

## **Laporan Tugas Kecil 2**

Image Compressor Using Quadtree

Algoritma Divide and Conquer



Disusun oleh:

Guntara Hambali

13523114

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

## Bab I

### Algoritma

Berikut adalah algoritma atau langkah-langkah *divide and conquer* yang ditempuh untuk melakukan kompresi gambar menggunakan Quadtree:

1. Pertama-tama, inisialisasi data gambar. Lalu, buat lah objek simpul akar untuk quadtree. Isi atribut dari objek akar tersebut sesuai dengan atributnya (pixel, tinggi blok, lebar blok, dsb.)
2. Lakukan perhitungan error (variansi, MAD, dll). Lalu cek syarat: eror di atas threshold, ukuran blok lebih besar dari minimum block size, dan ukuran blok setelah dibagi menjadi empat tidak kurang dari minimum block size. Jika semua syarat tersebut dipenuhi, lanjut ke langkah 3. Kalau salah satu syarat tidak terpenuhi, lanjut ke langkah 5
3. Bagi blok menjadi 4 bagian, dan jadikan 4 bagian tersebut sebagai simpul anak dari simpul blok tersebut.
4. Proses keempat anak dari simpul tersebut menggunakan langkah 2 (rekursif).
5. Ubah semua piksel dalam simpul tersebut menjadi rata-rata piksel blok. Akhirnya, proses untuk simpul tersebut selesai.

Seluruh simpul sekarang telah dibangkitkan dan telah memiliki atribut pixel hasil kompresi. Gambar hasil kompresi dapat dibuat dengan mengunjungi seluruh simpul yang telah dibangkitkan.

Pseudocode:

```
procedure MakeQuadTree (node) :  
    if (error > threshold AND  
        (width * height)/4 >= minBlockSize then:  
        children ← SplitIntoFour (node)  
        for each child in children:  
            MakeQuadtree (child)  
    else  
        node.pixel ← ComputeAverageColor (node.region)  
        node.isLeaf ← true
```

## Bab II

### Source Code

Berikut adalah implementasi algoritma kompresi gambar menggunakan quadtree yang saya buat menggunakan bahasa pemrograman c++:

#### Kelas QuadTreeNode

```
#ifndef QUADTREE_NODE_HPP
#define QUADTREE_NODE_HPP

#include <vector>
#include <cstdint>
#include <FreeImage.h>

struct RGB {
    uint8_t r, g, b;
};

class QuadTreeNode {
public:
    int x, y, width, height;
    int depth;
    std::vector<RGB> pixels;
    std::vector<QuadTreeNode*> children;
    RGB averageColor;

    QuadTreeNode(int x, int y, int w, int h, int d = 0)
        : x(x), y(y), width(w), height(h), depth(d) {}

    ~QuadTreeNode() {
        for (auto* child : children) {
            delete child;
        }
    }
};
```

```

    }
}

void extractNode(const unsigned char* data, int imgWidth, int imgHeight);
void splitBlock();
void renderNode(unsigned char* output, int imgWidth, int imgHeight) const;
void renderAtDepth(BYTE* bits, int width, int height, int maxDepth) const;
};

#endif

```

### Kelas ImgCompressor

```

#ifndef IMGCOMPRESSOR_HPP
#define IMGCOMPRESSOR_HPP

#include <string>
#include <vector>
#include <cstdint>
#include "QuadTreeNode.hpp"
#include "metrics.hpp"
#include "imageio.hpp"

class ImgCompressor {
private:
    std::string inputPath, outputPath;
    int method;
    double threshold;
    int minBlockSize;
    double targetCompression;

    int width, height, channels;

```

```
FIBITMAP* originalData;
QuadtreeNode* root;

size_t originalSize;
size_t compressedSize;
int treeDepth;
int nodeCount;

public:
    ImgCompressor(const std::string& inPath, const std::string& outPath, int m, double t, int
minSize, double target);
    ~ImgCompressor();

    double findBestThreshold();
    double runCompression(double currentThreshold);
    void process();
    void divide(QuadtreeNode* node, double currentThreshold);
    void generateGif(const std::string& gifPath, int duration);

    size_t getOriginalSize() const;
    size_t getCompressedSize() const;
    double getCompressionRatio() const;
    int getTreeDepth() const;
    int getNodeCount() const;

    void printStats(double execTimeInSeconds) const;
};

#endif
```

