Laporan Tugas Kecil 1 IF2211 Strategi Algoritma

Semester II Tahun Akademik 2024/2025

Kompresi Citra Dengan Metode QuadTree

Disusun Oleh :

Muhammad Jibril Ibrahim 13523085

Muhammad Luqman Hakim 13523044

Sekolah Teknik Elektro dan Informatika

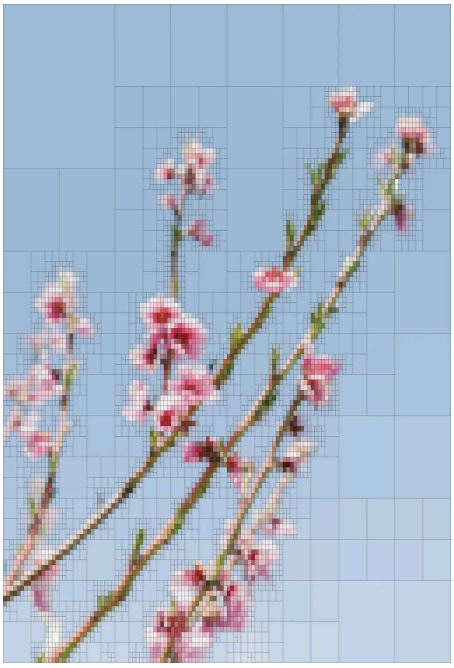
Institut Teknologi Bandung

2025

DAFTAR ISI

BAB 1: PENDAHULUAN

1.1. Deskripsi Tugas



Gambar 1. QuadTree dalam kompresi citra

(Sumber: https://medium.com/@tannerwyork/quadtrees-for-image-processing-302536c95c00)

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

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

BAB 2: PENYELESAIAN & ALGORITMA

2.1. Algoritma Divide & Conquer

Kompresi Citra ini dilakukan dengan langkah-langkah berikut:

1. Hitung variansi sebuah blok (berbentuk persegi panjang) pada gambar, pada mulanya blok ini mencakup seluruh gambar dengan fungsi variansi yang dispesifikasikan.
2. Jika nilainya lebih besar dari maksimum dan blok masih bisa dibagi menjadi 4 bagian yang >= ukuran blok minimum, blok tersebut dibagi menjadi 4 bagian sama besar, ulangi tahap satu untuk setiap 4 blok yang lebih kecil tersebut
3. Jika blok memiliki nilai variansi yang cukup kecil atau blok tidak bisa dibagi lagi, setiap pixel pada blok tersebut diwarnai dengan pixel rata-rata dari blok tersebut.
4. Lanjutkan hingga semua blok gambar output sudah diwarnai.

2.2. Pseudocode

```
PROGRAM QUAD TREE COMPRESSION

KAMUS

type QTreeElmt:

  <row_start: int,

   row_end: int,

   col_start: int,

   col_end: int,

   pixel: Pixel,

   children: array[0..3] of pointer to QTreeElmt>

type QTree: pointer to QTreeElmt

type VarianceFunction: function(Image, QTree) -> real

type Image <...>

type Pixel <...>


procedure compress(Image img_input, Image img_output, real err_threshold,

                   integer min_area, QTree qt, VarianceFunction varfn)
```

```
procedure fill(Image img_output,QTree qt)

function average_pixel(Image img_input, QTree qt)

img_input, img_output : Image

min_area : integer

err_threshold : real

varfn : VarianceFunction


ALGORITMA PROGRAM UTAMA

input(img_input)

input(min_area)

input(err_threshold)

input(varfn)

compress(img_input, img_output, min_area, err_threshold, varfn)


Procedure compress(Image img_input, Image img_output, real err_threshold,
integer min_area, QTree qt, VarianceFunction varfn):

KAMUS LOKAL

  var : real

  c : QTree

ALGORITMA

  var <- varfn(img_input, qt) {{ hitung variansi dari satu blok }}

  if (var >= error_threshold and

      area(qt) div 4 >= min_area) then {{ kondisi blok harus dibagi 4 }}

    qt.subdivide()

    c traversal qt.children:

      compress(img_input, img_ouput, error_threshold, min_area, c, varfn)

  else {{ isi warna pixel rata-rata dari blok }}

    qt.pixel <- average_pixel(img_input, qt)

    fill(img_output, qt)
```

BAB 3: IMPLEMENTASI ALGORITMA

3.1. Struktur Repository

```
quadtree
├── assets
│   └── flower.gif
├── bin
│   └── app
├── docs
│   └── Tucil2_13523044_13523085
├── src
│   ├── cli.c
│   ├── cli.h
│   ├── compression.c
│   ├── compression.h
│   ├── file_size.c
│   ├── file_size.h
│   ├── main.c
│   ├── qtree.c
│   ├── qtree.h
│   ├── variance.c
│   └── variance.h
├── test
│   ├── tc
│   │   ├── usan.png
│   │   └── tree.jpg
│   └── solusi
│       ├── tc1.png
│       ├── tc1.gif
│       ├── tc2.png
│       ├── tc2.gif
│       ├── tc3.png
│       ├── tc3.gif
│       ├── tc4.png
│       ├── tc4.gif
│       ├── tc5.png
│       ├── tc5.gif
│       ├── tc6.jpeg
│       ├── tc6.gif
│       ├── tc7.jpeg
│       ├── tc7.gif
```

```
├── .gitignore
├── makefile
└── README.md
```

