Laporan Tugas Kecil 2

Syahrizal Bani Khairan - 13523063

Untuk kuliah IF2211 Strategi Algoritma

5 April 2025

1. Identifikasi Masalah

Permasalahan yang menjadi topik tugas ini adalah kompresi gambar menggunakan metode Quadtree dengan pendekatan divide and conquer. Gambar yang dikompresi terdiri dari sejumlah piksel dengan nilai intensitas warna tertentu. Oleh karena itu, komponen dalam permasalahan ini adalah:

1. **Gambar**: representasi visual yang terdiri dari piksel-piksel membutuhkan struktur data tertentu. Berdasarkan data piksel, gambar akan dibagi menjadi blok yang berwarna mirip dan setiap piksel pada blok akan dinormalisasikan.
2. **Quadtree**: struktur data hierarkis yang digunakan untuk membagi gambar ke dalam bagian-bagian lebih kecil berdasarkan keseragaman piksel. Gambar akan direpresentasikan sebagai pohon yang setiap nodenya dapat memiliki empat anak.
3. **Pengukuran keseragaman/error**: untuk mengukur keseragaman piksel, dapat digunakan berbagai macam metode. Bagian gambar yang memiliki piksel seragam akan memiliki error yang rendah dan dapat dinormalisasikan agar gambar menjadi lebih sederhana.

Pada awalnya, gambar dapat memiliki banyak detail/informasi visual. Ada kemungkinan bahwa sebagian detail tersebut dapat disederhanakan, misalnya area yang memiliki piksel berwarna mirip dapat dijadikan seragam. Tentunya detail dari gambar asli akan hilang dan kualitasnya berkurang, tetapi dengan penyederhanaan seperti ini ukuran file gambar dapat menjadi lebih kecil. Sebuah area yang memiliki warna yang sama dapat dikenali oleh encoder gambar dan dikompresi dengan lebih baik. Pada tugas ini, akan diimplementasikan aplikasi yang menyederhanakan gambar menggunakan Quadtree.

2. Spesifikasi

A. Algoritma Divide and Conquer Quadtree

Pada dasarnya, metode kompresi ini akan mempartisi gambar menjadi sejumlah subgambar dengan piksel berwarna mirip yang kemudian disederhanakan. Dengan menggunakan struktur data Quadtree, gambar dibagi menjadi empat blok setara dan setiap blok tersebut akan dibagi kembali. Langkah detailnya sebagai berikut

1. **Tahap Conquer**. Akan dibentuk Quadtree untuk informasi kompresi gambar. Pada awalnya, seluruh gambar adalah satu blok utuh. Lalu, setiap blok akan dibagi menjadi empat blok yang kemudian akan dibagi kembali:
    1.1. Cek apakah blok memiliki error di bawah ambang batas tertentu. Metode perhitungan error akan dijelaskan selanjutnya. Blok yang memiliki error yang minimal dianggap seragam dan tidak perlu dibagi.
    1.2. Cek apakah blok memiliki ukuran di bawah ambang batas tertentu atau jika blok tersebut jika dibagi memiliki ukuran di bawah ambang batas. Blok yang terlalu kecil tidak perlu dibagi.
    1.3. Jika blok tidak termasuk ke dalam 2 kategori di atas, blok akan dibagi. Blok dibagi menggunakan 2 garis: garis horizontal yang melalui titik tengah tinggi blok dan garis vertikal yang melalui titik tengah lebar blok. Karena piksel diskret, pembagian tidak akan sama persis. Pembagian menghasilkan empat blok baru.
    1.4. Lakukan tahap conquer untuk setiap blok yang baru dibuat, hingga tidak ada blok yang dapat dibagi,
2. **Tahap Merge**. Setelah gambar dibagi sepenuhnya, gambar akan direkonstruksi berdasarkan Quadtree yang dibentuk. Gambar telah dipartisi menjadi blok yang berhubungan dengan simpul daun quadtree. Lakukan traversal Quadtree, dimulai dengan akar Quadtree:
    2.1. Jika simpul bukan merupakan simpul daun, traversal ke setiap simpul anak untuk rekonstruksi gambar.
    2.2. Jika simpul merupakan simpul daun, hitung rata-rata warna pada blok gambar tersebut. Gambar blok pada gambar keluaran dengan warna rata-rata tersebut.

Ada tiga parameter dalam algoritma kompresi ini, yaitu metode pengukuran error, ambang batas error, dan a,bang batas ukuran blok. Nilai parameter akan mempengaruhi hasil gambar kompresi. Terdapat *tradeoff* dalam kompresi;secara kasarnya gambar dapat dikompresi menjadi lebih kecil dengan kualitas gambar yang lebih buruk, atau gambar dapat dikompresi minimal dengan kualitas gambar yang lebih baik.

Dalam implementasinya, Quadtree terdiri dari simpul-simpul Quadtree. Simpul ini menyimpan informasi berupa batas tepi blok gambar, error dalam blok, dan warna rata-rata pada blok. Quadtree tidak terdiri dari bagian-bagian image secara langsung, tetapi terkait dengan gambar asal melalui referensi.

B. Pengukuran Error

Program menyediakan empat pilihan pengukuran error yaitu variance, mean absolute deviation, max pixel difference, dan entropy. Semakin kecil error pada blok, semakin seragam warna piksel pada blok. Metode pengukuran dilakukan sesuai tabel di bawah

Metode	Formula

Variance	$\sigma_c^2 = \frac{1}{N}\sum_{i=1}^{N}(P_{i,c} - \mu_c)^2$		
	$\sigma_{RGB}^2 = \frac{\sigma_R^2 + \sigma_G^2 + \sigma_B^2}{3}$		
	σ_c^2	=	Variansi tiap kanal warna c (R, G, B) dalam satu blok
	$P_{i,c}$	=	Nilai piksel pada posisi *i* untuk kanal warna *c*
	μ_c	=	Nilai rata-rata tiap piksel dalam satu blok
	N	=	Banyaknya piksel dalam satu blok
Mean Absolute Deviation (MAD)	$MAD_c = \frac{1}{N}\sum_{i=1}^{N}\lvert P_{i,c} - \mu_c \rvert$		
	$MAD_{RGB} = \frac{MAD_R + MAD_G + MAD_B}{3}$		
	MAD_c	=	Mean Absolute Deviation tiap kanal warna c (R, G, B) dalam satu blok
	$P_{i,c}$	=	Nilai piksel pada posisi *i* untuk kanal warna *c*
	μ_c	=	Nilai rata-rata tiap piksel dalam satu blok
	N	=	Banyaknya piksel dalam satu blok
Max Pixel Difference	$D_c = max(P_{i,c}) - min(P_{i,c})$		
	$D_{RGB} = \frac{D_R + D_G + D_B}{3}$		
	D_c	=	Selisih antara piksel dengan nilai max dan min tiap kanal warna c (R, G, B) dalam satu blok
	$P_{i,c}$	=	Nilai piksel pada posisi *i* untuk channel warna *c*
Entropy	$H_c = -\sum_{i=1}^{N} P_c(i) log_2(P_c(i))$		
	$H_{RGB} = \frac{H_R + H_G + H_B}{3}$		
	H_c	=	Nilai entropi tiap kanal warna c (R, G, B) dalam satu blok

	$P_c(i)$	=	Probabilitas piksel dengan nilai i dalam satu blok untuk tiap kanal warna c (R, G, B)

C. Input/Output dan Representasi Gambar

Untuk melakukan pemrosesan gambar, akan digunakan *library* berikut:

1. CImg
2. libjpeg
3. libpng
4. zlib

Dalam algoritma kompresi, dibutuhkan representasi data yang memungkinan akses baca dan tulis data nilai RGB per piksel. Akses blok dilakukan untuk bagian gambar dengan ukuran dan batasan tertentu. Dibutuhkan juga fungsionalitas *loading* dan *saving* file gambar. Dengan spesifikasi ini, akan diimplementasikan sebuah wrapper class dari objek CImg.

