalar color(avgColor[2], avgColor[1], avgColor[0]);

        // Draw rectangle with average color
        Point topLeft(block.getX(), block.getY());
    }
}

```



```

        Point bottomRight(block.getX() + block.getWidth(), block.getY() +
block.getHeight());

        // Fill the rectangle with color
        rectangle(output, topLeft, bottomRight - Point(1, 1), color, -1);
    }
    else if (currentDepth < targetDepth)
    {
        for (int i = 0; i < 4; i++)
        {
            if (child[i] != nullptr)
            {
                child[i]->drawAtDepth(output, targetDepth, currentDepth + 1);
            }
        }
    }
}

void Quadtree::renderAtDepth(Mat &output, int depth)
{
    if (root == nullptr) {
        return;
    }

    // Clear the output image to white
    output = Mat(height, width, CV_8UC3, Scalar(255, 255, 255));

    // Draw the quadtree at the specified depth
    root->drawAtDepth(output, depth);
}

```

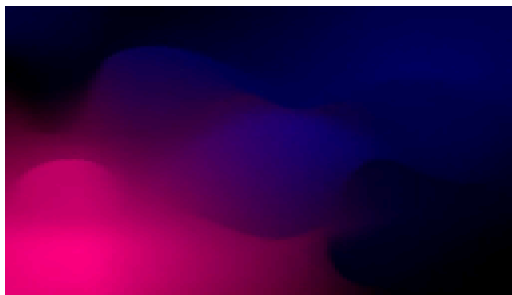
Test Case: Input & Output

Note: seluruh hasil GIF disimpan pada Google Drive di Lampiran.

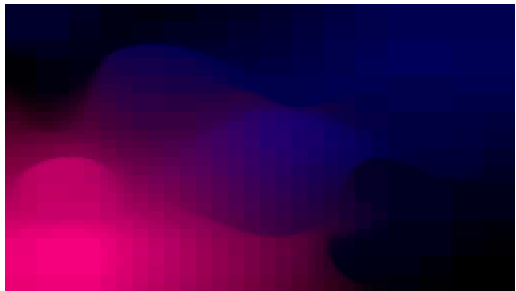
1. Test Case 1



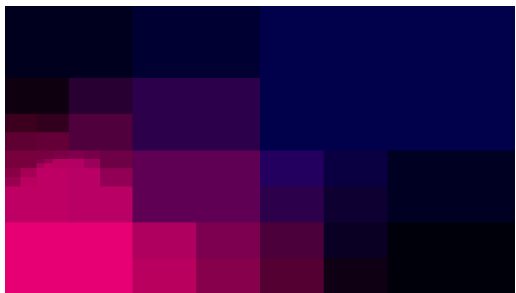
Auto (Compression Percentage)

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Compression Percentage = 0.1</i>		Waktu eksekusi: 144 ms Ukuran original: 96139 bytes Ukuran compressed: 87399 bytes Persentase kompresi: 9.091% Kedalaman pohon: 8 Banyak simpul: 21845

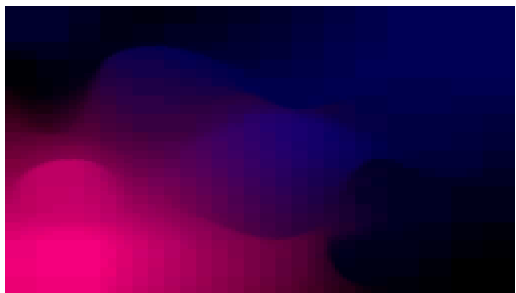
Variance

Parameter	Output	Description
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 125 ms Ukuran original: 96139 bytes Ukuran compressed: 34736 bytes Persentase kompresi: 63.869% Kedalaman pohon: 8 Banyak simpul: 1853

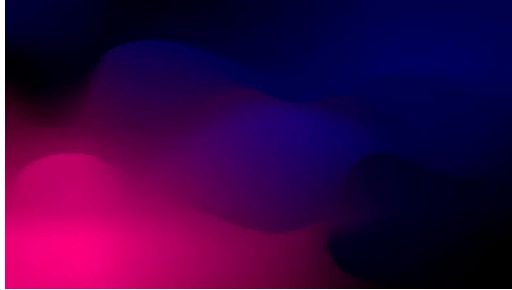
MAD

Parameter	Output	Description
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 29 ms Ukuran original: 96139 bytes Ukuran compressed: 9097 bytes Persentase kompresi: 90.5377% Kedalaman pohon: 7 Banyak simpul: 65

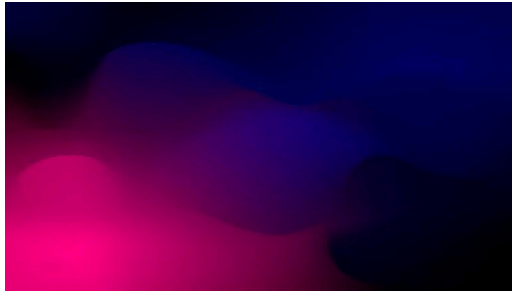
