

LAPORAN TUGAS KECIL #2
KOMPRESI GAMBAR DENGAN METODE QUADTREE

IF2211– Strategi Algoritma



Dosen:

Dr.Nur Ulfa Maulidevi, S.T, M.Sc.

Farrell Jabaar Altafataza 10122057

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

Daftar Isi	2
Penjelasan Algoritma	3
Isi Program.....	4
Percobaan Algoritma.....	9
Analisis Kompleksitas.....	12
Lampiran	13

Penjelasan Algoritma

Algoritma Divide and Conquer adalah strategi pemecahan masalah yang memecah suatu persoalan besar menjadi beberapa bagian lebih kecil, menyelesaikan tiap bagian secara rekursif, lalu menggabungkan hasil-hasilnya untuk membentuk solusi akhir. Algoritma ini umumnya dibagi menjadi 3 bagian utama., yaitu *divide*, *conquer*, dan *combine*. Metode *Quadtree* dalam kompresi gambar merupakan implementasi langsung dari algoritma divide and conquer, di mana gambar dibagi secara rekursif menjadi empat kuadran yang lebih kecil (*divide*), kemudian masing-masing kuadran diperiksa homogenitasnya berdasarkan metrik tertentu seperti variansi atau MAD (*conquer*), dan hasil akhirnya digunakan untuk merepresentasikan gambar dalam bentuk blok-blok yang lebih sederhana dan terkompresi (*combine*). Pendekatan ini memungkinkan mengurangi ukuran data secara signifikan sambil tetap mempertahankan kualitas visual.

Program ini memulai proses kompresi gambar dengan meminta pengguna untuk memasukkan nama file gambar yang ingin dikompres. Lalu, pengguna diminta memilih metode pengukuran keseragaman blok (seperti variansi, MAD, *Max Pixel Difference*, atau *entropy*), menentukan nilai ambang batas (*threshold*), dan ukuran blok terkecil yang diizinkan untuk dibagi lagi. Setelah parameter ditentukan, program menjalankan proses kompresi dengan membangun struktur pohon *Quadtree* secara rekursif. Gambar awal dianggap sebagai satu blok besar, lalu diperiksa apakah blok tersebut cukup seragam berdasarkan metode dan *threshold* dipilih. Jika tidak seragam, blok dibagi menjadi empat bagian yang lebih kecil, dan proses ini terus berlanjut sampai semua blok memenuhi syarat keseragaman atau sudah mencapai ukuran minimum. Setelah semua blok ditentukan, program merekonstruksi gambar berdasarkan struktur pohon tersebut. Tiap blok seragam diganti dengan nilai rata-rata atau nilai representatif, sehingga membentuk gambar versi kompresi. Gambar hasil kompresi disimpan ke file output, dan program menampilkan informasi seperti waktu eksekusi, ukuran file sebelum dan sesudah kompresi, persentase penghematan ukuran, serta kedalaman dan jumlah *nodes* pada pohon.

