CMPE 150 – ASSIGNMENT 3

PROBLEM DESCRIPTION

In this project, we are tasked to write a java program which does ditigal image processing. There are four parts of the project. First is the part where the program takes a PPM file and read the pixel values into a 3D array and writes the exact 3D array to another ppm file. Second, the program calculates an average of three pixels and write a black-and-white version of the input image as ppm. Third, the program performs convolution operation using a 2D array (called “ kernel filter”) on the input image and writes the result as another ppm image. Finally, the program checks the values of the neighboring pixels to see if they are within a given range and alter these pixels to have equal values using recursive methods. This is called color quantization. Then, the program will write the quantized image to a final ppm file.

PROBLEM SOLUTION

The program works using configurations, therefore, in the main method, the first argument is read to an integer value called “mode”. The program works in different modes, which represent which method should be used. Second, there’s always an input file which contains the image and it’s always given as the second argument of the program. A new file is created and a scanner reads the input file. The first thing to do afterwards is reading the pixel values into a 3D array because most of the methods in the program takes an 3D array as a parameter. The first method *ppmToArray* takes a scanner which contains a ppm file and copy its contents to a 3D array using for loops.

In ppmToArray, there’s a scanner which contains a PPM file. PPM files have the file type P3 and the sizes of the pixels at the top. So, these values are read and a 3D array is created using the sizes. After that, using for loops for each dimension and nextInt() method, values of the pixels are transferred to the 3D array. The inner nested loop is the channel values.

If mode is 0, program takes a ppm file and writes it to another file using a method called *printArray.* The method takes two parameters, a 3D array and the mode, because each mode it prints to a different file. Inside the method, there’s 4 if statements which check for the value of the mode. If it’s 0, the method prints an array to the “output.ppm” file using a PrintStream, if 2, “black-and-white.ppm”, if 3, “quantized.ppm” and if 1, “convolution.ppm”. There’s also nested for loops inside if statements to print the values of the array.

If mode is 1, in the main method, the program applies a black and white filter to the image using a method called *blackWhite.* The method takes an array as a parameter. Using nested loops, the method adds up the values of three pixels which have the same x and y coordinates but different color channels and then divides the sum to 3 to get their average. After that, replaces the all three pixels’ values with the average in the array. Eventually, with different tones of gray, returned array is a black and white version of the original image. Using *printArray* method, program prints the new black and white array to “black-and-white.ppm” file with a PrintStream object. It also prints the file type, the length of row and column of the array and the maximum pixel value to the top of every ppm file.

If mode is 2, the 2nd argument of the program is a .txt File containing the information of a kernel filter. A new file is created which contains the filter.txt and it is transferred to a scanner. After that, a new method *txtToArray* , which takes a scanner as a parameter, creates a 2D array for the filter and using nested loops, transfers the filter to the array. In the main method, the program applies convolution using a method called *convolution* . The method takes an 3D array, the image, and a 2D array, the filter , as a parameter. The first two integers keep and sum are neutral values for the multiplication and addition. There’s 5 nested loops, 3 for the 3D array ( row, column, channel) and 2 for 2D array (filterRow, filterColumn) . The way they are listed are also matters because my method goes column by column and row by row and channel by channel, respectively. The sum of a kernel filter applied on a part of image array is then transferred to a new array which is smaller than the original array because of the size of the filter. After that, the new *convolutionArray* is returned and printed using *PrintArray* method to the file “convolution.ppm”

If mode is 3, program does the quantization using the same named method. This method also has a helper method which is also called quantization because it’s recursive. Inside the first quantization method which takes 2 parameters, a 3D array and a range, there’s another Boolean 3D array at the same size with the image array. This *check* array is used to check if the the pixel which has the same coordinate already been checked or painted. If the Boolean value of a pixel is true, it means it’s already checked. *There’s* 3 nested loops to quantize ( check for the neighbors of a pixel and change their value with the reference pixel if they are in range ) every pixel. Outer loop is for the channel values(variable z), middle is for columns(variable y) and the inner is for rows(variable x). Inside the nested loops, an integer named keep keeps the value of that pixel and the helper method *quantization* is called with the parameters *image array, Boolean check array, the coordinates of the pixel, x, y, z, the value of the reference pixel, integer keep, and the range*.

The helper method quantization has a if statement at the beginning. It says that *if the pixel which has x y z coordinates doesn’t have a neighbor (it may be at corners or edges), or it is not in given range or it’s Boolean value is true , the method exits.* Otherwise it alters the value of the pixel with the integer *keep*, and changes its Boolean value to true, and calls itself 6 times for each neighbor. When all Boolean values become true, eventually, the helper method exits and the first quantization method returns the modified image array. In the main method*,* the *printArray* method is called again with mode 3 and the quantized image array is printed to the file “quantize.ppm”.

