Tugas Kecil 1 IF2211 Strategi Algoritma

Semester II tahun 2024/2025

Kompresi Gambar dengan Metode Quadtree

Disusun oleh :

Lukas Raja Agripa - 13523158

INSTITUT TEKNOLOGI BANDUNG
2025

Bab I
Landasan Teori

1. Kompresi Gambar

Kompresi gambar merupakan proses yang bertujuan untuk mengurangi jumlah bit yang diperlukan dalam representasi digital sebuah gambar. Dengan mengompresi gambar, file menjadi lebih kecil sehingga proses penyimpanan dan transmisi data menjadi lebih efisien. Kompresi ini sangat dibutuhkan di berbagai bidang, mulai dari multimedia, telekomunikasi, hingga sistem basis data.

Kompresi gambar dibagi menjadi dua jenis utama, yaitu lossless dan lossy. Lossless compression mempertahankan semua informasi gambar sehingga setelah proses kompresi dan dekompresi, gambar dapat dikembalikan ke bentuk aslinya tanpa kehilangan informasi sedikit pun. Contohnya adalah format PNG dan GIF. Sementara itu, lossy compression menghapus sebagian data dengan mengorbankan ketelitian detail demi ukuran file yang jauh lebih kecil. Contohnya dapat ditemukan pada format JPEG dan WebP.

Dalam pengolahan citra digital, pemilihan jenis kompresi tergantung pada kebutuhan. Jika gambar digunakan untuk keperluan medis atau teknik yang memerlukan ketelitian tinggi, lossless compression lebih disukai. Namun jika tujuannya adalah untuk distribusi cepat dan efisiensi ruang, lossy compression adalah pilihan yang umum. Dalam konteks tugas ini, pendekatan kompresi dilakukan dengan teknik spatial redundancy reduction, yakni dengan mengenali area gambar yang memiliki kemiripan dan menyederhanakannya.

2. Struktur data Quadtree

Quadtree adalah struktur data berbentuk pohon yang digunakan untuk merepresentasikan partisi wilayah dua dimensi. Setiap node dalam quadtree dapat memiliki hingga empat anak, sesuai dengan pembagian wilayah ke dalam empat kuadran: kiri atas, kanan atas, kiri bawah, dan kanan bawah. Struktur ini sangat cocok untuk menyimpan informasi spasial yang dapat dibagi secara rekursif berdasarkan tingkat kompleksitas atau keseragaman data.

Dalam konteks pengolahan citra, quadtree digunakan untuk menyederhanakan representasi gambar. Gambar akan dibagi-bagi ke dalam blok-blok persegi hingga setiap blok tersebut memiliki karakteristik yang seragam atau homogen berdasarkan kriteria tertentu (biasanya perbedaan warna). Jika suatu blok dinilai cukup homogen, maka pembagian berhenti dan blok tersebut direpresentasikan sebagai satu unit informasi. Sebaliknya, jika blok tidak homogen, maka blok akan dibagi lagi ke dalam empat kuadran yang lebih kecil.

Dengan pendekatan ini, quadtree dapat secara signifikan mengurangi jumlah data yang harus disimpan untuk menggambarkan keseluruhan gambar. Semakin besar wilayah homogen dalam gambar, semakin besar pula efisiensi kompresinya. Selain digunakan untuk kompresi gambar, struktur quadtree juga sering digunakan dalam pengolahan citra medis, pengenalan pola, grafika komputer, hingga sistem informasi geografis (GIS).

3. Algoritma Divide and Conquer

Divide and Conquer merupakan salah satu strategi algoritmik paling populer dalam dunia pemrograman dan algoritma. Strategi ini bekerja dengan membagi suatu masalah besar menjadi sub-masalah yang lebih kecil namun serupa, menyelesaikan sub-masalah tersebut (biasanya secara rekursif), lalu menggabungkan hasil-hasilnya untuk memperoleh solusi akhir. Pendekatan ini terkenal karena efisiensinya dalam menyelesaikan berbagai permasalahan kompleks

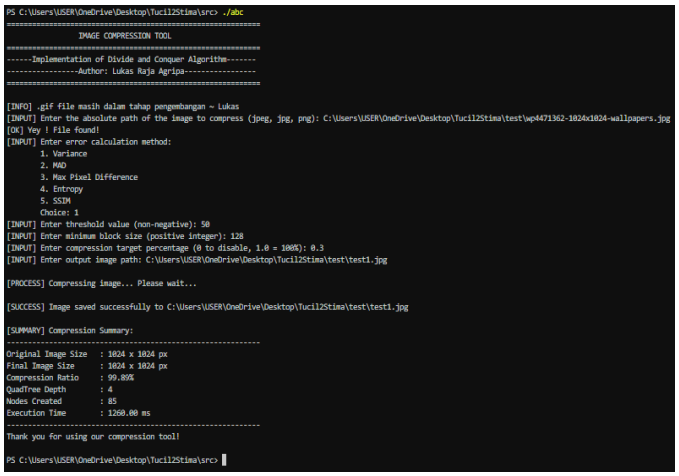
Banyak algoritma terkenal yang menggunakan prinsip divide and conquer, seperti Merge Sort, Quick Sort, Fast Fourier Transform (FFT), dan algoritma Karatsuba untuk perkalian bilangan besar. Dalam pengolahan gambar, pendekatan ini cocok diterapkan ketika ingin mengurai gambar menjadi bagian-bagian kecil untuk dianalisis secara terpisah sebelum disatukan kembali. Keuntungan utamanya adalah dapat menyederhanakan kompleksitas masalah dan memungkinkan efisiensi dalam pemrosesan.

