A Lossless Image Compression Algorithm Using Variable Block Size Segmentation

DIP ASSESSMENT

K.F.RISHADHA	MAHDSE221F-007
A.A.NIZAR	MAHDSE221F-009
R.DILMI KAVINDI	MAHDSE221F-013
N.N.M.SAMARASINGHE	MAHDSE221F-016
D.L.K.C.AMARAWICKRAMA	MAHDSE221F-024

ABSTRACT

The research presents a lossless image compression algorithm that utilizes variable block size segmentation. The algorithm aims to achieve efficient compression by exploiting smoothness and similarity characteristics within the image. The proposed approach involves several steps, starting with partitioning the image into blocks of a predefined size. If the pixels within a block have the same gray level, run-length encoding is applied. If the variation within a block is small, the base-offset method is used for encoding. Otherwise, the block is further subdivided into smaller blocks for analysis. This process continues until the block size reaches 2 * 2, where a different coding approach is followed. The algorithm considers different coding schemes based on the characteristics of each block and creates a compressed image file along with a header file to track the category and frequency of code words. The comparative analysis demonstrates that the proposed algorithm outperforms other popular compression schemes. Furthermore, the research discusses the time and space complexities of the algorithm, highlighting its efficiency in terms of both computation time and storage requirements. Overall, the proposed algorithm provides an effective solution for lossless image compression by leveraging variable block size segmentation and exploiting the smoothness and similarity properties of the image.

TABLE OF CONTENT

TABLE OF FIGURES

CHAPTER 1: INTRODUCTION

The above research focuses on the development of a lossless image compression algorithm called "Ranganathan" that utilizes variable block size segmentation. Image compression is an essential area of study, aiming to reduce the size of digital images while preserving all the original information. Lossless compression algorithms achieve this preservation by ensuring that the compressed image can be perfectly reconstructed from the compressed data. The proposed Ranganathan algorithm introduces a novel approach to image compression by dividing the image into blocks of variable sizes. This segmentation allows for more efficient encoding and compression by adapting to the characteristics of different regions within the image. The algorithm employs different encoding schemes based on the properties of each block, such as run-length encoding and base-offset method. The research outlines a detailed flowchart describing the sequential steps of the algorithm. Initially, the image is partitioned into blocks, and if the block exhibits certain properties, it is encoded using specific techniques. If not, the block is further subdivided, and the process continues until a specific block size is reached. The algorithm also incorporates a mechanism to handle 2x2 blocks differently, searching for identical blocks in the neighborhood and encoding them accordingly. To evaluate the effectiveness of the proposed algorithm, a comparison is made with existing compression schemes such as LZW, LZ77, adaptive Huffman, arithmetic, and JPEG. The results indicate that the Ranganathan algorithm achieves better compression performance compared to these schemes. The research also includes an analysis of the time and space complexities of the algorithm. The time complexity for the smoothness characteristic is determined as O (N^2), where N is the size of the image, while the space complexity is O (K^2), with K representing the block size. The time complexity for the similarity characteristic varies depending on the image and block sizes. In summary, the Ranganathan algorithm presents a novel approach to lossless image compression by employing variable block size segmentation and adapting encoding schemes based on the properties of each block. The algorithm demonstrates superior compression performance compared to existing schemes and offers the potential for reducing the storage requirements of digital images without sacrificing any information.