IMPLEMENTATION

import java.io.File;

import java.io.FileNotFoundException;

import java.io.PrintStream;

import java.util.Arrays;

import java.util.Scanner;

public class ES2018400183 {

public static void main(String[] args) throws FileNotFoundException {

int mode=Integer.parseInt(args[0]);

File originalImage = new File(args[1]);

Scanner x = new Scanner(originalImage); //takes the original ppm file

int[][][] image = ppmToArray(x); //Reads the contents of the PPM file into a 3D array

if(mode==0) { //if mode is 0 it just prints the file

printArray(image,0) //prints the array to the output file

}if(mode==1) { //if mode is 1 it does black and white process

printArray(blackWhite(image),1);

}if(mode==2) { //if mode is 2 it does the convolution part

File filter = new File(args[2]); //it reads the filter file to a scanner

Scanner y = new Scanner(filter);

int[][] filterArray = txtToArray(y); //using a method, reads the contents of the TXT file into a 2D array

int[][][] convolutionArray = new int[image.length-filterArray.length+1][image[0].length-filterArray[0].length+1][3];

//creates a new array which has less pixels than the original image

printArray(convolution(image, convolutionArray,filterArray),2); //prints the array to the convolution file

}if(mode==3) { //if mode is 2 it does the quantization

int range = Integer.parseInt(args[2]); //takes the 2nd argument as range

printArray(quantization(image, range),3); //prints the array to the quantized file

}

}

public static int[][][] ppmToArray(Scanner x) throws FileNotFoundException {

//the method to read the contents of the PPM file into a 3D array

x.nextLine(); //skips the P3 part

int rowLen = x.nextInt(); //reads the row length of the pixels

int colLen = x.nextInt(); //reads the column length of the pixels

int maxValue = x.nextInt(); //reads the maximum color value

int[][][] photo = new int[rowLen][colLen][3]; //creates a new 3D array for the pixel value for(int i = 0; i < photo.length; i++) { //this nested loops reads the pixels to the array

for(int k=0; k < photo[0].length; k++) {

for(int z=0; z < 3; z++) {

photo[i][k][z] = x.nextInt();

}

}

}

return photo; //returns the array

}

public static int[][] txtToArray(Scanner y) throws FileNotFoundException { //the method to read the contents of TXT file into a 2D array

String len=y.nextLine(); //takes the "3x3" etc. part

String row=len.substring(0,1) //takes the row and column length separately

String column=len.substring(2);

int len1=Integer.parseInt(row); //turns the sizes to integers

int len2=Integer.parseInt(column);

int[][] filter = new int[len1][len2] //this nested loops reads the contents of the filter to a 2D array

for(int u=0; u<len1; u++) {

for(int f=0; f<len2; f++) {

filter[u][f]=y.nextInt();

}

}

return filter; //returns the 2D array

}

public static void printArray(int[][][] image, int mode) throws FileNotFoundException {

//the method to print an array to a file

if(mode==0) { //if mode is 0, prints the whole array to the "output" file

PrintStream output = new PrintStream("output.ppm");

output.println("P3"); output.println(image.length+" "+image[0].length);

output.println("255");

for(int i = 0; i < image.length; i++) {

for(int k=0; k < image[0].length; k++) {

for(int z=0; z < 3; z++) {

output.print(image[i][k][z] + " ");

}

}

}

}

if(mode==1) { //if mode is 1, prints the whole array to the "black-and-white" file

PrintStream output = new PrintStream(new File("black-and-white.ppm"));

output.println("P3"); //prints the file type and the length and the maximum color value

output.println(image.length +" "+image[0].length);

output.println("255");

for(int i = 0; i < image.length; i++) {

for(int k=0; k < image[0].length; k++) {

for(int z=0; z < 3; z++) {

output.print(image[i][k][z] + " ");

}

}

}

}

if(mode==2) { //if mode is 2, prints the whole array to the "convolution" file

PrintStream output = new PrintStream("convolution.ppm");

output.println("P3"); //prints the file type and the length and the maximum color value

output.println(image.length +" "+image[0].length);

output.println("255");

for(int i = 0; i < image.length; i++) {

for(int k=0; k < image[0].length; k++) {

for(int z=0; z < 3; z++) {

output.print(image[i][k][z] + " ");

}

}

}

}

if(mode==3) { //if mode is 3, prints the whole array to the "quantized" file

PrintStream output = new PrintStream("quantized.ppm");

output.println("P3"); //prints the file type and the length and the maximum color value

output.println(image.length +" "+image[0].length);

output.println("255");

for(int i = 0; i < image.length; i++) {

for(int k=0; k < image[0].length; k++) {

for(int z=0; z < 3; z++) {

output.print(image[i][k][z] + " ");

