

Laporan Tugas Kecil

Kompresi Gambar Dengan Metode

Quadtree

**Tugas Kecil 2 - IF2211 Strategi Algoritma Semester II Tahun
2024/2025**



Disusun oleh

Muhammad Adha Ridwan - 13523098
Naufarrel Zhafif Abhista - 13523149

Daftar Isi

Daftar Isi.....	1
1. Pendahuluan.....	2
2. Penjelasan Algoritma Divide And Conquer.....	3
2.1. Divide.....	3
2.2. Conquer.....	6
2.3. Combine.....	6
2.4. Pseudocode.....	6
3. Source Code.....	8
3.1. Struktur Proyek.....	8
3.2. Dependency.....	9
3.3. Main.cpp.....	10
3.4. IO.cpp.....	12
3.5. quadtree.cpp.....	13
3.6. quadtree.cpp.....	14
3.7. image_error.cpp.....	17
3.8. image_utils.cpp.....	20
3.9. Bonus.....	24
3.10. Lainnya.....	27
4. Uji Coba.....	28
5. Analisis.....	28
5.1. Analisis Kompleksitas Waktu.....	28
5.2. Analisis Kompleksitas Ruang.....	29
5.3. Analisis Kompresi.....	30
5.4. Analisis Metode.....	30
6. Lampiran.....	31

1. Pendahuluan

Algoritma *Divide and Conquer* (DnC) adalah paradigma algoritma yang memecah sebuah masalah besar menjadi beberapa submasalah yang lebih kecil, menyelesaikan masing-masing submasalah secara rekursif, lalu menggabungkan hasilnya untuk mendapatkan solusi akhir. Salah satu implementasi algoritma *Divide and Conquer* adalah penggunaan Quadtree untuk kompresi gambar.

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.

2. Penjelasan Algoritma Divide And Conquer

Algoritma Divide and Conquer digunakan dalam tugas ini untuk menyelesaikan permasalahan kompresi gambar menggunakan struktur data Quadtree. Algoritma bekerja dengan membagi gambar menjadi blok-blok yang lebih kecil secara rekursif, kemudian menganalisis keseragaman warna tiap blok. Jika suatu blok tidak cukup seragam (nilai error melebihi threshold), maka blok tersebut dibagi lagi menjadi empat bagian kuadran. Proses ini terus berlangsung hingga setiap blok mencapai tingkat keseragaman yang ditentukan atau ukuran minimum yang diperbolehkan.

Dengan pendekatan ini, proses kompresi menjadi lebih efisien karena hanya bagian-bagian gambar yang memiliki variasi tinggi yang dibagi lebih lanjut, sedangkan area yang sudah cukup seragam tidak dipecah lagi. Hal ini memungkinkan pengurangan ukuran file tanpa mengorbankan detail penting, sesuai prinsip dari algoritma Divide and Conquer.

Pada laporan ini, seperti perintah spesifikasi, yang akan dibahas spesifik adalah algoritma dari *divide and conquer*. Untuk komponen pelengkap, seperti *export* solusi, validasi input, dan sebagainya, akan dilampirkan dalam *source code* dan juga Pranala Repositori.

Berikut adalah tahapan dalam melakukan DnC melalui Quadtree.

2.1. Divide

1. Mulai dari seluruh gambar sebagai blok awal.
2. Periksa apakah blok tersebut seragam dengan menghitung **error** menggunakan metode tertentu: variansi, MAD, Max Pixel Difference, entropy, atau SSIM, sesuai dengan tabel di bawah.

Tabel X. Metode Pengukuran Error

Metode	Formula	Penjelasan
Varians	$\sigma_c^2 = \frac{1}{N} \sum_{i=1}^N (P_{i,c} - \mu_c)^2$	Variansi mengukur seberapa tersebar nilai piksel terhadap rata-rata dalam satu blok.
	$\sigma_{RGB}^2 = \frac{\sigma_R^2 + \sigma_G^2 + \sigma_B^2}{3}$	Jika variansi kecil, berarti piksel-piksel cenderung seragam. Metode ini cocok untuk mendeteksi blok-blok dengan noise ringan atau
	<ul style="list-style-type: none">• σ_c^2 = Variansi tiap kanal• $P_{i,c}$ = Nilai piksel pada posisi i	

	<ul style="list-style-type: none"> di kanal c μ_c = Rata-rata piksel dalam satu blok N = Banyaknya piksel dalam satu blok 	<p>gradasi halus.</p> <p>Dapat menjadi opsi saat ingin kompresi yang sensitif terhadap fluktuasi nilai warna kecil.</p> <p>Karena nilai piksel 8-bit (0–255), variansi maksimum (threshold) adalah $(255^2) = 65025$.</p>
<i>Mean Absolute Deviation</i> (MAD)	$MAD_c = \frac{1}{N} \sum_{i=1}^N P_{i,c} - \mu_c $ $MAD_{RGB} = \frac{MAD_R^2 + MAD_G^2 + MAD_B^2}{3}$ <ul style="list-style-type: none"> MAD_c^2 = MAD tiap kanal $P_{i,c}$ = Nilai piksel pada posisi i di kanal c μ_c = Rata-rata piksel dalam satu blok N = Banyaknya piksel dalam satu blok 	<p>MAD menghitung rata-rata dari selisih absolut setiap nilai piksel terhadap rata-rata. Karena hanya menghitung selisih, ia lebih tahan terhadap outlier ekstrem dibanding variansi.</p> <p>Cocok digunakan saat ingin hasil kompresi yang <i>robust</i> terhadap perbedaan ekstrim kecil di blok.</p> <p>Asumsikan terdapat piksel dengan nilai 255 dan mean piksel 0, maka threshold maksimum yang dimiliki metode ini adalah $255 - 0 = 255$.</p>
<i>Max Pixel Differences</i>	$D_c = \max(P_{i,c}) - \min(P_{i,c})$ $D_{RGB} = \frac{D_R + D_G + D_B}{3}$ <ul style="list-style-type: none"> 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 di kanal c 	<p>Metode ini mengukur perbedaan maksimum antara piksel paling terang dan paling gelap dalam satu blok.</p> <p>Jika nilai maksimumnya kecil, blok dianggap seragam. Artinya, perhitungannya sederhana dan cepat. Metode ini cocok untuk implementasi ringan atau kompresi cepat dengan hasil kasar.</p> <p>Threshold maksimum yang dimiliki metode ini adalah $255 (\text{max}) - 0 (\text{min}) = 255$.</p>

Entropi	$H_c = - \sum_{i=1}^N P_c(i) \log_2(P_c(i))$	<p>Entropi mengukur ketidakpastian atau keragaman distribusi nilai piksel dalam satu blok. Semakin tinggi entropi, semakin acak dan bervariasi isi blok.</p> <p>Cocok untuk kasus di mana detail tekstur dan distribusi warna penting.</p> <p>Karena $\log_2 256 = 8$, maka threshold maksimumnya adalah 8.</p>
	$H_{RGB} = \frac{H_R + H_G + H_B}{3}$	
	<ul style="list-style-type: none"> • H_c = Nilai entropi tiap kanal warna dalam satu blok • $P_{i,c}$ = Nilai piksel pada posisi i di kanal c 	
<i>Structural Similarity Index</i> (SSIM)	<p>Rumus Asli:</p> $SSIM_{c(x,y)} = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(2\sigma_{xy,c} + C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + \sigma_{y,c}^2 + C_2)}$ <p>Rumus Modifikasi:</p> $SSIM_{c(x,y)} = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + C_2)}$	<p>SIM membandingkan dua citra (yakni, blok asli dan blok hasil kompresi) dari segi:</p> <ul style="list-style-type: none"> • Luminance (kecerahan) dengan μ, • Contrast (perbedaan intensitas) dengan σ^2, • Structure (pola tekstur) dengan σ_{xy}. <p>Dalam konteks tugas ini, blok hasil kompresi selalu berupa blok monoton dengan warna rata-rata. Oleh karena itu, variansi dan kovarians dari blok hasil kompresi bernilai nol. Akibatnya, rumus SSIM dapat disederhanakan, seperti yang terlihat pada kolom di samping.</p> <p>Nilai konstanta koreksi diperoleh dari referensi jurnal (Dicantumkan di lampiran)</p> <p>Hasil dari SSIM berada pada $[0, 1]$. Sehingga, threshold maksimumnya adalah 1.</p>
	$SSIM_{RGB} = w_R SSIM_R + w_G SSIM_G + w_B SSIM_B$ <ul style="list-style-type: none"> • $\mu_{x,c}$ = Rata-rata piksel gambar x dalam satu blok • $\mu_{y,c}$ = Rata-rata piksel gambar y dalam satu blok • $\sigma_{x,c}^2$ = Variansi gambar x tiap kanal • $\sigma_{y,c}^2$ = Variansi gambar y tiap kanal • $\sigma_{xy,c}$ = Kovariansi gambar x, y tiap kanal • C_1, C_2 = Konstanta koreksi ($C_1 = 0.1, C_2 = 0.3$) • w = Bobot untuk kanal warna 	

- Jika **error melebihi threshold** dan ukuran blok masih lebih besar dari minimum block size, maka blok dibagi menjadi **empat sub-blok kuadran** (kiri atas, kanan atas, kiri bawah, kanan bawah).

2.2. Conquer

- Lakukan proses yang sama (perhitungan error dan pengecekan threshold + ukuran) pada setiap sub-blok secara rekursif.
- Proses ini berlanjut hingga seluruh blok memenuhi kondisi berhenti, yaitu:
 - Error lebih kecil atau sama dengan threshold, atau
 - Ukuran blok tidak dapat dibagi lagi karena mencapai minimum block size.