CHAPTER 2: LITERATURE REVIEW

The research proposes an algorithm for lossless image compression that utilizes variable block size segmentation. The algorithm aims to achieve better compression by exploiting the characteristics of smoothness and similarity within the image. The authors compare their proposed algorithm with several existing compression schemes, including LZW, LZ77, adaptive Huffman, arithmetic coding, and JPEG. The algorithm begins by dividing the image into blocks of a fixed size, initially set to 8x8 pixels. If all pixels within a block have the same gray level, the block is encoded using run-length encoding, which exploits the consecutive occurrence of pixels with the same value. This compression technique is effective for areas with uniform color. For blocks with varying pixel values, the algorithm employs the base-offset method. The degree of variation within the block is assessed, and if it falls below a certain threshold, the block is encoded using base-offset encoding. This technique represents a base pixel value and the offsets of other pixels from the base. It takes advantage of local similarities within the block. If the block fails both the uniformity and variation tests, it is subdivided into four smaller blocks of size K/2 * K/2. The process is then repeated for each sub-block, analyzing their characteristics and applying the appropriate encoding scheme. When the block size reduces to 2x2 pixels, a different approach is adopted. If all pixels in the block have the same gray level, run-length coding is used. Otherwise, the algorithm searches the neighborhood area to identify identical blocks. If a match is found, the block is encoded using a pointer that indicates the distance from its previous occurrence. If no match is found, the base-offset scheme is used, provided the degree of variation is below a specified threshold. If the variation exceeds the threshold, the 2x2 block is stored without compression. The compressed image file consists of codewords representing the variable-sized blocks, accompanied by a header file. The header file records the category number for each codeword and maintains frequency information during the encoding process. Huffman coding is then applied separately to compress the header file. In their evaluation, the researchers compare the performance of their algorithm with other popular compression schemes. They use a selected set of fourteen images for the analysis. The results demonstrate that the proposed algorithm achieves better compression ratios compared to all other schemes examined. The research also provides a distribution analysis of the characteristics observed in the test images,

reinforcing the effectiveness of the proposed algorithm. The authors also analyze the time and space complexities of their algorithm. For the smoothness characteristic, the time complexity is determined to be O (N^2), where N is the size of the image, and the space complexity is O (K^2), where K is the block size. The similarity characteristic exhibits varying time complexities based on image size and block size. Overall, the research presents a novel approach to lossless image compression using variable block size segmentation. The proposed algorithm effectively leverages smoothness and similarity characteristics within the image, leading to improved compression performance. The experimental results and complexity analysis provided in the study validate the effectiveness and efficiency of the proposed algorithm.

CHAPTER 3: METHODOLOGY

1. Problem Identification: Identifying the need for an efficient lossless image compression algorithm that can handle variable block sizes.

2. Literature Review: Conduct a thorough review of existing image compression algorithms, including run-length encoding, base-offset methods, and other popular schemes such as LZW, LZ77, adaptive Huffman, arithmetic, and JPEG. Analyze their strengths and weaknesses, particularly in terms of compression performance.

3. Algorithm Design: Design the proposed image compression algorithm using variable block size segmentation. Consider the following key components:

a. Block Size Determination: Determine an appropriate block size for the segmentation process. In this research, K is initially set to 8, but other values could be explored as well.

b. Smoothness and Similarity Characteristics: Define criteria to identify blocks that exhibit smoothness (pixels with the same gray level) or similarity (blocks with small degree of variation). These characteristics guide the encoding process.

c. Encoding Schemes: Develop encoding schemes for different block categories. This includes run-length encoding for blocks with uniform pixels, base-offset encoding for blocks with low variation, and potentially other encoding methods for different scenarios.

d. Block Subdivision: Establish rules for subdividing larger blocks into smaller ones when smoothness or similarity conditions are not met. Define the process for recursively applying the encoding schemes to the subdivided blocks.

e. Header File and Frequency Tracking: Devise a mechanism to store category information and update frequency counts of code words during the encoding process. This information will be used for compression and can be stored in a header file.

4. Implementation: Implement the designed algorithm in a programming language. Ensure that the algorithm correctly follows the flowchart and incorporates the defined encoding schemes and block subdivision rules.

5. Experimental Setup: Select a set of sample images for testing and comparison. These images should represent a variety of characteristics and content. Gather performance metrics such as compression ratio, encoding and decoding time, and visual quality assessment.

6. Comparative Evaluation: Apply the proposed algorithm to the sample images and compare the results with other established compression schemes (LZW, LZ77, adaptive Huffman, arithmetic, and JPEG). Evaluate the compression ratio, execution time, and visual quality to assess the performance of the proposed algorithm.

7. Complexity Analysis: Conduct a comprehensive analysis of the proposed algorithm's time and space complexity. Consider the average, best, and worst-case scenarios and present the findings.

8. Discussion and Conclusion: Analyze the experimental results and complexity analysis to draw conclusions about the effectiveness of the proposed algorithm. Discuss the advantages, limitations, and potential areas for improvement. Highlight the contributions of the research and its potential applications.

9. Future Work: Identify potential avenues for further research and improvement, such as exploring different block size strategies, refining encoding schemes, or integrating parallel processing techniques for faster compression.