Objek CImg adalah objek yang dapat menyimpan nilai RGB gambar. Struktur data yang digunakan CImg cukup sederhana, yaitu buffer linear yang menyimpan secara berurut channel warna kemudian lokasi piksel (tidak *interleaved*). Seluruh nilai piksel channel red akan diikuti nilai piksel channel blue dan seterusnya. Objek CImg juga memiliki method manipulasi gambar sederhana. Implementasi wrapper objek ini akan dapat memenuhi kebutuhan algoritma kompresi.

Objek CImg dapat melakukan operasi Input/Output file gambar. Gambar dengan format JPG dan PNG tidak disupport secara native oleh CImg tetapi dapat dilakukan dengan ekstensi *library* libjpeg dan libpng(yang membutuhkan zlib).

3. Implementasi

A. Struktur Data

A.1. Error

Fungsionalitas pengukuran error diimplementasikan dalam satu file header sebagai fungsi generik statik yang menerima masukan iterator. Fungsi generik memungkinkan fleksibilitas struktur data selama bisa mendefinisikan iterator. Hal ini mengurangi beban pengembangan saat menentukan struktur data gambar yang tepat.

error.hpp

```
#ifndef ERROR_HPP
#define ERROR_HPP

#include <map>
```

```cpp
#include <cmath>

enum ErrorMethod {
    VARIANCE = 1,
    MEAN_ABSOLUTE_DEVIATION = 2,
    MAX_PIXEL_DIFFERENCE = 3,
    ENTROPY = 4
};

class ErrorMetrics {
public:
    // Used to calculate the error value of pixels based on various methods
    // Require Iterator object such as the one defined in Image::Iterator
    template <typename Iterator>
    static double calculateChannelError(ErrorMethod method, Iterator begin, Iterator end) {
        if (begin == end) {
            return 0;   // Empty iterator
        }
        switch (method) {
            case VARIANCE:
                return calculateVariance(begin, end);
            case MEAN_ABSOLUTE_DEVIATION:
                return calculateMeanAbsoluteDeviation(begin, end);
            case MAX_PIXEL_DIFFERENCE:
                return calculateMaxPixelDifference(begin, end);
            case ENTROPY:
                return calculateEntropy(begin, end);
            default:
                return 0;
        }
    }

    // Aggregates the error values of each channel into a single value
    static double calculateError(ErrorMethod method, double r, double g, double b) {
        return (r + g + b) / 3;
    }

private:
    /* Error calculation */
    // Pixels should have unsigned char type

    template <typename Iterator>
    static double calculateVariance(Iterator begin, Iterator end) {
        // Variance error
        // By Var(X) = E[(X-E[X])^2]
        double sum = 0, mean = 0;
        int count = 0;

        // Calculate mean first
        for (auto it = begin; it != end; ++it) {
            sum += *it;
            count++;
```

```
        }
        mean = sum / count;

        double variance = 0;
        for (auto it = begin; it != end; ++it) {
            variance += std::pow(*it - mean, 2);
        }
        return variance / count;

    }

    template <typename Iterator>
    static double calculateMeanAbsoluteDeviation(Iterator begin, Iterator end) {
        // Mean Absolute Deviation error
        // By MAD(X) = E[|X - E[X]|]
        double sum = 0, mean = 0;
        int count = 0;

        // Calculate mean first
        for (auto it = begin; it != end; ++it) {
            sum += *it;
            count++;
        }
        mean = sum / count;

        double mad = 0;
        for (auto it = begin; it != end; ++it) {
            mad += std::abs(*it - mean);
        }
        return mad / count;
    }

    template <typename Iterator>
    static double calculateMaxPixelDifference(Iterator begin, Iterator end) {
        // Max Pixel Difference error
        // By MaxDiff(X) = max(X) - min(X)
        typename Iterator::value_type min = *begin, max = *begin;
        for (auto it = begin; it != end; ++it) {
            if (*it < min) min = *it;
            if (*it > max) max = *it;
        }
        return max - min;
    }

    template <typename Iterator>
    static double calculateEntropy(Iterator begin, Iterator end) {
        // Entropy error
        // By H = -Σ p(x) * log2(p(x))

        // Calculate frequency of each pixel value for p(x)
        std::map<typename Iterator::value_type, int> histogram;
        int count = 0;
```

```
        for (auto it = begin; it != end; ++it) {
            histogram[*it]++;
            count++;
        }

        double entropy = 0;
        for (auto& [color, frequency] : histogram) {
            double probability = static_cast<double>(frequency) / count;    // p(x) = frequency /
count
            entropy -= probability * std::log2(probability);
        }

        return entropy;
    }
};

#endif
```

A.2. Image

Class Image adalah wrapper dari objek CImg dari library dengan nama yang sama. Image memiliki operasi input/output file gambar melalui konstruktor dan method save. Nilai piksel channel tertentu dari gambar diakses melalui interface iterator. Karena akses biasanya dilakukan pada blok bagian gambar, didefinisikan method beginBlock dan endBlock untuk mengakses pixel dalam blok tertentu. Kemudian, manipulasi gambar hanyalah berupa pengisian, didefinisikan method paintBlockPixel.

