using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Drawing;

using System.IO;

using System.Windows.Forms;

namespace IntelligentScissors

{

public partial class MainForm : Form

{

public MainForm()

{

InitializeComponent();

}

Point point;

Point endpoint;

int w;

Stopwatch stopwatch;

List<int> anchorpoints = new List<int>();

int shortest\_path\_value = 0, base\_value = 0;

List<KeyValuePair<int, RGBPixel>> imageorg = new List<KeyValuePair<int, RGBPixel>>();

RGBPixel[,] ImageMatrix;

// RGBPixel[,] ImageMatrix2;

string testdir;

private void btnOpen\_Click(object sender, EventArgs e)//O(V)

{

// MessageBox.Show(point.X.ToString());

OpenFileDialog openFileDialog1 = new OpenFileDialog();

if (openFileDialog1.ShowDialog() == DialogResult.OK)

{

//Open the browsed image and display it

string OpenedFilePath = openFileDialog1.FileName;

testdir = Path.GetDirectoryName(OpenedFilePath);

ImageMatrix = ImageOperations.OpenImage(OpenedFilePath);

ImageOperations.DisplayImage(ImageMatrix, pictureBox1);//O(V)

}

txtWidth.Text = ImageOperations.GetWidth(ImageMatrix).ToString();

txtHeight.Text = ImageOperations.GetHeight(ImageMatrix).ToString();

stopwatch = Stopwatch.StartNew();

//ImageOperations.dijkstra(1695577, 2055619);

ImageOperations.initAdjacencyList(ImageMatrix);//O(V)

stopwatch.Stop();

MessageBox.Show(stopwatch.Elapsed.ToString());

}

#region automated testing

private void testgraph()//O(E)

{

Vector2D[] x = ImageOperations.pixelEnergies;

List<KeyValuePair<int, double>>[] adjacencyList = ImageOperations.adjacencyList;

// File.AppendText("testenergies.txt");

StreamWriter f = new StreamWriter("testgraph.txt", append: false);

DialogResult d = MessageBox.Show("sample?", "test type", MessageBoxButtons.YesNo);

if (d == DialogResult.Yes)

{

int index = 0;

f.WriteLine("The constructed graph");

f.WriteLine("");

foreach (List<KeyValuePair<int, double>> a in adjacencyList)//O(E)

{

f.WriteLine(" The index node" + index);

f.WriteLine("Edges");

foreach (KeyValuePair<int, double> z in a)

{

f.WriteLine("edge from " + index + " To " + z.Key.ToString() + " With Weights " + z.Value);

}

f.WriteLine("\n\n");

index++;

}

f.Close();

//Open the browsed image and display it

string OpenedFilePath = (!(testdir[testdir.Length - 1].ToString().Equals("1"))) ? testdir + @"\" + "output" + testdir[testdir.Length - 1] + ".txt" : testdir + @"\" + "output.txt";

StreamReader trueoutput = new StreamReader(OpenedFilePath);

StreamReader output = new StreamReader(OpenedFilePath);

int wrong = 0;

while (true)//O(E)

{

string s = trueoutput.ReadLine();

string s1 = output.ReadLine();

if (s != null)

{

if (s.Equals(s1))

{

continue;

}

else

{

wrong++;

break;

}

}

else

break;

}

output.Close();

trueoutput.Close();

MessageBox.Show(wrong == 0 ? "test completed successfully :)" : "wrong lines=" + wrong.ToString() + "\n SAD");

}

else if (d == DialogResult.No)

{

int index = 0;

f.WriteLine("Constructed Graph: (Format: node\_index|edges:(from, to, weight)... )");

foreach (List<KeyValuePair<int, double>> a in adjacencyList)//O(E)

{

f.Write(index + "|edges:");

foreach (KeyValuePair<int, double> z in a)

{

f.Write("(" + index + "," + z.Key.ToString() + "," + z.Value + ")");

}

f.WriteLine("");

index++;

}

f.WriteLine("Graph construction took: " + stopwatch.Elapsed.TotalSeconds.ToString());

f.Close();

//Open the browsed image and display it

string OpenedFilePath = (!testdir[testdir.Length - 1].ToString().Equals("1")) ? testdir + @"\" + "output" + testdir[testdir.Length - 1] + ".txt" : testdir + @"\" + "output.txt";

StreamReader trueoutput = new StreamReader(OpenedFilePath);