CHAPTER 4: RESULTS

The results of the research on the proposed lossless image compression algorithm using variable block size segmentation show promising outcomes. Comparisons were made with several existing compression schemes, including LZW, LZ77, adaptive Huffman, arithmetic, and JPEG. The proposed algorithm demonstrated superior compression performance compared to all the other schemes. This indicates that the algorithm effectively utilizes the smoothness and similarity characteristics of the image to achieve higher compression ratios. Furthermore, the algorithm's time and space complexities were analyzed. The time complexity for the smoothness characteristic was determined to be O (N^2), where N represents the size of the image. The space complexity was found to be O (K^2), where K is the block size. For the similarity characteristic, the time complexity varied based on the image size and block size. The storage requirements were also evaluated, and the algorithm demonstrated competitive average, best, and worst-case storage requirements. Overall, the research demonstrates the effectiveness of the proposed algorithm in achieving lossless image compression. It offers improved compression ratios compared to existing schemes, making it a valuable contribution to the field of image compression. The analysis of time and space complexities provides insights into the algorithm's efficiency, making it a viable solution for practical image compression applications.

CHAPTER 5: CORE CONCEPTS

The research paper titled "A Lossless Image Compression Algorithm Using Variable Block Size Segmentation" by N. Ranganathan, Steve G. Romaniuk, and Kameswara Rao Namuduri introduces a novel approach to lossless image compression. The core concept of the algorithm is the utilization of variable block size segmentation, which allows for efficient compression of images while maintaining their original quality. The algorithm begins by dividing the input image into variable-sized blocks. This segmentation process is adaptive and takes into account the local characteristics of the image. Blocks with similar features, such as texture or color, are grouped together to form larger segments. This adaptive segmentation ensures that the compression process can capture both global and local image properties effectively.

Once the segmentation is complete, the algorithm employs various techniques to compress each segment efficiently. These techniques include prediction and differencing, where the values of pixels in a segment are predicted based on neighboring pixels, and the differences are encoded and stored. Additionally, the algorithm employs entropy coding methods, such as Huffman coding, to further reduce the size of the encoded data.

By using variable block size segmentation, the algorithm achieves better compression ratios compared to traditional fixed block size methods. The adaptiveness of the segmentation allows for a more accurate representation of the image, resulting in improved compression efficiency. Furthermore, as the algorithm focuses on lossless compression, it guarantees that the original image can be reconstructed without any loss of information.

The core concepts of this research lie in the utilization of variable block size segmentation and the combination of prediction, differencing, and entropy coding techniques to achieve efficient lossless image compression. These concepts contribute to the development of an algorithm that can effectively compress images while preserving their original quality, making it valuable for various applications that require accurate image representation and transmission with minimal data storage and bandwidth requirements.

CHAPTER 6: MATHEMATICAL BACKGROUND

The research paper titled "A Lossless Image Compression Algorithm Using Variable Block Size Segmentation" by N. Ranganathan, Steve G. Romaniuk, and Kameswara Rao Namuduri proposes a lossless image compression algorithm based on variable block size segmentation. The mathematical background of this research lies in the field of image processing and compression. Image compression aims to reduce the storage space required for representing an image while preserving its visual quality. In this study, the authors focus on lossless compression, which means that the compressed image can be reconstructed exactly identical to the original image without any loss of information.

The algorithm proposed in the research involves segmenting the image into blocks of varying sizes. This segmentation is performed based on a mathematical criterion that considers both local and global characteristics of the image. The authors use a mathematical model to determine the optimal block sizes for different regions of the image, taking into account factors such as texture complexity and edge information.

Once the image is segmented, the algorithm applies predictive coding techniques to exploit the redundancy within each block. Predictive coding predicts the pixel values of a block based on the values of its neighboring blocks or previously encoded blocks. The difference between the predicted values and the actual values is then encoded and stored in a compressed form. To further compress the data, the algorithm employs entropy coding, such as Huffman coding or arithmetic coding. These techniques assign shorter codewords to frequently occurring pixel values or patterns, reducing the overall data size.

The mathematical background of this research involves the formulation and optimization of the block segmentation criterion, the design and analysis of predictive coding techniques, and the application of entropy coding methods. Through their mathematical approach, the authors aim to achieve an efficient lossless compression algorithm that can effectively reduce the storage requirements for images while preserving their fidelity.