Max Pixel Difference

Parameter	Output	Description
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 74 ms Ukuran original: 96139 bytes Ukuran compressed: 36413 bytes Persentase kompresi: 62.1246% Kedalaman pohon: 8 Banyak simpul: 2049

Entropy

Parameter	Output	Description
<i>Threshold = 0.1</i> <i>Min. Block Size = 4</i>		Waktu eksekusi: 116 ms Ukuran original: 96139 bytes Ukuran compressed: 122114 bytes Persentase kompresi: -27.0182% Kedalaman pohon: 9 Banyak simpul: 62101


SSIM

Parameter	Output	Description
<i>Threshold = 0.1</i> <i>Min. Block Size = 4</i>		Waktu eksekusi: 169 ms Ukuran original: 18949 bytes Ukuran compressed: 129523 bytes Persentase kompresi: -583.535% Kedalaman pohon: 9 Banyak simpul: 71061

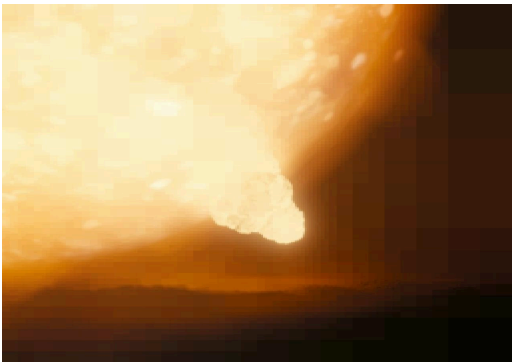
2. Test Case 2



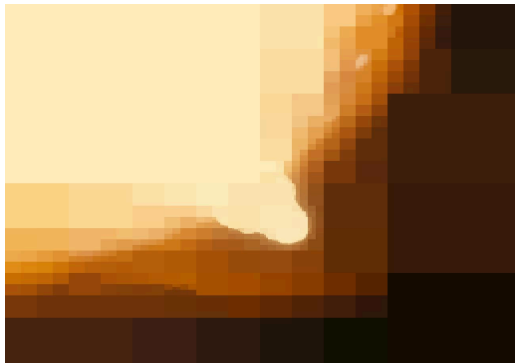
Auto (Compression Percentage)

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Compression Percentage = 0.1</i>		Waktu eksekusi: 667 ms Ukuran original: 230035 bytes Ukuran compressed: 125458 bytes Persentase kompresi: 45.4613% Kedalaman pohon: 9 Banyak simpul: 87381


Variance

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold = 10 Min. Block Size = 10</i>		Waktu eksekusi: 642 ms Ukuran original: 230035 bytes Ukuran compressed: 96182 bytes Persentase kompresi: 58.1881% Kedalaman pohon: 9 Banyak simpul: 11781


MAD

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 188 ms Ukuran original: 230035 bytes Ukuran compressed: 36165 bytes Persentase kompresi: 84.2785% Kedalaman pohon: 9 Banyak simpul: 733


Max Pixel Difference

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 365 ms Ukuran original: 230035 bytes Ukuran compressed: 104753 bytes Persentase kompresi: 54.4621% Kedalaman pohon: 9 Banyak simpul: 16085

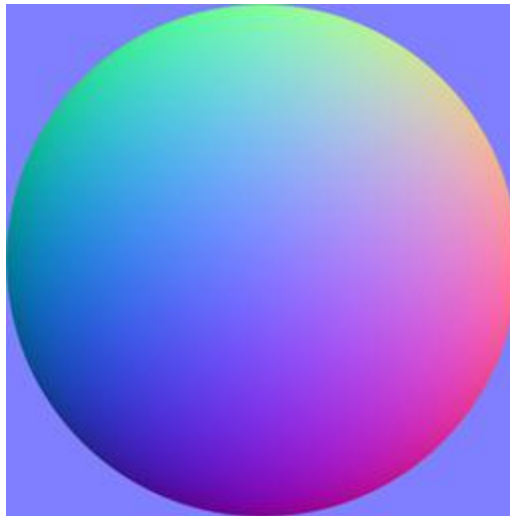
Entropy

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold</i> = 0.1 <i>Min. Block Size</i> = 4		Waktu eksekusi: 564 ms Ukuran original: 230035 bytes Ukuran compressed: 121145 bytes Persentase kompresi: 47.3363% Kedalaman pohon: 10 Banyak simpul: 200801