3.2. Source Code

1. Struktur Data dan Algoritma Quadtree

```c
typedef struct _QTreeNode{
    /** node's row */
    int row;
    /** node's column */
    int col;
    /** node's width */
    int width;
    /** node's height */
    int height;
    /** content of the quadtree */
    struct _data {
        /** Internal node's first child */
        struct _QTreeNode *first_child;
        /** leaf node's RGB value */
        RGB rgb;
    } data;
    /** type of the quadtree, its' value is either INTERNAL_NODE or LEAF_NODE */
    char node_type;
} *QTreeNode;
```

```c
#include "qtree.h"
#include <assert.h>
#include <stdint.h>
#include <stdlib.h>

#define max(a, b) ((a) > (b) ? (a) : (b))

int qtree_depth_helper(QTreeNode first_child){
    int depth = 1;
    if (NODE(first_child).node_type == INTERNAL_NODE){
        int child_depth = qtree_depth_helper(first_child: NODE(first_child).data.first_child);
        depth = max(depth, child_depth+1);
    }
    if (NODE(first_child+1).node_type == INTERNAL_NODE){
        int child_depth = qtree_depth_helper(first_child: NODE(first_child+1).data.first_child);
        depth = max(depth, child_depth+1);
    }
    if (NODE(first_child+2).node_type == INTERNAL_NODE){
        int child_depth = qtree_depth_helper(first_child: NODE(first_child+2).data.first_child);
        depth = max(depth, child_depth+1);
    }
    if (NODE(first_child+3).node_type == INTERNAL_NODE){
        int child_depth = qtree_depth_helper(first_child: NODE(first_child+3).data.first_child);
        depth = max(depth, child_depth+1);
    }
    return depth;
}

int qtree_depth(QTreeNode tree){
    if (NODE(tree).node_type == LEAF_NODE){
        return 0;
    }
    return qtree_depth_helper(first_child: NODE(tree).data.first_child);
}
```

```c
int qtree_n_nodes_helper(QTreeNode first_child){
    int n = 4;
    if (NODE(first_child).node_type == INTERNAL_NODE){
        n += qtree_n_nodes_helper(first_child: NODE(first_child).data.first_child);
    }
    if (NODE(first_child + 1).node_type == INTERNAL_NODE){
        n += qtree_n_nodes_helper(first_child: NODE(first_child + 1).data.first_child);
    }
    if (NODE(first_child + 2).node_type == INTERNAL_NODE){
        n += qtree_n_nodes_helper(first_child: NODE(first_child + 2).data.first_child);
    }
    if (NODE(first_child + 3).node_type == INTERNAL_NODE){
        n += qtree_n_nodes_helper(first_child: NODE(first_child + 3).data.first_child);
    }
    return n;
}

int qtree_n_nodes(QTreeNode tree){
    if (tree == NULL){
        return 0;
    } else if (NODE(tree).node_type == LEAF_NODE){
        return 1;
    }
    return 1 + qtree_n_nodes_helper(first_child: NODE(tree).data.first_child);
}

QTreeNode qtree_new(int width, int height) {
    QTreeNode res = malloc(size: sizeof(struct _QTreeNode));
    res->width = width;
    res->height = height;
    res->row = res->col = 0;
    res->node_type = LEAF_NODE;
    res->data.rgb.r = res->data.rgb.g = res->data.rgb.b = 0;
```

```c
}

QTreeNode qtree_subdivide(QTreeNode node) {
    QTreeNode next = malloc(size: sizeof(struct _QTreeNode) * 4);
    node->node_type = INTERNAL_NODE;
    node->data.first_child = next;

    NODE(next).node_type = LEAF_NODE;
    NODE(next).row = NODE(node).row;
    NODE(next).col = NODE(node).col;
    NODE(next).width = NODE(node).width / 2;
    NODE(next).height = NODE(node).height / 2;

    NODE(next+1).node_type = LEAF_NODE;
    NODE(next+1).row = NODE(node).row;
    NODE(next+1).col = NODE(node).col + NODE(node).width / 2;
    NODE(next+1).width = (NODE(node).width+1) / 2;
    NODE(next+1).height = NODE(node).height / 2;

    NODE(next+2).node_type = LEAF_NODE;
    NODE(next+2).row = NODE(node).row + NODE(node).height / 2;
    NODE(next+2).col = NODE(node).col;
    NODE(next+2).width = NODE(node).width / 2;
    NODE(next+2).height = (NODE(node).height+1) / 2;

    NODE(next+3).node_type = LEAF_NODE;
    NODE(next+3).row = NODE(node).row + NODE(node).height / 2;
    NODE(next+3).col = NODE(node).col + NODE(node).width / 2;
    NODE(next+3).width = (NODE(node).width+1) / 2;
    NODE(next+3).height = (NODE(node).height+1) / 2;
    return next;
}

void destroy_children(QTreeNode first_child){
```

```c
void destroy_children(QTreeNode first_child){
    for (int i = 0; i < 4; i++){
        if (NODE(first_child+i).node_type == INTERNAL_NODE){
            destroy_children(first_child: NODE(first_child+i).data.first_child);
        }
    }
    free(ptr: first_child);
}
void qtree_destroy(QTreeNode qtree){
    if(qtree->node_type == INTERNAL_NODE){
        QTreeNode next = qtree->data.first_child;
        destroy_children(first_child: next);
    }
    free(ptr: qtree);
}
```