2.3. Combine

- Setelah pembagian selesai, setiap blok akhir (leaf node) disimpan dengan nilai **rata-rata RGB** dari piksel-pikselnya.
- Struktur pohon Quadtree digunakan untuk merekonstruksi gambar hasil kompresi.
- Blok-blok seragam akan diwarnai dengan nilai rata-rata, sehingga mengurangi kompleksitas data dan ukuran gambar.

2.4. Pseudocode

Ketiga langkah tersebut dapat digambarkan dengan pseudocode berikut:

```
Function CompressQuadtree(image, x, y, width, height):
    // === Divide ===
    block ← extractBlock(image, x, y, width, height)
    error ← computeError(block)

    // === Conquer ===
    if (error > threshold AND width > minBlockSize AND height > minBlockSize):
        midWidth ← width / 2
        midHeight ← height / 2

        topLeft ← CompressQuadtree(image, x, y, midWidth, midHeight)
        topRight ← CompressQuadtree(image, x + midWidth, y, midWidth, midHeight)
        bottomLeft ← CompressQuadtree(image, x, y + midHeight, midWidth, midHeight)
        bottomRight ← CompressQuadtree(image, x + midWidth, y + midHeight, midWidth, midHeight)
```

```

    return InternalNode(topLeft, topRight, bottomLeft, bottomRight)

    // === Combine ===
else:
    avgColor ← computeAverageColor(block)
    return LeafNode(x, y, width, height, avgColor)

```

Alur program tersebut sudah dijelaskan secara umum melalui poin 2.1 hingga 2.3. Sehingga, pada subbab ini, hanya dibahas beberapa hal dari pseudocode ini. Beberapa hal tersebut, antara lain:

1. **Fungsi CompressQuadtree** merupakan fungsi rekursif utama yang mengimplementasikan paradigma *divide and conquer*. Fungsi ini menerima parameter posisi koordinat blok saat ini, ukuran blok, serta gambar asli sebagai referensi.
2. **Langkah "Divide"** dilakukan dengan mengekstrak blok gambar dan menghitung nilai error-nya, menggunakan salah satu metode pengukuran error yang telah ditentukan pengguna, seperti variansi, MAD, atau SSIM. Nilai error ini digunakan sebagai dasar keputusan apakah blok tersebut akan dibagi lebih lanjut.
3. **Langkah "Conquer"** tercermin dalam proses rekursif ketika blok dibagi menjadi empat kuadran jika nilai error melebihi threshold dan ukuran blok masih lebih besar dari batas minimum. Pemanggilan rekursif dilakukan untuk masing-masing sub-blok: kiri atas, kanan atas, kiri bawah, dan kanan bawah.
4. **Langkah "Combine"** dilakukan saat kondisi pembagian tidak terpenuhi, yaitu ketika blok dianggap cukup seragam atau terlalu kecil untuk dibagi lagi. Pada tahap ini, dihitung rata-rata warna blok, dan simpul daun (LeafNode) dibentuk dengan informasi tersebut.
5. Struktur pohon **Quadtree dibentuk secara bottom-up**, di mana simpul-simpul daun akan disatukan dalam simpul internal (InternalNode) yang merepresentasikan pembagian sebelumnya. Struktur ini digunakan untuk merekonstruksi gambar hasil kompresi.

Hasil dari Quadtree ini kemudian diolah dari matrix of RGB menjadi citra/gambar kembali yang telah terkompresi pikselnya dibandingkan gambar asli.

3. Source Code

Pada bagian ini, akan diperjelas secara singkat mengenai struktur proyek dan juga modularitas dari setiap kelas. Pada dokumen ini, karena keterbatasan halaman, *penjelasan kode dari suatu file hanya akan dilakukan per bagian, bukan keseluruhan file itu sendiri*. Untuk melihat isi suatu file secara penuh, dapat dilihat secara langsung repositorinya, yang berada di Bab 6 (Lampiran).

3.1. Struktur Proyek

Proyek ini memiliki struktur direktori dan file sebagai berikut:

```
Tucil2_13523098_13523149
├── Makefile
├── README.md
└── bin
    ├── FreeImage.dll
    ├── FreeImagePlus.dll
    ├── linuxver
    └── winver.exe
└── doc
    └── Laporan Tucil2_13523098_13523149.pdf
└── src
    ├── header
    │   ├── FreeImage.h
    │   ├── FreeImagePlus.h
    │   ├── image_error.hpp
    │   ├── image_utils.hpp
    │   ├── io.hpp
    │   └── quadtree.hpp
    ├── image_error.cpp
    ├── image_utils.cpp
    ├── io.cpp
    └── lib
        ├── libfreeimage.a
        ├── libfreeimage.dll.a
        ├── libfreeimageplus.a
        └── libfreeimageplus.dll.a
    ├── main.cpp
    ├── quadtree.cpp
    └── quadtreenode.cpp
└── test
```

Penjelasan masing-masing direktori dan file:

- **bin/**: Berisi file executable untuk menjalankan program

- linuxver merupakan executable untuk menjalankan program dalam sistem operasi Linux.
 - winver.exe merupakan executable untuk menjalankan program dalam sistem operasi Windows.
 - FreeImage.dll dan FreeImagePlus.dll merupakan *dependency* untuk winver.exe sehingga dapat dijalankan pada Windows.
- **doc/**: Direktori untuk dokumentasi proyek dalam bentuk **.pdf**.
- **src/**: Berisi kode sumber utama proyek.
 - **header/**: Mengandung header untuk mengompilasi program.
 - **lib/**: Mengandung sumber kode dari FreeImage agar program dapat dikompilasi.
 - **main.cpp** : Driver utama program.
 - **io.cpp** : Bertanggung jawab atas alur *stream input* dan *output*.
 - **image_error.cpp** : Mengandung metode penentuan error, sesuai dengan Tabel X.
 - **image_utils.cpp** : Bertanggung jawab atas pengolahan gambar, sebelum dan setelah dikompresi.
 - **quadtree.cpp**: Bertanggung jawab atas kompresi gambar.
 - **quadtreeinode.cpp** : Struktur dasar untuk **quadtree.cpp**
- **test/**: Direktori untuk file uji coba dalam bentuk gambar dan hasil dalam bentuk gambar dengan format file yang sama dalam folder output.
- **Makefile**: Bertanggung jawab untuk melakukan kompilasi program sehingga menghasilkan executable yang dapat dijalankan. *Makefile ini hanya bisa dijalankan di Linux/WSL*.
- **README.md**: File readme yang mencakup deskripsi, persyaratan, instalasi, dan cara penggunaan aplikasi.

3.2. *Dependency*

Program ini menggunakan pustaka FreeImage sebagai library eksternal utama untuk membaca dan menulis file gambar. FreeImage merupakan library open source yang mendukung berbagai format gambar, seperti BMP, PNG, JPEG, dan lainnya, serta menyediakan fungsi-fungsi manipulasi citra dalam bentuk struktur data yang efisien untuk diolah oleh algoritma.

FreeImage dipilih karena:

- Kompatibilitas format gambar luas, termasuk format umum seperti .jpg, .png, dan .bmp.
- Mudah diintegrasikan ke dalam program berbasis C++.

- Menyediakan akses langsung ke buffer piksel, yang sangat berguna dalam pengolahan blok-blok gambar untuk algoritma kompresi Quadtree.
- Cross-platform, dapat digunakan baik di Windows maupun Linux.
- Relatif ringan untuk pustaka pembacaan gambar.

Supaya program dapat dikompilasi, perlu dilakukan penginstalan pustaka FreeImage (Juga Wrapper FreeImagePlus untuk C++). Berikut langkah pernyiapannya.

Linux

Jalankan perintah berikut di terminal Linux/WSL (Kami menggunakan Ubuntu 22.04 LTS).

```
sudo apt-get install libfreeimage3 libfreeimageplus3 libfreeimage-dev
```

Windows

Pastikan telah mempunyai MinGW-W64. Di terminal MSYS2, jalankan perintah berikut.

```
pacman -S mingw64/mingw-w64-x86_64-freeimage
pacman -Syu
```

3.3. Main.cpp

Main.cpp merupakan awal mula dari program saat dijalankan pengguna. Secara umum, Main.cpp bertanggung jawab atas antarmuka pengguna. Fungsionalitas utamanya mencakup:

1. Inisialisasi Input

Main.cpp memanfaatkan kelas IO untuk memproses dan memvalidasi argumen yang diberikan pengguna saat menjalankan program. Input mencakup:

- Path absolut gambar input
- Metode perhitungan error
- Threshold
- Minimum block size
- Path absolut output gambar terkompresi
- (Opsional) GIF output



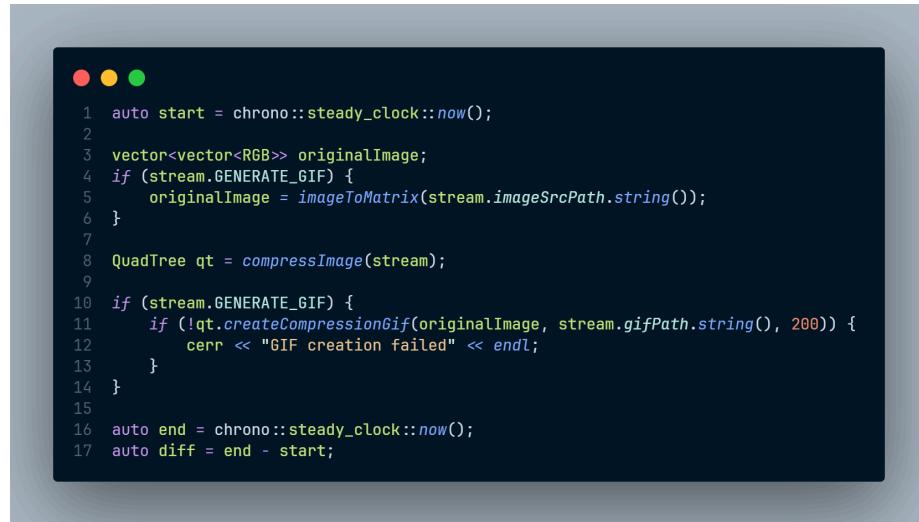
```
1 int main(int argc, char const *argv[])
2 {
3     IO stream;
4
5     if (!stream.initInput(argc, argv)) return 0;
```

Jika keluar dari proses input, program selesai dengan exit code 0.