SSIM

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold</i> = 0.1 <i>Min. Block Size</i> = 4		Waktu eksekusi: 818 ms Ukuran original: 230035 bytes Ukuran compressed: 121155 bytes Persentase kompresi: 47.3319% Kedalaman pohon: 10 Banyak simpul: 219477

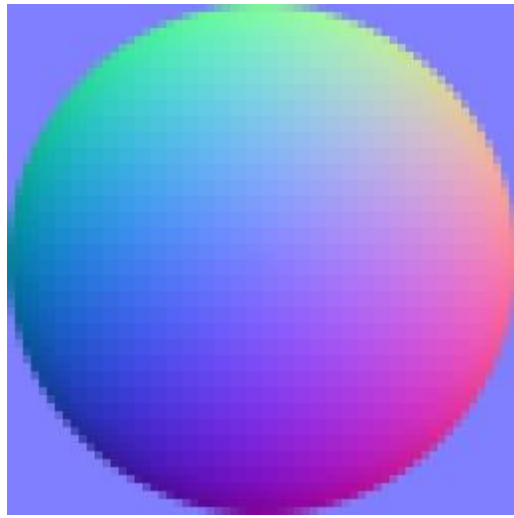
3. Test Case 3



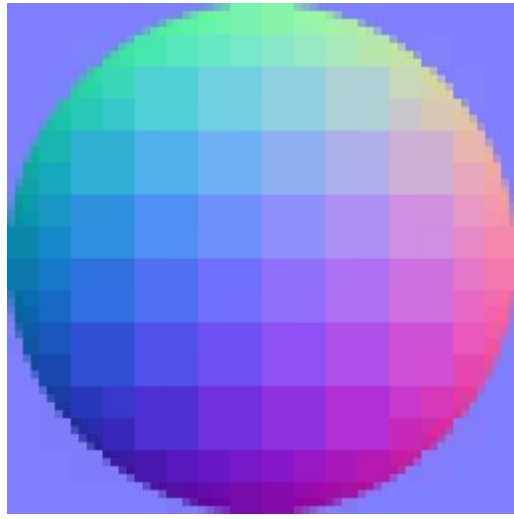
Auto (Compression Percentage)

Parameter	Output	Description
Compression Percentage = 0.1		Waktu eksekusi: 29 ms Ukuran original: 22683 bytes Ukuran compressed: 11559 bytes Persentase kompresi: 49.0411% Kedalaman pohon: 8 Banyak simpul: 21845

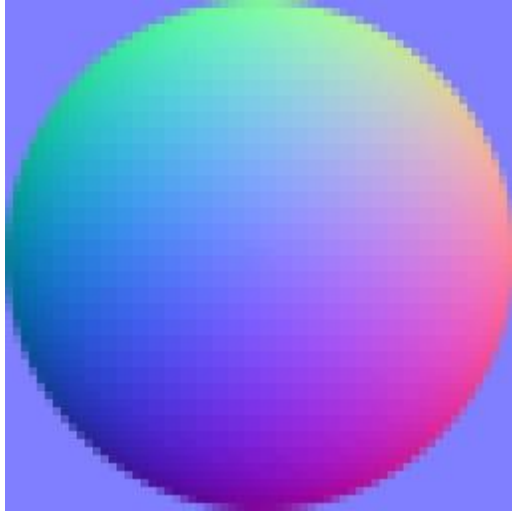
Variance

Parameter	Output	Description
Threshold = 10 Min. Block Size = 10		Waktu eksekusi: 32 ms Ukuran original: 22683 bytes Ukuran compressed: 8387 bytes Persentase kompresi: 63.0252% Kedalaman pohon: 7 Banyak simpul: 2161


MAD

Parameter	Output	Description
Threshold = 10 Min. Block Size = 10		Waktu eksekusi: 9 ms Ukuran original: 22683 bytes Ukuran compressed: 5022 bytes Persentase kompresi: 77.8601% Kedalaman pohon: 7 Banyak simpul: 617


Max Pixel Difference

Parameter	Output	Description
<i>Threshold</i> = 10 <i>Min. Block Size</i> = 10		Waktu eksekusi: 15 ms Ukuran original: 22683 bytes Ukuran compressed: 8941 bytes Persentase kompresi: 60.5828% Kedalaman pohon: 7 Banyak simpul: 2577

Entropy

Parameter	Output	Description
<i>Threshold</i> = 0.1 <i>Min. Block Size</i> = 4		Waktu eksekusi: 29 ms Ukuran original: 22683 bytes Ukuran compressed: 11559 bytes Persentase kompresi: 49.0411% Kedalaman pohon: 8 Banyak simpul: 18341

SSIM

<i>Parameter</i>	<i>Output</i>	<i>Description</i>
<i>Threshold = 0.1</i> <i>Min. Block Size = 4</i>		Waktu eksekusi: 32 ms Ukuran original: 22683 bytes Ukuran compressed: 11559 bytes Persentase kompresi: 49.0411% Kedalaman pohon: 8 Banyak simpul: 21845

Analisis Algoritma *Divide & Conquer* Quadtree

Bagian utama algoritma Divide & Conquer dapat dilihat pada berikut:

```
Quadtree::Quadtree(int width, int height, const vector<vector<vector<int>>>
*rgbMatrix, int varianceMethod, double varianceThreshold, int minBlockSize) :
width(width), height(height), rgbMatrix(rgbMatrix)
{
    TreeNode::varianceMethod = varianceMethod;
    TreeNode::varianceThreshold = varianceThreshold;
    TreeNode::minBlockSize = minBlockSize;

    root = nullptr;
}

void Quadtree::compressImage()
{
    if (root == nullptr)
    {
        root = new TreeNode(0, 0, width, height, rgbMatrix);
    }
}
```

Class Quadtree berperan sebagai *root node* atau *node awal*. Kompleksitas inisialisasi adalah $O(1)$, *looping* akan terjadi dalam TreeNode

```
TreeNode::TreeNode(int x, int y, int width, int height, const
vector<vector<vector<int>>> *rgbMatrix)
: block(x, y, width, height, TreeNode::minBlockSize,
TreeNode::varianceThreshold, TreeNode::varianceMethod, rgbMatrix), isLeaf(true)
{
    for (int i = 0; i < 4; i++)
    {
        child[i] = nullptr;
    }
    subdivide(rgbMatrix);
}

void TreeNode::subdivide(const vector<vector<vector<int>>> *rgbMatrix)
{
    if (!block.calcIsValid())
    {
        isLeaf = false;

        int childWidth = block.getWidth() / 2;
        int childHeight = block.getHeight() / 2;

        int remainderWidth = block.getWidth() % 2;
        int remainderHeight = block.getHeight() % 2;
    }
}
```