2. Algoritma Kompresi

```c
void quad_tree_compression(const unsigned char *rgb_input,
                           unsigned char       *rgb_output,
                           int                  width,
                           int                  height,
                           double               error_threshold,
                           int                  minimum_block_size,
                           VarianceFunction    *variance_fn,
                           QTreeNode            tree) {
    int row_start = NODE(tree).row;
    int col_start = NODE(tree).col;
    int row_end = row_start + NODE(tree).height - 1;
    int col_end = col_start + NODE(tree).width - 1;

    RGB temp = average_rgb(rgb_data: rgb_input, row_start, col_start, row_end,
        col_end, width, height);
    NODE(tree).data.rgb = temp;

    // can't subdivide further
    if (NODE(tree).width * NODE(tree).height / 4 < minimum_block_size) {
        goto compute_color;
    }

    double variance = (*variance_fn)(rgb_data: rgb_input, row_start, col_start, row_end,
                                     col_end, width, height);

    if (variance >= error_threshold) {
        QTreeNode next = qtree_subdivide(node_idx: tree);

        for (int i = 0; i < 4; i++) {
            quad_tree_compression(rgb_input, rgb_output, width, height,
                                  error_threshold, minimum_block_size,
                                  variance_fn, tree: next + i);
        }
        return;
    }

compute_color:;
    fill_color(rgb_output, width, height, tree);
    return;
}
```

```
void fill_color(unsigned char       *rgb_output,
                int                  width,
                int                  height,
                QTreeNode            tree) {
    int row_start = NODE(tree).row;
    int col_start = NODE(tree).col;
    int row_end = row_start + NODE(tree).height - 1;
    int col_end = col_start + NODE(tree).width - 1;
    unsigned char r = NODE(tree).data.rgb.r;
    unsigned char g = NODE(tree).data.rgb.g;
    unsigned char b = NODE(tree).data.rgb.b;
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            rgb_output[(i*width+j)*3] = r;
            rgb_output[(i*width+j)*3+1] = g;
            rgb_output[(i*width+j)*3+2] = b;
        }
    }
}
```