image.hpp

```
#ifndef IMAGE_HPP
#define IMAGE_HPP

#define cimg_display 0      // Disable CImg display methods thus not requiring X11 or gdi
#define cimg_use_jpeg 1     // Use libjpeg for JPEG support
#define cimg_use_png 1      // Use libpng for PNG support
#include "CImg.h"

#include <stdexcept>

typedef unsigned char Quantum;      // Unit of subpixel value

// Channel index e.g. index 0 is used to access the red channel of a pixel
enum Channels {
    RED = 0,
    GREEN = 1,
    BLUE = 2,
    ALPHA = 3
};

class Image {
```

```
private:
    // Image object
    cimg_library::CImg<Quantum> img;
public:
    // Constructors and destructors
    // From file
    Image(std::string address);
    // Image object with given dimensions and color
    Image(int width, int height, Quantum r, Quantum g, Quantum b);
    // Copy constructor
    Image(const Image &other);
    ~Image();

    // Dimension getters
    int getSize() const;
    int getWidth() const;
    int getHeight() const;

    // Pixel setters
    void paintBlockPixel(int rowStart, int colStart, int rowEnd, int colEnd, Quantum r, Quantum
g, Quantum b, bool addBorder);

    // Save the image to a file
    void save(std::string address);

    // For accessing pixel values
    class Iterator {
    private:
        const cimg_library::CImg<Quantum>& img;
        Channels channel;
        int startRow, startCol;
        int endRow, endCol;
        int currentRow, currentCol;
    public:
        // Read only iterator
        Iterator(const cimg_library::CImg<Quantum>& imgref, Channels channel,
            int startRow, int startCol,
            int endRow, int endCol)
            : img(imgref), channel(channel), startRow(startRow), startCol(startCol),
            endRow(endRow), endCol(endCol), currentRow(startRow), currentCol(startCol) { };

        // Iterator traits
        using value_type = Quantum;
        using iterator_category = std::input_iterator_tag;

        // Accessor methods
        // Read-only
        Quantum operator*() const {
            if (currentRow < startRow || currentRow > endRow || currentCol < startCol ||
currentCol > endCol) {
                throw std::out_of_range("Iterator out of bounds.");
            }
```

```
            return img(currentCol, currentRow, 0, channel);
        }

        // Iterator controls
        Iterator& operator++() {
            ++currentCol;
            if (currentCol > endCol) {  // Move to the next row
                currentCol = startCol;
                ++currentRow;
            }
            // end iterator must be the increment of currentRow==endRow and currentCol==endCol to
act as stop condition
            return *this;
        }
        bool operator==(const Iterator &other) const {
            return currentRow == other.currentRow && currentCol == other.currentCol;
        }
        bool operator!=(const Iterator &other) const { return !(*this == other); }
    };

    // Use these methods to iterate over a channel of an image subblock, as to be used with the
error calculation and QuadTree formation
    Iterator beginBlock(int startRow, int startCol, int endRow, int endCol, Channels channel)
const;
    Iterator endBlock(int startRow, int startCol, int endRow, int endCol, Channels channel)
const;
};

#endif
```

image.cpp

```
#include <stdexcept>
#include <string>
#include "image.hpp"

// Constructors and destructors
// From file
Image::Image(std::string address) {
    cimg_library::CImg<Quantum> image(address.c_str());
    if (image.is_empty()) {
        throw std::runtime_error("Image not found or empty.");
    }
    if (image.spectrum() < 3) {
        throw std::runtime_error("Image do not have at least RGB channels.");
    }

    // Image with alpha channel is set to opaque
    if (image.spectrum() == 4) {
        image.channel(Channels::ALPHA).fill(1);
    }
```

```
    this->img = image;

}
// Image object with given dimensions and color
Image::Image(int width, int height, Quantum r, Quantum g, Quantum b) {
    cimg_library::CImg<Quantum> image(width, height, 1, 3, 0);  // width, height, depth, channel
count, pixel initial value
    Quantum pixel[3] = { r, g, b };
    image.draw_rectangle(0, 0, width - 1, height - 1, pixel);

    this->img = image;
}
// Copy constructor
Image::Image(const Image &other) {
    // Deep copy. Does not share buffer
    this->img = cimg_library::CImg<Quantum>(other.img, false);
}

Image::~Image() { }

// Dimension getters
int Image::getSize() const { return img.width() * img.height(); }
int Image::getWidth() const { return img.width(); }
int Image::getHeight() const { return img.height(); }

// Pixel setters
void Image::paintBlockPixel(int rowStart, int colStart, int rowEnd, int colEnd, Quantum r,
Quantum g, Quantum b, bool addBorder) {
    // Check if the coordinates are within the image bounds
    if (rowStart < 0 || colStart < 0 || rowEnd >= img.height() || colEnd >= img.width()) {
        throw std::out_of_range("Coordinates are out of bounds.");
    }

    // Set the pixel values in the specified block
    Quantum pixel[3] = { r, g, b };
    img.draw_rectangle(colStart, rowStart, colEnd, rowEnd, pixel);

    // Add border if requested
    if (addBorder) {
        // Thin (1 pixel) black border on top and right side of the block
        // Does not paint border on image border
        Quantum borderPixel[3] = { 0, 0, 0 }; // Black border
        if (rowStart != 0) {
            img.draw_rectangle(colStart, rowStart, colEnd, rowStart, borderPixel); // Top border
        }
        if (colEnd != img.width() - 1) {
            img.draw_rectangle(colEnd, rowStart, colEnd, rowEnd, borderPixel); // Right border
        }
    }
}

// Save the image to a file
```

```
void Image::save(std::string address) {
    // Save the image to the specified address
    img.save(address.c_str());
}


/* Iterator */
// Modifiable iterator
Image::Iterator Image::beginBlock(int startRow, int startCol, int endRow, int endCol, Channels
channel) const {
    return Image::Iterator(img, channel, startRow, startCol, endRow, endCol);
}
Image::Iterator Image::endBlock(int startRow, int startCol, int endRow, int endCol, Channels
channel) const {
    return Image::Iterator(img, channel, endRow+1, startCol, endRow+1, startCol);
}
```

A.3. QuadTree

Quadtree diimplementasi menggunakan dua kelas yaitu `QuadTree` dan `QuadTreeNode`. `QuadTree` menyimpan pointer ke akar quadtree, data mengenai pohon secara keseluruhan, dan referensi ke input objek Image untuk perhitungan warna dan error. Method untuk melakukan divide and conquer berada di kelas ini. Sementara itu, `QuadTreeNode` berkaitan erat dengan blok pada gambar seperti batas tepi, error, dan warna rata-rata. `QuadTreeNode` menyimpan pointer ke simpul anak yang maksimum sebanyak empat.

Saat `QuadTree` dibentuk, pohon hanya terdiri dari satu simpul yaitu akar. Dengan memanggil method `divide`, pohon akan mencoba membagi setiap simpul daun pada pohon sesuai kriteria yang dijelaskan sebelumnya. Method `divideExhaust` akan membagi pohon hingga tidak ada lagi simpul daun yang bisa dibagi. Error dari block akan dihitung tepat setelah simpul terkait dibentuk. Rekonstruksi gambar dilakukan dengan memanggil method `merge` dari `QuadTree`. Method tersebut akan menghitung warna rerata dan kemudian menggabungkan seluruh simpul daun untuk membentuk objek Image baru yang telah dikompresi.