```

        child[0] = new TreeNode(block.getX(), block.getY(), childWidth, childHeight,
rgbMatrix);
        child[1] = new TreeNode(block.getX() + childWidth, block.getY(), childWidth
+ remainderWidth, childHeight, rgbMatrix);
        child[2] = new TreeNode(block.getX(), block.getY() + childHeight,
childWidth, childHeight + remainderHeight, rgbMatrix);
        child[3] = new TreeNode(block.getX() + childWidth, block.getY() +
childHeight, childWidth + remainderWidth, childHeight + remainderHeight, rgbMatrix);
    }
    else
    {
        isLeaf = true;
    }
}

```

TreeNode adalah *node-node* dalam Quadtree. Saat sebuah TreeNode diinstansiasi akan membuat sebuah *list of* TreeNode yang diinisialisasi dengan nullptr, list of TreeNode ini berfungsi sebagai tempat *child root* jika subdivisi berhasil. TreeNode akan melakukan subdivisi, jika memenuhi kondisi maka akan menjadi *leaf node* jika tidak maka akan membelah menjadi 4 *child node*. Kompleksitas algoritma ini adalah $O(n\log(n))$.

Implementasi Bonus

1. Bonus 1: *Compression Percentage*

```
tuple<int, double, int, string> findOptimalSettingsBinarySearch(
    double targetCompression,
    const string &inputImagePath,
    const vector<vector<vector<int>>> &rgbMatrix,
    int width,
    int height)
{
    const long long originalFileSize = getFileSize(inputImagePath);
    if (originalFileSize <= 0)
    {
        cout << "Size file original tidak dapat ditentukan." << endl
              << endl;
        return make_tuple(1, 50.0, 4, inputImagePath);
    }

    int bestMethod = 1;
    double bestThreshold = 0.0;
    int bestMinBlockSize = 0;
    double bestCompressionDiff = 1.0;

    string tempOutputPath = inputImagePath.substr(0,
inputImagePath.find_last_of('.') + "_temp_compressed" +
inputImagePath.substr(inputImagePath.find_last_of('.')));

    cout << "Memulai proses . . ." << endl;
    cout << "Mungkin memakan banyak waktu akibat iterasi seluruh metode . . ." <<
endl
        << endl;

    for (int method = 1; method <= 5; method++)
    {
        double minThreshold, maxThreshold;
        if (method == 5)
        {
            minThreshold = 0.1;
            maxThreshold = 0.95;
        }
        else if (method == 4)
        {
            minThreshold = 0.1;
            maxThreshold = 8.0;
        }
        else if (method == 1)
        {
            minThreshold = 1.0;
            maxThreshold = 255.0 * 255.0;
        }
        else
```

```

{
    minThreshold = 1.0;
    maxThreshold = 255.0;
}

int minBlockSizes[] = {1, 4, 9, 16};

string methodNames[] = {
    "Variance",
    "Mean Absolute Deviation (MAD)",
    "Max Pixel Difference",
    "Entropy",
    "Structural Similarity Index (SSIM)"};
cout << "Metode (" << method << ") " << methodNames[method - 1] << " . . ."
<< endl;

for (int blockSize : minBlockSizes)
{
    double lowThreshold = minThreshold;
    double highThreshold = maxThreshold;

    for (int iteration = 0; iteration < 10; iteration++)
    {
        double midThreshold = lowThreshold + (highThreshold - lowThreshold)
/ 2.0;

        try
        {
            Quadtree quadtree(width, height, &rgbMatrix, method,
midThreshold, blockSize);
            quadtree.compressImage();

            if (!quadtree.saveCompressedImage(tempOutputPath))
            {
                cout << "Gagal menyimpan temp." << endl
<< endl;
                continue;
            }

            long long compressedSize = getFileSize(tempOutputPath);
            if (compressedSize <= 0)
            {
                cout << "Gagal membaca." << endl
<< endl;
                continue;
            }

            double achievedCompression = 1.0 -
(static_cast<double>(compressedSize) / originalFileSize);

            double compressionDiff = abs(achievedCompression -
targetCompression);
            if (compressionDiff < bestCompressionDiff)

```