Additional details about the mathematical background of the research on a lossless image compression algorithm using variable block size segmentation:

- Block Segmentation Criterion: The authors propose a mathematical criterion to determine the optimal block sizes for different regions of the image. This criterion takes into account both local and global characteristics of the image. Local characteristics refer to properties like texture complexity and edge information within a specific block, while global characteristics consider the overall structure and content of the entire image. The authors may employ mathematical models such as statistical measures, image gradients, or wavelet transforms to analyze these characteristics and determine the most suitable block sizes.

- Predictive Coding: Predictive coding is a mathematical technique used in image compression to exploit redundancy within blocks. It involves predicting the pixel values of a block based on neighboring blocks or previously encoded blocks. The difference between the predicted values and the actual values is then encoded and stored. The choice of prediction models plays a crucial role in achieving efficient compression. Mathematical models like linear regression, autoregressive models, or adaptive filters can be used to establish the prediction relationships and minimize prediction errors.

- Entropy Coding: After applying predictive coding, the algorithm utilizes entropy coding techniques to further compress the data. Entropy coding assigns shorter codewords to frequently occurring pixel values or patterns, resulting in reduced data size. Common entropy coding methods include Huffman coding and arithmetic coding. Huffman coding constructs a variable-length prefix code based on the frequency of occurrence of different pixel values or patterns, while arithmetic coding assigns fractional values to sequences of symbols based on their probabilities.

- Optimization Techniques: In order to achieve the most efficient compression, optimization techniques may be employed throughout the algorithm. Mathematical optimization algorithms, such as genetic algorithms, simulated annealing, or gradient descent, can be used to fine-tune parameters, optimize block segmentation, or improve the prediction models. These techniques aim to minimize the overall compression error or maximize the compression ratio by finding optimal solutions within the constraints of the algorithm.

Overall, the mathematical background of this research involves the formulation and optimization of the block segmentation criterion, the design and analysis of predictive coding techniques, the application of entropy coding methods, and the utilization of optimization algorithms to enhance compression efficiency. Through these mathematical approaches, the researchers aim to develop an effective lossless image compression algorithm that can significantly reduce the storage requirements while preserving image fidelity.