## File ImgCompressor.cpp

```
#include "header/ImgCompressor.hpp"
#include "header/gif.h"
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cmath>
#include <FreeImage.h>
#include <sys/stat.h>

using namespace std;

#define GIF_TEMP_MALLOC malloc
#define GIF_TEMP_FREE free

ImgCompressor::ImgCompressor(const string& inPath, const string&
outPath, int m, double t, int minSize, double target)
    : inputPath(inPath), outputPath(outPath), method(m), threshold(t),
  minBlockSize(minSize),
    targetCompression(target), width(0), height(0), channels(3),
    originalData(nullptr), root(nullptr),
    originalSize(0), compressedSize(0), treeDepth(0), nodeCount(0)
{}

ImgCompressor::~ImgCompressor() {
    if (originalData) FreeImage_Unload((FIBITMAP*)originalData);
    if (root) delete root;
}

static size_t getFileSize(const string& filename) {
    struct stat stat_buf;
```

```

    return stat(filename.c_str(), &stat_buf) == 0 ? stat_buf.st_size :
0;
}

double ImgCompressor::findBestThreshold() {
    double low = 0.0, high = 100.0;
    double bestThreshold = threshold;
    double epsilon = 0.01;
    int maxIter = 20;

    for (int i = 0; i < maxIter; ++i) {
        double mid = (low + high) / 2.0;
        threshold = mid;
        double ratio = runCompression(threshold);

        if (abs(ratio - targetCompression) < epsilon) {
            bestThreshold = mid;
            break;
        }

        if (ratio < targetCompression) {
            low = mid;
        } else {
            high = mid;
        }

        bestThreshold = mid;
    }

    return bestThreshold;
}

```

```

double ImgCompressor::runCompression(double currentThreshold) {
    if (root) delete root;
    root = new QuadtreeNode(0, 0, width, height);
    BYTE* bits = FreeImage_GetBits(originalData);

    root->extractNode(bits, width, height);
    divide(root, currentThreshold);

    FIBITMAP* outImage = FreeImage_Allocate(width, height, 24);
    BYTE* outBits = FreeImage_GetBits(outImage);

    root->renderNode(outBits, width, height);
    saveImage(outputPath, outImage);

    FreeImage_Unload(outImage);
    compressedSize = getFileSize(outputPath);

    return getCompressionRatio();
}

void ImgCompressor::process() {
    FreeImage_Initialise();

    FIBITMAP* image = loadImage(inputPath, width, height);
    if (!image) {
        cerr << "Failed to load image using FreeImage." << endl;
        return;
    }
}

```



```

    originalData = image;
    originalSize = getFileSize(inputPath);

    if (targetCompression > 0.0) {
        threshold = findBestThreshold();
    }

    runCompression(threshold);
    FreeImage_DeInitialise();
}

void ImgCompressor::divide(QuadtreeNode* node, double
currentThreshold) {
    nodeCount++;
    treeDepth = max(treeDepth, node->depth);

    double error = computeError(node->pixels, method);
    bool canSplit = (node->height * node->width / 4 >= minBlockSize);
    if (error <= currentThreshold || !canSplit) {
        return;
    }

    node->splitBlock();
    for (auto* child : node->children) {
        child->extractNode(FreeImage_GetBits((FIBITMAP*)originalData),
width, height);
        divide(child, currentThreshold);
    }
}

void ImgCompressor::generateGif(const string& gifPath, int duration) {