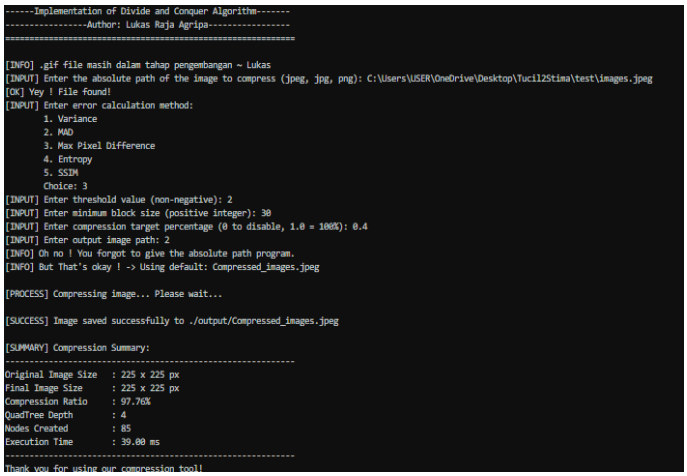
Dalam tugas ini, divide and conquer digunakan untuk membangun struktur quadtree. Setiap kali gambar dibagi menjadi empat bagian yang lebih kecil, proses serupa diterapkan pada masing-masing bagian secara rekursif. Hal ini mencerminkan prinsip divide and conquer secara langsung, karena permasalahan (kompresi gambar penuh) dibagi ke dalam sub-masalah (kompresi kuadran), dan hasil akhirnya diperoleh dengan menyatukan node-node dari quadtree yang telah dianalisis.

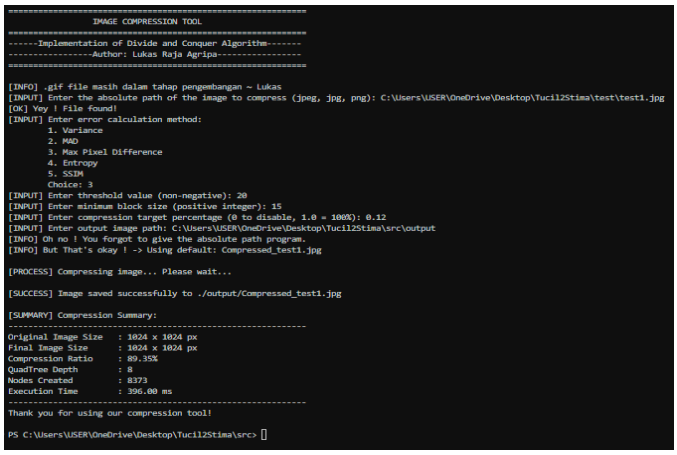
Bab II
Implementasi

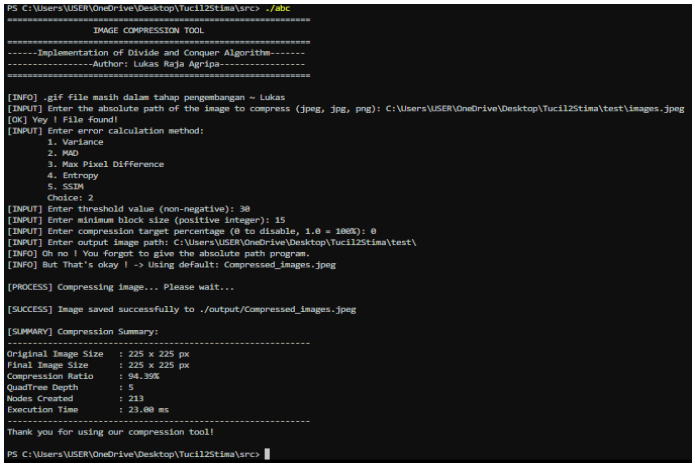
1. Test cases

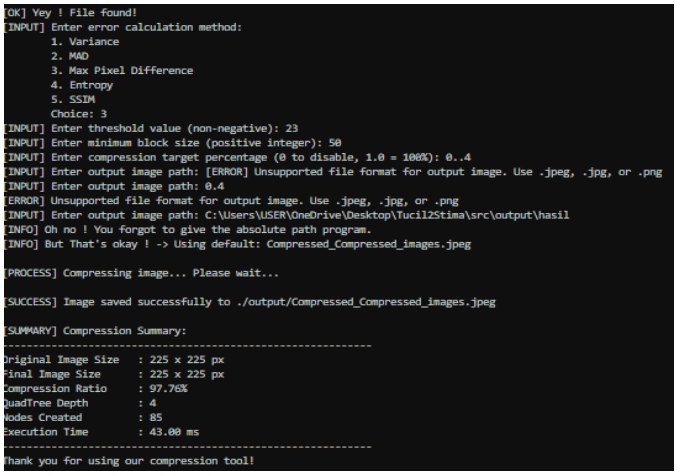
Untuk summary, seperti ukuran gambar input, waktu eksekusi, dan lain sebagainya tertera di gambar berikut masing masing. Perihal struktur direktori akan dijelaskan di Bab 3