2. Proses Kompresi

Setelah input tervalidasi, program akan memulai proses kompresi gambar menggunakan struktur data QuadTree. Fungsi compressImage() akan:

- Membentuk struktur pohon Quadtree secara rekursif
- Mengompres blok berdasarkan nilai error
- Melakukan normalisasi warna pada blok yang tidak dibagi lagi



```
1 auto start = chrono::steady_clock::now();
2
3 vector<vector<RGB>> originalImage;
4 if (stream.GENERATE_GIF) {
5     originalImage = imageToMatrix(stream.imageSrcPath.string());
6 }
7
8 QuadTree qt = compressImage(stream);
9
10 if (stream.GENERATE_GIF) {
11     if (!qt.createCompressionGif(originalImage, stream.gifPath.string(), 200)) {
12         cerr << "GIF creation failed" << endl;
13     }
14 }
15
16 auto end = chrono::steady_clock::now();
17 auto diff = end - start;
```

Jika fitur visualisasi kompresi diaktifkan, program akan menyimpan proses kompresi sebagai animasi GIF menggunakan method createCompressionGif() dari objek Quadtree. Proses ini membutuhkan gambar asli yang telah dimuat sebelumnya. Kelas chrono juga digunakan untuk menghitung runtime program saat melakukan kompresi.

3. Menampilkan Statistik Hasil

Setelah proses selesai, program akan menampilkan informasi penting:

- Waktu eksekusi
- Ukuran gambar sebelum kompresi
- Ukuran gambar setelah kompresi
- Persentase kompresi
- Kedalaman maksimum pohon Quadtree
- Jumlah total node yang digunakan
- Gambar hasil pada alamat yang sudah ditentukan
- GIF hasil kompresi pada alamat yang sudah ditentukan [Opsional]



```

1 cout << "Execution time: " << chrono::duration<double, milli>(diff).count() << " ms" << '\n';
2 cout << "Original image size: " << originalSize << " bytes" << '\n';
3 cout << "Compressed image size: " << compressedSize << " bytes" << '\n';
4 cout << "Compression percentage: " << compressionPercentage << "%" << '\n';
5 cout << "Max tree depth: " << qt.getMaxDepth() << '\n';
6 cout << "Total nodes: " << qt.getNodeCount() << '\n';

```

3.4. IO.cpp

IO.cpp bertanggung jawab terhadap pengelolaan input/output dari pengguna. Kode ini mengatur proses interaktif CLI agar pengguna dapat mengisi parameter kompresi dengan benar. Fungsionalitas utamanya mencakup:

1. Validasi Jawaban Ya/Tidak

Metode `validYN(msg)` digunakan untuk meminta konfirmasi dari pengguna dengan input Y/y atau N/n. Jika input tidak valid, pengguna akan diminta ulang hingga valid.

2. Validasi Input Tiap Atribut

Metode `inputSrc()`, `inputMethod()`, `inputThreshold()`, `inputMinBlock()`, `inputDest()`, dan `inputGifPath()` meminta pengguna untuk memasukkan nama file gambar dari folder `test/`. Program akan mengecek keberadaan file:

- Untuk `inputSrc()`, jika file tidak ditemukan, pengguna dapat memilih untuk mencoba lagi.
- Untuk `inputMethod()`, input dibatasi dari integer 1 - 5 sebagai opsi.
- Setiap metode input dilengkapi dengan failsafe bila tidak sesuai dengan yang diharapkan.

Setiap method tersebut bertipe boolean. Jika method sukses dilaksanakan, akan mengembalikan true. Kelima atribut tersebut disatukan dalam method initInput().

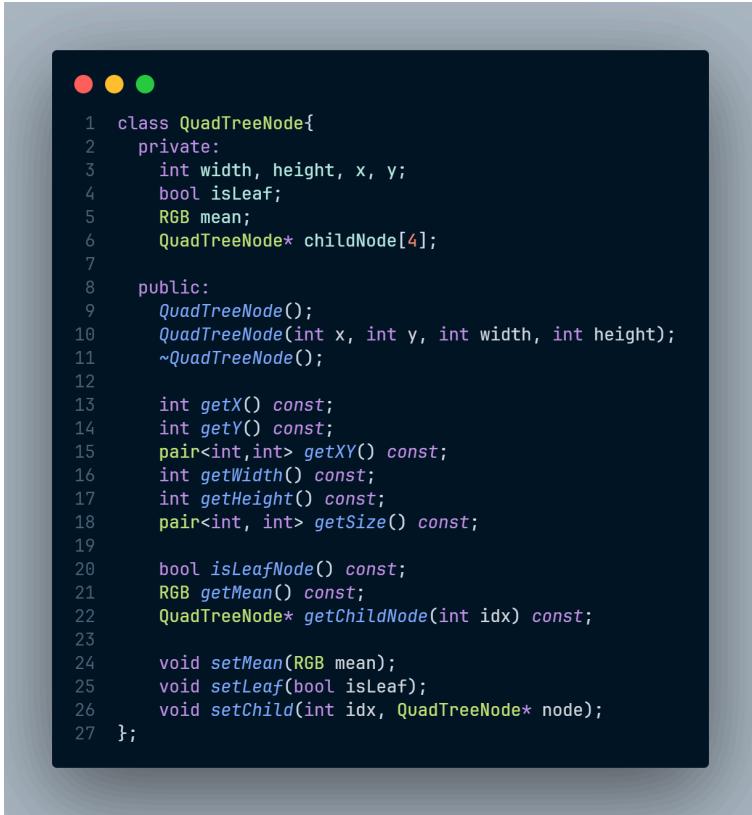
The screenshot shows a terminal window with three colored dots (red, yellow, green) at the top. The code is displayed in a dark-themed terminal:

```
1 bool IO::initInput(int argc, char const *argv[]) {
2     if (!inputSrc(argv)) return false;
3     if (!inputMethod()) return false;
4     if (!inputThreshold()) return false;
5     if (!inputMinBlock()) return false;
6     if (!inputDest(argv)) return false;
7     if (!inputGifPath(argv)) return false;
8
9     return true;
10 }
```

3.5. quadtree.cpp

File ini mengimplementasikan kelas QuadTreeNode yang berada di dalam quadtree.hpp, yang merepresentasikan simpul pada struktur data Quadtree. Setiap simpul (node) menyimpan informasi tentang posisi, ukuran, dan anak-anaknya dalam struktur Quadtree, yang berisi maksimal empat (4) anak.

Setiap nama metode dalam QuadTreeNode sudah cukup “menjelaskan” apa yang dilakukan dalam QuadTreeNode, sehingga hanya akan dilampirkan isi headernya.

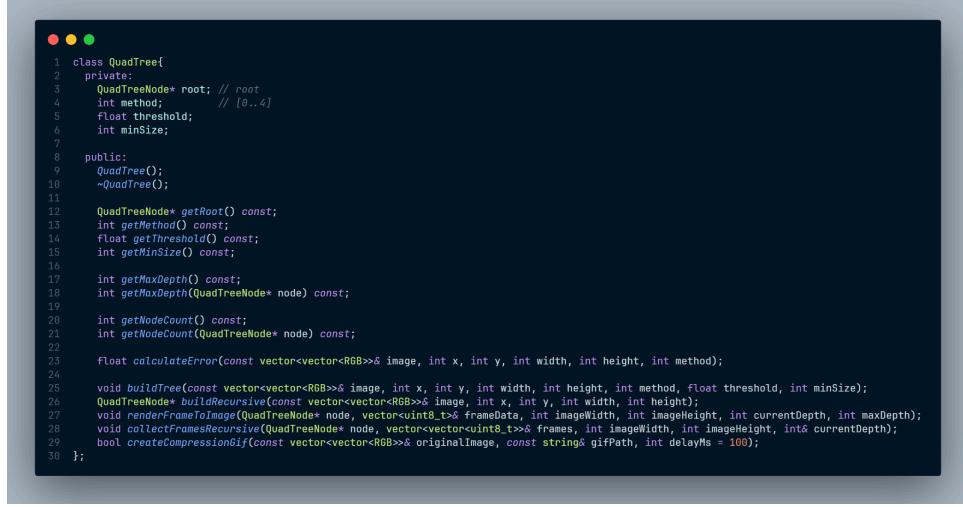


```
1  class QuadTreeNode{
2  private:
3      int width, height, x, y;
4      bool isLeaf;
5      RGB mean;
6      QuadTreeNode* childNode[4];
7
8  public:
9      QuadTreeNode();
10     QuadTreeNode(int x, int y, int width, int height);
11     ~QuadTreeNode();
12
13     int getX() const;
14     int getY() const;
15     pair<int,int> getXY() const;
16     int getWidth() const;
17     int getHeight() const;
18     pair<int, int> getSize() const;
19
20     bool isLeafNode() const;
21     RGB getMean() const;
22     QuadTreeNode* getChildNode(int idx) const;
23
24     void setMean(RGB mean);
25     void setLeaf(bool isLeaf);
26     void setChild(int idx, QuadTreeNode* node);
27 };
```

- Atribut width dan height berfungsi sebagai indikator ukuran blok.
- Atribut x dan y berfungsi sebagai letak blok. Atau, dengan kata lain, menandakan koordinat dalam gambar yang kemudian akan menjadi blok untuk Quadtree.
- Atribut mean berfungsi untuk menyimpan rata-rata warna dalam suatu blok.
- Quadtree, sesuai namanya, memiliki empat (4) anak, sehingga anak dari Quadtree dialokasikan sebanyak empat.