StreamReader output = new StreamReader(OpenedFilePath);

int wrong = 0;

while (true)//O(V)

{

string s = trueoutput.ReadLine();

string s1 = output.ReadLine();

if (s != null)

{

if (s.Equals(s1))

{

continue;

}

else

{

wrong++;

break;

}

}

else

break;

}

output.Close();

trueoutput.Close();

MessageBox.Show(wrong == 0 ? "test completed successfully :)" : "wrong lines=" + wrong.ToString() + "\n SAD");

}

}

private void btnGaussSmooth\_Click(object sender, EventArgs e)//O(E)

{

//double sigma = double.Parse(txtGaussSigma.Text);

//int maskSize = (int)nudMaskSize.Value;

//ImageMatrix = ImageOperations.GaussianFilter1D(ImageMatrix, maskSize, sigma);

//ImageOperations.DisplayImage(ImageMatrix, pictureBox2);

testgraph();//O(E)

}

#endregion

private void panel1\_Paint(object sender, PaintEventArgs e)

{

}

private void pictureBox1\_Click(object sender, EventArgs e)//O(V)

{

MouseEventArgs ee = (MouseEventArgs)e;

point = ee.Location;

w = ImageOperations.GetWidth(ImageMatrix);

anchorpoints.Add(point.Y \* w + point.X);

int start = (point.Y) \* w + (point.X);

Point p = pictureBox1.Location;

ImageOperations.init = false;

base\_value = shortest\_path\_value;

imageorg.Clear();//O(V)

}

private void pictureBox1\_MouseMove(object sender, MouseEventArgs e)//O(VLog(V)+E)

{

endpoint = e.Location;

shortest\_path\_value = base\_value;

if (!point.IsEmpty)

{

if (imageorg.Count > 0)

{

while (imageorg.Count > 0)//O(V)

{

int index = imageorg[0].Key;

int x = index % w, y = index / w;

//ImageMatrix[y, x].blue = imageorg[0].Value.blue;

//ImageMatrix[y, x].green = imageorg[0].Value.green;

//ImageMatrix[y, x].red = imageorg[0].Value.red;

((Bitmap)pictureBox1.Image).SetPixel(x, y, Color.FromArgb(imageorg[0].Value.blue, imageorg[0].Value.green, imageorg[0].Value.red));

imageorg.RemoveAt(0);

}

// ImageOperations.DisplayImage(ImageMatrix, pictureBox1);

}

int start = (point.Y) \* w + (point.X);

int end = (endpoint.Y) \* w + (endpoint.X);

if (ImageOperations.GetHeight(ImageMatrix) < Math.Abs(panel1.Height) && ImageOperations.GetWidth(ImageMatrix) < Math.Abs(panel1.Width))

{

ImageOperations.start = 0;

ImageOperations.end = ImageOperations.numOfPixels;

ImageOperations.w = ImageOperations.GetWidth(ImageMatrix);

ImageOperations.h = ImageOperations.GetHeight(ImageMatrix);

}

else

{

if(ImageOperations.start!= Math.Abs(pictureBox1.Location.X) + w \* Math.Abs(pictureBox1.Location.Y))

ImageOperations.init = false;

ImageOperations.start = Math.Abs(pictureBox1.Location.X) + w \* Math.Abs(pictureBox1.Location.Y) ;

ImageOperations.end = Math.Abs(pictureBox1.Location.X) + panel1.Width + w \* (Math.Abs(pictureBox1.Location.Y) + panel1.Height);

ImageOperations.w = panel1.Width;

ImageOperations.h = panel1.Height;

}

if (!ImageOperations.init)

ImageOperations.dijkstra(start, end);//O(VLog(V)+E)

if (ImageOperations.parent != null && ImageOperations.parent.ContainsKey(end))

{

if (ImageOperations.parent[end] == -1 && end != start)

ImageOperations.dijkstra(start, end);//O(VLog(V)+E)

if (ImageOperations.parent[start] != -1)

ImageOperations.dijkstra(start, end);//O(VLog(V)+E)

}

else

{

ImageOperations.dijkstra(start, end);//O(VLog(V)+E)

}

Dictionary<int, int> pp = ImageOperations.parent;

while (true)//O(V)

{

int x = end % w;

int y = end / w;

if (!pp.ContainsKey(end))

break;

if (end == -1)

{

break;

}

else

{

shortest\_path\_value += 1;