```

        {
            bestCompressionDiff = compressionDiff;
            bestMethod = method;
            bestThreshold = midThreshold;
            bestMinBlockSize = blockSize;
        }

    if (achievedCompression < targetCompression)
    {
        if (method == 5)
        {
            lowThreshold = midThreshold;
        }
        else
        {
            highThreshold = midThreshold;
        }
    }
    else
    {
        if (method == 5)
        {
            highThreshold = midThreshold;
        }
        else
        {
            lowThreshold = midThreshold;
        }
    }
}
catch (const exception &e)
{
    cout << "Error: " << e.what() << endl
         << endl;
}
}
}
cout << endl;

return make_tuple(bestMethod, bestThreshold, bestMinBlockSize, tempOutputPath);
}

```

Source code di atas merupakan fungsi untuk mencari parameter *terbaik* untuk mencapai persentase kompresi terdekat dengan input. *Binary search* digunakan untuk mencari *threshold* terbaik, dan *looping* dilakukan untuk melakukan *testing* terhadap setiap *method* serta mencari *minimum block size* terbaik.

2. Bonus 2: Structural Similarity Index (SSIM)

Structural Similarity Index Measure (SSIM) merupakan metrik berbasis persepsi manusia yang digunakan untuk mengukur kemiripan antara dua citra (dalam kasus ini adalah 2 gambar). Dalam konteks kompresi gambar menggunakan Quadtree, SSIM digunakan untuk mengevaluasi tingkat keseragaman dalam sebuah blok gambar atau dalam kata lain menjadi salah satu *error method*. Jika blok dianggap cukup mirip secara struktur, maka tidak perlu dipecah lebih lanjut. Pada program ini, SSIM diimplementasikan pada *source code* Block.hpp dan Block.cpp dalam *function* getStructSimIdx() yang menghitung SSIM pada blok dan mengembalikannya untuk evaluasi lebih lanjut. Implementasi getStructSimIdx() dilakukan dalam tahapan seperti berikut:

a. Inisialisasi Konstanta

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

Melihat *source code* di atas, terlihat terdapat konstanta K1 dan K2 yang merupakan konstanta kecil untuk menstabilkan pembagian saat nilai *mean* atau *variance* mendekati nol. Terdapat juga konstanta L yang merupakan batasan maksimum nilai piksel dalam gambar 8-bit, serta terdapat juga C1 dan C2 yang merupakan konstanta nilai stabilisasi sesuai standar *paper* SSIM.

b. Perhitungan *Channel* Warna (R, G, B)

```
for (int channel = 0; channel < 3; channel++)
{
    // Menghitung SSIM untuk masing-masing channel
    ...
}
```

Masing-masing *channel* warna (R, G, B) dihitung secara terpisah. Setelahnya, SSIM dari ketiga *channel* pun digabungkan menggunakan bobot atas dasar persepsi manusia:

Channel	Weight	Dasar Weight
Red (R)	0.3	Sensitivitas sedang terhadap warna merah
Green (G)	0.59	Mata paling sensitif terhadap warna hijau
Blue (B)	0.11	Mata paling minim sensitif terhadap warna biru

```

if (channel == 0)
{
    ssimSum += 0.3 * ssim;
}
else if (channel == 1)
{
    ssimSum += 0.59 * ssim;
}
else
{
    ssimSum += 0.11 * ssim;
}

```

c. Perhitungan Statistik *Channel*

Pada tahap ini, untuk setiap *channel* warna akan dihitung:

- Rata-rata nilai piksel μ_x dan μ_y
 - μ_x dihitung dari data piksel asli pada blok.
 - μ_y diambil dari rata-rata piksel blok (`averageRGB[channel]`).
- Varians σ_x^2 dan σ_y^2
 - σ_x^2 dihitung dari deviasi piksel terhadap μ_x .
 - σ_y^2 biasanya nol karena blok dikompresi menjadi warna rata-rata.
- Kovariansi σ_{xy}
 - Dalam implementasi ini dianggap nol karena y merupakan rata-rata.