3.6. quadtree.cpp

File ini berisi implementasi dari kelas QuadTree. Berbeda dengan class QuadTreeNode yang merupakan pohon dasar untuk program, QuadTree telah disesuaikan untuk melakukan pemrosesan kompresi gambar, mengimplementasikan Divide And Conquer.



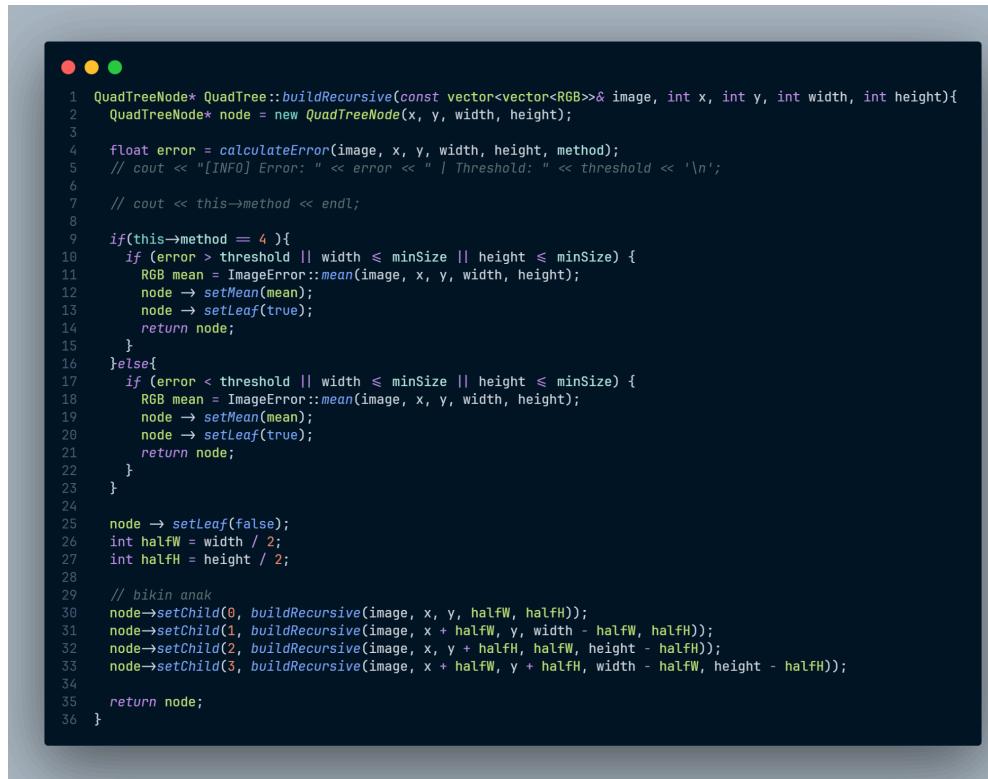
```

1  class QuadTree{
2
3     private:
4         QuadTreeNode* root; // root
5         int method;          // [0..4]
6         float threshold;
7         int minSize;
8
9     public:
10    QuadTree();
11    ~QuadTree();
12    QuadTreeNode* getRoot() const;
13    int getMethod() const;
14    float getThreshold() const;
15    int getMinSize() const;
16    int getMaxDepth() const;
17    int getMaxDepth(QuadTreeNode* node) const;
18    int getNodeCount() const;
19    int getNodeCount(QuadTreeNode* node) const;
20    int getFrameCount() const;
21    int getFrameCount(QuadTreeNode* node) const;
22
23    float calculateError(const vector<vector<RGB>>& image, int x, int y, int width, int height, int method);
24
25    void buildTree(const vector<vector<RGB>>& image, int x, int y, int width, int height, int method, float threshold, int minSize);
26    QuadTreeNode* buildRecursive(const vector<vector<RGB>>& image, int x, int y, int width, int height);
27    void renderFrameToImage(QuadTreeNode* node, vector<uint8_t>& frameData, int imageWidth, int imageHeight, int currentDepth, int maxDepth);
28    void collectFramesRecursive(QuadTreeNode* node, vector<vector<uint8_t>>& frames, int imageWidth, int imageHeight, int& currentDepth);
29    bool createCompressionGif(const vector<vector<RGB>>& originalImage, const string& gifPath, int delayMs = 100);
30 };

```

- Atribut root berisi QuadTreeNode yang akan dipakai untuk proses kompresi
- Atribut method, threshold, dan minSize, dipakai untuk menyimpan informasi yang diperlukan dalam proses kompresi.

Method yang mengimplementasikan Divide And Conquer berada pada method BuildRecursive(). Berikut akan dijelaskan isi dari Method tersebut.



```

1  QuadTreeNode* QuadTree::buildRecursive(const vector<vector<RGB>>& image, int x, int y, int width, int height){
2     QuadTreeNode* node = new QuadTreeNode(x, y, width, height);
3
4     float error = calculateError(image, x, y, width, height, method);
5     // cout << "[INFO] Error: " << error << " | Threshold: " << threshold << '\n';
6
7     // cout << this->method << endl;
8
9     if(this->method == 4){
10        if(error > threshold || width <= minSize || height <= minSize) {
11            RGB mean = ImageError::mean(image, x, y, width, height);
12            node->setMean(mean);
13            node->setLeaf(true);
14            return node;
15        }
16    }else{
17        if(error < threshold || width <= minSize || height <= minSize) {
18            RGB mean = ImageError::mean(image, x, y, width, height);
19            node->setMean(mean);
20            node->setLeaf(true);
21            return node;
22        }
23    }
24
25    node->setLeaf(false);
26    int halfW = width / 2;
27    int halfH = height / 2;
28
29    // bikin anak
30    node->setChild(0, buildRecursive(image, x, y, halfW, halfH));
31    node->setChild(1, buildRecursive(image, x + halfW, y, width - halfW, halfH));
32    node->setChild(2, buildRecursive(image, x, y + halfH, halfW, height - halfH));
33    node->setChild(3, buildRecursive(image, x + halfW, y + halfH, width - halfW, height - halfH));
34
35    return node;
36 }

```

Berikut akan dijelaskan isi dari kode tersebut per-snippet.

```
● ○ ●
1 QuadTreeNode* QuadTree::buildRecursive(const vector<vector<RGB>>& image, int x, int y, int width, int height){
2     QuadTreeNode* node = new QuadTreeNode(x, y, width, height);
3
4     float error = calculateError(image, x, y, width, height, method);
```

Ketika dimulai, node baru akan dibuat sesuai dengan parameter terkait. Error kemudian akan dihitung berdasarkan metode penghitungan error yang dipilih.

```
● ○ ●
1 if (error < threshold || width * height <= minSize) {
2     RGB mean = ImageError::mean(image, x, y, width, height);
3     node -> setMean(mean);
4     node -> setLeaf(true);
5     return node;
6 }
```

Setelah error dihitung, dilakukan pengecekan apakah error masih dibawah threshold atau ukuran telah kurang dari minSize. Jika salah satu syarat ini terpenuhi, maka node tersebut akan menjadi leaf. Kemudian, piksel pada blok ini dirata-ratakan dan rata-ratanya disimpan pada node tersebut.

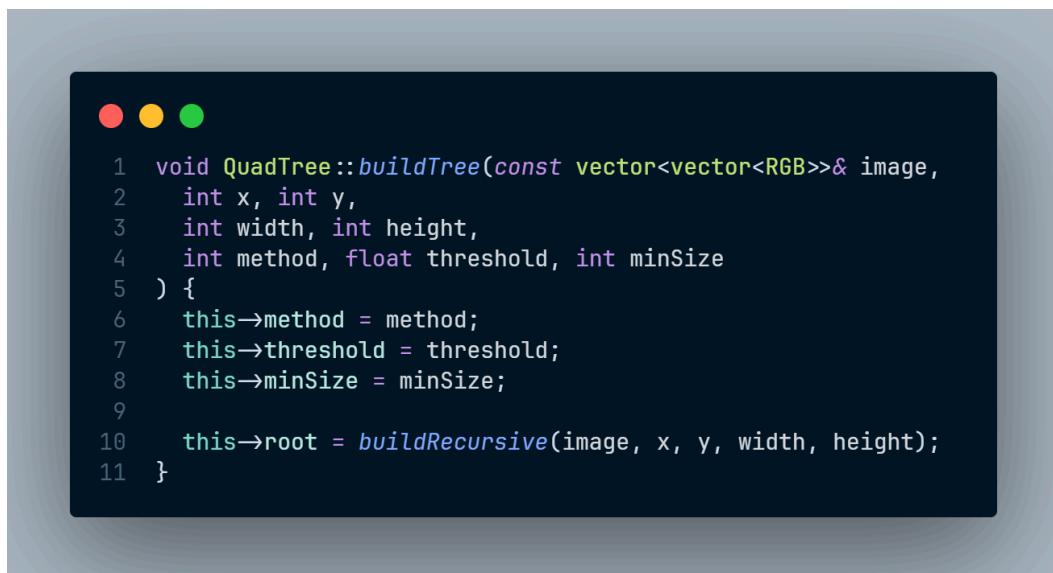
```
● ○ ●
1 node -> setLeaf(false);
2 int halfW = width / 2;
3 int halfH = height / 2;
4
5 // bikin anak
6 node->setChild(0, buildRecursive(image, x, y, halfW, halfH));
7 node->setChild(1, buildRecursive(image, x + halfW, y, width - halfW, halfH));
8 node->setChild(2, buildRecursive(image, x, y + halfH, halfW, height - halfH));
9 node->setChild(3, buildRecursive(image, x + halfW, y + halfH, width - halfW, height - halfH));
10
11 return node;
```

Jika tidak sesuai dengan kondisi if di atas, algoritma dilanjutkan dengan menandai bahwa simpul saat ini (node) bukanlah daun, karena ia akan dibagi menjadi sub-blok. Selanjutnya, program menghitung ukuran setengah dari lebar dan tinggi blok saat ini, yang akan digunakan untuk menentukan ukuran masing-masing anak node.