}

output.print("\t"); //prints a tab between every third group of pixels

}

output.println(); //prints a new line

}

}

}

public static int[][][] blackWhite(int[][][] image) {

//method to Calculate the color-channel average values and convert the colored image to black-and-white.

int sum=0; //integer used to add the 3 pixels' values

int avg=0; //integer used to keep the average of the three pixel group

for(int i = 0; i < image.length; i++) { // nested loops to add the 3 pixels' value

for(int k=0; k < image[0].length; k++) {

for(int z=0; z < 3; z++) {

sum=sum+image[i][k][z];

}

avg=sum/3; //takes the average of the 3 pixels' sum

for(int z=0; z<3; z++) { //transfers the average value to all 3 pixels

image[i][k][z]=avg;

}

sum=0; //sets the average and sum values back to zero

avg=0;

}

}

return image; //returns the image array

}

public static int[][][] convolution(int[][][] image, int[][][] convolutionArray, int[][] filterArray) {

//method to apply a kernel filter to an image

int keep=1; //neutral element for multiplication of a pixel and the value of the filter

int sum=0; //neutral element for addition of all multiplications

for(int row = 0; row<convolutionArray.length; row++) { // Nested loops to apply convolution to each color channel of the image separately

for(int column = 0; column<convolutionArray[0].length; column++) {

for(int channel=0; channel<3; channel++) { for(int filterRow=0; filterRow<filterArray.length; filterRow++) {

for(int filterColumn=0; filterColumn<filterArray[0].length; filterColumn++) {

keep=image[row+filterRow][column+filterColumn][channel]\*filterArray[filterRow][filterColumn];

sum=sum+keep;

keep=1;

}

}

if(sum>255) { //if sum values are greater than maximum value or negative, changes them to proper value

sum=255;

}

else if(sum<0) {

sum=0;

}

convolutionArray[row][column][channel]=sum; //before moving the filter, transfers the sum to the convolution array

sum=0; //sets the sum back to zero

}

}

}

return blackWhite(convolutionArray); //returns the array after applying black and white filter

}

public static int[][][] quantization(int[][][] image, int range) { //method to do quantization

boolean[][][] check = new boolean[image.length][image[0].length][3]; //creates a boolean array to check if the pixels are already painted

int keep=0; //keeps the value of the pixel whose neighbors' will be checked

for(int z=0; z < 3; z++) { //does the quantization for every pixel using for loops

for(int x = 0; x < image.length; x++) {

for(int y=0; y < image[0].length; y++) {

keep=image[x][y][z] ; //keeps the value of the pixel

quantized(image, x, y, z, check, range, keep); //calls the helper quantization method

}

}

}

return image; //returns the quantized array

}

public static void quantized(int[][][] image, int x, int y, int z, boolean[][][] check, int range, int keep) {

//helper method for quantization

if(x<0 || x>=image.length ||y<0 || y>=image[0].length || z<0 || z>=3 || image[x][y][z]>keep+range || image[x][y][z]<keep-range || check[x][y][z]==true ) {

//if the pixel doesn't have neighbors(out of bounds), or if it is not in range, or it has already painted(boolean value for that pixel must be true), exits the method

return;

}else { //else, makes the neighbor's value same as the reference pixel and makes its boolean value true(makes it painted)

image[x][y][z]=keep;

check[x][y][z]=true;

quantized(image, x+1, y, z, check, range, keep); //does the whole process recursively to all neighbors of a pixel in an order

quantized(image, x-1, y, z, check, range, keep);

quantized(image, x, y+1, z, check, range, keep);

quantized(image, x, y-1, z, check, range, keep);

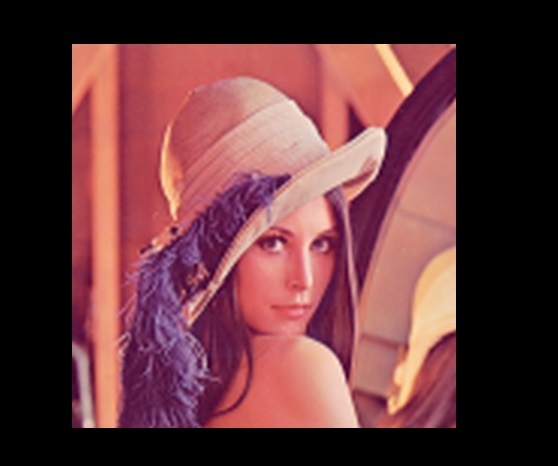
quantized(image, x, y, z+1, check, range, keep);

quantized(image, x, y, z-1, check, range, keep);

}

}

}

OUTPUT OF THE PROGRAM

CONCLUSION

The problem is solved correctly because not only in ppm view, the pixels are in the fully correct value when checked from DiffChecker. Although it may be shorter using methods for the usage of for loops to print arrays, it is much more readable that way. It is a “clear” code with correct outputs and it can be easily read by humans.