`quadtree.hpp`

```
#ifndef QUADTREE_HPP
#define QUADTREE_HPP

#include <array>
#include <memory>
#include "image.hpp"
#include "error.hpp"

#define QUADTREE_MAX_DEPTH 50

class QuadTreeNode {
```

```
public:
    // Children
    std::array<std::unique_ptr<QuadTreeNode>, 4> children;

    // Node data
    double averageR, averageG, averageB;
    double error;

    // Block boundary
    int rowStart, colStart;
    int rowEnd, colEnd;

    bool isDivisible;   // Default to true
    bool isLeaf;        // Default to true

    // Constructor
    QuadTreeNode();
    QuadTreeNode(int rowStart, int colStart, int rowEnd, int colEnd);
    ~QuadTreeNode() {};

    // Dimension getter
    int getWidth() const { return colEnd - colStart + 1; }
    int getHeight() const { return rowEnd - rowStart + 1; }
    int getArea() const { return getWidth() * getHeight(); }

    // Error calculation that set the error attribute
    void calculateError(const Image& image, ErrorMethod errorMethod);

    // Average calculation that set the averageR, averageG, and averageB attributes
    void calculateAverage(const Image& image);
};

class QuadTree {
private:
    // Root node
    std::unique_ptr<QuadTreeNode> root;

    // Image to be compressed
    const Image& image;

    // Tree information
    int nodeCount;  // Root, leaves and internal nodes
    int treeDepth;  // Incremented with each divide call
    int depthOnLastColorCalc; // Depth of the last average color calculation

    // Compression parameters
    int minBlockArea;
    double errorThreshold;
    ErrorMethod errorMethod;

    // Calculate all node's average color
    void calculateAverageColor() const;
```

```
    // Divide nodes
    int divideNode(QuadTreeNode& node);

    // Merge nodes up to variable depth. Calculate average RGB value from each leaf node
    void mergeNodeDepth(QuadTreeNode& node, Image& outputImage, int depth, bool addBorder) const;

    // Merge nodes on variable error threshold
    void mergeNodeThreshold(QuadTreeNode& node, Image& outputImage, double errorThreshold, bool
addBorder) const;

public:
    // Constructor and destructor
    QuadTree(const Image& image, int minBlockSize, double errorThreshold, ErrorMethod
errorMethod);
    ~QuadTree();

    // Getters
    int getNodeCount() const;
    int getTreeDepth() const;

    // Divide all current divisible leaf nodes per level
    int divide();

    // Divide until exhaustion
    void divideExhaust();

    // Merge the current tree into an Image
    Image merge(int depth=-1, bool addBorder=false) const;

    // Merge with variable error threshold. Unused.
    Image mergeThreshold(double errorThreshold, bool addBorder=false) const;

};

#endif
```