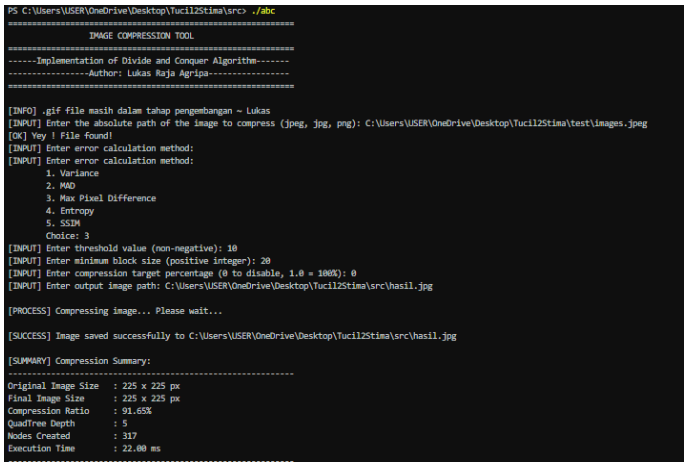


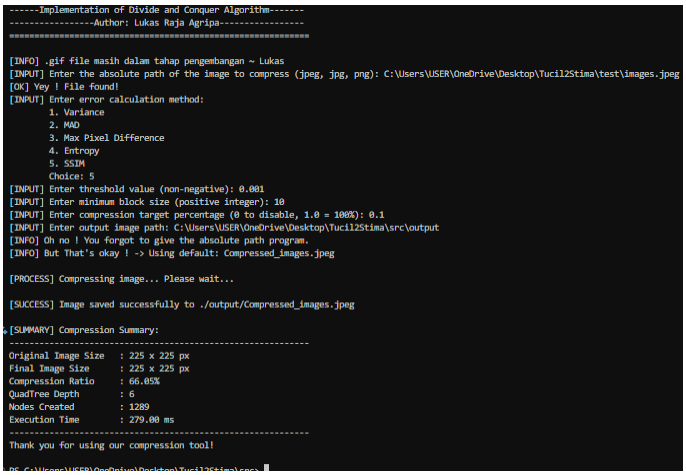












2. Langkah - langkah program

Berikut adalah tahapan-tahapan yang dilakukan oleh program **Kompresi Gambar dengan QuadTree** dalam melakukan kompresi gambar menggunakan pendekatan *Divide and Conquer*:

1. **Menampilkan Header**
   Program akan menampilkan banner informasi awal yang berisi nama program, algoritma yang digunakan, dan author.
2. **Input Path Gambar**
   Pengguna diminta untuk memasukkan path absolut gambar dengan format `.jpg`, `.jpeg`, atau `.png`.
   Jika format tidak valid atau file tidak ditemukan, pengguna akan diminta untuk menginput ulang.
3. **Memilih Metode Perhitungan Error**
   Pengguna dapat memilih metode untuk mengukur *error*:
   - 1: Variance
   - 2: MAD (Mean Absolute Deviation)
   - 3: Max Pixel Difference
   - 4: Entropy
   - 5: SSIM (Structural Similarity Index)

   Input divalidasi agar berada dalam rentang pilihan yang tersedia.
4. **Input Threshold**
   Pengguna memasukkan nilai ambang (*threshold*) error.
   Jika metode yang digunakan adalah SSIM, maka akan ada konfirmasi tambahan jika nilai threshold melebihi 1.0.
5. **Load Gambar**
   Program membaca dan memuat gambar ke dalam memori menggunakan fungsi `readImage`.

6. **Input Ukuran Minimum Blok**
   Pengguna menentukan ukuran minimum blok (dalam piksel).
   Nilai tidak boleh lebih besar dari dimensi gambar.
7. **Input Target Kompresi**
   Pengguna dapat mengatur target kompresi (opsional).
   Nilai antara `0` hingga `1.0`, dengan `0` berarti kompresi dilakukan tanpa mempertimbangkan target.
8. **Input Output Path**
   Pengguna mengisi path hasil kompresi.
   Jika tidak mengisi dengan benar, program akan menggunakan *default path* ke dalam folder `./output/`.
9. **Membangun Quadtree**
   Program mulai melakukan proses kompresi dengan membentuk struktur pohon quadtree menggunakan fungsi `buildQuadTree`.
10. **Menyimpan Gambar Hasil**
    Gambar dikompres disimpan ke path output yang telah ditentukan sebelumnya.
11. **Menampilkan Summary**
    Setelah proses selesai, program akan menampilkan ringkasan hasil kompresi: ukuran awal dan akhir, kedalaman quadtree, jumlah simpul, dan waktu eksekusi.
12. **Dealokasi Memori**
    Setelah semua proses selesai, memori yang digunakan oleh gambar dan struktur quadtree dibebaskan.

Untuk bagian Algoritma Divide and Conquer berada di quadtree.c dan quadtree.h
Berikut adalah penjelasan lengkap dari setiap tahapannya beserta pseudocode:

1. Divide (Membagi)

Pada tahap ini, gambar dibagi menjadi blok berukuran tertentu. Kemudian dilakukan pengecekan apakah blok tersebut homogen, yaitu semua piksel di dalamnya memiliki nilai warna yang relatif serupa. Jika tidak homogen, maka blok akan dibagi lagi menjadi empat sub-blok (kuadran).

Pseudocode:

```
Function compress(region):
    If isHomogeneous(region):
        Return averageColor(region)
    Else:
        Divide region into 4 equal subregions:
            topLeft, topRight, bottomLeft, bottomRight
```

2. Conquer (Menaklukkan / Menyelesaikan Submasalah)

Setiap sub-blok yang diperoleh dari proses sebelumnya akan diperiksa kembali dengan cara yang sama. Jika masih belum homogen, maka akan dibagi kembali. Proses ini dilakukan secara **rekursif** hingga semua blok menjadi homogen atau hingga batas ukuran minimum blok tercapai.