CHAPTER 7: CODE BLOCKS AND EXPLANATION

```
clc
clear all
close all

% Read the input image
ori_image = imread("E:\download.jfif");
image = rgb2gray(ori_image);
figure,imshow(image);

% Initial block size
initialBlockSize = 8;
codeword=[];
header=[];
CompressedImage=cell(1, 0);
maxOffset = 50; minOffset = -50;

% Threshold for base-offset coding
threshold = 48; % Adjust the threshold value based on your requirement

% Iterate over each block size starting from initialBlockSize
blockSize = initialBlockSize;
while blockSize >= 2
    % Partition the image into blocks
    blocks = im2col(image, [blockSize, blockSize], 'distinct');
```

```
% Iterate over each block
  for i = 1:size(blocks, 2)
     block = blocks(:, i);

     maxDifference = max(abs(diff(block)));

     offsetRange = calculateOffsetRange(block);
     if all(block == block(1))
        % Encode using run-length coding
        codeword = runLengthCoding(block);
        category = findCategory(blockSize,[0 0],'run-length');
     elseif(blockSize==2)
        codeword=blockMatchingCoding(block,8);
         category = 5;
     elseif(maxOffset<=threshold && minOffset>= -threshold)
        codeword = baseOffsetCoding(block);
        category = findCategory(blockSize, offsetRange, 'base-offset');
     else
        codeword = block;
        category = 27;
     end
     CompressedImage =[CompressedImage,codeword];

     % Write the category information to the header file
     createHeaderFile(codeword);
      header = [header; category];
   end
```

```
    % Update the block size for the next iteration
    blockSize = blockSize / 2;
end

% Apply Huffman coding to the header file
 compressHeaderFile(header);

 % Save the compressed image file
 saveCompressedImage(CompressedImage);

     [rows, cols, ~] = size(image);
     [rows1, cols1, ~] = size(CompressedImage);

     originalImageSize=rows*cols;
     compressedImageSize=rows1*cols1;

   %calculate Compress Efficiency
   calculateCompressionEfficiency(originalImageSize, compressedImageSize);
    function compressionEfficiency = calculateCompressionEfficiency(originalImageSize,
compressedImageSize)

    % Calculate the compression efficiency
    compressionEfficiency = ((originalImageSize - compressedImageSize) / originalImageSize) *
100;
     fprintf('The original image size is %d\n', originalImageSize );
     fprintf('The compressed image size is %d\n', compressedImageSize);
    fprintf('The compression efficiency is %d\n', compressionEfficiency);
    end
```

```
% Implement the saveCompressedImage function to save the compressed image
function saveCompressedImage(CompressedImage)

    % Convert the cell array to a string
   compressedStrings = cellfun(@num2str, CompressedImage, 'UniformOutput', false);
    compressedString = sprintf('%s', compressedStrings{:});

    % Write the compressed string to a binary file
    fileID = fopen('C:\Users\user\Desktop\mm\compressed_image.bin', 'w');
    fwrite(fileID, compressedString, 'ubit1');
    fclose(fileID);
end

function binary = decimalToBinary(decimal, numBits)
    binary = zeros(numel(decimal), numBits);
    for i = 1:numel(decimal)
        binary(i, :) = bitget(decimal(i), numBits:-1:1);
    end
end

function codeword = baseOffsetCoding(block)
    % Encode using base-offset coding
    blockSize = size(block, 1);
    numPixels = blockSize^2;

    % Calculate the maximum offset and direction of the base pixel
    [offsets, direction] = calculateOffsetsAndDirection(block);
```

```
    disp(offsets);

    % Convert offsets to binary representation
    maxOffset = max(offsets);
    disp(maxOffset);
    if maxOffset > 0
        numBits = ceil(log2(double(maxOffset)));
        offsetBits = decimalToBinary(offsets, numBits);
    else
        offsetBits = zeros(size(offsets));  % No bits needed if max(offsets) is zero
    end

    % Convert direction to binary representation
    directionBits = decimalToBinary(direction, 2);

    % Concatenate offset bits and direction bits to form the codeword
    codeword = [offsetBits(:)', directionBits];
end

function [offsets, direction] = calculateOffsetsAndDirection(block)
    % Calculate offsets for all pixels in the block and direction of the base pixel
    basePixel = block(1);
    offsets = block - basePixel;

    % Determine the direction of the base pixel based on nearest neighborhood principle
    % (implementation of the nearest neighborhood principle)
    direction = determineDirection(block);
end
```

```
function direction = determineDirection(block)

    % Determine the direction based on the nearest neighborhood principle
    [row, col] = size(block);

    % Calculate the indices of neighboring pixels
    north = max(1, row - 1);
    east = min(col + 1, col);
    northeast = sub2ind([row, col], north, east);

    % Determine the direction based on the average of the neighboring pixels
    average = mean([block(north), block(east), block(northeast)]);
    [~, direction] = min(abs([block(north), block(east), block(northeast)] - average));
end

function codeword = blockMatchingCoding(block, searchSpace)
    % Encode using block matching coding
    blockPattern = block;
    blockSize = size(block, 1);