quadtree.cpp

```
#include "quadtree.hpp"
#include "image.hpp"

/* QuadTreeNode */
QuadTreeNode::QuadTreeNode()
    : rowStart(0), colStart(0), rowEnd(0), colEnd(0),
    averageR(0), averageG(0), averageB(0), error(0), isDivisible(true), isLeaf(true) {
    for (int i = 0; i < 4; i++) {
        children[i] = nullptr;
    }
}

QuadTreeNode::QuadTreeNode(int rowStart, int colStart, int rowEnd, int colEnd)
```

```
    : rowStart(rowStart), colStart(colStart), rowEnd(rowEnd), colEnd(colEnd),
    averageR(0), averageG(0), averageB(0), error(0), isDivisible(true), isLeaf(true) {
    for (int i = 0; i < 4; i++) {
        children[i] = nullptr;
    }
}

// Error calculation that set the error attribute
void QuadTreeNode::calculateError(const Image& image, ErrorMethod errorMethod){
    if (rowStart < 0 || colStart < 0 || rowEnd >= image.getHeight() || colEnd >=
image.getWidth()) {
        throw std::out_of_range("Block dimensions are out of bounds.");
    }

    double errorR, errorG, errorB;

    errorR = ErrorMetrics::calculateChannelError(errorMethod,
        image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::RED),
        image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::RED));
    errorG = ErrorMetrics::calculateChannelError(errorMethod,
        image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::GREEN),
        image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::GREEN));
    errorB = ErrorMetrics::calculateChannelError(errorMethod,
        image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::BLUE),
        image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::BLUE));

    error = ErrorMetrics::calculateError(errorMethod, errorR, errorG, errorB);
}

// Average calculation that set the averageR, averageG, and averageB attributes
void QuadTreeNode::calculateAverage(const Image& image){
    averageR = 0;
    averageG = 0;
    averageB = 0;
    int count = getArea();
    if (isLeaf) {
        // Each channel iterated separately to optimize cache hit due to CImg data structure
        for (auto it = image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::RED);
            it != image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::RED); ++it) {
            averageR += *it;
        }
        for (auto it = image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::GREEN);
        it != image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::GREEN); ++it) {
            averageG += *it;
        }
        for (auto it = image.beginBlock(rowStart, colStart, rowEnd, colEnd, Channels::BLUE);
        it != image.endBlock(rowStart, colStart, rowEnd, colEnd, Channels::BLUE); ++it) {
            averageB += *it;
        }

        averageR /= count;
        averageG /= count;
```

```
        averageB /= count;
    } else {
        // Calculate average by way of weighted average of children
        for (int i = 0; i < 4; i++) {
            if (children[i] == nullptr) {
                throw std::runtime_error("Child node is null.");
            }
            children[i]->calculateAverage(image);
            averageR += children[i]->averageR * children[i]->getArea();
            averageG += children[i]->averageG * children[i]->getArea();
            averageB += children[i]->averageB * children[i]->getArea();
        }
        averageR /= count;
        averageG /= count;
        averageB /= count;
    }

}



/* ------------------------------------------------*/
/* QuadTree */

// Constructor and destructor
QuadTree::QuadTree(const Image& image, int minBlockArea, double errorThreshold, ErrorMethod
errorMethod)
    : image(image), nodeCount(1), treeDepth(1), depthOnLastColorCalc(0),
    minBlockArea(minBlockArea), errorThreshold(errorThreshold), errorMethod(errorMethod) {
    root = std::make_unique<QuadTreeNode>(0, 0, image.getHeight()-1, image.getWidth()-1);
    root->calculateError(image, errorMethod);
}
QuadTree::~QuadTree() {}

// Getters
int QuadTree::getNodeCount() const { return nodeCount; }
int QuadTree::getTreeDepth() const { return treeDepth; }

// Calculate average color of all nodes
void QuadTree::calculateAverageColor() const {
    if (root == nullptr) {
        throw std::runtime_error("Root node is null.");
    }
    if (depthOnLastColorCalc == treeDepth) {
        // Average color already calculated
        return;
    }
    // Recursively calculate average color for each node
    root->calculateAverage(image);
}

/* Divide and conquer */
```

```
// Divide nodes per level
// Returns the number of nodes divided
int QuadTree::divideNode(QuadTreeNode& node) {
    int count = 0;
    if (!node.isLeaf) {
        // Case 1: Inner node
        // Count how many nodes are created on this subtree
        for (int i = 0; i < 4; i++) {
            if (node.children[i] == nullptr) {
                throw std::runtime_error("Child node is null.");
            }
            count += divideNode(*node.children[i]);
        }
    } else if (!node.isDivisible) {
        // Case 2: Node has been checked to be indivisible
        // Node is ignored
        count = 0;
    } else {
        // Case 3: Leaf node
        // Check if it is divisible
        if (node.getArea() <= minBlockArea) {
            // The node is not larger than the minimum block size
            node.isDivisible = false;
            count = 0;
        } else if ((node.colEnd-node.colStart) * (node.rowEnd-node.rowStart) / 4 < minBlockArea)
{
            // If divided, the node will be smaller than the minimum block size
            node.isDivisible = false;
            count = 0;
        } else if (node.error <= errorThreshold) {
            // The node is below the error threshold/the pixels are similar
            node.isDivisible = false;
            count = 0;
        } else {
            // Divide the node
            int rowMid = (node.rowStart + node.rowEnd) / 2;
            int colMid = (node.colStart + node.colEnd) / 2;

            // Create children
            // Divided into these 4 panels in order:
            // 0 1
            // 2 3
            node.isLeaf = false;
            node.children[0] = std::make_unique<QuadTreeNode>(node.rowStart, node.colStart,
rowMid, colMid);
            node.children[1] = std::make_unique<QuadTreeNode>(node.rowStart, colMid+1, rowMid,
node.colEnd);
            node.children[2] = std::make_unique<QuadTreeNode>(rowMid+1, node.colStart,
node.rowEnd, colMid);
            node.children[3] = std::make_unique<QuadTreeNode>(rowMid+1, colMid+1, node.rowEnd,
node.colEnd);
```

```
            for (int i = 0; i < 4; i++) {
                if (node.children[i] == nullptr) {
                    throw std::runtime_error("Child node is null.");
                }
                // Calculate error for each child node
                node.children[i]->calculateError(image, errorMethod);
            }
            count = 4;
        }

    }
    return count;
}

// Merge nodes. Calculate average RGB value from each leaf node
void QuadTree::mergeNodeDepth(QuadTreeNode& node, Image& outputImage, int depth, bool addBorder)
const {
    // depth == 0   : Do nothing
    // depth == 1   : Fill the block with the average color
    // depth > 1    : Merge children nodes if exist
    // depth == -1  : Merge all leaf nodes

    // node.isLeaf is only set to false when children of a node are created

    if (depth == 0) {
        // Do nothing
    } else if (depth == 1 || node.isLeaf) {
        // Fill the block with the average color
        // Average color should already be calculated
        outputImage.paintBlockPixel(node.rowStart, node.colStart, node.rowEnd, node.colEnd,
            node.averageR, node.averageG, node.averageB, addBorder);
    } else if (depth > 1 && !node.isLeaf) {
        // Merge nodes up to a certain depth which may not be leaf nodes
        // Or merge all leaf nodes
        for (int i = 0; i < 4; i++) {
            mergeNodeDepth(*node.children[i], outputImage, depth-1, addBorder);
        }
    } else if (depth == -1 && !node.isLeaf){
        for (int i = 0; i < 4; i++) {
            mergeNodeDepth(*node.children[i], outputImage, depth, addBorder);
        }
    }
}

// Merge nodes on variable error threshold
void QuadTree::mergeNodeThreshold(QuadTreeNode& node, Image& outputImage, double errorThreshold,
bool addBorder) const {
    // Blocks that already have low error is immediately merged even if it has children
    if (node.error < errorThreshold || node.isLeaf) {
        // Fill the block with the average color
        // Average color should already be calculated
```

```
        outputImage.paintBlockPixel(node.rowStart, node.colStart, node.rowEnd, node.colEnd,
            node.averageR, node.averageG, node.averageB, addBorder);
    } else if (!node.isLeaf) {
        // Merge children nodes
        for (int i = 0; i < 4; i++) {
            mergeNodeThreshold(*node.children[i], outputImage, errorThreshold, addBorder);
        }
    }
}

// Divide all current divisible leaf nodes per level
int QuadTree::divide() {
    int count = divideNode(*root);
    nodeCount += count;
    if (count > 0) { treeDepth++; }
    return count;
}

// Divide until exhaustion
void QuadTree::divideExhaust() {
    // Divide until no more nodes can be divided
    int count;
    do {
        count = divide();
    } while (count > 0 && treeDepth < QUADTREE_MAX_DEPTH);
}

// Merge the current tree into an Image up to a certain depth
Image QuadTree::merge(int depth, bool addBorder) const {
    // Create a copy of the original image
    Image outputImage = image;
    if (depth < -1) {
        throw std::invalid_argument("Depth must be greater than or equal to -1.");
    }
    if (depth > treeDepth) {
        throw std::invalid_argument("Depth must be less than or equal to the current tree
depth.");
    }

    calculateAverageColor(); // Calculate average color for each node
    mergeNodeDepth(*root, outputImage, depth, addBorder);
    return outputImage;
}

// Merge with variable error threshold
Image QuadTree::mergeThreshold(double errorThreshold, bool addBorder) const {
    // Create a copy of the original image
    Image outputImage = image;
    if (errorThreshold < 0) {
        throw std::invalid_argument("Error threshold must be greater than or equal to 0.");
    }
```

```
    calculateAverageColor(); // Calculate average color for each node
    mergeNodeThreshold(*root, outputImage, errorThreshold, addBorder);
    return outputImage;
}
```