```

```

int delay = duration / 10;

GifWriter writer = {};
GifBegin(&writer, gifPath.c_str(), width, height, delay);

for (int d = 0; d <= treeDepth; ++d) {
    FIBITMAP* frameImage = FreeImage_Allocate(width, height, 24);
    BYTE* frameBits = FreeImage_GetBits(frameImage);
    root->renderAtDepth(frameBits, width, height, d);

    std::vector<uint8_t> rgba(width * height * 4);
    for (int y = 0; y < height; ++y) {
        int flippedY = height - 1 - y;
        for (int x = 0; x < width; ++x) {
            int i = flippedY * width + x;
            BYTE* pixel = frameBits + i * 3;
            int j = y * width + x;
            rgba[j * 4 + 0] = pixel[FI_RGBA_RED];
            rgba[j * 4 + 1] = pixel[FI_RGBA_GREEN];
            rgba[j * 4 + 2] = pixel[FI_RGBA_BLUE];
            rgba[j * 4 + 3] = 255;
        }
    }

    GifWriteFrame(&writer, rgba.data(), width, height, delay);
    FreeImage_Unload(frameImage);
}

// Frame akhir (hasil kompresi)
for (int i = 0; i < 4; ++i) {

```

```

        FIBITMAP* finalImage = FreeImage_Allocate(width, height, 24);
        BYTE* finalBits = FreeImage_GetBits(finalImage);
        root->renderNode(finalBits, width, height);

        std::vector<uint8_t> rgba(width * height * 4);
        for (int y = 0; y < height; ++y) {
            int flippedY = height - 1 - y;
            for (int x = 0; x < width; ++x) {
                int i = flippedY * width + x;
                BYTE* pixel = finalBits + i * 3;
                int j = y * width + x;
                rgba[j * 4 + 0] = pixel[FI_RGBA_RED];
                rgba[j * 4 + 1] = pixel[FI_RGBA_GREEN];
                rgba[j * 4 + 2] = pixel[FI_RGBA_BLUE];
                rgba[j * 4 + 3] = 255;
            }
        }

        GifWriteFrame(&writer, rgba.data(), width, height, delay);
        FreeImage_Unload(finalImage);
    }

    GifEnd(&writer);
}

void ImgCompressor::printStats(double execTime) const {
    std::cout << "\n=== COMPRESSION STATS ===\n";
    std::cout << "Execution Time          : " << execTime << "
seconds\n";
    std::cout << "Original File Size      : " << originalSize << "\n";
    std::cout << "Compressed File Size   : " << compressedSize << "\n";
}

```

```

        std::cout << "Compression Ratio      : " << std::fixed <<
std::setprecision(2) << getCompressionRatio() * 100 << " %\n";

        std::cout << "Tree Depth          : " << treeDepth << "\n";

        std::cout << "Tree Nodes          : " << nodeCount << "\n";

        std::cout << "Compressed image saved to: " << outputPath << "\n";
    }

size_t ImgCompressor::getOriginalSize() const {
    return originalSize;
}

size_t ImgCompressor::getCompressedSize() const {
    return compressedSize;
}

double ImgCompressor::getCompressionRatio() const {
    return 1 - (double)compressedSize / (double)originalSize;
}

int ImgCompressor::getTreeDepth() const {
    return treeDepth;
}

int ImgCompressor::getNodeCount() const {
    return nodeCount;
}

```

## File QuadTreeNode.cpp

```
#include "header/QuadtreeNode.hpp"
```

```

#include <numeric>
#include <algorithm>

void QuadtreeNode::extractNode(const unsigned char* data, int
imgWidth, int imgHeight) {
    pixels.clear();
    long long sumR = 0, sumG = 0, sumB = 0;
    int count = 0;

    for (int j = y; j < y + height && j < imgHeight; ++j) {
        for (int i = x; i < x + width && i < imgWidth; ++i) {
            int idx = (j * imgWidth + i) * 3;
            RGB color = { data[idx], data[idx + 1], data[idx + 2] };
            pixels.push_back(color);
            sumR += color.r;
            sumG += color.g;
            sumB += color.b;
            count++;
        }
    }
    averageColor.r = sumR / count;
    averageColor.g = sumG / count;
    averageColor.b = sumB / count;
}

void QuadtreeNode::splitBlock() {
    int halfW = width / 2;
    int halfH = height / 2;
    children.push_back(new QuadtreeNode(x, y, halfW, halfH, depth +
1)); // atas kiri