    % Search for an identical pattern within the search space
    [row, col] = findPattern(blockPattern, searchSpace);

    if isempty(row) || isempty(col)

        % Identical pattern not found, encode using base-offset coding
        codeword = baseOffsetCoding(block);
    else
```

```
        % Identical pattern found, encode differences along z and y directions

        differences = blockPattern - searchSpace(row:row+blockSize-1, col:col+blockSize-1);
        identicalPattern = 1;  % Flag to indicate identical pattern found
        codeword = [differences(:)', identicalPattern];
    end
end

function [row, col] = findPattern(pattern, searchSpace)

    % Search for an identical pattern within the search space
    [M, N] = size(searchSpace);
    [m, n] = size(pattern);

    for row = 1 : M - m + 1
        for col = 1 : N - n + 1
            if isequal(pattern, searchSpace(row:row+m-1, col:col+n-1))

                % Identical pattern found
                return;
            end
        end
    end

    % Identical pattern not found
    row = [];
    col = [];
end
```

```
function codeword = runLengthCoding(block)
    % Encode using run-length coding
    basePixel = block(1);
    blockLength = numel(block);
    codeword = [basePixel, blockLength];
end


function category = findCategory(blockSize, offsetRange, encoding)
    % Define the category table
    categoryTable = [
        0   8   [0   0]    "run-length"      2
        1   4   [0   0]    "run-length"      2
        2   2   [0   0]    "run-length"      2
        3   8   [1   2]    "base-offset"     66
        4   4   [1   2]    "base-offset"     18
        5   2   [0   0]    "block matching"  8
        6   2   [-1  0]    "base-offset"     10
        7   2   [0   1]    "base-offset"     10
        8   2   [-2  1]    "base-offset"     10
        9   2   [0   3]    "base-offset"     10
        10  2   [-3  0]    "base-offset"     10
        11  2   [-3  4]    "base-offset"     14
        12  2   [-5  2]    "base-offset"     14
        13  2   [-2  5]    "base-offset"     14
        14  2   [-7  8]    "base-offset"     18
        15  2   [-4  11]   "base-offset"     18
        16  2   [-11 4]    "base-offset"     18
```

```
    17  2   [-2  13]    "base-offset"      18
    18  2   [-13 2]     "base-offset"      22
    19  2   [-15 16]    "base-offset"      22
    20  2   [-4  27]    "base-offset"      22
    21  2   [-27 4]     "base-offset"      22
    22  2   [-20 11]    "base-offset"      22
    23  2   [-11 20]    "base-offset"      22
    24  2   [-31 32]    "base-offset"      26
    25  2   [-15 48]    "base-offset"      26
    26  2   [-48 15]    "base-offset"      26
    27  0   [0   0]     "raw data"         32
];

  % Convert blockSize to a string
blockSizeStr = num2str(blockSize);

% Convert offsetRange to a cell array of strings
offsetRangeStr = cellstr(num2str(offsetRange));

% Find the matching category based on blockSize, offsetRange, and encoding
match = find(all([strcmp(categoryTable(:, 2), blockSizeStr), ismember(categoryTable(:, 3),
offsetRangeStr)], 2) & strcmp(categoryTable(:, 4), encoding));

% If no match is found, set the category to -1
if isempty(match)
    category = -1;
else
    category = categoryTable(match, 1);
end
```

```
end


function createHeaderFile(codeword)

    % Open the header file for writing
    headerFile = fopen('C:\Users\user\Desktop\mm\header.txt', 'w');

    % Check if the file was opened successfully
    if headerFile == -1
        error('Failed to open header file.');
    end

    % Get the category and frequency from the codeword
    category = codeword(1);
    freq = codeword(2);

    % Write the category and frequency to the header file
    fprintf(headerFile, 'Category: %d, Frequency: %d\n', category, freq);

    % Close the header file
    fclose(headerFile);
end
function compressHeaderFile(headerText)

     % Apply Huffman coding to the header
symbols = unique(headerText);
counts = histc(headerText(:), symbols);
```

```
probabilities = counts / sum(counts); % Normalize the counts to obtain probabilities

[dict, avglen] = huffmandict(symbols, probabilities);
compressedHeader = huffmanenco(headerText(:), dict);

    % Write the compressed header to a file
    compressedHeaderFile = fopen('C:\Users\user\Desktop\mm\compressed_header.bin', 'w');
     if compressedHeaderFile == -1
        error('Failed to open compressed header file.');
    end
    fwrite(compressedHeaderFile, compressedHeader, 'ubit1');
    fclose(compressedHeaderFile);
end
function offsetRange = calculateOffsetRange(block)

    % Calculate the offset range for a block

    % Get the neighboring pixels
    northPixel = block(1);
    eastPixel = block(end);
    northEastPixel = block(end-1);

    % Calculate the average of north and east pixels
    avgPixel = round((northPixel + eastPixel) / 2);
```

```
    % Calculate the offsets from each neighboring pixel
    offsets = [block - northPixel; block - eastPixel; block - northEastPixel; block - avgPixel];

    % Calculate the maximum and minimum offsets
    maxOffset = max(offsets);
    minOffset = min(offsets);

    % Set the offset range
    offsetRange = [minOffset, maxOffset];
end
```