3. Algoritma Error Measurement

```
RGB average_rgb(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
            int col_end,int width, int height){
    RGB res;
    double r = 0, g = 0, b = 0;
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            r += (double) rgb_data[(i * width + j) * 3];
            g += (double) rgb_data[(i * width + j) * 3 + 1];
            b += (double) rgb_data[(i * width + j) * 3 + 2];
        }
    }
    int area = (row_end - row_start + 1) * (col_end - col_start + 1);
    res.r = (unsigned char) (r / area);
    res.g = (unsigned char) (g / area);
    res.b = (unsigned char) (b / area);
    return res;
}

void average(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
            int col_end,int width, int height, double *avg_r, double *avg_g, double *avg_b){
    double r = 0, g = 0, b = 0;
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            r += (double) rgb_data[(i * width + j) * 3];
            g += (double) rgb_data[(i * width + j) * 3 + 1];
            b += (double) rgb_data[(i * width + j) * 3 + 2];
        }
    }
    int area = (row_end - row_start + 1) * (col_end - col_start + 1);
    *avg_r = r / area;
    *avg_g = g / area;
    *avg_b = b / area;
}
```

```
double variance(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
                int col_end,int width, int height) {
    double res;
    double mr, mg, mb, sr=0, sg=0, sb=0;
    average(rgb_data, row_start, col_start, row_end, col_end, width, height, avg_r: &mr, avg_g: &mg, avg_b: &mb);
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            double diff_r = rgb_data[(i * width + j) * 3] - mr;
            double diff_g = rgb_data[(i * width + j) * 3 + 1] - mg;
            double diff_b = rgb_data[(i * width + j) * 3 + 2] - mb;
            sr += diff_r * diff_r;
            sg += diff_g * diff_g;
            sb += diff_b * diff_b;
        }
    }

    int area = (row_end - row_start + 1) * (col_end - col_start + 1);
    return (sr + sg + sb)/(area*3);
}

double mean_absolute_deviation(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
                               int col_end, int width, int height) {
    double res;
    double mr, mg, mb, sr = 0, sg = 0, sb = 0;
    average(rgb_data, row_start, col_start, row_end, col_end, width, height, avg_r: &mr, avg_g: &mg, avg_b: &mb);
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            sr += fabs(x: rgb_data[(i * width + j) * 3] - mr);
            sg += fabs(x: rgb_data[(i * width + j) * 3 +   1] - mg);
            sb += fabs(x: rgb_data[(i * width + j) * 3 +   2] - mb);
        }
    }
    int area = (row_end - row_start + 1) * (col_end - col_start + 1);
    return (sr + sg + sb)/(area*3);
}
```

```
double max_pixel_difference(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
                            int col_end, int width, int height) {
    double max_r = 0, min_r = 255;
    double max_g = 0, min_g = 255;
    double max_b = 0, min_b = 255;
    double r, g, b;
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            r = (double) rgb_data[(i * width + j) * 3];
            g = (double) rgb_data[(i * width + j) * 3 + 1];
            b = (double) rgb_data[(i * width + j) * 3 + 2];
            max_r = r > max_r ? r : max_r;
            max_g = g > max_g ? g : max_g;
            max_b = b > max_b ? b : max_b;
            min_r = r < min_r ? r : min_r;
            min_g = g < min_g ? g : min_g;
            min_b = b < min_b ? b : min_b;
        }
    }

    return (max_r - min_r + max_g - min_g + max_b - min_b) / 3;
}
```

```c
double entropy(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
               int col_end, int width, int height) {
    int area = (row_end - row_start + 1) * (col_end - col_start + 1);
    double pr, pg, pb;
    double hr = 0, hg = 0, hb = 0;
    int freq_table[3][256];
    create_freq_table(rgb_data, row_start, col_start, row_end, col_end, width, height, freq_table);

    for (int i = 0; i < 256; i++) {
        if (freq_table[0][i] > 0) {
            double pr = (double) freq_table[0][i] / area;
            hr += pr * log2(x: pr);
        }
        if (freq_table[1][i] > 0) {
            double pg = (double) freq_table[1][i] / area;
            hg += pg * log2(x: pg);
        }
        if (freq_table[2][i] > 0) {
            double pb = (double) freq_table[2][i] / area;
            hb += pb * log2(x: pb);
        }
    }

    hr *= -1;
    hg *= -1;
    hb *= -1;

    return (hr + hg + hb) / 3;
}
```

```c
void create_freq_table(const unsigned char *rgb_data, int row_start, int col_start, int row_end,
                       int col_end, int width, int height, int freq_table[3][256]) {
    // initialize freq table
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 256; j++) {
            freq_table[i][j] = 0;
        }
    }

    // count frequency
    for (int i = row_start; i <= row_end; i++){
        for (int j = col_start; j <= col_end; j++){
            freq_table[0][rgb_data[(i * width + j) * 3]]++;
            freq_table[1][rgb_data[(i * width + j) * 3 + 1]]++;
            freq_table[2][rgb_data[(i * width + j) * 3 + 2]]++;
        }
    }
}
```

BAB 4: TESTING

4.1. Test Case 1

Input : File input tidak ditemukan

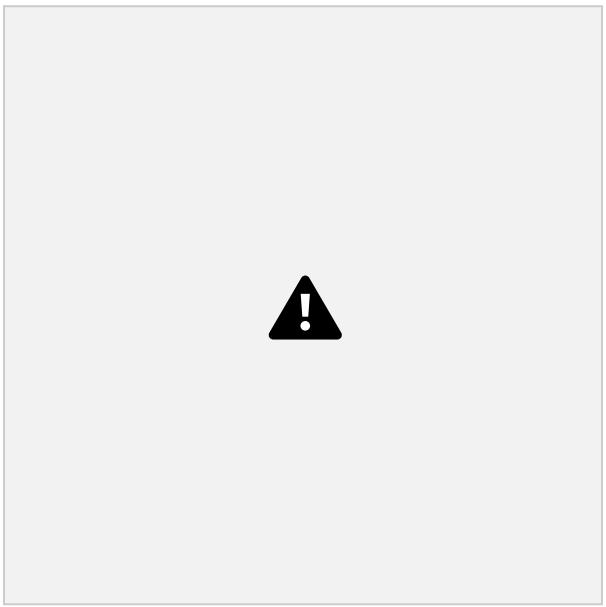
```
Insert absolute path to input image file:
 >>> skibidi toilet
File doesn't exist. please try again

Insert absolute path to input image file:
 >>>
```

Output : Mengulang dialog file path

4.2. Test Case 2

Input : Input valid, gambar png, error measurement method: variance, threshold = 10, min block = 2