```

```

        children.push_back(new QuadtreeNode(x + halfW, y, width - halfW,
halfH, depth + 1)); // atas kanan

        children.push_back(new QuadtreeNode(x, y + halfH, halfW, height -
halfH, depth + 1)); // bawah kiri

        children.push_back(new QuadtreeNode(x + halfW, y + halfH, width -
halfW, height - halfH, depth + 1)); // bawah kanan
    }

void QuadtreeNode::renderNode(unsigned char* output, int imgWidth, int
imgHeight) const {
    if (children.empty()) {
        for (int j = y; j < y + height && j < imgHeight; ++j) {
            for (int i = x; i < x + width && i < imgWidth; ++i) {
                int idx = (j * imgWidth + i) * 3;
                output[idx] = averageColor.r;
                output[idx + 1] = averageColor.g;
                output[idx + 2] = averageColor.b;
            }
        }
    } else {
        for (auto* child : children) {
            child->renderNode(output, imgWidth, imgHeight);
        }
    }
}

void QuadtreeNode::renderAtDepth(unsigned char* output, int imgWidth,
int imgHeight, int maxDepth) const {
    if (depth > maxDepth) return;

    if (children.empty() || depth == maxDepth) {
        for (int j = y; j < y + height && j < imgHeight; ++j) {

```

```

        for (int i = x; i < x + width && i < imgWidth; ++i) {
            int idx = (j * imgWidth + i) * 3;
            output[idx] = averageColor.r;
            output[idx + 1] = averageColor.g;
            output[idx + 2] = averageColor.b;
        }
    }
} else {
    for (auto* child : children) {
        child->renderAtDepth(output, imgWidth, imgHeight,
maxDepth);
    }
}
}
}

```

### File main.cpp

```

#include <iostream>
#include <string>
#include <chrono>
#include "header/ImgCompressor.hpp"
using namespace std;

int main() {
    string inputPath, outputPath, outputGifPath;
    int method, minBlockSize;
    double threshold, targetCompression;

    cout << "=== Kompresi GAMBAR MENGGUNAKAN QUADTREE ===\n";

    cout << "Masukkan path absolut gambar: ";
    getline(cin, inputPath);
}

```

```

    cout << "Pilih metode perhitungan error:\n";
    cout << "[1] Variance\n[2] Mean Absolute Deviation (MAD)\n[3] Entropy\n[4] Max Pixel Difference\n";
    cout << "Masukkan nomor metode: ";
    cin >> method;

    cout << "Masukkan nilai threshold (berdasarkan metode): ";
    cin >> threshold;

    cout << "Masukkan ukuran blok minimum: ";
    cin >> minBlockSize;

    cout << "Masukkan target rasio kompresi (0 untuk menonaktifkan): ";
    cin >> targetCompression;
    cin.ignore();

    cout << "Masukkan path absolut gambar hasil kompresi: ";
    getline(cin, outputPath);

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

    ImgCompressor compressor(inputPath, outputPath, method, threshold, minBlockSize, targetCompression);
    compressor.process();

    cout << "Masukkan path absolut GIF: ";
    getline(cin, outputGifPath);
    compressor.generateGif(outputGifPath, 1000);

```



```
    auto end = chrono::high_resolution_clock::now();  
    chrono::duration<double> execTime = end - start;  
  
    compressorprintStats(execTime.count());  
  
    return 0;  
}
```

### File imageio.cpp

```
#include "header/imageio.hpp"  
using namespace std;  
  
FIBITMAP* loadImage(const std::string& filename, int& width, int& height) {  
    FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(filename.c_str(),  
0);  
    if (fif == FIF_UNKNOWN) fif =  
FreeImage_GetFIFFromFilename(filename.c_str());  
    if (fif == FIF_UNKNOWN) return nullptr;  
  
    FIBITMAP* dib = FreeImage_Load(fif, filename.c_str());  
    if (!dib) return nullptr;  
  
    FIBITMAP* rgbImage = FreeImage_ConvertTo24Bits(dib);  
    FreeImage_Unload(dib);  
  
    width = FreeImage_GetWidth(rgbImage);  
    height = FreeImage_GetHeight(rgbImage);  
    return rgbImage;  
}
```

```
bool saveImage(const std::string& filename, FIBITMAP* image) {  
    return FreeImage_Save(FIF_PNG, image, filename.c_str(),  
        PNG_DEFAULT);  
}
```