RGBPixel pixel = new RGBPixel();

pixel.blue = ((Bitmap)pictureBox1.Image).GetPixel(x, y).R;

pixel.green = ((Bitmap)pictureBox1.Image).GetPixel(x, y).G;

pixel.red = ((Bitmap)pictureBox1.Image).GetPixel(x, y).B;

imageorg.Add(new KeyValuePair<int, RGBPixel>(end, pixel));

((Bitmap)pictureBox1.Image).SetPixel(x, y, Color.Red);

}

end = pp[end];

}

pictureBox1.Refresh();

txtGaussSigma.Text = shortest\_path\_value.ToString();

}

}

private void pictureBox1\_DoubleClick(object sender, EventArgs e)//O(VLog(V)+E)

{

MouseEventArgs ee = (MouseEventArgs)e;

point = ee.Location;

w = ImageOperations.GetWidth(ImageMatrix);

anchorpoints.Add(point.Y \* w + point.X);

int start = (point.Y) \* w + (point.X);

ImageOperations.init = false;

if (anchorpoints.Count > 1)

{

int end = (point.Y) \* w + (point.X);

Dictionary<int, int> pp = ImageOperations.parent;

imageorg.Clear();//O(V)

while (true)//O(V)

{

int x = end % w;

int y = end / w;

if (end == -1)

{

break;

}

else

{

((Bitmap)pictureBox1.Image).SetPixel(x, y, Color.Red);

}

end = pp[end];

}

}

int end1 = anchorpoints[0];

ImageOperations.start = Math.Abs(pictureBox1.Location.X) + w \* Math.Abs(pictureBox1.Location.Y);

ImageOperations.end = Math.Abs(pictureBox1.Location.X) + panel1.Width + w \* (Math.Abs(pictureBox1.Location.Y) + panel1.Height);

ImageOperations.w = panel1.Width;

ImageOperations.h = panel1.Height;

if (!ImageOperations.init)

ImageOperations.dijkstra(start, end1);//O(VLog(V)+E)

if (ImageOperations.parent != null && ImageOperations.parent.ContainsKey(end1))

{

if (ImageOperations.parent[end1] == -1 && end1 != start)

ImageOperations.dijkstra(start, end1);//O(VLog(V)+E)

if (ImageOperations.parent[start] != -1)

ImageOperations.dijkstra(start, end1);//O(VLog(V)+E)

}

else

{

ImageOperations.dijkstra(start, end1);//O(VLog(V)+E)

}

Dictionary<int, int> pp1 = ImageOperations.parent;

while (true)//O(V)

{

int x = end1 % w;

int y = end1 / w;

if (end1 == -1)

{

break;

}

else

{

((Bitmap)pictureBox1.Image).SetPixel(x, y, Color.Red);

}

end1 = pp1[end1];

}

point.X = 0;

point.Y = 0;

anchorpoints = new List<int>();

pictureBox1.Refresh();

}

private void pictureBox1\_MouseHover(object sender, EventArgs e)

{

}

}

}

using FibonacciHeap;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Drawing.Imaging;

using System.Windows.Forms;

///Algorithms Project

///Intelligent Scissors

///

namespace IntelligentScissors

{

/// <summary>

/// Holds the pixel color in 3 byte values: red, green and blue

/// </summary>

public struct RGBPixel

{

public byte red, green, blue;

}

public struct RGBPixelD

{

public double red, green, blue;

}

/// <summary>

/// Holds the edge energy between

/// 1. a pixel and its right one (X)

/// 2. a pixel and its bottom one (Y)

/// </summary>

public struct Vector2D

{

public double X { get; set; }

public double Y { get; set; }

}

/// <summary>

/// Library of static functions that deal with images

/// </summary>

public class ImageOperations

{

public static List<KeyValuePair<int, double>>[] adjacencyList;

public static Vector2D[] pixelEnergies;

public static Dictionary<int, double> cost;

public static Dictionary<int, int> parent;

public static int numOfPixels, start, end, w, h;

public static Dictionary<int, FibonacciHeapNode<int, double>> fheapnode;

public static FibonacciHeap<int, double> qu;

static int imageWidth;

public static bool init = false;

public static double s(int x, int y, int x1, int y1)

{

return Math.Abs(x - x1) + Math.Abs(y - y1);

}

public static void dijkstra(int point, int finish)//O(VLog(V)+E)

{

if (!init)