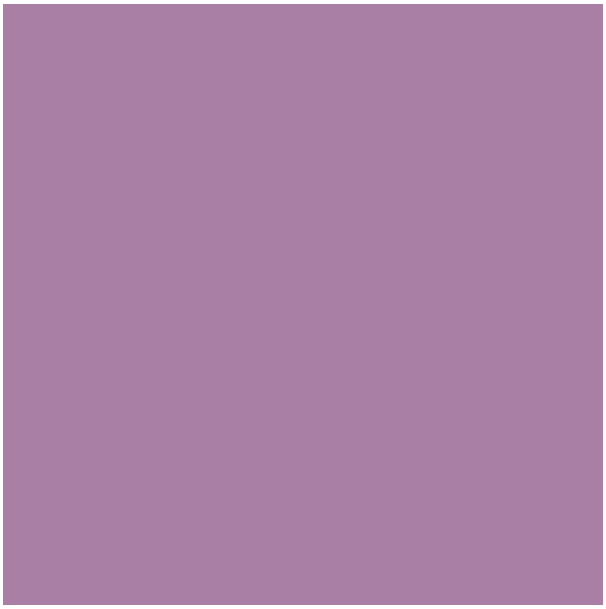


```
[*] Input Image Absolute Path  : ugly_michi.png
[*] Error Measurements Method  : variance
[*] Error Threshold            : 10.000000
[*] Minimum Block Size         : 2
[*] Output Image Absolute Path : res.png
[*] Output GIF Absolute Path   : res.gif

Compressing....

Success!
[*] Waktu eksekusi    : 48.343000 ms
[*] Kedalaman pohon   : 8
[*] Banyak node pohon: 20005
[*] Besar file awal   : 101733 byte
[*] Besar file akhir  : 61468 byte
[*] Rasio kompresi    : 39.58%
```

Output :



4.3. Test Case 3

Input : Input valid, gambar png, error measurement method: MAD, threshold = 10, min block = 2

Gambar masih sama dengan sebelumnya.

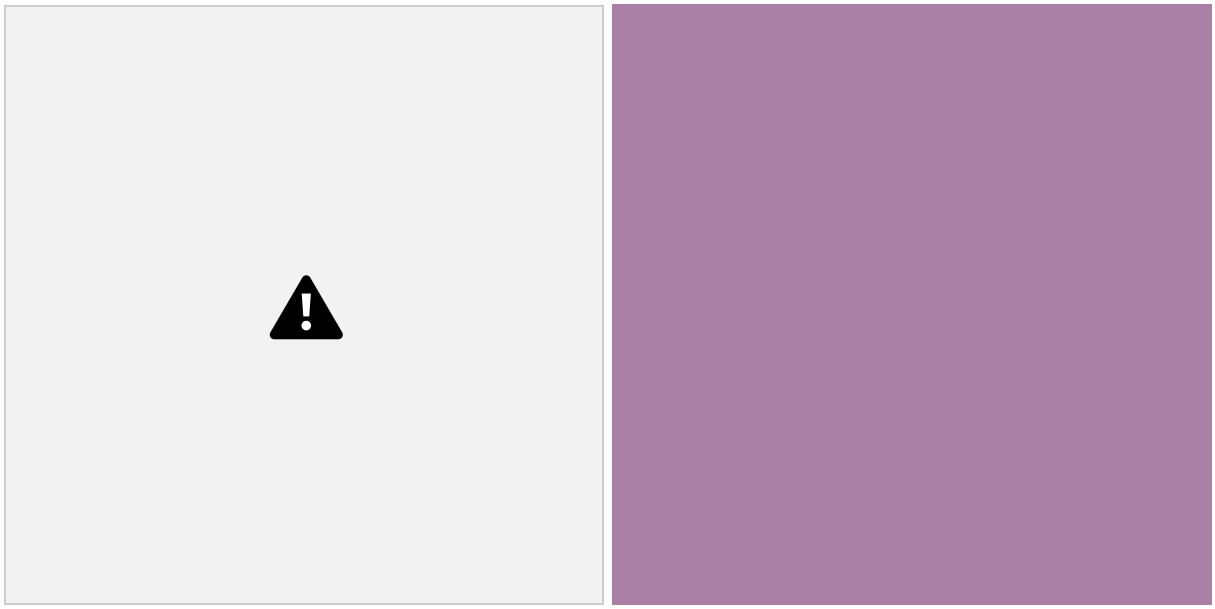
```
A QuadTree PNG Compressor by BoredAngel & LQMMM


[*] Input Image Absolute Path  : ugly_michi.png
[*] Error Measurements Method  : Mean Absolute Deviation
[*] Error Threshold            : 10.000000
[*] Minimum Block Size         : 2
[*] Output Image Absolute Path : res2.png
[*] Output GIF Absolute Path   : res2.gif

Compressing....

Success!
[*] Waktu eksekusi   : 48.965000 ms
[*] Kedalaman pohon  : 8
[*] Banyak node pohon: 13825
[*] Besar file awal  : 101733 byte
[*] Besar file akhir : 50442 byte
[*] Rasio kompresi   : 50.42%
```

Output :



4.4. Test Case 4

Input : Input valid, gambar png, error measurement method: max pixel difference, threshold = 10, min block = 2