## File metrics.cpp

```
#include "header/QuadtreeNode.hpp"  
#include "header/metrics.hpp"  
  
double computeVariance(const std::vector<RGB>& pixels) {  
    double sumR = 0, sumG = 0, sumB = 0;  
    for (const auto& p : pixels) {  
        sumR += p.r;  
        sumG += p.g;  
        sumB += p.b;  
    }  
    double n = pixels.size();  
    double meanR = sumR / n;  
    double meanG = sumG / n;  
    double meanB = sumB / n;  
  
    double varR = 0;  
    double varG = 0;  
    double varB = 0;  
    for (const auto& p : pixels) {  
        varR += pow(p.r - meanR, 2);  
        varG += pow(p.g - meanG, 2);  
        varB += pow(p.b - meanB, 2);  
    }  
    varR /= n;
```

```

    varG /= n;

    varB /= n;

    return (varR + varG + varB) / 3;
}

double computeMAD(const std::vector<RGB>& pixels) {
    double sumR = 0, sumG = 0, sumB = 0;

    for (const auto& p : pixels) {
        sumR += p.r;
        sumG += p.g;
        sumB += p.b;
    }

    double n = pixels.size();

    double meanR = sumR / n;
    double meanG = sumG / n;
    double meanB = sumB / n;

    double MADR = 0;
    double MADG = 0;
    double MADB = 0;

    for (const auto& p : pixels) {
        MADR += abs(p.r - meanR);
        MADG += abs(p.g - meanG);
        MADB += abs(p.b - meanB);
    }

    MADR /= n;
    MADG /= n;
    MADB /= n;

    return (MADR + MADG + MADB) / 3;
}

```

```

double computeEntropy(const std::vector<RGB>& pixels) {
    std::array<int, 256> histR{}, histG{}, histB{};

    for (const auto& p : pixels) {
        histR[p.r]++;
        histG[p.g]++;
        histB[p.b]++;
    }

    double n = static_cast<double>(pixels.size());

    auto channelEntropy = [n](const std::array<int, 256>& hist) ->
double {
        double entropy = 0.0;
        for (int count : hist) {
            if (count == 0) continue;
            double p = count / n;
            entropy -= p * std::log2(p);
        }
        return entropy;
    };

    double hR = channelEntropy(histR);
    double hG = channelEntropy(histG);
    double hB = channelEntropy(histB);

    return (hR + hG + hB) / 3.0;
}

double computeMaxDiff(const std::vector<RGB>& pixels) {
    uint8_t minR = 255, minG = 255, minB = 255;

```

```
uint8_t maxR = 0, maxG = 0, maxB = 0;

for (const auto& p : pixels) {
    minR = std::min(minR, p.r);
    minG = std::min(minG, p.g);
    minB = std::min(minB, p.b);

    maxR = std::max(maxR, p.r);
    maxG = std::max(maxG, p.g);
    maxB = std::max(maxB, p.b);
}

return ((maxR - minR) + (maxG - minG) + (maxB - minB)) / 3;
}



double computeError(const std::vector<RGB>& pixels, int method) {
    switch (method) {
        case 1: return computeVariance(pixels);
        case 2: return computeMAD(pixels);
        case 3: return computeEntropy(pixels);
        case 4: return computeMaxDiff(pixels);
        default: return 0.0;
    }
}
```

## Bab III



### Test Case

Berikut merupakan beragam kasus uji yang ditujukan untuk menguji keberjalanan program dan fitur-fiturnya. Gambar dalam pdf mungkin terkompresi. Oleh karena itu, saya sediakan dokumen docx di repositori github.