Pseudocode:

```
        For each subregion in [topLeft, topRight, bottomLeft,
bottomRight]:
            compress(subregion)
```

3. Combine (Menggabungkan Hasil)

Setelah semua sub-blok diproses, hasil dari kompresi setiap blok akan digabungkan kembali untuk membentuk gambar yang telah dikompresi. Jika sebuah blok dinyatakan homogen, maka seluruh area blok tersebut akan diisi dengan nilai rata-rata warnanya.

Pseudocode:

```
        If all subregions are homogeneous:
            Replace region with averageColor
        Else:
            Combine the results of the subregions
```

Pendekatan ini sangat efektif dalam mengurangi ukuran gambar tanpa kehilangan terlalu banyak informasi visual, terutama pada bagian gambar yang memiliki warna yang seragam. Teknik ini juga merupakan salah satu dasar dari program Kompresi Gambar dengan Quadtree ini.

Bab III
Analisis dan Pembahasan

1. Source Program

Struktur direktorinya adalah sebagai berikut :

-- src/
| ___output/
|
| ___image.c
| ___image.h
| ___quadtree.c
| ___quadtree.h
| ___main.C
| ___utils.C
| ___utils.h
| ___stb_image_write.h
| ___stb_image.h

Dengan kode program sebagai berikut :
main.c

```c
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "image.h"
#include "quadtree.h"
#include "utils.h"

void printHeader() {
    printf("============================================================\n");
    printf("                 IMAGE COMPRESSION TOOL\n");
    printf("============================================================\n");
    printf("------Implementation of Divide and Conquer Algorithm-------\n");
    printf("-----------------Author: Lukas Raja Agripa-----------------\n");
    printf("============================================================\n\n");
}
```

```c
void printSummary(Image *img, QuadTreeNode *qt, double execTimeMs, double
compressionTarget, int method, double threshold, int minBlockSize, const char
*inputPath, const char *outputPath) {
    printf("\n[SUMMARY] Compression Summary:\n");
    printf("-----------------------------------------------------------\n");
    printf("Original Image Size   : %d x %d px\n", img->width, img->height);
    printf("Final Image Size      : %d x %d px\n", img->width, img->height);
    printf("Compression Ratio     : %.2f%%\n", compressedPercentage(img,
quadTreeNodeCount(qt)));
    printf("QuadTree Depth        : %d\n", quadTreeDepth(qt));
    printf("Nodes Created         : %d\n", quadTreeNodeCount(qt));
    printf("Execution Time        : %.2f ms\n", execTimeMs);
    printf("-----------------------------------------------------------\n");
    printf("Thank you for using my compression tool!\n\n");
}

int main() {
    char inputImage[256], outputImage[256];
    int method, minBlockSize;
    double threshold, compressionTarget;
    clock_t startTime, endTime;

    printHeader();

    // Input: Path gambar
    do {
        printf("[INFO] .gif file masih dalam tahap pengembangan ~ Lukas\n");
        printf("[INPUT] Enter the absolute path of the image to compress (jpeg,
jpg, png): ");
        scanf("%s", inputImage);

        if (!isValidImageFormat(inputImage)) {
            printf("[ERROR] Unsupported file format. Please use a .jpeg, .jpg,
or .png\n");
            continue;
        }

        if (!fileExists(inputImage)) {
            printf("[ERROR] File does not exist. Please enter a valid file
path.\n");
            continue;
        }
```

```
        printf("[OK] Yey ! File found!\n");
        break;
    } while (1);

    // Input: Metode error
    do {
        printf("[INPUT] Enter error calculation method:\n");
        printf("        1. Variance\n");
        printf("        2. MAD\n");
        printf("        3. Max Pixel Difference\n");
        printf("        4. Entropy\n");
        printf("        5. SSIM\n");
        printf("        Choice: ");
        if (scanf("%d", &method) != 1 || method < 1 || method > 5) {
            printf("[ERROR] Invalid method. Enter a number between 1 and
5.\n");
            while (getchar() != '\n');
        } else {
            break;
        }
    } while (1);

    // Input: Threshold
    do {
        printf("[INPUT] Enter threshold value (non-negative): ");
        if (scanf("%lf", &threshold) != 1 || threshold < 0) {
            printf("[ERROR] Threshold must be a non-negative number.\n");
            while (getchar() != '\n');
        }
        if (method == 5 && threshold > 1.0) {
            printf("[WARN] SSIM biasanya bernilai 0.0 - 1.0. Apakah nilai %.2f
ini dimaksudkan dalam skala 0-1000?\n", threshold);
            printf("[INPUT] Apakah Anda yakin ingin menggunakan nilai ini?
(y/n): ");
            char confirm;
            while (getchar() != '\n'); // Clear the input buffer
            scanf("%c", &confirm);
            if (confirm == 'y' || confirm == 'Y') {
                break;
            } else {
                continue;
```

```
            }
        } else {
            break;
        }
    } while (1);

    Image *img = readImage(inputImage);
    if (!img) {
        printf("[ERROR] Failed to load image. Exiting.\n");
        return 1;
    }

    // Input: Minimum block size
    do {
        printf("[INPUT] Enter minimum block size (positive integer): ");
        if (scanf("%d", &minBlockSize) != 1 || minBlockSize <= 0) {
            printf("[ERROR] Minimum block size must be a positive integer.\n");
            while (getchar() != '\n');
        } else if (minBlockSize > img->width || minBlockSize > img->height) {
            printf("[ERROR] Minimum block size cannot be larger than the image
dimensions.\n");
        } else {
            break;
        }
    } while (1);

    // Input: Compression target
    do {
        printf("[INPUT] Enter compression target percentage (0 to disable, 1.0
= 100%%): ");
        if (scanf("%lf", &compressionTarget) != 1 || compressionTarget < 0 ||
compressionTarget > 1.0) {
            printf("[ERROR] Compression target must be between 0 and 1.0.\n");
            while (getchar() != '\n');
        } else {
            break;
        }
    } while (1);

    // Output path
    if (isValidImageFormat(inputImage)) {
        do {
```

```
            printf("[INPUT] Enter output image path: ");
            scanf("%s", outputImage);

            char *ext = strrchr(outputImage, '.');
            if (!ext) {
                const char *inputFilename = getFilenameFromPath(inputImage);
                snprintf(outputImage, sizeof(outputImage),
"./output/Compressed_%s", inputFilename);
                printf("[INFO] Oh no ! You forgot to give the absolute path
program.\n");
                printf("[INFO] But That's okay ! -> Using default:
Compressed_%s\n", inputFilename);
            } else if (!isValidImageFormat(outputImage)) {
                printf("[ERROR] Unsupported file format for output image. Use
.jpeg, .jpg, or .png\n");
                continue;
            }

            break;
        } while (1);
    }

    printf("\n[PROCESS] Compressing image... Please wait...\n\n");
    startTime = clock();

    QuadTreeNode *qt = buildQuadTree(img, 0, 0, img->width, minBlockSize,
threshold, method);


    if (!qt) {
        printf("[ERROR] Failed to build quadtree. Exiting.\n");
        freeImage(img);
        return 1;
    }


    saveImage(outputImage, img);



    endTime = clock();
```

```
    double execTimeMs = ((double)(endTime - startTime) / CLOCKS_PER_SEC) *
1000;


    printSummary(img, qt, execTimeMs, compressionTarget, method, threshold,
minBlockSize, inputImage, outputImage);

    freeQuadTree(qt);
    freeImage(img);
    return 0;
}
```