```

double muX = 0.0;
for (int i = y; i < y + height; ++i)
{
    for (int j = x; j < x + width; ++j)
    {

```

```

        muX += (*rgbMatrix)[i][j][channel];
    }
}
muX /= area;

double muY = averageRGB[channel];

double sigmaX = 0.0;
for (int i = y; i < y + height; ++i)
{
    for (int j = x; j < x + width; ++j)
    {
        double diff = (*rgbMatrix)[i][j][channel] - muX;
        sigmaX += diff * diff;
    }
}
sigmaX /= area;

double sigmaXY = 0.0;
for (int i = y; i < y + height; ++i)
{
    for (int j = x; j < x + width; ++j)
    {
        sigmaXY += ((*rgbMatrix)[i][j][channel] - muX) *
        ((*rgbMatrix)[i][j][channel] - muY);
    }
}
sigmaXY /= area;

double sigmaY = 0.0;
for (int i = y; i < y + height; ++i)
{
    for (int j = x; j < x + width; ++j)
    {
        double diff = (*rgbMatrix)[i][j][channel] - muY;
        sigmaY += diff * diff;
    }
}
sigmaY /= area;

```

d. Aplikasi Rumus SSIM

```

double numerator = (2 * muX * muY + C1) * (2 * sigmaXY + C2);
double denominator = (muX * muX + muY * muY + C1) * (sigmaX + sigmaY + C2);
double ssim = (denominator > 0) ? numerator / denominator : 1.0;

```

Implementasi rumus SSIM seperti pada *source code* di atas diimplementasikan atas dasar rumus:

$$SSIM(x, y) = [(2\mu_x\mu_y + C1)(2\sigma_{xy} + C2)] / [(\mu_x^2 + \mu_y^2 + C1)(\sigma_x^2 + \sigma_y^2 + C2)]$$

Nilai SSIM berada di antara -1 hingga 1, tapi pada praktik normal berada di antara 0 hingga 1.

3. Bonus 3: GIF Generation

Selain *output* gambar, terdapat juga pembuatan dan *output* GIF yang menggambarkan setiap langkah dekomposisi pada struktur Quadtree yang dibentuk. GIF yang dihasilkan akan memperlihatkan secara bertahap bagaimana gambar dipecah dari blok besar menjadi lebih detail, sesuai *depth* dari Quadtree: *frame* awal (1 blok besar atau *root*) – *frame-frame intermediate* (pembagian menjadi lebih banyak blok) – *frame* akhir (representasi penuh dari hasil kompresi Quadtree). Implementasi dari fitur ini dibagi menjadi beberapa komponen atau tahapan utama seperti berikut:

a. Struktur Class GIF

Implementasi *Class* GIF dibantu menggunakan *library* Magick++ pada *source code* GIF.hpp dan GIF.cpp sebagai *tools* utama dalam pembentukan GIF. Kelas ini akan menyimpan seluruh frame animasi, mengubah objek gambar OpenCV (`cv::Mat`) menjadi gambar Magick++ (`ImageMagick`), membuat *frame* dari setiap level dekomposisi Quadtree, dan akhirnya menyusun dan menyimpan GIF akhir. Berikut merupakan file *header* dari kelas GIF:

```
#ifndef GIF_HPP
#define GIF_HPP

#include <opencv2/opencv.hpp>
#include <string>
#include <vector>
#include <Magick++.h>
#include "Quadtree.hpp"

using namespace std;
using namespace cv;

class GIF {
private:
    vector<Mat> frames;
    int width;
    int height;
    int frameDelay;

    Magick::Image convertMatToMagickImage(const Mat& mat);

public:
    GIF(int width, int height, int frameDelay = 500);
};
```

```

    void addFrame(const Mat &frame);
    void generateFramesFromQuadtree(Quadtree &quadtree);
    bool saveGif(const string &outputPath);
};