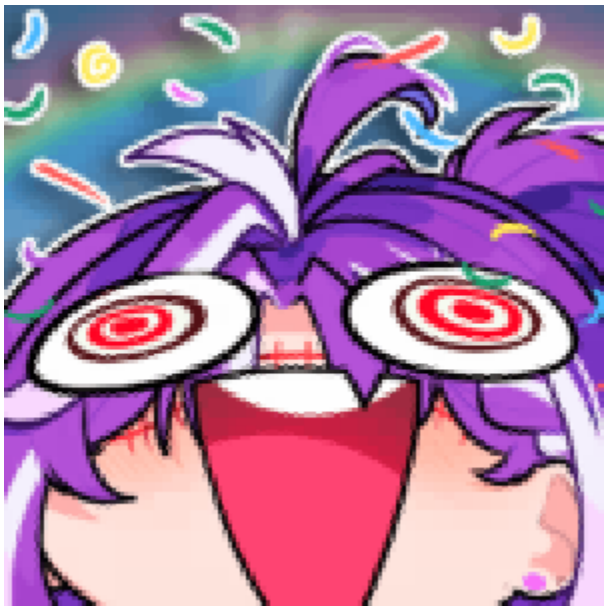
Gambar masih sama dengan sebelumnya

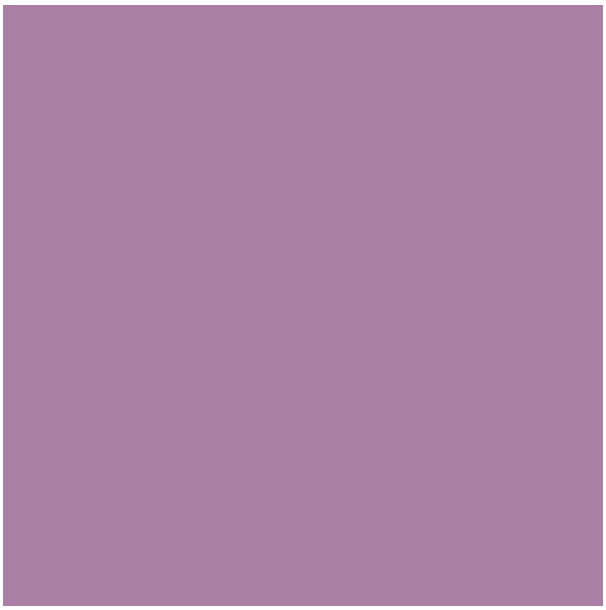
```
[*] Input Image Absolute Path  : ugly_michi.png
[*] Error Measurements Method  : Max Pixel Difference
[*] Error Threshold            : 10.000000
[*] Minimum Block Size         : 2
[*] Output Image Absolute Path : res1.png
[*] Output GIF Absolute Path   : res1.gif

Compressing....

Success!
[*] Waktu eksekusi    : 50.393000 ms
[*] Kedalaman pohon   : 8
[*] Banyak node pohon: 19273
[*] Besar file awal   : 101733 byte
[*] Besar file akhir  : 60358 byte
[*] Rasio kompresi    : 40.67%
```

Output :