B. Program Utama

main.cpp

```
#include <iostream>
#include <string>
#include <memory>
#include <chrono>
#include <filesystem>

#include "image.hpp"
#include "error.hpp"
#include "quadtree.hpp"

struct CompressionConfig {
    std::string inputImageAddress="";       // Image to be compressed
    std::string outputImageAddress="";      // Compressed image output address
    double errorThreshold=0.0;              // Error threshold for block division
    double compressionTarget=0.0;           // Compression percentage target (not implemented yet)
    int minBlockArea=1;                     // Minimum block size for block division (width,
height)
    ErrorMethod errorMethod=VARIANCE;       // Error calculation method to be used
};

void compress(std::unique_ptr<Image>& inputImage, std::unique_ptr<Image>& outputImage,
std::unique_ptr<QuadTree>& tree, const CompressionConfig& config) {
    tree = std::make_unique<QuadTree>(*inputImage, config.minBlockArea, config.errorThreshold,
config.errorMethod);
    tree->divideExhaust(); // Divide the image into blocks
    outputImage = std::make_unique<Image>(tree->merge(-1));
}

long long calculateFileSize(const std::string& filePath) {
    auto fileSize = std::filesystem::file_size(filePath);
    return std::filesystem::file_size(filePath);
}

int main(){
    /* INPUT */
    CompressionConfig config;

    std::cout << "Enter the input image address: ";
    std::getline(std::cin, config.inputImageAddress);
```

```
    std::cout << "Error method (enter the number)" << std::endl
              << "1. Variance | "
              << "2. Mean Absolute Deviation (MAD) | "
              << "3. Max Pixel Difference | "
              << "4. Entropy" << std::endl;

    int errorMethodInput;
    std::cin >> errorMethodInput;
    if (errorMethodInput < 1 || errorMethodInput > 4) {
        std::cerr << "Invalid error method." << std::endl;
        return 1;
    }
    config.errorMethod = static_cast<ErrorMethod>(errorMethodInput);

    std::cout << "Enter error threshold: ";
    std::cin >> config.errorThreshold;
    if (config.errorThreshold < 0) {
        std::cerr << "Invalid error threshold." << std::endl;
        return 1;
    }

    std::cout << "Enter minimum block area: ";
    std::cin >> config.minBlockArea;
    if (config.minBlockArea < 1) {
        std::cerr << "Invalid minimum block area." << std::endl;
        return 1;
    }

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear the input buffer
    std::cout << "Enter the output image address: ";
    std::getline(std::cin, config.outputImageAddress);

    /* PROCESS */
    std::unique_ptr<Image> inputImage;
    std::unique_ptr<Image> outputImage;
    std::unique_ptr<QuadTree> tree;
    try {
        inputImage = std::make_unique<Image>(config.inputImageAddress);
    } catch (const std::exception& e) {
        std::cerr << "Error loading image: " << e.what() << std::endl;
        return 1;
    }

    auto t1 = std::chrono::high_resolution_clock::now();

    compress(inputImage, outputImage, tree, config);

    auto t2 = std::chrono::high_resolution_clock::now();

    /* OUTPUT */
    long long filesizeBefore = calculateFileSize(config.inputImageAddress);
```

```cpp
    try {
        outputImage->save(config.outputImageAddress); // Save the compressed/merged image
    } catch (const std::exception& e) {
        std::cerr << "Error saving image: " << e.what() << std::endl;
        return 1;
    }
    long long filesizeAfter = calculateFileSize(config.outputImageAddress);
    double compressionPercentage = 100 * (1 - ((double)filesizeAfter / (double)filesizeBefore));
    auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1);

    std::cout << "------------------------------------------------------------" << std::endl;
    std::cout << "Compression excecution time: " << ms.count() << "ms" << std::endl;
    std::cout << "Image size before compression: " << filesizeBefore << " bytes" << std::endl;
    std::cout << "Image size after compression: " << filesizeAfter << " bytes" << std::endl;
    std::cout << "Compression percentage: " << compressionPercentage << "%" << std::endl;
    std::cout << "Tree depth: " << tree->getTreeDepth() << std::endl;
    std::cout << "Number of nodes: " << tree->getNodeCount() << std::endl;

    return 0;
}
```

4. Hasil Uji

Pengujian dilakukan menggunakan 8 gambar, 4 format jpg dan 4 format png. Untuk setiap format akan digunakan 4 metode pengukuran error berbeda. Gambar uji dan hasilnya juga berada di repository pada direktori test.

4.1. Format PNG

4.1.1. test1.png

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test1.png
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
1
Enter error threshold: 0.8
Enter minimum block area: 8
Enter the output image address: test/result1.png
------------------------------------------------------------
Compression excecution time: 3987ms
Image size before compression: 1683834 bytes
Image size after compression: 222080 bytes
Compression percentage: 86.8111%
Tree depth: 9
Number of nodes: 79353
```


Come Volunteers for
G-13
TG-01
PERFECTION!!
VANDELAY TECHNOLOGIES
"A Better Life"
VANDELAY
VANDELAY
VANDELAY


VANDELAY
VANDELAY

4.1.2. test2.png

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test2.png
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
2
Enter error threshold: 1.5
Enter minimum block area: 8
Enter the output image address: test/result2.png
-----------------------------------------------------------
Compression excecution time: 2738ms
Image size before compression: 4472173 bytes
Image size after compression: 642502 bytes
Compression percentage: 85.6333%
Tree depth: 10
Number of nodes: 332061
```

J. Nugent Fitch del et lith.

PHALÆNOPSIS STUARTIANA NOBILIS.

J. Nugent Fitch imp.

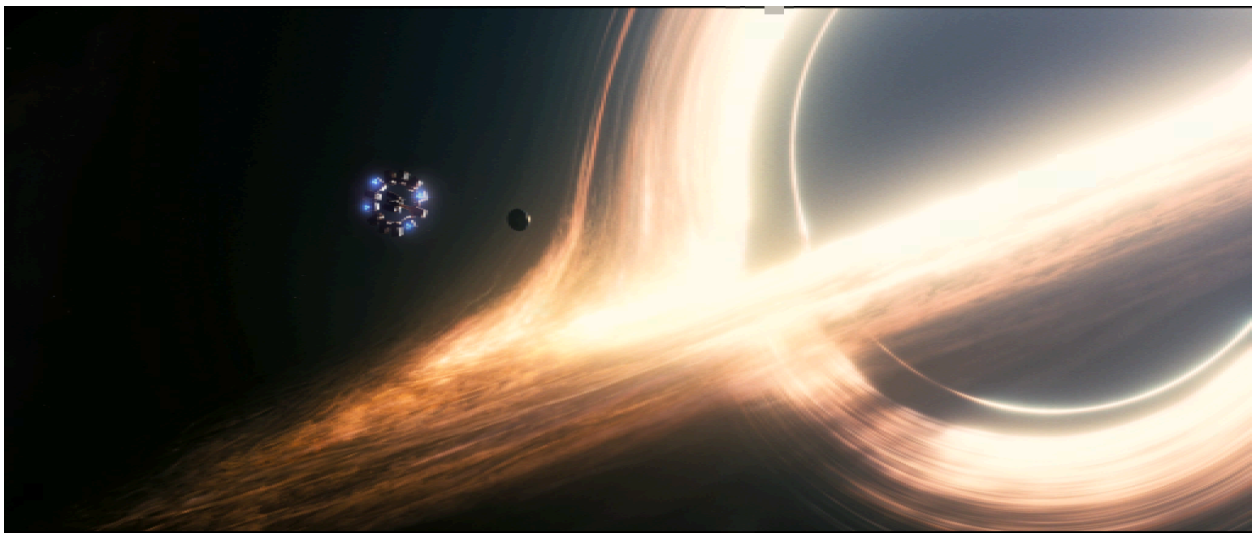
PHALÆNOPSIS STUARTIANA NOBILIS

4.1.3. test3.png

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test3.png
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
3
Enter error threshold: 5
Enter minimum block area: 4
Enter the output image address: test/result3.png
------------------------------------------------------------
Compression excecution time: 1655ms
Image size before compression: 645103 bytes
Image size after compression: 128691 bytes
Compression percentage: 80.0511%
Tree depth: 10
Number of nodes: 40981
```

4.1.4. test4.png

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test4.png
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
4
Enter error threshold: 1
Enter minimum block area: 4
Enter the output image address: test/result4.png
-----------------------------------------------------------
Compression excecution time: 9126ms
Image size before compression: 2502636 bytes
Image size after compression: 458315 bytes
Compression percentage: 81.6867%
Tree depth: 10
Number of nodes: 284905
```