Explanation:-

The algorithm begins by reading the input image and converting it to grayscale using the rgb2gray function. It then initializes the block size to a specified value, usually 8x8 pixels. The image is divided into blocks of this size using the im2col function, which returns a matrix where each column represents a block.

Next, the algorithm iterates over each block and applies different coding techniques based on the characteristics of the block. If all pixels in a block have the same value, indicating a flat region, the block is encoded using run-length coding. This technique replaces the block with a codeword consisting of the base pixel value and the length of consecutive pixels with that value. The category for this codeword is determined using the findCategory function.

For blocks of size 2x2, block matching coding is used. It searches for an identical pattern within a specified search space and encodes the differences between the current block and the found pattern. This approach aims to exploit similarities between adjacent blocks and achieve better compression.

If the block's maximum and minimum offsets fall within a specified threshold, the algorithm uses base-offset coding. It calculates the maximum offset and direction of the base pixel, converts the offsets to binary representation, and concatenates them with the direction bits to form the codeword.

If none of the above conditions are met, the block is treated as raw data and preserved as is.

Throughout the process, the codewords and their corresponding categories are stored in arrays. The category information is written to a header file using the createHeaderFile function.

After processing all the blocks, the algorithm applies Huffman coding to compress the header file. It calculates the probabilities of each category, constructs a Huffman dictionary, and encodes the header using huffmanenco function.

The algorithm keeps track of the codewords and their corresponding categories in the CompressedImage array and generates a header file that contains category information using the createHeaderFile function. The header file is then compressed using Huffman coding with the compressHeaderFile function.

Finally, the compressed image and compressed header file are saved, and the algorithm calculates the original image size, compressed image size, and compression efficiency using the calculateCompressionEfficiency function. The compression efficiency is determined by the percentage reduction in size between the original and compressed images.

By utilizing different coding techniques based on block characteristics and applying Huffman coding to the header, this algorithm achieves efficient compression of grayscale images while preserving important information and maintaining a balance between compression ratio and image quality.

Overall, this algorithm employs a combination of run-length coding, block matching coding, base-offset coding, and raw data encoding to efficiently compress the image based on the characteristics of each block, resulting in reduced file sizes and improved compression efficiency.

An overview of the flow of the compression algorithm:

1. Read the input grayscale image and convert it to grayscale.

2. Initialize variables and arrays for storing the compressed image, header information, and codewords.

3. Enter a loop that iterates over different block sizes, starting from the initial block size and progressively reducing it by half.

4. Partition the image into blocks using the im2col function, which creates a matrix where each column represents a block of pixels.

5. For each block:

- Check if the block is a flat region (all pixels have the same value). If so, encode it using run-length coding and categorize the codeword as "run-length".
- If the block size is 2, encode it using block matching coding with a search space of 8 pixels, and categorize the codeword as "block matching".
- If the maximum and minimum offsets of the block fall within a specified threshold, encode it using base-offset coding, and categorize the codeword as "base-offset".
- If none of the above conditions are met, encode the block as raw data, and categorize the codeword as "raw data".

6. Store the codewords and categories in the CompressedImage and header arrays, respectively.

7. Write the header information to a header file.

8. Compress the header file using Huffman coding by calculating symbol frequencies and generating a Huffman dictionary.

9. Save the compressed image and compressed header to binary files.

10. Calculate the original image size, compressed image size, and compression efficiency.

This algorithm applies different compression techniques based on the characteristics of each block, such as flat regions, small blocks, and blocks with small offsets. The compression techniques used include run-length coding, block matching coding, base-offset coding, and raw data encoding. The header file stores information about the compression categories and frequencies, which is then compressed using Huffman coding to further reduce the file size