Setelah itu, node dibagi menjadi empat kuadran yang masing-masing merepresentasikan sub-blok gambar: kuadran kiri atas, kanan atas, kiri bawah, dan kanan bawah. Keempat anak ini dibuat dengan memanggil fungsi buildRecursive() secara rekursif, dengan parameter posisi (x, y) dan ukuran (width, height) yang telah disesuaikan untuk masing-masing kuadran. Hasil dari masing-masing pemanggilan rekursif kemudian diset sebagai anak dari node saat ini dengan menggunakan setChild(index, childNode).

Dengan cara ini, pohon Quadtree dibangun secara top-down, yakni setiap node akan terus membagi dirinya menjadi empat sub-blok sampai kondisi penghentian tercapai (misalnya ukuran minimum blok atau nilai error di bawah threshold).

Untuk memanggil buildRecursive() pada root, cukup memanggil buildTree().



```
1 void QuadTree::buildTree(const vector<vector<RGB>>& image,
2     int x, int y,
3     int width, int height,
4     int method, float threshold, int minSize
5 ) {
6     this->method = method;
7     this->threshold = threshold;
8     this->minSize = minSize;
9
10    this->root = buildRecursive(image, x, y, width, height);
11 }
```

3.7. image_error.cpp

File ini mengimplementasikan berbagai **metode perhitungan error** yang digunakan dalam proses kompresi gambar berbasis Quadtree. Fungsi-fungsi ini berperan dalam menilai apakah suatu blok gambar dianggap seragam atau tidak. Jika nilai error di atas ambang batas (threshold), maka blok tersebut akan dibagi lebih lanjut.

Fungsi mean()



```
1 RGB ImageError::mean(const vector<vector<RGB>>& image, int x, int y, int width, int height) {
2     if (width <= 0 || height <= 0) {
3         return {0, 0, 0};
4     }
5
6     long sumR = 0, sumG = 0, sumB = 0;
7     int count = 0;
8
9     for (int j = y; j < y + height && j < image.size(); j++) {
10        for (int i = x; i < x + width && i < image[j].size(); i++) {
11            sumR += image[j][i].r;
12            sumG += image[j][i].g;
13            sumB += image[j][i].b;
14            count++;
15        }
16    }
17
18    if (count == 0) return {0, 0, 0};
19
20    // Ensure we don't exceed RGB bounds (0-255)
21    return {
22        static_cast<uint8_t>(sumR / count),
23        static_cast<uint8_t>(sumG / count),
24        static_cast<uint8_t>(sumB / count)
25    };
26 }
```

Fungsi ini menghitung rata-rata nilai RGB dari sebuah blok gambar. Nilai ini akan digunakan sebagai representasi warna blok dalam visualisasi hasil kompresi, serta menjadi dasar untuk penghitungan error (seperti variansi dan MAD). Jika blok kosong atau tidak valid, fungsi ini akan mengembalikan warna hitam {0, 0, 0}.

Fungsi variance()



```
1 float ImageError::variance(const vector<vector<RGB>>& image, int x, int y, int width, int height){
2     long long R = 0, G = 0, B = 0;
3     int dr = 0, dg = 0, db = 0;
4     float dim = float(width*height);
5     RGB imgMean = mean(image, x, y, width, height);
6
7     for (int j = y; j < y + height; j++){
8         for (int i = x; i < x + width; i++){
9             dr = image[j][i].r - imgMean.r;
10            dg = image[j][i].g - imgMean.g;
11            db = image[j][i].b - imgMean.b;
12            R += dr*dr;
13            G += dg*dg;
14            B += db*db;
15        }
16    }
17
18    return (R + G + B) / (3.0f*dim);
19 }
20 }
```

Menghitung variansi dari sebuah blok gambar. Variansi di sini menunjukkan seberapa tersebar nilai RGB piksel terhadap nilai rata-rata blok. Semakin tinggi nilai variansi, semakin tidak seragam blok tersebut. Nilai error dihitung dengan menjumlahkan variansi dari masing-masing channel (R, G, dan B), lalu dibagi 3.

Fungsi mad()



```
1 float ImageError::mad(const vector<vector<RGB>>& image, int x, int y, int width, int height){
2     long long R = 0, G = 0, B = 0;
3     float dim = float(width * height);
4     if (dim == 0) return 0.0f;
5
6     RGB imgMean = mean(image, x, y, width, height);
7
8     for (int j = y; j < y + height; j++){
9         for (int i = x; i < x + width; i++){
10             R += abs(image[j][i].r - imgMean.r);
11             G += abs(image[j][i].g - imgMean.g);
12             B += abs(image[j][i].b - imgMean.b);
13         }
14     }
15
16     return (R + G + B) / (3.0f * dim);
17 }
```

Fungsi ini menghitung rata-rata penyimpangan absolut nilai piksel terhadap nilai rata-rata warna blok, tidak seperti varians yang dikuadratkan terlebih dahulu. Nilai akhirnya adalah rata-rata dari penyimpangan pada channel R, G, dan B.

Fungsi maxDiff()



```
1 float ImageError::maxDiff(const vector<vector<RGB>>& image, int x, int y, int width, int height) {
2     int minR = 255, maxR = 0;
3     int minG = 255, maxG = 0;
4     int minB = 255, maxB = 0;
5
6     for (int j = y; j < y + height; ++j) {
7         for (int i = x; i < x + width; ++i) {
8             const RGB& pixel = image[j][i];
9
10            if (pixel.r < minR) minR = pixel.r;
11            if (pixel.r > maxR) maxR = pixel.r;
12
13            if (pixel.g < minG) minG = pixel.g;
14            if (pixel.g > maxG) maxG = pixel.g;
15
16            if (pixel.b < minB) minB = pixel.b;
17            if (pixel.b > maxB) maxB = pixel.b;
18        }
19    }
20
21    int dR = maxR - minR;
22    int dG = maxG - minG;
23    int dB = maxB - minB;
24
25    return (dR + dG + dB) / 3.0f;
26 }
```

Fungsi ini mencari selisih antara piksel maksimum dan minimum untuk masing-masing channel warna (R, G, B). Hasil akhirnya adalah rata-rata dari ketiga selisih tersebut. Metode ini sederhana dan sangat sensitif terhadap outlier.

Fungsi entropy()



```
1 float ImageError::entropy(const vector<vector<RGB>>& image, int x, int y, int width, int height) {
2     // histogram R, G, B
3     map<int, int> histR, histG, histB;
4     int totalPixels = width * height;
5
6     // hitung "tinggi" histogram
7     for (int j = y; j < y + height; j++) {
8         for (int i = x; i < x + width; i++) {
9             int r = image[j][i].r;
10            int g = image[j][i].g;
11            int b = image[j][i].b;
12
13            histR[r]++;
14            histG[g]++;
15            histB[b]++;
16        }
17    }
18
19    // Entropi histogram
20    auto calculateEntropy = [] (const map<int, int>& hist, int totalPixels) → float {
21        float entropy = 0.0f;
22        for (const auto& entry : hist) {
23            float probability = float(entry.second) / totalPixels;
24            if (probability > 0) {
25                entropy -= probability * log2(probability);
26            }
27        }
28        return entropy;
29    };
29
30    float entropyR = calculateEntropy(histR, totalPixels);
31    float entropyG = calculateEntropy(histG, totalPixels);
32    float entropyB = calculateEntropy(histB, totalPixels);
33
34    return (entropyR + entropyG + entropyB) / 3.0f;
35 }
```

Fungsi ini menghitung entropi (jumlah informasi) dari blok gambar berdasarkan histogram intensitas warna R, G, dan B. Semakin beragam nilai warna dalam blok, semakin tinggi entropinya. Histogram digunakan untuk menghitung distribusi probabilitas, lalu diterapkan rumus entropi Shannon:

$$H_c = - \sum_{i=1}^N P_c(i) \log_2(P_c(i))$$

di mana $P(i)$ adalah probabilitas munculnya warna i .