#### Test Case 1 (Variance)

Input	Output
 === Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 1 Masukkan nilai threshold (berdasarkan metode): 100 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0 Masukkan path absolut gambar hasil kompresi: D:/tree2.jpg Masukkan path absolut GIF: D:/tree2.gif	 === COMPRESSION STATS === Execution Time : 56.3076 seconds Original File Size : 5669327 Compressed File Size : 1833077 Compression Ratio : 67.67 % Tree Depth : 10 Tree Nodes : 602593 Compressed image saved to: D:/tree2.jpg

## Test Case 2 (MAD)


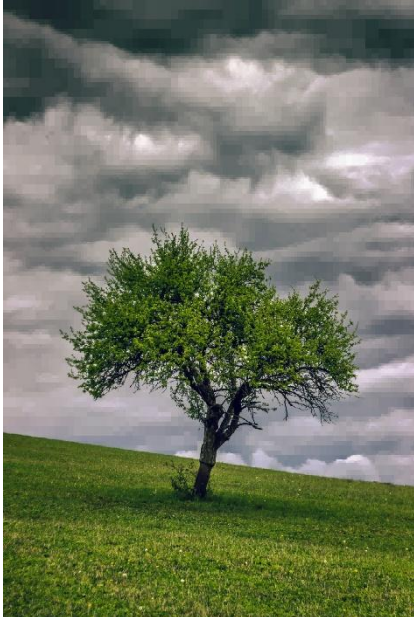
Input	Output
 <p>=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 2 Masukkan nilai threshold (berdasarkan metode): 9.375 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0 Masukkan path absolut gambar hasil kompresi: D:/treeMAD.jpg Masukkan path absolut GIF: D:/treeMAD.gif</p>	 <p>=== COMPRESSION STATS === Execution Time : 46.536 seconds Original File Size : 5669327 Compressed File Size : 1717556 Compression Ratio : 69.70 % Tree Depth : 10 Tree Nodes : 552433 Compressed image saved to: D:/treeMAD.jpg</p>

### Test Case 3 (Entropy)



Input	Output
<div data-bbox="295 285 704 898"></div> <p>=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 3 Masukkan nilai threshold (berdasarkan metode): 5 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0 Masukkan path absolut gambar hasil kompresi: D:/treeEntropy.jpg Masukkan path absolut GIF: D:/treeEntropy.gif</p>	<div data-bbox="914 285 1323 898"></div> <p>=== COMPRESSION STATS === Execution Time : 52.0019 seconds Original File Size : 5669327 Compressed File Size : 1757859 Compression Ratio : 68.99 % Tree Depth : 10 Tree Nodes : 544609 Compressed image saved to: D:/treeEntropy.jpg</p>





#### Test Case 4 (Max Pixel Difference)

Input	Output
<div data-bbox="295 285 706 898"></div> <p data-bbox="203 903 755 1501">=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 4 Masukkan nilai threshold (berdasarkan metode): 62.5 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0 Masukkan path absolut gambar hasil kompresi: D:/treeMax.jpg Masukkan path absolut GIF: D:/treeMax.gif</p>	<div data-bbox="914 285 1325 898"></div> <p data-bbox="820 903 1388 1186">=== COMPRESSION STATS === Execution Time : 49.8842 seconds Original File Size : 5669327 Compressed File Size : 1666543 Compression Ratio : 70.60 % Tree Depth : 10 Tree Nodes : 513877 Compressed image saved to: D:/treeMax.jpg</p>

### Test Case 5 (Target Compression using MAD)

Input	Output
<div data-bbox="295 285 706 898"></div> <p>=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 2 Masukkan nilai threshold (berdasarkan metode): 0 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0.7 Masukkan path absolut gambar hasil kompresi: D:/treeTargetMAD.jpg Best Threshold: 9.375 Masukkan path absolut GIF: D:/treeTargetMAD.gif</p>	<div data-bbox="914 285 1325 898"></div> <p>=== COMPRESSION STATS === Execution Time : 58.967 seconds Original File Size : 5669327 Compressed File Size : 1717556 Compression Ratio : 69.70 % Tree Depth : 10 Tree Nodes : 2399970 Compressed image saved to: D:/treeTargetMAD.jpg</p>

## Test Case 6 (Target Compression using Entropy)

Input	Output
<div data-bbox="295 285 706 898"></div> <p>=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: 3 Masukkan nilai threshold (berdasarkan metode): 1 Masukkan ukuran blok minimum: 8 Masukkan target rasio kompresi (0 untuk menonaktifkan): 0.7 Masukkan path absolut gambar hasil kompresi: D:/treeTargetEntropy.jpg Best Threshold: 5.0293 Masukkan path absolut GIF: D:/treeTargetEntropy.gif</p>	<div data-bbox="914 285 1325 898"></div> <p>=== COMPRESSION STATS === Execution Time : 104.699 seconds Original File Size : 5669327 Compressed File Size : 1712302 Compression Ratio : 69.80 % Tree Depth : 10 Tree Nodes : 5176488 Compressed image saved to: D:/treeTargetEntropy.jpg</p>