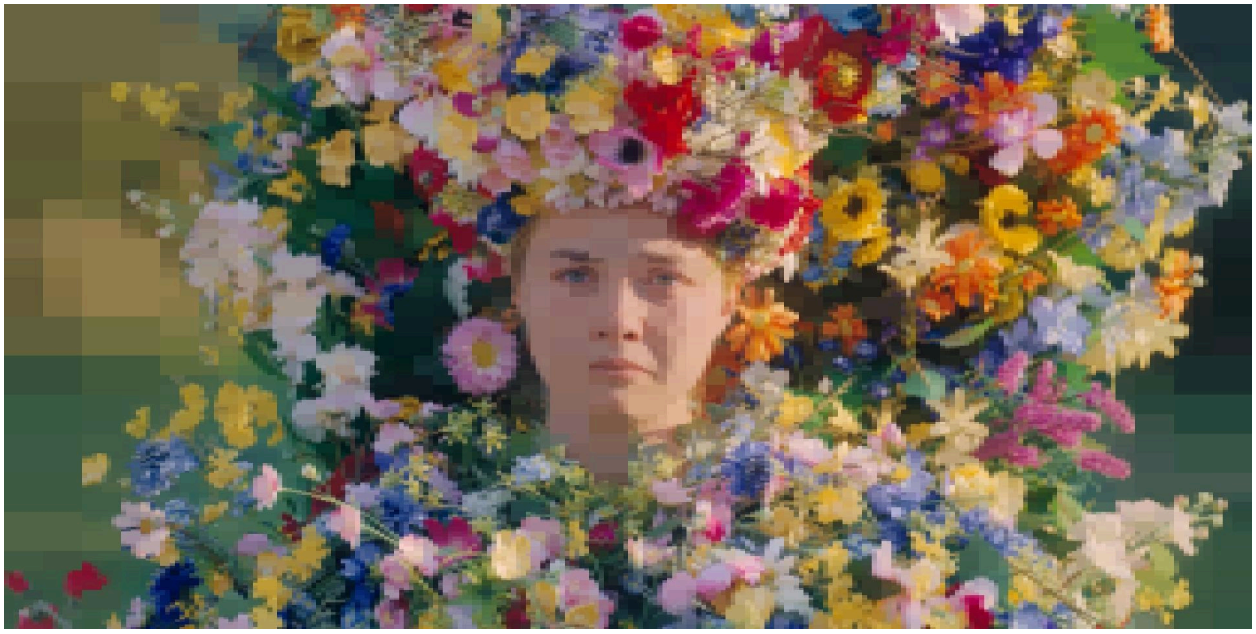
4.2. Format JPG

4.2.1. test5.jpg

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test5.jpg
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
1
Enter error threshold: 2
Enter minimum block area: 4
Enter the output image address: test/result5.jpg
------------------------------------------------------------
Compression excecution time: 3617ms
Image size before compression: 1067073 bytes
Image size after compression: 457514 bytes
Compression percentage: 57.1244%
Tree depth: 9
Number of nodes: 68741
```

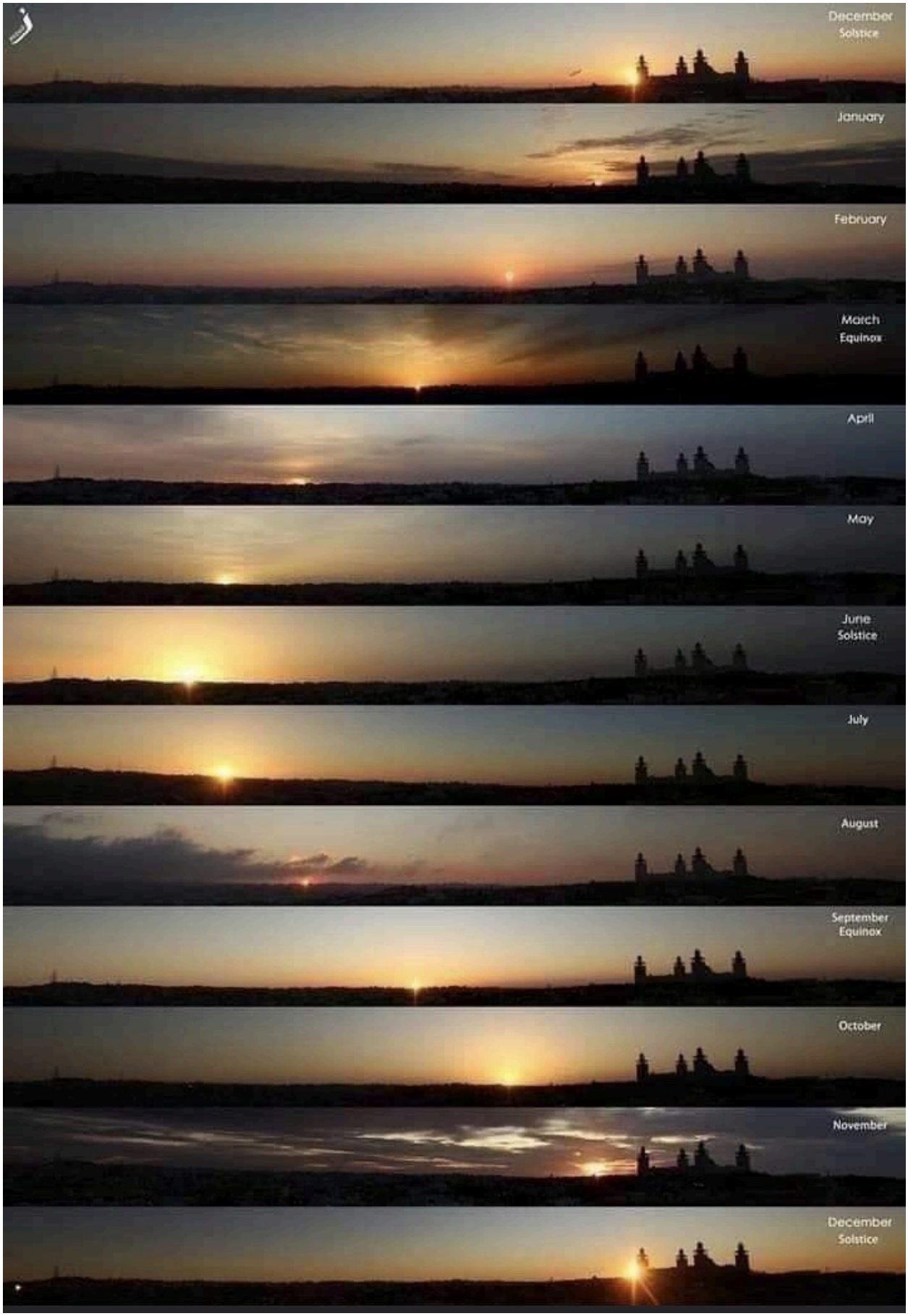
4.2.2. test6.jpg

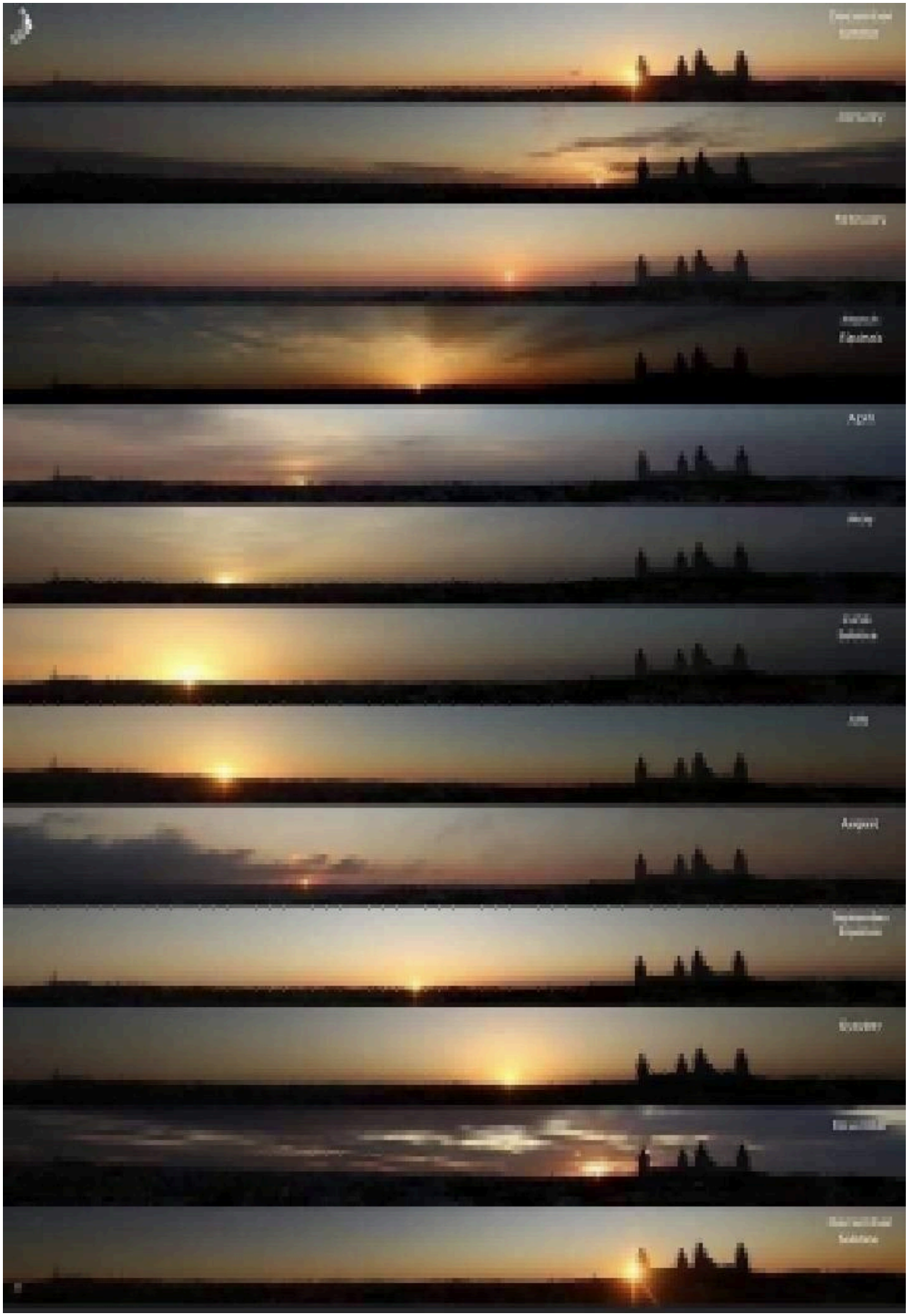
```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test6.jpg
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
2
Enter error threshold: 8
Enter minimum block area: 16
Enter the output image address: test/result6.jpg
------------------------------------------------------------
Compression excecution time: 2868ms
Image size before compression: 217438 bytes
Image size after compression: 1107910 bytes
Compression percentage: -409.529%
Tree depth: 9
Number of nodes: 53849
```



4.2.3. test7.jpg

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test7.jpg
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
3
Enter error threshold: 5
Enter minimum block area: 2
Enter the output image address: test/result7.jpg
------------------------------------------------------------
Compression excecution time: 1230ms
Image size before compression: 61222 bytes
Image size after compression: 322515 bytes
Compression percentage: -426.796%
Tree depth: 10
Number of nodes: 92485
```