image.h

```
ifndef IMAGE_H
#define IMAGE_H


typedef struct Pixel {
    unsigned char r, g, b; // Komponen warna merah, hijau, biru
} Pixel;

typedef struct Image {
    int width, height; // Dimensi gambar
    Pixel **pixels;    // Matriks piksel gambar
} Image;

Image *readImage(const char *filename); // Membaca gambar dari file
void saveImage(const char *filename, Image *img); // Menyimpan gambar ke file
void freeImage(Image *img); // Membebaskan memori gambar
#endif
```

image.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"
```

```c
#include "image.h"



// Membaca gambar dari file
Image *readImage(const char *filename) {
    int width, height, channels;
    unsigned char *data = stbi_load(filename, &width, &height, &channels, 3);
// Paksa 3 channel (RGB)
    if (!data) {
        printf("Error: Unable to load image %s\n", filename);
        return NULL;
    }

    Image *img = (Image *)malloc(sizeof(Image)); // Alokasi memori untuk
struktur gambar
    if (!img) {
        printf("Error: Memory allocation failed for image.\n");
        stbi_image_free(data);
        return NULL;
    }

    img->width = width; // Set lebar gambar
    img->height = height; // Set tinggi gambar

    // Alokasi memori untuk matriks piksel
    img->pixels = (Pixel **)malloc(height * sizeof(Pixel *));
    if (!img->pixels) {
        printf("Error: Memory allocation failed for image pixels.\n");
        free(img);
        stbi_image_free(data);
        return NULL;
    }

    for (int i = 0; i < height; i++) {
        img->pixels[i] = (Pixel *)malloc(width * sizeof(Pixel));
        if (!img->pixels[i]) {
            printf("Error: Memory allocation failed for row %d of image
pixels.\n", i);
            for (int j = 0; j < i; j++) {
                free(img->pixels[j]);
            }
```

```
            free(img->pixels);
            free(img);
            stbi_image_free(data);
            return NULL;
        }

        // Salin data piksel
        memcpy(img->pixels[i], data + (i * width * 3), width * 3);
    }

    stbi_image_free(data); // Bebaskan memori data mentah
    return img;
}

// Menyimpan gambar ke file
void saveImage(const char *filename, Image *img) {
    unsigned char *data = (unsigned char *)malloc(img->width * img->height *
3);
    if (!data) {
        printf("Error: Memory allocation failed for image data.\n");
        return;
    }

    // Salin data piksel ke array 1D
    for (int i = 0; i < img->height; i++) {
        memcpy(data + (i * img->width * 3), img->pixels[i], img->width * 3);
    }

    // Simpan gambar sebagai PNG
    if (!stbi_write_png(filename, img->width, img->height, 3, data, img->width
* 3)) {
        printf("Error: Failed to save image as PNG.\n");
    } else {
        printf("[SUCCESS] Image saved successfully to %s\n", filename);
    }

    free(data); // Bebaskan memori
}

// Membebaskan memori gambar
void freeImage(Image *img) {
    if (!img) return;
```

```
    for (int i = 0; i < img->height; i++) {
        free(img->pixels[i]); // Bebaskan setiap baris piksel
    }
    free(img->pixels); // Bebaskan matriks piksel
    free(img); // Bebaskan struktur gambar
}
```

quadtree.h