3.8. image_utils.cpp

Supaya gambar dapat diolah, gambar terlebih dahulu diubah menjadi matrix. Oleh karena itu, tools ini perlu diimplementasikan. Selain itu, gambar juga harus dapat dibentuk dari tree yang sudah ada. Lebih lengkapnya, berikut isi dari image_utils.hpp

```
1  namespace ImageUtils{
2      vector<vector<RGB>> imageToMatrix(const string& filename);
3      void matrixToImage(const vector<vector<RGB>>& image, const string& filename);
4
5      void fillCompressedImage(QuadTreeNode* node, vector<vector<RGB>>& compressedImage);
6
7      QuadTree compressImage(IO streams);
8  }
```

Berikut akan dijelaskan isi dari file tersebut.

Fungsi `imageToMatrix()`

```
1  vector<vector<RGB>> ImageUtils::imageToMatrix(const string& filename){
2      fipImage img;
3      img.load(filename.c_str());
4      if (!img.isValid()) {
5          cout << "Could not read the image: " << filename << endl;
6          exit(1);
7      }
8
9      int x = img.getWidth();
10     int y = img.getHeight();
11     vector<vector<RGB>> image(y, vector<RGB>(x));
12
13     for (int i = 0; i < y; i++){
14         for (int j = 0 ; j < x; j++){
15             RGBQUAD color;
16             if (img.getPixelColor(j, i, &color)) { // Perbaikan indeks
17                 image[i][j].r = color.rgbRed;
18                 image[i][j].g = color.rgbGreen;
19                 image[i][j].b = color.rgbBlue;
20             }
21         }
22     }
23
24     return image;
25 }
```

Fungsi ini bertugas untuk mengubah gambar dari file menjadi representasi matriks 2D piksel RGB. Proses diawali dengan memuat gambar menggunakan `fipImage`. Jika gambar tidak valid atau gagal dimuat, program akan berhenti dengan pesan error. Setelah itu, dimensi gambar diambil, dan struktur `vector<vector<RGB>>` dibuat untuk menyimpan data warna tiap piksel. Lalu, dilakukan iterasi dua tingkat (baris dan kolom) untuk membaca setiap piksel menggunakan

getPixelColor() dan menyimpannya dalam bentuk struktur RGB. Perlu diperhatikan bahwa koordinat yang digunakan di getPixelColor(j, i) mengikuti urutan kolom dulu lalu baris, sesuai spesifikasi FreeImage.

Fungsi matrixToImage()



```
1 void ImageUtils::matrixToImage(const vector<vector<RGB>>& image, const string& filename) {
2     if (image.empty() || image[0].empty()) { // Cek agar tidak segmentation fault
3         cout << "Error: Image matrix is empty!" << endl;
4         return;
5     }
6
7     int height = image.size();
8     int width = image[0].size();
9
10    fipImage img(FIT_BITMAP, width, height, 24);
11    if (!img.isValid()) {
12        cout << "Failed to create image!" << endl;
13        return;
14    }
15
16    for (int y = 0; y < height; y++) {
17        for (int x = 0; x < width; x++) {
18            RGBQUAD color;
19            color.rgbRed = image[y][x].r;
20            color.rgbGreen = image[y][x].g;
21            color.rgbBlue = image[y][x].b;
22            img.setPixelColor(x, y, &color);
23        }
24    }
25
26    if (!img.save(filename.c_str())) {
27        cout << "Failed to save image: " << filename << endl;
28    }
29 }
```

Kebalikan dari imageToMatrix(), fungsi ini mengubah matriks 2D RGB menjadi file gambar dan menyimpannya ke disk. Fungsi ini terlebih dahulu memvalidasi bahwa input matriks tidak kosong agar terhindar dari segmentation fault. Setelah itu, gambar baru dibuat menggunakan fipImage, dan diisi dengan warna dari matriks piksel melalui setPixelColor. Format gambar default yang digunakan adalah 24-bit bitmap. Setelah seluruh piksel dipetakan, gambar disimpan menggunakan save(). Jika proses penyimpanan gagal, akan ditampilkan pesan error.

Fungsi fillCompressedImage()

```

1 void ImageUtils::fillCompressedImage(QuadTreeNode* node, vector<vector<RGB>>& image){
2     if (node->isLeafNode()) {
3         int x = node->getX();
4         int y = node->getY();
5         int width = node->getWidth();
6         int height = node->getHeight();
7         RGB mean = node->getMean();
8
9         for (int i = y; i < min(y + height, (int)image.size()); ++i) {
10             for (int j = x; j < min(x + width, (int)image[0].size()); ++j) {
11                 image[i][j] = mean;
12             }
13         }
14     } else {
15         for (int k = 0; k < 4; ++k) {
16             fillCompressedImage(node->getChildNode(k), image);
17         }
18     }
19 }
```

Fungsi ini digunakan untuk merekonstruksi hasil kompresi berdasarkan pohon Quadtree. Proses dilakukan secara rekursif. Jika simpul yang sedang diproses adalah leaf node, maka seluruh area yang diwakili node tersebut akan diisi dengan warna rata-rata (mean) dari node. Jika bukan leaf, maka fungsi dipanggil ulang secara rekursif untuk masing-masing dari 4 anak node. Dengan demikian, fungsi ini menyusun ulang gambar hasil kompresi berdasarkan struktur Quadtree yang telah dibentuk sebelumnya.

Fungsi compressImage()

```

1 QuadTree ImageUtils::compressImage(Io streams){
2     vector<vector<RGB>> image = imageToMatrix(streams.imageSrcPath.string());
3     QuadTree qt;
4     qt.buildTree(image, 0, 0, image[0].size(), image.size(), streams.method-1, streams.VAR_THRESHOLD, streams.MIN_BLOCK_SIZE);
5     vector<vector<RGB>> compressedImage(image.size(), vector<RGB>(image[0].size()));
6     fillCompressedImage(qt.getRoot(), compressedImage);
7     matrixToImage(compressedImage, streams.imageDestPath.string());
8
9     return qt;
10 }
```

Ini adalah fungsi utama yang menggabungkan semua proses: membaca gambar, membangun pohon Quadtree, mengisi gambar hasil kompresi, dan menyimpannya. Langkah pertama adalah memanggil imageToMatrix() untuk membaca gambar asli. Kemudian pohon dibentuk dengan buildTree(), di mana parameter penting seperti metode error, threshold, dan ukuran blok minimum ikut digunakan. Setelah pohon terbentuk, dibuat matriks kosong untuk menampung hasil kompresi, yang diisi lewat fillCompressedImage(). Terakhir, hasilnya disimpan sebagai gambar dengan matrixToImage(). Fungsi ini mengembalikan

objek QuadTree supaya statistiknya dapat ditampilkan di layar output oleh kelas lain.

3.9. Bonus

3.9.1. SSIM (dalam image_error.cpp)

Salah satu bonus yang diajukan dalam tugas ini adalah SSIM. Telah dibuat implementasi dari SSIM berdasarkan Bab 2 sebagai berikut.

```

1 // Original
2 const vector<vector<RGB>> original,
3 const vector<vector<RGB>> compressed,
4 int x, int y, int width, int height,
5 char channel, float C1, float C2
6 ) {
7     // Original block
8     float meanx = 0.0f;
9     float varx = 0.0f;
10
11    // Compressed block
12    float meany = 0.0f;
13    float vary = 0.0f;
14
15    int pixelCount = width * height;
16
17    //Means
18    for (int j = 0; j < height; j++) {
19        for (int i = 0; i < width; i++) {
20            float valX, valY;
21
22            if (channel == 'r') {
23                valX = original[j][i+x].r;
24                valY = compressed[j][i].r;
25            } else if (channel == 'g') {
26                valX = original[j][i+x].g;
27                valY = compressed[j][i].g;
28            } else { // 'b'
29                valX = original[j][i+x].b;
30                valY = compressed[j][i].b;
31            }
32
33            meanx += valX;
34            meany += valY;
35        }
36
37        meanx /= pixelCount;
38        meany /= pixelCount;
39    }
}

```



```

1 // Variances
2 for (int j = 0; j < height; j++) {
3     for (int i = 0; i < width; i++) {
4         float valX, valY;
5
6         if (channel == 'r') {
7             valX = original[j][i+x].r;
8             valY = compressed[j][i].r;
9         } else if (channel == 'g') {
10            valX = original[j][i+x].g;
11            valY = compressed[j][i].g;
12        } else { // 'b'
13            valX = original[j][i+x].b;
14            valY = compressed[j][i].b;
15        }
16
17        float dx = valX - meanx;
18        float dy = valY - meany;
19
20        varx += dx * ox;
21        vary += dy * oy;
22    }
23
24    varx /= pixelCount;
25    vary /= pixelCount;
26
27    // cout << "ox2: " << ox2 << " oy2: " << oy2 << endl;
28    // cout << "px: " << px << " py: " << py << endl;
29
30    ox = (meanx * meanx + meany * meany + C1) / (varx + vary + C2);
31    oy = (meanx * meany + meany * meany + C1) / (varx + vary + C2);
32
33    float numerator = (2 * ox * oy * C1) + (C2);
34    float denominator = (meanx * meanx + meany * meany + C1) + (varx + vary + C2);
35
36    if (denominator < 0.0001f) return 0.8f; // Avoid division by very small numbers
37    float rawResult = numerator / denominator;
38    return rawResult;
39 }

```

Fungsi ini menghitung nilai SSIM per channel warna (R, G, atau B) pada dua blok gambar. Parameter yang diberikan mencakup dua matriks gambar (original dan compressed), posisi dan ukuran blok (x, y, width, height), serta konstanta stabilisasi (C1, C2) dan karakter channel ('r', 'g', atau 'b').