December
Solstice
January
February
March
Equinox
April
May
June
Solstice
July
August
September
Equinox
October
November
December
Solstice



4.2.4. test8.jpg

```
PS E:\Project\Kuliah\IF2211 Strategi Algoritma\Tucil2_13523063> make run
./bin/main
Enter the input image address: test/test8.jpg
Error method (enter the number)
1. Variance | 2. Mean Absolute Deviation (MAD) | 3. Max Pixel Difference | 4. Entropy
4
Enter error threshold: 2
Enter minimum block area: 4
Enter the output image address: test/result8.jpg
------------------------------------------------------------
Compression excecution time: 5040ms
Image size before compression: 89075 bytes
Image size after compression: 250033 bytes
Compression percentage: -180.699%
Tree depth: 9
Number of nodes: 40053
```

5. Analisis

- Untuk analisis kompleksitas, operasi yang diperhitungkan adalah akses baca dan tulis pixel yang setiap pixelnya memiliki 3 channel. Dalam algoritma divide and conquer, sebuah persoalan dibagi menjadi sejumlah upa-persoalan yang kemudian solusi dari upa-persoalan tersebut digabungkan. Gambar akan dibagi menjadi blok yang kemudian akan dibagi kembali yang pada akhirnya akan digabungkan kembali.
    - Misal suatu blok gambar memiliki n pixel. Blok ini akan dibagi menjadi empat blok yang memiliki n/4 pixel (sebagai aproksimasi).
    - Di persoalan ini, tiga jenis operasi utama akan dipertimbangkan: perhitungan error, perhitungan warna normal, dan normalisasi/rekonstruksi gambar. Perhitungan error maksimal melakukan 3n akses pixel, begitu pun operasi lain.
    - Dalam pembagian upa-persoalan, terdapat dua batasan pembagian. Untuk penyederhanaan, anggap pembagian hanya menganggap kondisi ukuran blok terkecil. Perhitungan error tetap dilakukan, hanya saja tidak menganggap kondisi ambang batas, seakan tidak bisa diraih. Ini merupakan kasus dimana tree dibagi menjadi full.
    - Maka

$$T(n) = \begin{cases} 3n & \text{if } n < n_0 \\ 4T\left(\frac{n}{4}\right) + 3n + n & \text{if } n > n_0 \end{cases}$$

Ini adalah kasus kedua teorema master dimana $4n = \Theta(n^{log_4 4}) = \Theta(n)$. Dapat disimpulkan kompleksitas algoritma ini adalah

$$T(n) = n\,log(n)$$

6. Lampiran

Link spesifikasi tugas: Spesifikasi Tugas Kecil 2 Stima 2024/2025

Link repository: https://github.com/rizalkhairan/Tucil2_13523063

Checklist pengerjaan tugas:

-

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 (kelompok) sendiri	v	