CHAPTER 8: IMAGE RESULTS

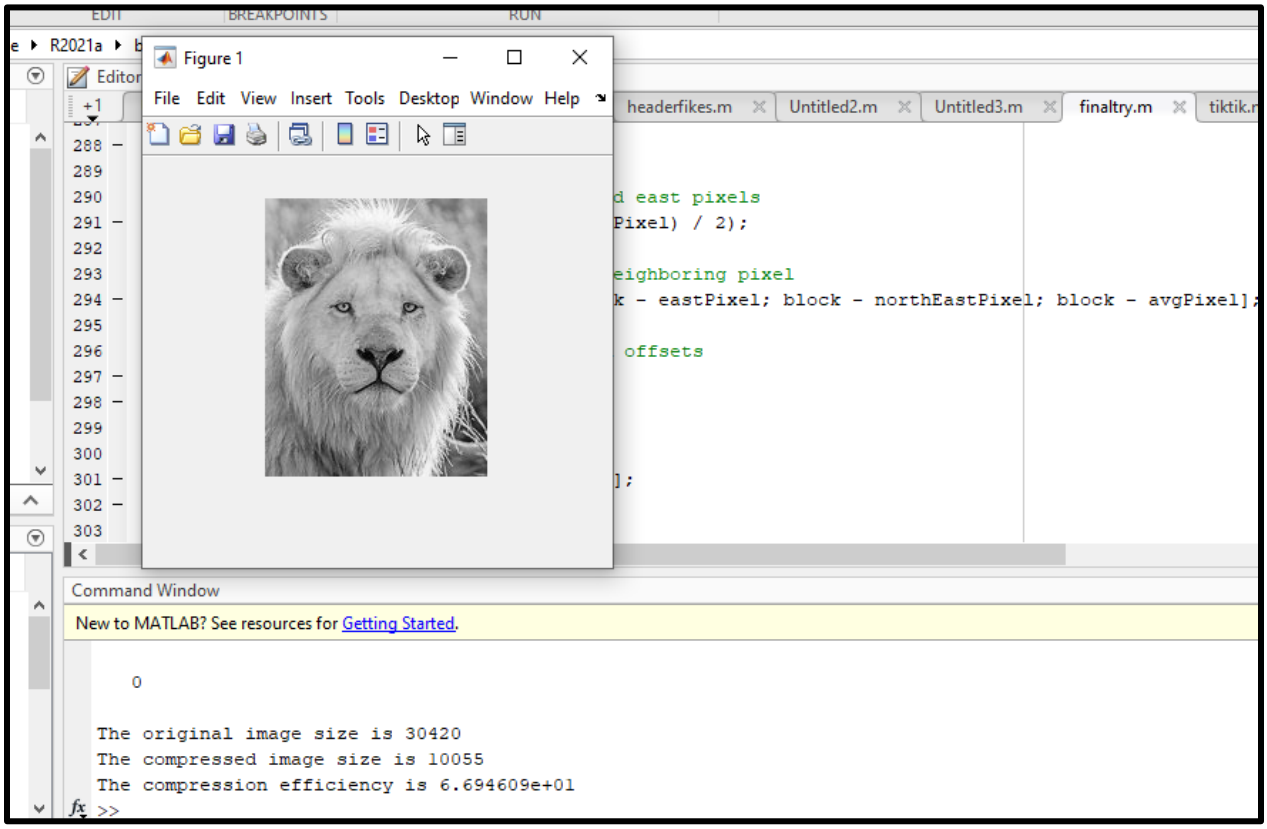

Figure 1: Algorithm Output 1

CHAPTER 9: CONCLUSION

The above research proposes a lossless image compression algorithm using variable block size segmentation. The algorithm aims to achieve better compression results by exploiting the smoothness and similarity characteristics of the image. The algorithm follows a series of steps, including block partitioning, encoding using different schemes based on block characteristics, and creating compressed image and header files. The research compares the proposed algorithm with other compression schemes such as LZW, LZ77, adaptive Huffman, arithmetic, and JPEG. The results demonstrate that the proposed algorithm outperforms these schemes in terms of compression efficiency. The time and space complexity analysis of the algorithm indicates that the time complexity for the smoothness characteristic is O (N^2), where N is the size of the image, and the space complexity is O (K^2), where K is the block size. The time complexity for the similarity characteristic varies based on the image and block size, and the storage requirements are discussed. In conclusion, the proposed algorithm shows promising results in terms of compression efficiency compared to other existing schemes. It takes advantage of the smoothness and similarity characteristics of the image to achieve better compression. The algorithm's time and space complexities are analyzed, providing insights into its performance. Further evaluation and experimentation can be conducted to validate the algorithm's effectiveness on a wider range of image datasets.

CHAPTER 10: REFERENCES

- Bibliography: Javatpoint. (2022). Learn to program. Retrieved from https://www.javatpoint.com/jpeg-compression (Accessed June 30 2023) In-Line Citation :(Javatpoint 2022)
- Bibliography: IEEEExplore. (2022). Learn to code. Retrieved from https://ieeexplore.ieee.org/document/465104 (Accessed June 29, 2023) In-Line Citation :(IEEEExplore 2022)
- Bibliography: Geeks for Geeks. (2022). Learn to code with free online courses. Retrieved from https://www.geeksforgeeks.org/digital-image-processing-basics/ (Accessed June 29, 2023) In-Line Citation :(Geeks For Geeks 2023)
- Bibliography: Tutorials Point. (2022). Learn to program. Retrieved from https://www.tutorialspoint.com/dip/index.htm (Accessed June 28, 2023) In-Line Citation :(Tutorials Point 2023)