Langkah pertama adalah menghitung rata-rata intensitas piksel untuk blok asli dan blok kompresi, berdasarkan channel yang dipilih. Nilai mean dari kedua blok tersebut disimpan dalam meanx dan meany. Setelah mean diperoleh, fungsi kemudian menghitung variansi dari tiap blok dengan

mengiterasi piksel lagi dan mengakumulasi deviasi kuadrat dari masing-masing nilai terhadap rata-ratanya, lalu disimpan di varx dan vary.

Setelah mendapatkan mean dan variansi, rumus SSIM per channel dihitung menggunakan formula:

$$SSIM_{c(x,y)} = \frac{(2\mu_{x,c}\mu_{y,c} + C_1)(C_2)}{(\mu_{x,c}^2 + \mu_{y,c}^2 + C_1)(\sigma_{x,c}^2 + C_2)}$$

Karena pada kasus kompresi ini tidak digunakan nilai kovarian (σ_{xy}) secara eksplisit, rumus disederhanakan dan hanya menggunakan mean dan variansi dari masing-masing blok. Jika penyebut (denominator) terlalu kecil (mendekati nol), maka fungsi mengembalikan nilai default 0.8f untuk menghindari pembagian dengan angka sangat kecil. Nilai akhir SSIM disimpan sebagai rawResult.



```
1 float ImageError::ssim(const vector<vector<RGB>>& image, int x, int y, int width, int height) {
2     // Constants to stabilize division with weak denominator
3     const float C1 = (0.01f * 255) * (0.01f * 255);
4     const float C2 = (0.03f * 255) * (0.03f * 255);
5
6     // Get mean of the block
7     RGB blockMean = mean(image, x, y, width, height);
8
9     // Create a block with uniform color (the compressed version)
10    vector<vector<RGB>> compressedBlock(height, vector<RGB>(width, blockMean));
11
12    // Calculate SSIM for each channel
13    float ssimR = calculateChannelSSIM(image, compressedBlock, x, y, width, height, 'r', C1, C2);
14    float ssimG = calculateChannelSSIM(image, compressedBlock, x, y, width, height, 'g', C1, C2);
15    float ssimB = calculateChannelSSIM(image, compressedBlock, x, y, width, height, 'b', C1, C2);
16
17    // Equal weights for each channel
18    float wR = 0.33f;
19    float wG = 0.33f;
20    float wB = 0.33f;
21
22    float result = wR * ssimR + wG * ssimG + wB * ssimB;
23
24    // Ensure the result is between 0 and 1
25    return max(0.0f, min(1.0f, result));
26 }
```

Fungsi ssim() adalah pembungkus utama yang menghitung nilai SSIM total dari satu blok dengan mempertimbangkan ketiga channel RGB. Fungsi ini diawali dengan menetapkan dua konstanta C1 dan C2 yang diperlukan untuk menjaga stabilitas saat melakukan pembagian dengan penyebut yang mendekati nol.

Pertama-tama, fungsi menghitung rata-rata warna dari blok (menggunakan fungsi mean()) dan membentuk versi blok yang dikompresi (compressedBlock) yang seluruhnya diisi dengan warna rata-rata tersebut.

Hal ini meniru hasil akhir kompresi yang menyederhanakan blok menjadi satu warna rata-rata.

Kemudian, fungsi calculateChannelSSIM() dipanggil untuk masing-masing channel (r, g, b) untuk mendapatkan nilai SSIM-nya. Nilai-nilai SSIM per channel tersebut kemudian digabung menggunakan bobot yang sama rata (0.33f untuk masing-masing channel) untuk menghasilkan nilai akhir result. Nilai ini kemudian dijepit agar tetap berada dalam rentang [0, 1] menggunakan max(0.0f, min(1.0f, result)).

3.9.2. GIF Proses Kompresi (gif_maker.cpp)

Bonus selanjutnya yang dikerjakan adalah GIF dari proses kompresi gambar. Kami menggunakan gif.h oleh Charlie Tangora. Kemudian, dibungkus menggunakan gif_maker.cpp



```
1 #include "header/gif_maker.hpp"
2 #include "header/gif.h"
3 #include <cstdio>
4
5 bool createAnimatedGif(const std::vector<std::vector<uint8_t>> &frames,
6                         int width, int height,
7                         int delayMs,
8                         const std::string &outputPath)
9 {
10     if (frames.empty() || width <= 0 || height <= 0)
11         return false;
12
13     GifWriter writer = {};
14     if (!GifBegin(&writer, outputPath.c_str(), width, height, delayMs))
15         return false;
16
17     // Each frame is assumed to be 3 channels (RGB).
18     // gif.h expects 4 channels per pixel (RGBA), so we convert.
19     for (const auto &frame : frames)
20     {
21         std::vector<uint8_t> rgba(width * height * 4);
22         for (int i = 0; i < width * height; i++)
23         {
24             rgba[i * 4 + 0] = frame[i * 3 + 2];
25             rgba[i * 4 + 1] = frame[i * 3 + 1];
26             rgba[i * 4 + 2] = frame[i * 3 + 0];
27             rgba[i * 4 + 3] = 255;
28         }
29         if (!GifWriteFrame(&writer, rgba.data(), width, height, delayMs))
30         {
31             GifEnd(&writer);
32             return false;
33         }
34     }
35     GifEnd(&writer);
36     return true;
37 }
```

Fungsi createAnimatedGif() bertugas untuk membuat file GIF animasi dari sekumpulan frame gambar. Setiap frame disimpan dalam bentuk

`vector<uint8_t>` yang merepresentasikan nilai warna piksel dalam format RGB (3 channel). Parameter fungsi ini mencakup daftar frame (frames), dimensi gambar (width, height), delay antar frame (delayMs), dan jalur file output (outputPath).

Pertama-tama, fungsi memverifikasi bahwa daftar frame tidak kosong dan dimensi gambar valid. Jika tidak memenuhi syarat, fungsi akan mengembalikan false. Kemudian, objek GifWriter diinisialisasi dan fungsi GifBegin() dipanggil untuk memulai proses penulisan file GIF ke outputPath. Jika inisialisasi gagal, proses dihentikan.

Setiap frame diproses dalam loop. Karena library gif.h mengharuskan setiap piksel terdiri dari 4 channel (RGBA), sedangkan input hanya 3 channel (RGB), maka fungsi melakukan konversi manual. Untuk setiap piksel, nilai R, G, dan B diambil dari frame dan disisipkan ke dalam rgba, serta nilai alpha (A) diatur ke 255 (penuh opacity). Susunan warna dalam gif.h menggunakan format RGBA, tetapi indeks pengisian dilakukan dalam urutan terbalik (R = index 2, G = index 1, B = index 0) karena urutan input dari frame.

Setelah konversi selesai, GifWriteFrame() digunakan untuk menuliskan frame ke file GIF. Jika terjadi kegagalan saat menulis frame, maka writer ditutup dan fungsi mengembalikan false. Setelah semua frame berhasil diproses, GifEnd() dipanggil untuk menutup writer dan fungsi mengembalikan true sebagai tanda keberhasilan.

3.10. Lainnya

3.10.1. header/

Folder header/ berisi semua file header (.hpp) yang mendefinisikan struktur dan fungsi yang digunakan di berbagai bagian program. Berikut adalah strukturnya.

```
header/
└── FreeImage.h
└── FreeImagePlus.h
└── gif.h
└── gif_maker.hpp
└── image_error.hpp
└── image_utils.hpp
└── io.hpp
└── quadtree.hpp
```

3.10.2. lib/

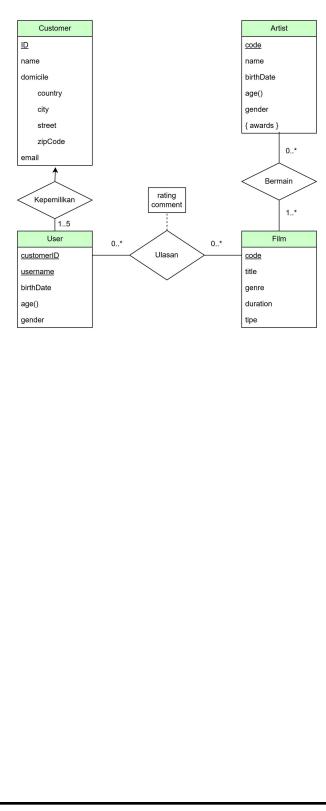
Folder lib/ berisi library eksternal atau file dependensi pihak ketiga yang diperlukan agar program bisa berjalan. Isi folder lib sendiri adalah library FreeImage yang diperlukan agar program dapat dikompilasikan pada Windows. Berikut adalah strukturnya.

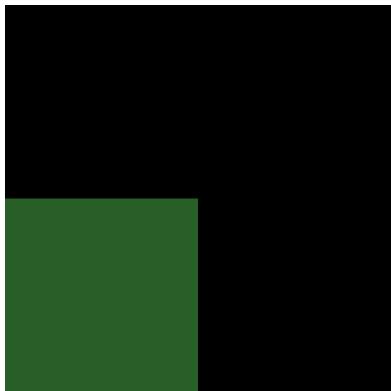


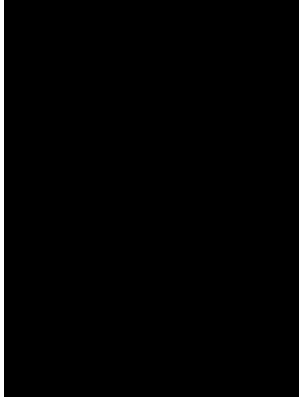
4. Uji Coba

Program diuji coba dengan menggunakan file berformat .jpg dan .png. Sample dan hasilnya dapat dilihat pada folder /test.