{

fheapnode = new Dictionary<int, FibonacciHeapNode<int, double>>(numOfPixels);

qu = new FibonacciHeap<int, double>(0);

cost = new Dictionary<int, double>(numOfPixels);

parent = new Dictionary<int, int>(numOfPixels);

int z = 0;

for (int j = start + w; j < end; j += imageWidth)//O(V)

{

for (int i = j - w; i < j; i++)

{

cost[i] = (double.MaxValue);

parent[i] = (-1);

FibonacciHeapNode<int, double> node = new FibonacciHeapNode<int, double>(i, cost[i]);

fheapnode[i] = (node);

qu.Insert(node);//O(1)

}

z++;

}

cost[point] = 0;

qu.DecreaseKey(fheapnode[point], cost[point]);//O(1)

}

init = true;

while (!qu.IsEmpty())//itr v

{

FibonacciHeapNode<int, double> u = qu.Min();//O(1)

if (u.Data == finish)

return;

u = qu.RemoveMin();//O(Log(V))

foreach (var x in adjacencyList[u.Data])

{

//if (cost[u.Data] + x.Value+s(x.Key%imageWidth, x.Key / imageWidth, finish % imageWidth, finish / imageWidth) < cost[x.Key])

if (cost.ContainsKey(x.Key))

if (cost[u.Data] + x.Value < cost[x.Key])

{

// cost[x.Key] = x.Value + cost[u.Data]+ s(x.Key % imageWidth, x.Key / imageWidth, finish % imageWidth, finish / imageWidth);

cost[x.Key] = x.Value + cost[u.Data];

qu.DecreaseKey(fheapnode[x.Key], cost[x.Key]);//O(1)

parent[x.Key] = u.Data;

}

}

}

}

// Checks if new pixel coordinates are valid

private static bool check(int rowIndex, int columnIndex, RGBPixel[,] ImageMatrix)//O(1)

{

return rowIndex >= 0 && rowIndex < GetHeight(ImageMatrix)

&& columnIndex >= 0 && columnIndex < GetWidth(ImageMatrix);

}

// Constructs the adjacency list for an image

public static void initAdjacencyList(RGBPixel[,] ImageMatrix)//O(V)

{

int imageHeight = GetHeight(ImageMatrix);

imageWidth = GetWidth(ImageMatrix);

numOfPixels = imageHeight \* imageWidth;

adjacencyList = new List<KeyValuePair<int, double>>[numOfPixels];

// These were reversed \*\*\* (matters for the first sample case)

int[] rowsOffset = { -1, 0, 1, 0 };

int[] columnsOffset = { 0, -1, 0, 1 };

pixelEnergies = new Vector2D[numOfPixels];

for (int i = 0; i < numOfPixels; i++)//O(V)

{

// 20 x 10 image

// 100 ?

// 100 / 20 = 10 row index=y

// 100 % 20 = 0 columns=x

// 10 \* 10 + 0

// 4-Connectivity

adjacencyList[i] = new List<KeyValuePair<int, double>>(4);

// These were wrong \*\*\* (doesn't matter for the first sample case)

int pixelRowIndex = i / imageWidth;

int pixelColumnIndex = i % imageWidth;

// These were flipped x == pixelColumnIndex and y == pixelRowIndex \*\*\* (matters for the first sample case??)

Vector2D energies = CalculatePixelEnergies(pixelColumnIndex, pixelRowIndex, ImageMatrix);

pixelEnergies[i] = energies;

for (int j = 0; j < 4; j++)

{

if (check(pixelRowIndex + rowsOffset[j], pixelColumnIndex + columnsOffset[j], ImageMatrix))

{

// From (x, y) to [0 , numOfPixels[

int adjacentPixelIndex = (pixelRowIndex + rowsOffset[j]) \* imageWidth + (pixelColumnIndex + columnsOffset[j]);

if (j == 0)

{

adjacencyList[i].Add(new KeyValuePair<int, double>(adjacentPixelIndex, pixelEnergies[adjacentPixelIndex].Y));

}

else if (j == 1)

{

adjacencyList[i].Add(new KeyValuePair<int, double>(adjacentPixelIndex, pixelEnergies[adjacentPixelIndex].X));

}

else if (j == 2)

{

adjacencyList[i].Add(new KeyValuePair<int, double>(adjacentPixelIndex, energies.Y));

}

else

{

adjacencyList[i].Add(new KeyValuePair<int, double>(adjacentPixelIndex, energies.X));

}

}

}

}

}

/// <summary>

/// Open an image and load it into 2D array of colors (size: Height x Width)