Isi Program

```
#include <iostream>
#include <vector>
#include <opencv2/opencv.hpp>
#include <memory>
#include <chrono>
#include <filesystem>
using namespace std;
using namespace cv;
using namespace chrono;
namespace fs = std::filesystem;

struct Color {
    int r, g, b;
    Color(int _r, int _g, int _b) : r(_r), g(_g), b(_b) {}
    Color() : r(0), g(0), b(0) {}
};

Color calculateAverageColor(const vector<vector<Color>>& block) {
    long sumR = 0, sumG = 0, sumB = 0;
    int pixelCount = block.size() * block[0].size();

    for (const auto& row : block) {
        for (const auto& pixel : row) {
            sumR += pixel.r;
            sumG += pixel.g;
            sumB += pixel.b;
        }
    }

    return {
        static_cast<int>(sumR / pixelCount),
        static_cast<int>(sumG / pixelCount),
        static_cast<int>(sumB / pixelCount)
    };
}

class QuadTreeNode {
public:
    int x, y;
    int width, height;
    Color averageColor;
    bool isLeaf;
    unique_ptr<QuadTreeNode> children[4];

    QuadTreeNode(int x, int y, int width, int height)
        : x(x), y(y), width(width), height(height), isLeaf(true) {}
};
```

```

int countNodes(const QuadTreeNode* node) {
    if (!node) return 0;
    if (node->isLeaf) return 1;
    int count = 1;
    for (const auto& child : node->children) {
        count += countNodes(child.get());
    }
    return count;
}

int calculateTreeDepth(const QuadTreeNode* node) {
    if (!node || node->isLeaf) return 1;
    int maxDepth = 0;
    for (const auto& child : node->children) {
        maxDepth = max(maxDepth, calculateTreeDepth(child.get()));
    }
    return maxDepth + 1;
}

vector<vector<Color>> loadImage(const string filename) {
    Mat img = imread(filename);
    vector<vector<Color>> image(img.rows, vector<Color>(img.cols));

    for (size_t i = 0; i < img.rows; i++) {
        for (size_t j = 0; j < img.cols; j++) {
            Vec3b pixel = img.at<Vec3b>(i, j);
            image[i][j] = Color(pixel[2], pixel[1], pixel[0]);
        }
    }
    return image;
}

int selectVarianceMethod() {
    cout << "Select Variance Calculation Method:\n";

    cout << "1. Variance\n";

    cout << "2. Mean Absolute Deviation (MAD)\n";

    cout << "3. Max Pixel Difference\n";

    cout << "4. Entropy\n";

    int choice;

    cin >> choice;

    return choice;
}

float getThreshold() {
    float threshold;
    cout << "Enter the variance threshold: ";
    cin >> threshold;
    return threshold;
}

int getMinimumBlockSize() {
    int minBlockSize;
    cout << "Enter the minimum block size: ";
    cin >> minBlockSize;
    return minBlockSize;
}

```

```

int getMinimumBlockSize() {
    int minBlockSize;
    cout << "Enter the minimum block size: ";
    cin >> minBlockSize;
    return minBlockSize;
}

float calculateVariance(const vector<vector<Color>>& imageBlock) {
    float meanR = 0, meanG = 0, meanB = 0;
    int pixelCount = imageBlock.size() * imageBlock[0].size();

    for (const auto& row : imageBlock) {
        for (const auto& pixel : row) {
            meanR += pixel.r;
            meanG += pixel.g;
            meanB += pixel.b;
        }
    }
    meanR /= pixelCount;
    meanG /= pixelCount;
    meanB /= pixelCount;

    float varR = 0, varG = 0, varB = 0;
    for (const auto& row : imageBlock) {
        for (const auto& pixel : row) {
            varR += pow(pixel.r - meanR, 2);
            varG += pow(pixel.g - meanG, 2);
            varB += pow(pixel.b - meanB, 2);
        }
    }
    varR /= pixelCount;
    varG /= pixelCount;
    varB /= pixelCount;

    return (varR + varG + varB) / 3.0f;
}

float calculateMAD(const vector<vector<Color>>& block) {
    float meanR = 0, meanG = 0, meanB = 0;
    int pixelCount = block.size() * block[0].size();

    for (const auto& row : block) {
        for (const auto& pixel : row) {
            meanR += pixel.r;
            meanG += pixel.g;
            meanB += pixel.b;
        }
    }
    meanR /= pixelCount;
    meanG /= pixelCount;
    meanB /= pixelCount;

    float madR = 0, madG = 0, madB = 0;
    for (const auto& row : block) {
        for (const auto& pixel : row) {
            madR += abs(pixel.r - meanR);
            madG += abs(pixel.g - meanG);
            madB += abs(pixel.b - meanB);
        }
    }
    madR /= pixelCount;
    madG /= pixelCount;
    madB /= pixelCount;

    return (madR + madG + madB) / 3;
}

float calculateMaxDiff(const vector<vector<Color>>& block) {
    if (block.empty() || block[0].empty()) return 0;

    int minR = 255, maxR = 0;
    int minG = 255, maxG = 0;
    int minB = 255, maxB = 0;

    for (const auto& row : block) {
        for (const auto& pixel : row) {
            minR = min(minR, pixel.r);
            maxR = max(maxR, pixel.r);
            minG = min(minG, pixel.g);
            maxG = max(maxG, pixel.g);
            minB = min(minB, pixel.b);
            maxB = max(maxB, pixel.b);
        }
    }

    float diffR = static_cast<float>(maxR - minR);
    float diffG = static_cast<float>(maxG - minG);
    float diffB = static_cast<float>(maxB - minB);

    return (diffR + diffG + diffB) / 3;
}