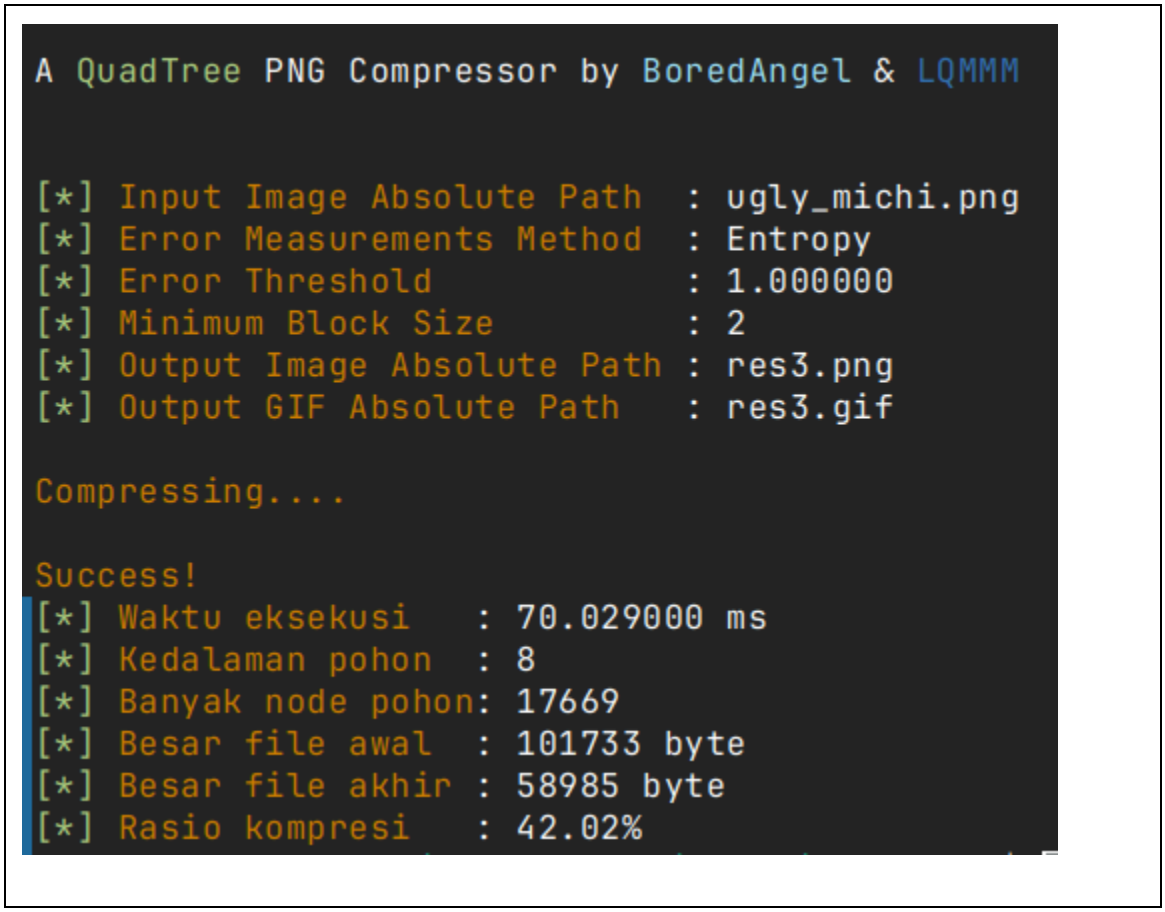




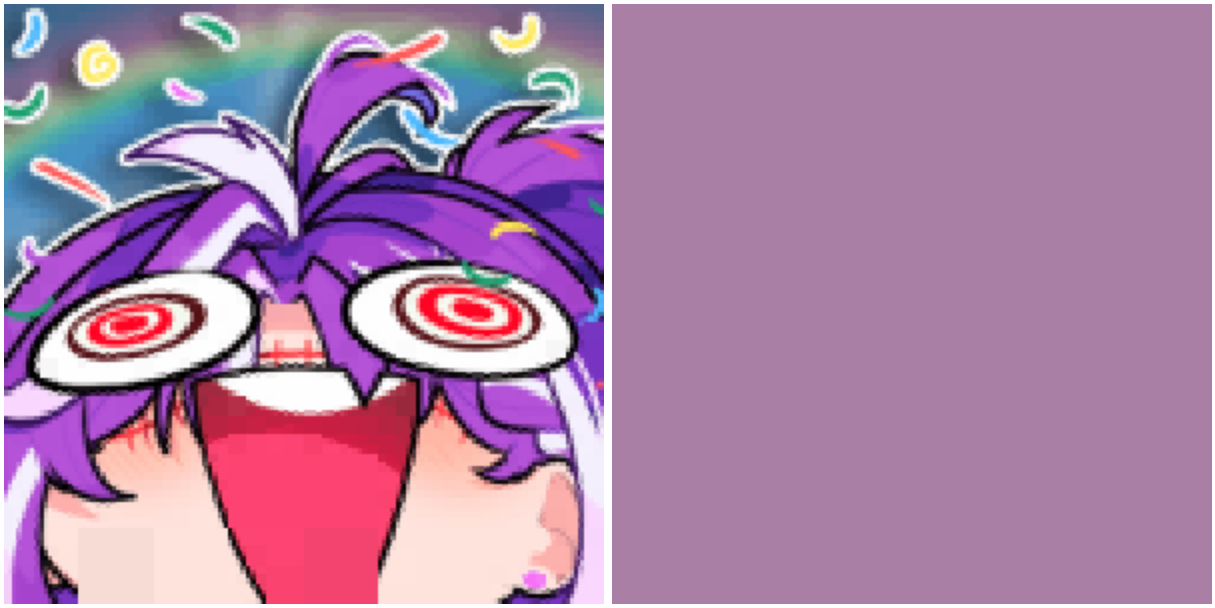
4.5. Test Case 5

Input : Input valid, error measurement method: entropy, threshold = 5, min block = 2

Gambar masih sama dengan sebelumnya



Output :