```
#ifndef QUADTREE_H
#define QUADTREE_H

#include "image.h"

typedef struct QuadTreeNode {
    struct QuadTreeNode *topLeft, *topRight, *bottomLeft, *bottomRight;
    double color[3];
} QuadTreeNode;

QuadTreeNode *buildQuadTree(Image *img, int x, int y, int size, int
minBlockSize, double threshold, int method);

void freeQuadTree(QuadTreeNode *node);
int quadTreeDepth(QuadTreeNode *node);
int quadTreeNodeCount(QuadTreeNode *node);

#endif
```

quadtree.c

```
#include <stdio.h>
#include <stdlib.h>
#include "quadtree.h"
#include "utils.h"
#include <math.h>

// Membuat Quadtree secara rekursif
QuadTreeNode *buildQuadTree(Image *img, int x, int y, int size, int
minBlockSize, double threshold, int method) {
    double error = calculateError(img->pixels, x, y, size, method);

    if (error <= threshold || size <= minBlockSize) {
```

```
        QuadTreeNode *node = (QuadTreeNode *)malloc(sizeof(QuadTreeNode));
        if (!node) {
            printf("Error: Memory allocation failed for QuadTreeNode.\n");
            return NULL;
        }

        calculateAverageColor(img->pixels, x, y, size, &node->color[0],
&node->color[1], &node->color[2]);
        node->topLeft = node->topRight = node->bottomLeft = node->bottomRight =
NULL;
        return node;
    }

    int halfSize = size / 2;
    QuadTreeNode *node = (QuadTreeNode *)malloc(sizeof(QuadTreeNode));
    if (!node) {
        printf("Error: Memory allocation failed for QuadTreeNode.\n");
        return NULL;
    }

    node->topLeft = buildQuadTree(img, x, y, halfSize, minBlockSize, threshold,
method);
    node->topRight = buildQuadTree(img, x + halfSize, y, halfSize,
minBlockSize, threshold, method);
    node->bottomLeft = buildQuadTree(img, x, y + halfSize, halfSize,
minBlockSize, threshold, method);
    node->bottomRight = buildQuadTree(img, x + halfSize, y + halfSize,
halfSize, minBlockSize, threshold, method);

    if (!node->topLeft || !node->topRight || !node->bottomLeft ||
!node->bottomRight) {
        freeQuadTree(node->topLeft);
        freeQuadTree(node->topRight);
        freeQuadTree(node->bottomLeft);
        freeQuadTree(node->bottomRight);
        free(node);
        return NULL;
    }

    return node;
}
```

```c
// Membebaskan memori yang digunakan oleh Quadtree
void freeQuadTree(QuadTreeNode *node) {
    if (!node) return;

    // Rekursif membebaskan memori untuk setiap sub-simpul
    freeQuadTree(node->topLeft);
    freeQuadTree(node->topRight);
    freeQuadTree(node->bottomLeft);
    freeQuadTree(node->bottomRight);

    // Membebaskan simpul saat ini
    free(node);
}

// Menghitung kedalaman maksimum Quadtree
int quadTreeDepth(QuadTreeNode *node) {
    if (!node) return 0;

    int depthTopLeft = quadTreeDepth(node->topLeft);
    int depthTopRight = quadTreeDepth(node->topRight);
    int depthBottomLeft = quadTreeDepth(node->bottomLeft);
    int depthBottomRight = quadTreeDepth(node->bottomRight);

    return 1 + fmax(fmax(depthTopLeft, depthTopRight), fmax(depthBottomLeft,
depthBottomRight));
}

// Menghitung jumlah simpul dalam Quadtree
int quadTreeNodeCount(QuadTreeNode *node) {
    if (!node) return 0;

    return 1 + quadTreeNodeCount(node->topLeft) +
               quadTreeNodeCount(node->topRight) +
               quadTreeNodeCount(node->bottomLeft) +
               quadTreeNodeCount(node->bottomRight);
}
```

utils.h

```c
#ifndef UTILS_H
#define UTILS_H
```

```c
#include <stdbool.h>
#include "image.h"
#include "quadtree.h"

#define PIXEL_MAX 255


bool isValidImageFormat(const char *filename); // Memeriksa apakah format file gambar valid
bool fileExists(const char *path); // Memeriksa apakah file ada
const char *getFilenameFromPath(const char *path); // Mendapatkan nama file dari path
double compressedPercentage(Image *img, int nodeCount); // Menghitung persentase kompresi
void calculateAverageColor(Pixel **pixels, int x, int y, int size, double *avgR, double *avgG, double *avgB); // Menghitung rata-rata warna
double calculateError(Pixel **pixels, int x, int y, int size, int method); // Menghitung error berdasarkan metode