```

```

float calculateEntropy(const vector<vector<Color>>& block) {
    if (block.empty() || block[0].empty()) return 0;

    unordered_map<int, int> histR, histG, histB;
    int pixelCount = block.size() * block[0].size();

    for (const auto& row : block) {
        for (const auto& pixel : row) {
            histR[pixel.r] += 1;
            histG[pixel.g] += 1;
            histB[pixel.b] += 1;
        }
    }

    float entropyR = 0, entropyG = 0, entropyB = 0;

    for (const auto& pair : histR) {
        float probability = static_cast<float>(pair.second) / pixelCount;
        entropyR -= probability * log2(probability);
    }

    for (const auto& pair : histG) {
        float probability = static_cast<float>(pair.second) / pixelCount;
        entropyG -= probability * log2(probability);
    }

    for (const auto& pair : histB) {
        float probability = static_cast<float>(pair.second) / pixelCount;
        entropyB -= probability * log2(probability);
    }

    return (entropyR + entropyG + entropyB) / 3;
}

float calculateError(const vector<vector<Color>>& block, int method) {
    switch (method) {
        case 1: return calculateVariance(block);
        case 2: return calculateMAD(block);
        case 3: return calculateMaxDiff(block);
        case 4: return calculateEntropy(block);
        default: throw runtime_error("Invalid error calculation method");
    }
}

unique_ptr<QuadTreeNode> buildQuadtree(const vector<vector<Color>>& image,
    int x, int y, int width, int height,
    float threshold, int minBlockSize,
    int method) {
    auto node = make_unique<QuadTreeNode>(x, y, width, height);

    vector<vector<Color>> block(height, vector<Color>(width));
    for (size_t i = 0; i < height; i++) {
        for (size_t j = 0; j < width; j++) {
            block[i][j] = image[y + i][x + j];
        }
    }

    float error = 0;
    switch (method) {
        case 1: error = calculateVariance(block); break;
        case 2: error = calculateMAD(block); break;
        case 3: error = calculateMaxDiff(block); break;
        case 4: error = calculateEntropy(block); break;
        default: throw runtime_error("Invalid method");
    }

    if (error <= threshold || width <= minBlockSize || height <= minBlockSize) {
        node->isLeaf = true;
        node->averageColor = calculateAverageColor(block);
        return node;
    }

    int halfWidth = width / 2;
    int halfHeight = height / 2;

    node->isLeaf = false;
    node->children[0] = buildQuadtree(image, x, y, halfWidth, halfHeight, threshold, minBlockSize,
        method);
    node->children[1] = buildQuadtree(image, x + halfWidth, y, halfWidth, halfHeight,
        threshold, minBlockSize, method);
    node->children[2] = buildQuadtree(image, x, y + halfHeight, halfWidth, halfHeight,
        threshold, minBlockSize, method);
    node->children[3] = buildQuadtree(image, x + halfWidth, y + halfHeight, halfWidth, halfHeight,
        threshold, minBlockSize, method);

    return node;
}

```

```

void reconstructImage(Mat& outputImage, const QuadTreeNode* node) {
    if (node->isLeaf) {
        Rect rect(node->x, node->y, node->width, node->height);
        rectangle(outputImage, rect,
            Scalar(node->averageColor.b, node->averageColor.g, node->averageColor.r),
            FILLED);
    }
    else {
        for (const auto& child : node->children) {
            if (child) {
                reconstructImage(outputImage, child.get());
            }
        }
    }
}

int main() {
    try {
        string filename;
        cout << "Enter image filename: ";
        cin >> filename;

        auto start = high_resolution_clock::now();
        auto image = loadImage(filename);

        int method = selectVarianceMethod();
        float threshold = getThreshold();
        int minBlockSize = getMinimumBlockSize();

        auto root = buildQuadtree(image, 0, 0, image[0].size(), image.size(),
            threshold, minBlockSize, method);

        Mat original = imread(filename);
        Mat compressed(original.size(), original.type());
        reconstructImage(compressed, root.get());

        auto end = high_resolution_clock::now();
        duration<double> duration = end - start;
        cout << "Execution time: " << duration.count() << " seconds" << endl;

        string outputFilename;
        cout << "Enter output filename: ";
        cin >> outputFilename;
        imwrite(outputFilename, compressed);

        uintmax_t compressedSize = fs::file_size(outputFilename);
        cout << "Compressed image size: " << compressedSize << " bytes" << endl;

        uintmax_t originalSize = fs::file_size(filename);
        cout << "Original image size: " << originalSize << " bytes" << endl;

        double compressionPercentage = 100.0 * (originalSize - compressedSize) / originalSize;
        cout << "Compression percentage: " << compressionPercentage << "%" << endl;





        cout << "Tree depth: " << calculateTreeDepth(root.get()) << endl;
        cout << "Number of nodes: " << countNodes(root.get()) << endl;
        cout << "Output saved to " << outputFilename << endl;







    }
    catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
        return 1;
    }





    return 0;
};