**Test Case 7 (Metode tidak valid)**

Input	Output
=== Kompresi GAMBAR MENGGUNAKAN QUADTREE === Masukkan path absolut gambar: D:/tree.jpg Pilih metode perhitungan error: [1] Variance [2] Mean Absolute Deviation (MAD) [3] Entropy [4] Max Pixel Difference Masukkan nomor metode: <b>5</b>	[ERROR] Metode tidak valid! Pilih angka antara 1 hingga 4.

## Bab IV

### Analisis Kompleksitas

Pada program ini, mari kita berfokus kepada bagian program yang menerapkan *divide and conquer* untuk dianalisis kompleksitas algoritmanya. Sekarang, kita asumsikan gambar yang diberikan berukuran  $N \times N$  pixel. Setiap split, 4 anak dihasilkan. Sekarang, kita lihat bagaimana kompleksitas program saat simpul dibagi terus hingga `minBlockSize`? (asumsi threshold selalu dilewati). Maka, dari ukuran awal  $N$ , kita bisa membagi gambar sebanyak  $\log_2 \left( \frac{N}{\text{minBlockSize}} \right)$  kali. Sementara itu, jumlah simpul adalah:

$$1 + 4 + 16 + \dots + 4^k$$

Dengan  $k$  adalah frekuensi gambar dipecah (yang dalam hal ini sudah ditemukan  $\log_2(N/\text{minBlockSize})$ ). Artinya, kompleksitas menjadi:

$$T(n) = 1 + 4 + 16 + \dots + 4^{\log_2 \left( \frac{N}{\text{minBlockSize}} \right)} = O \left( \frac{N^2}{\text{minBlockSize}} \right)$$

Lebih worst case lagi, apabila `minBlockSize` bernilai 1. Sehingga pada kasus terburuk, kompleksitas algoritma tersebut adalah  $O(N^2)$

## Bab V

### Implementasi Bonus

#### TargetCompression

TargetCompression dibuat dengan cara mencari threshold yang menghasilkan nilai kompresi paling mendekati target. Artinya metode ini dilakukan secara iteratif (binary search) untuk mencoba threshold berapa yang menghasilkan compression rasio paling mendekati target. Oleh karena itu, target ini sangat bergantung pada jumlah iterasinya agar semakin konvergen ke nilai target yang diinginkan.

#### Gif

Fungsi `generateGif` membuat animasi GIF dengan merekam proses kompresi gambar menggunakan struktur Quadtree. Untuk setiap kedalaman pohon dari 0 hingga maksimum, program membuat frame gambar baru, mewarnai blok-bloknya sesuai hasil kompresi pada kedalaman tersebut, lalu membalik citra secara vertikal agar orientasinya benar, dan menyimpannya sebagai frame dalam GIF. Setelah semua kedalaman selesai, ditambahkan beberapa frame akhir yang menunjukkan hasil kompresi penuh agar tampak lebih lama di akhir animasi. Proses ini menggunakan `FreeImage` untuk mengatur piksel dan `gif.h` untuk menyimpan file GIF

## Bab VI

### Lampiran

#### Repositori program

[https://github.com/guntarahmbi/Tucil2\\_13523114](https://github.com/guntarahmbi/Tucil2_13523114)

#### Sumber untuk gif header

<https://github.com/charlietangora/gif-h/blob/master/gif.h>

#### Pernyataan

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	V	
2	Program berhasil dijalankan	V	
3	Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	V	
4	Mengimplementasi seluruh metode perhitungan error wajib	V	
5	[Bonus] Implementasi persentase kompresi sebagai parameter tambahan	V	
6	[Bonus] Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error		V
7	[Bonus] Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar	V	
8	Program dan laporan dibuat sendiri	V	