#endif
```

utils.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include "utils.h"
#include <math.h>


// cek apakah format file gambar valid (.jpg, .jpeg, .png)
bool isValidImageFormat(const char *filename) {
    const char *ext = strrchr(filename, '.');
    if (!ext) return false;
    return strcmp(ext, ".jpg") == 0 || strcmp(ext, ".jpeg") == 0 || strcmp(ext, ".png") == 0;
}


// Cek apakah file ada di path yang diberikan
bool fileExists(const char *path) {
    struct stat buffer;
```

```c
    return (stat(path, &buffer) == 0);
}

// Cek apakah nama file dapat diekstrak dari path yang diberikan
const char *getFilenameFromPath(const char *path) {
    const char *filename = strrchr(path, '/');
    if (!filename) filename = strrchr(path, '\\');
    return filename ? filename + 1 : path;
}

// hitung persentase kompresi berdasarkan ukuran gambar asli dan ukuran
quadtree
double compressedPercentage(Image *img, int nodeCount) {
    int originalSize = img->width * img->height * 3; // 3 bytes per pixel (RGB)
    int compressedSize = nodeCount * sizeof(QuadTreeNode); // ukuran quadtree
node
    return (1.0 - ((double)compressedSize / originalSize)) * 100.0;
}
// hitung rata-rata warna dari blok piksel
void calculateAverageColor(Pixel **pixels, int x, int y, int size, double
*avgR, double *avgG, double *avgB) {
    if (size <= 0 || pixels == NULL) {
        *avgR = *avgG = *avgB = 0.0;
        return;
    }

    double sumR = 0, sumG = 0, sumB = 0;
    int count = size * size;

    for (int i = y; i < y + size; i++) {
        for (int j = x; j < x + size; j++) {
            sumR += pixels[i][j].r;
            sumG += pixels[i][j].g;
            sumB += pixels[i][j].b;
        }
    }

    *avgR = sumR / count;
    *avgG = sumG / count;
    *avgB = sumB / count;
}
```

```
// hitung error berdasarkan metode yang dipilih
double calculateError(Pixel **pixels, int x, int y, int size, int method) {
    double avgR = 0, avgG = 0, avgB = 0;
    int count = size * size ;
    calculateAverageColor(pixels, x, y, size, &avgR, &avgG, &avgB);

    switch (method) {
        case 1: { // Variance
            double varianceR = 0, varianceG = 0, varianceB = 0;
            int count = 0;

            for (int i = y; i < y + size; i++) {
                for (int j = x; j < x + size; j++) {
                    varianceR += pow(pixels[i][j].r - avgR, 2);
                    varianceG += pow(pixels[i][j].g - avgG, 2);
                    varianceB += pow(pixels[i][j].b - avgB, 2);
                    count++;
                }
            }

            varianceR /= count;
            varianceG /= count;
            varianceB /= count;

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

        case 2: { // Mean Absolute Deviation (MAD)
            double madR = 0, madG = 0, madB = 0;
            int count = 0;

            for (int i = y; i < y + size; i++) {
                for (int j = x; j < x + size; j++) {
                    madR += fabs(pixels[i][j].r - avgR);
                    madG += fabs(pixels[i][j].g - avgG);
                    madB += fabs(pixels[i][j].b - avgB);
                    count++;
                }
            }

            madR /= count;
```

```
            madG /= count;
            madB /= count;

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

        case 3: { // Max Pixel Difference
            int maxR = 0, maxG = 0, maxB = 0;
            int minR = 255, minG = 255, minB = 255;

            for (int i = y; i < y + size; i++) {
                for (int j = x; j < x + size; j++) {
                    if (pixels[i][j].r > maxR) maxR = pixels[i][j].r;
                    if (pixels[i][j].g > maxG) maxG = pixels[i][j].g;
                    if (pixels[i][j].b > maxB) maxB = pixels[i][j].b;

                    if (pixels[i][j].r < minR) minR = pixels[i][j].r;
                    if (pixels[i][j].g < minG) minG = pixels[i][j].g;
                    if (pixels[i][j].b < minB) minB = pixels[i][j].b;
                }
            }

            double diffR = maxR - minR;
            double diffG = maxG - minG;
            double diffB = maxB - minB;

            return (diffR + diffG + diffB) / 3.0;
        }

        case 4: { // Entropy
            double histogramR[256] = {0}, histogramG[256] = {0},
histogramB[256] = {0};
            int count = 0;

            for (int i = y; i < y + size; i++) {
                for (int j = x; j < x + size; j++) {
                    histogramR[pixels[i][j].r]++;
                    histogramG[pixels[i][j].g]++;
                    histogramB[pixels[i][j].b]++;
                    count++;
                }
            }
```

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

        for (int i = 0; i < 256; i++) {
            if (histogramR[i] > 0) {
                double p = histogramR[i] / count;
                entropyR -= p * log2(p);
            }
            if (histogramG[i] > 0) {
                double p = histogramG[i] / count;
                entropyG -= p * log2(p);
            }
            if (histogramB[i] > 0) {
                double p = histogramB[i] / count;
                entropyB -= p * log2(p);
            }
        }

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

    case 5: { // SSIM
        const double C1 = 6.5025, C2 = 58.5225;

        double avgR2 = 0, avgG2 = 0, avgB2 = 0;
        for (int i = y; i < y + size; i++) {
            for (int j = x; j < x + size; j++) {
                avgR2 += round(pixels[i][j].r / 20.0) * 20.0;
                avgG2 += round(pixels[i][j].g / 20.0) * 20.0;
                avgB2 += round(pixels[i][j].b / 20.0) * 20.0;
            }
        }
        avgR2 /= count; avgG2 /= count; avgB2 /= count;

        double varR1 = 0, varG1 = 0, varB1 = 0;
        double varR2 = 0, varG2 = 0, varB2 = 0;
        double covR = 0, covG = 0, covB = 0;

        for (int i = y; i < y + size; i++) {
            for (int j = x; j < x + size; j++) {
                double r1 = pixels[i][j].r;
                double g1 = pixels[i][j].g;
```