#endif

```

b. Pembuatan *Frame* dari Quadtree

```

void generateFramesFromQuadtree(Quadtree &quadtree);

```

Method di atas ini akan menghapus seluruh *frame* yang sudah tersimpan sebelumnya, mengambil *depth* maksimum dari pohon Quadtree, dan membuat satu *frame* untuk setiap *depth*. Selanjutnya, *method* akan memanggil *renderAtDepth()* yang berasal dari kelas Quadtree untuk menghasilkan gambar pohon pada *depth* spesifik. Terakhir, hasil gambar akan ditambahkan ke daftar *frame* GIF.

c. Render Gambar atau *Frame* Berdasarkan *Depth*

Untuk melakukan *rendering* gambar berdasarkan *depth* yang spesifik, diimplementasikan dua *helper methods* pada kelas Quadtree dan TreeNode:

```

void Quadtree::renderAtDepth(Mat &output, int depth)

```

Method *renderAtDepth()* akan menginisialisasi gambar *output* dengan mengosongkannya atau dalam kata lain memberikan warna putih polos. Selanjutnya, *method* *drawAtDepth()* akan dipanggil pada *root node* serta menyusun kembali gambar sesuai struktur pohon pada *depth* tertentu.

```

void TreeNode::drawAtDepth(Mat &output, int targetDepth, int currentDepth)

```

Pada *method* di atas, bila *target depth* sudah tercapai atau *node* kini merupakan *leaf node* yang lebih dangkal dari *target*, maka blok akan digambar dengan warna rata-rata. Jika belum sampai kedalaman *target*, maka *method* akan dipanggil kembali secara rekursif untuk seluruh *child node*-nya.

d. Konversi dan Penyimpanan ke GIF

Pada tahap terakhir pembentukan GIF, gambar OpenCV akan dikonversi menjadi gambar Magick++ serta disimpan dalam format GIF. Implementasi ini direalisasikan dalam dua *helper methods*:

```
Magick::Image convertMatToMagickImage(const Mat& mat);
```

Method di atas akan mengonversi gambar `cv::Mat` ke format Magick++ Image. *Method* ini menangani gambar RGB, Grayscale, dan BGRA, serta mengonversi ke format yang kompatibel dengan ImageMagick.

```
bool saveGif(const string &outputPath);
```

Method di atas akan mengonversi seluruh frame `cv::Mat` ke `Magick::Image` dengan bantuan *method* sebelumnya (`convertMatToMagickImage`) dengan aturan *delay* antar-*frame* (`frameDelay / 10`, karena Magick++ menggunakan satuan 1/100 detik). Animasi pun di-set agar melakukan *looping* selamanya dan hasil seluruh *frame* pun akan disimpan sebagai file `.gif`.

e. Integrasi dalam Main.cpp

```
GIF gifGenerator(width, height, gifFrameDelay);
gifGenerator.generateFramesFromQuadtree(quadtree);

if (gifGenerator.saveGif(outputGIFPath))
{
    cout << "GIF berhasil disimpan di: " << outputGIFPath << endl;
}
else
{
    cout << "Gagal menyimpan GIF!" << endl;
}
```

Setelah seluruh *method* atau *tools* pembentukan *frame* masing-masing *depth* pada Quadtree dan GIF, seluruh *methods* tersebut dapat dieksekusi pada file *main* dengan tahap: (1) inisialisasi generator GIF dengan ukuran gambar dan *delay frame* yang diinginkan, (2) hasil dari Quadtree digunakan untuk membuat animasi tiap level, dan (3) GIF disimpan pada *output path* yang telah ditentukan.

Lampiran

Repository: [OxNathaniel/Tucil2_13523003_13523013](#)

Google Drive GIF: [Tucil2_13523003_13523013_GIF](#)

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 wajib	✓	
5	[Bonus] Implementasi persentase kompresi sebagai parameter wajib	✓	
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	✓	