```


Percobaan Algoritma

<i>Original Image</i>	<i>Compressed Image</i>	<i>Parameters</i>	<i>Output</i>
		Metode: Entropy (4) Threshold: 1.5 Minimum Block Size: 8	Execution time: 7.71734 seconds Enter output filename: compressed.jpg Compressed image size: 19523 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 6 Number of nodes: 1365 Output saved to compressed.jpg
		Metode: Variance (1) Threshold: 10.0 Minimum Block Size: 16	Execution time: 5.16844 seconds Enter output filename: compressed.jpg Compressed image size: 15084 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 5 Number of nodes: 341 Output saved to compressed.jpg

		Metode: MAD (2) Threshold: 5.0 Minimum Block Size: 4	Execution time: 12.587 seconds Enter output filename: compressed_3.jpg Compressed image size: 21839 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 7 Number of nodes: 3949 Output saved to compressed_3.jpg
		Metode: Max Pixel Difference (3) Threshold: 30.0 Minimum Block Size: 16	Execution time: 2.58279 seconds Enter output filename: compressed.jpg Compressed image size: 15075 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 5 Number of nodes: 337 Output saved to compressed.jpg
		Metode: Variance (1) Threshold: 15.0 Minimum Block Size: 8	Execution time: 2.42845 seconds Enter output filename: compressed.jpg Compressed image size: 19471 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 6 Number of nodes: 1321 Output saved to compressed.jpg

		Metode: Entropy (4) Threshold: 0.5 Minimum Block Size: 2	<div>Execution time: 4.11268 seconds Enter output filename: 1.jpg Compressed image size: 22692 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 8 Number of nodes: 18197 Output saved to 1.jpg</div>
		Metode: MAD (2) Threshold: 20.0 Minimum Block Size: 8	<div>Execution time: 2.43806 seconds Enter output filename: 2.jpg Compressed image size: 16371 bytes Original image size: 11199 bytes Compression percentage: 1.64718e+17% Tree depth: 6 Number of nodes: 649 Output saved to 2.jpg</div>

Analisis Kompleksitas

Kompleksitas waktu algoritma bergantung kepada input gambar dan pemilihan metode. Jika gambar sangat kompleks, maka setiap blok terus dibelah hingga ukuran minimum. Pada kasus biasa, kompleksitas waktu akan lebih minim terutama di bagian gambar yang rata. Metode Variance dan Entropy cenderung lebih mahal secara komputasi dibandingkan Max Pixel Difference dan MAD. Kompleksitas waktu pada algoritma ini adalah $O(n^2)$.

Kompleksitas ruang dari algoritma ini sangat bergantung pada jumlah blok yang dihasilkan selama proses rekursif. Dalam kasus terburuk, ketika gambar memiliki banyak detail atau noise sehingga tidak memenuhi ambang batas error pada setiap pembagian, seluruh gambar akan terus dibagi hingga ukuran blok mencapai batas minimum. Hal ini menyebabkan jumlah blok yang terbentuk mendekati jumlah piksel dalam gambar. Oleh karena itu, kompleksitas ruang dalam kondisi terburuk adalah $O(n^2)$, di mana n adalah panjang sisi gambar (dengan asumsi gambar berukuran $n \times n$).

https://github.com/altftz/Tucil2_10122057

Lampiran