Asli	Hasil	Keterangan
	 <p>Execution time: 302.667 ms Original image size: 118492 bytes Compressed image size: 94357 bytes Compression percentage: 20.3685% Max tree depth: 10 Total nodes: 20337</p>	Variance Threshold = 500 Min Size = 8

	<p>Execution time: 771.921 ms Original image size: 222615 bytes Compressed image size: 192675 bytes Compression percentage: 13.4492% Max tree depth: 10 Total nodes: 81565</p>	Maximum Difference Threshold =150 Min Size = 8
	<p>Execution time: 4210.42 ms Original image size: 771150 bytes Compressed image size: 437434 bytes Compression percentage: 43.2751% Max tree depth: 11 Total nodes: 50649</p>	Variance Threshold = 500 Min Size = 10
	 <p>Execution time: 251.688 ms Original image size: 135400 bytes</p>	Metode Maximum Difference dengan threshold 20 dan minblock 4.

	<p>Compressed image size: 157629 bytes Compression percentage: -16.4173% Max tree depth: 10 Total nodes: 225913</p>	
	 	<p>Variance Threshold = 500 Min Size = 8</p>
	<p>Execution time: 136.479 ms Original image size: 498062 bytes Compressed image size: 30489 bytes Compression percentage: 93.8785% Max tree depth: 9 Total nodes: 6841</p>	
		<p>Entropy Threshold = 3 Min Size = 8</p>
	<p>Execution time: 1181.9 ms</p>	

	<p>Original image size: 83680 bytes Compressed image size: 51354 bytes Compression percentage: 38.6305% Max tree depth: 8 Total nodes: 75345</p>	
	 <p>Execution time: 16.0645 ms Original image size: 20512 bytes Compressed image size: 188 bytes Compression percentage: 99.0835% Max tree depth: 0 Total nodes: 1</p>	<p>Gambar memiliki format grayscale/black and white. Hal ini karena pembacaan error dianggap sudah dibawah threshold, menyebabkan gambar yang muncul hanyalah gambar dengan pixel hitam.</p>

Untuk gif, supaya dapat bergerak, dapat dilihat langsung di repository (Karena pdf tidak mendukung gif)

5. Analisis

5.1. Analisis Kompleksitas Waktu

Kompleksitas waktu dari algoritma kompresi gambar menggunakan Quadtree dapat diperkirakan sebesar $O(WH \times \log \max(W, H))$, dengan W dan H masing-masing merupakan lebar dan tinggi gambar. Estimasi ini berasal dari dua faktor utama: jumlah total piksel yang diproses dan kedalaman maksimum dari rekursi pembagian blok.

Dalam setiap pemrosesan blok, algoritma menghitung error berdasarkan semua piksel di dalam blok tersebut. Dengan demikian, seluruh piksel gambar sebanyak $W \times H$ berpotensi untuk dibaca selama proses. Selain itu, karena Quadtree membagi blok menjadi empat bagian secara rekursif, proses pembagian ini akan terus dilakukan hingga mencapai ukuran blok minimum. Secara teoritis, proses pembagian ini memiliki kedalaman maksimum sebesar $\log_2 W$ untuk arah horizontal dan $\log_2 H$ untuk arah vertikal. Maka, batas atas kedalaman rekursi yang mungkin terjadi adalah $\log_2 \max(W, H)$.

Dengan menggabungkan kedua aspek tersebut, total kerja komputasi menjadi $O(WH \times \log \max(W, H))$. Artinya, semakin besar dimensi gambar, maka waktu eksekusi akan bertambah secara proporsional terhadap jumlah piksel dan secara logaritmik terhadap resolusi tertingginya. Estimasi kompleksitas ini berlaku pada kasus terburuk, di mana pembagian blok dilakukan terus-menerus hingga mencapai batas terkecil.

5.2. Analisis Kompleksitas Ruang

Kompleksitas ruang dari algoritma kompresi gambar berbasis Quadtree dapat diperkirakan sebesar $O(N)$, di mana N adalah jumlah node yang dibentuk dalam struktur pohon Quadtree. Jumlah node ini bergantung pada seberapa banyak blok gambar yang dibagi selama proses rekursif. Dalam kasus terburuk — misalnya gambar sangat bervariasi sehingga seluruh blok dibagi hingga ukuran minimum — jumlah node bisa mendekati $O(WH)$ karena setiap piksel atau kelompok kecil piksel akan direpresentasikan oleh satu simpul daun (leaf).

Setiap node pada Quadtree menyimpan informasi posisi (x, y), ukuran (width, height), status apakah ia leaf atau bukan, rata-rata warna blok, dan maksimal 4 pointer ke anak. Karena node-node ini dibuat secara dinamis selama rekursi, maka penggunaan ruang akan bertambah secara proporsional terhadap jumlah node. Oleh karena itu, kompleksitas ruang untuk menyimpan struktur pohon adalah $O(N)$, yang dalam kasus terburuk bisa mencapai $O(WH)$.

Selain itu, program juga menggunakan dua buah matriks piksel selama eksekusi: satu untuk menyimpan gambar asli (originalImage) dan satu lagi untuk menyimpan hasil kompresi (compressedImage). Masing-masing memiliki ukuran $O(WH)$. Maka, total kompleksitas ruang program dapat ditulis sebagai $O(WH + N)$

Karena N dalam kasus terburuk juga bisa sebesar WH , maka penggunaan ruang secara keseluruhan tetap berskala linear terhadap ukuran gambar.

5.3. Analisis Kompresi

- Semakin **rendah threshold**, semakin tinggi kualitas gambar hasil kompresi karena lebih banyak detail dipertahankan, tetapi ukuran pohon dan waktu eksekusi meningkat.
- Penggunaan **SSIM** sebagai metode error cenderung memberikan **kompresi berkualitas lebih baik secara visual**, meskipun lebih mahal secara perhitungan.
- Pengaturan **minimum block size** yang lebih besar menurunkan waktu eksekusi dan simpul, tetapi berisiko menghilangkan detail kecil.

5.4. Analisis Metode

Pemilihan metode dan pengaturan threshold sangat memengaruhi hasil akhir kompresi—baik dari sisi kualitas visual maupun ukuran file. Berikut adalah analisis masing-masing metode beserta rekomendasi nilai threshold berdasarkan hasil percobaan praktikal:

1. Variance

Metode ini mengukur seberapa besar penyebaran nilai RGB dalam suatu blok terhadap rata-rata warnanya. Semakin besar variansi, semakin tidak seragam blok tersebut. Range praktikal metode ini berada pada threshold 500 – 2000. Nilai variansi umumnya tinggi jika blok memiliki perbedaan warna mencolok. Threshold terlalu rendah akan memicu terlalu banyak pembagian dan memperlambat proses. Threshold sekitar 1000 memberikan hasil yang seimbang antara kualitas dan ukuran file.

Metode ini cocok untuk gambar alami atau foto dengan gradasi warna yang halus.

2. Mean Absolute Deviation (MAD)

Metode ini menghitung rata-rata selisih absolut tiap piksel terhadap rata-rata blok. Lebih stabil terhadap outlier dibanding variansi. Range praktikal thresholdnya berada di 15 – 60. Hal ini karena MAD berbasis selisih absolut (bukan kuadrat), nilai yang dihasilkan cenderung lebih kecil dari variansi. Threshold di atas 50 biasanya sudah cukup longgar untuk menghasilkan kompresi yang signifikan.

Metode ini cocok untuk gambar dengan batas warna yang lebih tajam (misalnya gambar kartun atau diagram).

3. Maximum Pixel Difference

Mengukur selisih antara piksel maksimum dan minimum di dalam blok. Metode ini sangat sensitif terhadap perbedaan ekstrem. Range praktikalnya berada pada

25 – 100. Hal ini karena threshold kecil seperti 30–50 sudah cukup untuk menandai blok tidak seragam. Threshold tinggi bisa membuat terlalu banyak blok dianggap seragam padahal tidak.

Metode ini kurang cocok untuk gambar alami dengan noise halus, tapi sangat efisien untuk gambar sederhana seperti line art.

4. Entropy

Mengukur tingkat keragaman distribusi warna dalam blok, berdasarkan histogram frekuensi piksel. Semakin tinggi entropi, semakin kompleks blok tersebut. Range praktikalnya berada pada 3.5 – 6.0. Hak ini karena entropi maksimum untuk 8-bit channel adalah 8 (Dari $\log 256 = 8$). Gambar dengan entropi ≥ 5 dianggap sangat variatif. Threshold 4.5 adalah titik tengah yang baik untuk menjaga kualitas. Karena entropy mengukur tingkat distribusi warna, entropy cocok untuk gambar dengan banyak detail halus, seperti tekstur atau pemandangan.

5. Structural Similarity Index (SSIM) (Bonus)

Metode ini membandingkan blok terhadap versi terkompresinya dengan mempertimbangkan luminansi, kontras, dan struktur secara bersamaan. Nilainya berada dalam rentang 0 sampai 1, dengan 0 berarti identik sempurna (Karena kami sudah inverskan di implementasi ini agar selaras dengan metode yang lain). Range praktikalnya berada di 0.05 – 0.25. Tidak seperti metode lain, threshold SSIM semakin tinggi artinya semakin ketat. Threshold 0.9 artinya hanya blok yang sangat mirip yang boleh dipertahankan.

SSIM sangat cocok untuk menjaga kualitas visual dan digunakan ketika preservasi detail penting, namun komputasinya paling mahal.

6. Lampiran

Repositori: [Tucil2_13523098_13523149](#)

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 dan laporan dibuat (kelompok) sendiri	✓	