4.6. Test Case 6

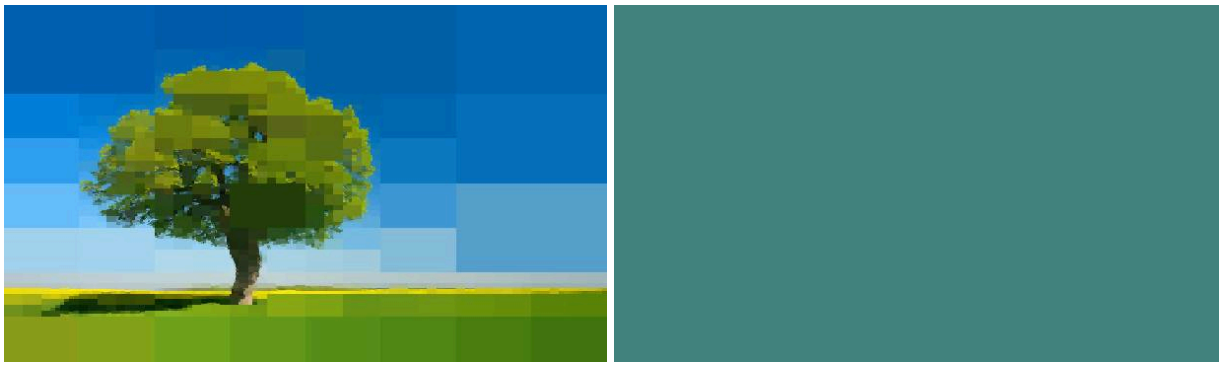
Input : Input valid berupa file jpeg, error measurement method: variance, threshold = 500, min block = 1

```
[*] Input Image Absolute Path  : C:\Users\Lenovo\Downloads\tree.jpeg
[*] Error Measurements Method  : variance
[*] Error Threshold            : 1000.000000
[*] Minimum Block Size         : 1
[*] Output Image Absolute Path : C:\Users\Lenovo\Downloads\tree1.jpeg
[*] Output GIF Absolute Path   : C:\Users\Lenovo\Downloads\tree1.gif

Compressing....

Success!
[*] Waktu eksekusi   : 24.000000 ms
[*] Kedalaman pohon  : 9
[*] Banyak node pohon: 5905
[*] Besar file awal  : 42166 byte
[*] Besar file akhir : 16466 byte
[*] Rasio kompresi   : 60.95%
```

Output :



4.7. Test Case 7

Input : Input valid berupa file jpeg, error measurement method: entropy, threshold = 2.5, min block = 1

Foto sama seperti sebelumnya

```
[*] Input Image Absolute Path  : C:\Users\Lenovo\Downloads\tree.jpeg
[*] Error Measurements Method  : Entropy
[*] Error Threshold            : 2.500000
[*] Minimum Block Size         : 1
[*] Output Image Absolute Path : C:\Users\Lenovo\Downloads\tree4.jpeg
[*] Output GIF Absolute Path   : C:\Users\Lenovo\Downloads\tree4.gif

Compressing....

Success!
[*] Waktu eksekusi   : 64.000000 ms
[*] Kedalaman pohon  : 9
[*] Banyak node pohon: 45545
[*] Besar file awal  : 42166 byte
[*] Besar file akhir : 26859 byte
[*] Rasio kompresi   : 36.30%
```

Output :

BAB 5: Analisis

5.1. Analisis Kompleksitas

5.1.1. Fungsi Rata-Rata Pixel

Untuk bagian gambar dengan panjang m dan lebar n, kompleksitas untuk menghitung rata-rata pixel adalah O(mn) karena algoritma rata-rata membutuhkan penjumlahan setiap nilai pixel.

5.1.2. Fungsi Variansi

Variansi dihitung dengan dua tahap. Tahap pertama adalah menghitung rata-rata pixel (O(mn)), dan tahap selanjutnya adalah menghitung rata-rata kuadrat selisih pixel dengan rata-rata. Karena tahap kedua juga menghitung setiap pixel sekali, kompleksitasnya juga O(mn). Sehingga kompleksitas keseluruhannya adalah O(mn).

5.1.3. Fungsi MAD

Dengan alasan yang serupa dengan fungsi variansi, kompleksitas algoritma ini juga O(mn).

5.1.4. Fungsi Max Pixel Difference

Fungsi ini perlu mencari nilai maksimum dan minimum setiap pixel. Algoritma pencarian nilai ekstrim sudah umum diketahui bergantung linear dengan banyak elemen yang ditelusuri, pada kasus ini O(mn). Sehingga kompleksitas keseluruhannya adalah O(mn).

5.1.5. Fungsi Entropy

Fungsi ini perlu mencari frekuensi kemunculan setiap pixel. Sehingga langkah pertama algoritma ini menelusuri setiap elemen sekali, dengan kompleksitas O(mn). Saat menghitung entropi, karena hanya terdapat 256 nilai pixel yang mungkin, waktu untuk menghitung entropi adalah O(1). Hal ini karena banyak kemungkinan pixel adalah konstan. Sehingga kompleksitas keseluruhannya adalah O(mn).

5.1.6. Fungsi Kompresi

Waktu yang digunakan untuk menyelesaikan suatu subproblem adalah O(mn). Sedangkan satu subproblem terbagi menjadi 4 bagian yang sama besar. Sehingga, dengan teorema Master:

$$T(mn) = 4T(\frac{mn}{4}) + O(mn)$$

$$T(mn) = O((mn)^{log_4 4}(log\ mn)^{0+1}) = O(mn\ log(mn)),$$

dengan m panjang gambar dan n lebar gambar.

LAMPIRAN

Github Repository

Program dapat diakses pada https://github.com/carasiae/quadtree

Tabel Poin

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