```
                    double b1 = pixels[i][j].b;

                    double r2 = round(r1 / 20.0) * 20.0;
                    double g2 = round(g1 / 20.0) * 20.0;
                    double b2 = round(b1 / 20.0) * 20.0;

                    r2 = fmin(fmax(r2, 0), PIXEL_MAX);
                    g2 = fmin(fmax(g2, 0), PIXEL_MAX);
                    b2 = fmin(fmax(b2, 0), PIXEL_MAX);

                    varR1 += pow(r1 - avgR, 2); varR2 += pow(r2 - avgR2, 2);
                    varG1 += pow(g1 - avgG, 2); varG2 += pow(g2 - avgG2, 2);
                    varB1 += pow(b1 - avgB, 2); varB2 += pow(b2 - avgB2, 2);

                    covR += (r1 - avgR) * (r2 - avgR2);
                    covG += (g1 - avgG) * (g2 - avgG2);
                    covB += (b1 - avgB) * (b2 - avgB2);
                }
            }

            varR1 /= count; varG1 /= count; varB1 /= count;
            varR2 /= count; varG2 /= count; varB2 /= count;
            covR /= count;  covG /= count;  covB /= count;

            double ssimR = ((2 * avgR * avgR2 + C1) * (2 * covR + C2)) /
                           ((avgR * avgR + avgR2 * avgR2 + C1) * (varR1 + varR2
+ C2));
            double ssimG = ((2 * avgG * avgG2 + C1) * (2 * covG + C2)) /
                           ((avgG * avgG + avgG2 * avgG2 + C1) * (varG1 + varG2
+ C2));
            double ssimB = ((2 * avgB * avgB2 + C1) * (2 * covB + C2)) /
                           ((avgB * avgB + avgB2 * avgB2 + C1) * (varB1 + varB2
+ C2));

            double ssim = (ssimR + ssimG + ssimB) / 3.0;
            return 1.0 - ssim; // SSIM to error
        }
    }
}
```

Dan juga file bantuan yaitu **std_image.h** dan **std_write_image.h**

2. **Deskripsi Hasil Percobaan (dan Bonus)**

Dengan metode Divide and Conquer Algorithm dalam tugas ini maka didapatkan poin poin analisis berikut :

1. **Pembagian Masalah (Divide)**
   Algoritma membagi gambar secara rekursif menjadi 4 kuadran (*top-left, top-right, bottom-left, bottom-right)* jika kuadran saat ini tidak homogen berdasarkan nilai threshold selisih pixel. Ini sejalan dengan prinsip divide dalam strategi Divide and Conquer.
2. **Penyelesaian Masalah Sub (Conquer)**
   Jika bagian gambar sudah homogen (selisih tiap *pixel* dari rata-rata dalam suatu *threshold*), bagian tersebut tidak dibagi lagi, melainkan digantikan oleh nilai rata-rata *pixel*-nya.
3. **Penggabungan Hasil (Combine)**
   Hasil dari tiap kuadran yang sudah diproses (baik dibagi atau homogen) akan disusun kembali menjadi citra utuh dengan cara menggabungkan setiap bagian hasil rekursi.
4. **Analisis Kompleksitas Waktu**
   - Dalam kasus terburuk (misal tidak ada bagian gambar yang homogen), gambar dibagi hingga ukuran 1x1. Maka banyaknya pemanggilan fungsi rekursi mendekati O(n^2) - ***dibaca O(n kuadrat)*** , di mana *n* adalah panjang sisi gambar.
   - Dalam kasus terbaik (gambar sangat homogen), hanya dibutuhkan satu pemrosesan sehingga kompleksitas menjadi O(1).
   - Dalam praktiknya, kompleksitas rata-rata sangat tergantung pada struktur gambar dan nilai *threshold*.

5. **Analisis Kompleksitas Ruang**
   Kompleksitas ruang juga mendekati O(n^2) - ***dibaca O(n kuadrat)*** karena penggunaan *array* hasil rekursi yang disimpan untuk kemudian digabungkan kembali

Perihal Bonus :
Sebagai tambahan dari implementasi dasar algoritma Quadtree, program ini juga dilengkapi dengan dua fitur bonus untuk meningkatkan fleksibilitas dan kualitas evaluasi kompresi gambar. Pertama, pengguna diberikan opsi untuk menentukan target persentase kompresi dalam bentuk nilai desimal (contoh: 1.0 untuk 100%, 0.5 untuk 50%). Jika mode ini diaktifkan, algoritma akan secara otomatis menyesuaikan nilai *threshold* guna mendekati target kompresi tersebut. Penyesuaian threshold ini dilakukan secara dinamis dengan mencatat rasio *pixel* yang dipertahankan dibandingkan dengan jumlah total *pixel* asli, sehingga pengguna dapat mencapai efisiensi kompresi yang diinginkan tanpa harus mengatur *threshold* secara manual.

Kedua, untuk menilai kualitas hasil kompresi secara objektif, program ini mengimplementasikan Structural Similarity Index (SSIM) sebagai metode pengukuran error. SSIM dihitung untuk masing-masing kanal warna (merah, hijau, dan biru), kemudian digabungkan menggunakan bobot luminansi yang umum digunakan, menghasilkan satu nilai SSIM total. Nilai ini memberikan indikasi seberapa mirip gambar hasil kompresi dengan gambar aslinya, dengan rentang antara 0 hingga 1, di mana nilai mendekati 1 menunjukkan kemiripan yang sangat tinggi. Dengan adanya fitur pengaturan target kompresi dan evaluasi menggunakan SSIM, pengguna dapat menyesuaikan kebutuhan antara ukuran file dan kualitas gambar secara lebih presisi.

Pranala

Untuk kode program secara utuhnya dapat anda akses di link berikut :
https://github.com/rlukassa/Tucil2_13523158

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 sendiri	✓	