/// </summary>

/// <param name="ImagePath">Image file path</param>

/// <returns>2D array of colors</returns>

public static RGBPixel[,] OpenImage(string ImagePath)//O(V)

{

Bitmap original\_bm = new Bitmap(ImagePath);

int Height = original\_bm.Height;

int Width = original\_bm.Width;

RGBPixel[,] Buffer = new RGBPixel[Height, Width];

unsafe

{

BitmapData bmd = original\_bm.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite, original\_bm.PixelFormat);

int x, y;

int nWidth = 0;

bool Format32 = false;

bool Format24 = false;

bool Format8 = false;

if (original\_bm.PixelFormat == PixelFormat.Format24bppRgb)

{

Format24 = true;

nWidth = Width \* 3;

}

else if (original\_bm.PixelFormat == PixelFormat.Format32bppArgb || original\_bm.PixelFormat == PixelFormat.Format32bppRgb || original\_bm.PixelFormat == PixelFormat.Format32bppPArgb)

{

Format32 = true;

nWidth = Width \* 4;

}

else if (original\_bm.PixelFormat == PixelFormat.Format8bppIndexed)

{

Format8 = true;

nWidth = Width;

}

int nOffset = bmd.Stride - nWidth;

byte\* p = (byte\*)bmd.Scan0;

for (y = 0; y < Height; y++)

{

for (x = 0; x < Width; x++)

{

if (Format8)

{

Buffer[y, x].red = Buffer[y, x].green = Buffer[y, x].blue = p[0];

p++;

}

else

{

Buffer[y, x].red = p[0];

Buffer[y, x].green = p[1];

Buffer[y, x].blue = p[2];

if (Format24) p += 3;

else if (Format32) p += 4;

}

}

p += nOffset;

}

original\_bm.UnlockBits(bmd);

}

return Buffer;

}

/// <summary>

/// Get the height of the image

/// </summary>

/// <param name="ImageMatrix">2D array that contains the image</param>

/// <returns>Image Height</returns>

public static int GetHeight(RGBPixel[,] ImageMatrix)//O(1)

{

return ImageMatrix.GetLength(0);

}

/// <summary>

/// Get the width of the image

/// </summary>

/// <param name="ImageMatrix">2D array that contains the image</param>

/// <returns>Image Width</returns>

public static int GetWidth(RGBPixel[,] ImageMatrix)//O(1)

{

return ImageMatrix.GetLength(1);

}

/// <summary>

/// Calculate edge energy between

/// 1. the given pixel and its right one (X)

/// 2. the given pixel and its bottom one (Y)

/// </summary>

/// <param name="x">pixel x-coordinate</param>

/// <param name="y">pixel y-coordinate</param>

/// <param name="ImageMatrix">colored image matrix</param>

/// <returns>edge energy with the right pixel (X) and with the bottom pixel (Y)</returns>

public static Vector2D CalculatePixelEnergies(int x, int y, RGBPixel[,] ImageMatrix)//O(1)

{

if (ImageMatrix == null) throw new Exception("image is not set!");

Vector2D gradient = CalculateGradientAtPixel(x, y, ImageMatrix);

double gradientMagnitude = Math.Sqrt(gradient.X \* gradient.X + gradient.Y \* gradient.Y);

double edgeAngle = Math.Atan2(gradient.Y, gradient.X);

double rotatedEdgeAngle = edgeAngle + Math.PI / 2.0;

Vector2D energy = new Vector2D();

energy.X = 1 / Math.Abs(gradientMagnitude \* Math.Cos(rotatedEdgeAngle));

energy.Y = 1 / Math.Abs(gradientMagnitude \* Math.Sin(rotatedEdgeAngle));

if (double.IsInfinity(energy.X))

energy.X = 1e16;

if (double.IsInfinity(energy.Y))

energy.Y = 1e16;

return energy;

}

/// <summary>

/// Display the given image on the given PictureBox object

/// </summary>

/// <param name="ImageMatrix">2D array that contains the image</param>

/// <param name="PicBox">PictureBox object to display the image on it</param>

public static void DisplayImage(RGBPixel[,] ImageMatrix, PictureBox PicBox)//O(V)

{

// Create Image:

//==============

int Height = ImageMatrix.GetLength(0);

int Width = ImageMatrix.GetLength(1);

Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb);

unsafe

{

BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite, ImageBMP.PixelFormat);

int nWidth = 0;

nWidth = Width \* 3;

int nOffset = bmd.Stride - nWidth;

byte\* p = (byte\*)bmd.Scan0;

for (int i = 0; i < Height; i++)

{

for (int j = 0; j < Width; j++)

{

p[0] = ImageMatrix[i, j].red;

p[1] = ImageMatrix[i, j].green;

p[2] = ImageMatrix[i, j].blue;

p += 3;

}

p += nOffset;

}

ImageBMP.UnlockBits(bmd);

}

PicBox.Image = ImageBMP;

}

/// <summary>

/// Apply Gaussian smoothing filter to enhance the edge detection

/// </summary>

/// <param name="ImageMatrix">Colored image matrix</param>

/// <param name="filterSize">Gaussian mask size</param>

/// <param name="sigma">Gaussian sigma</param>

/// <returns>smoothed color image</returns>

public static RGBPixel[,] GaussianFilter1D(RGBPixel[,] ImageMatrix, int filterSize, double sigma)

{

int Height = GetHeight(ImageMatrix);

int Width = GetWidth(ImageMatrix);

RGBPixelD[,] VerFiltered = new RGBPixelD[Height, Width];

RGBPixel[,] Filtered = new RGBPixel[Height, Width];

// Create Filter in Spatial Domain:

//=================================

//make the filter ODD size

if (filterSize % 2 == 0) filterSize++;

double[] Filter = new double[filterSize];

//Compute Filter in Spatial Domain :

//==================================

double Sum1 = 0;

int HalfSize = filterSize / 2;

for (int y = -HalfSize; y <= HalfSize; y++)

{

//Filter[y+HalfSize] = (1.0 / (Math.Sqrt(2 \* 22.0/7.0) \* Segma)) \* Math.Exp(-(double)(y\*y) / (double)(2 \* Segma \* Segma)) ;

Filter[y + HalfSize] = Math.Exp(-(double)(y \* y) / (double)(2 \* sigma \* sigma));

Sum1 += Filter[y + HalfSize];

}

for (int y = -HalfSize; y <= HalfSize; y++)

{

Filter[y + HalfSize] /= Sum1;

}

//Filter Original Image Vertically:

//=================================

int ii, jj;

RGBPixelD Sum;

RGBPixel Item1;

RGBPixelD Item2;

for (int j = 0; j < Width; j++)

for (int i = 0; i < Height; i++)

{

Sum.red = 0;

Sum.green = 0;

Sum.blue = 0;

for (int y = -HalfSize; y <= HalfSize; y++)

{

ii = i + y;

if (ii >= 0 && ii < Height)

{

Item1 = ImageMatrix[ii, j];

Sum.red += Filter[y + HalfSize] \* Item1.red;

Sum.green += Filter[y + HalfSize] \* Item1.green;

Sum.blue += Filter[y + HalfSize] \* Item1.blue;

}

}

VerFiltered[i, j] = Sum;

}

//Filter Resulting Image Horizontally:

//===================================

for (int i = 0; i < Height; i++)

for (int j = 0; j < Width; j++)

{

Sum.red = 0;

Sum.green = 0;

Sum.blue = 0;

for (int x = -HalfSize; x <= HalfSize; x++)

{

jj = j + x;

if (jj >= 0 && jj < Width)

{

Item2 = VerFiltered[i, jj];

Sum.red += Filter[x + HalfSize] \* Item2.red;

Sum.green += Filter[x + HalfSize] \* Item2.green;

Sum.blue += Filter[x + HalfSize] \* Item2.blue;

}

}

Filtered[i, j].red = (byte)Sum.red;

Filtered[i, j].green = (byte)Sum.green;

Filtered[i, j].blue = (byte)Sum.blue;

}

return Filtered;

}

#region Private Functions

/// <summary>

/// Calculate Gradient vector between the given pixel and its right and bottom ones

/// </summary>

/// <param name="x">pixel x-coordinate</param>

/// <param name="y">pixel y-coordinate</param>

/// <param name="ImageMatrix">colored image matrix</param>

/// <returns></returns>

private static Vector2D CalculateGradientAtPixel(int x, int y, RGBPixel[,] ImageMatrix)//O(1)

{

Vector2D gradient = new Vector2D();

RGBPixel mainPixel = ImageMatrix[y, x];

double pixelGrayVal = 0.21 \* mainPixel.red + 0.72 \* mainPixel.green + 0.07 \* mainPixel.blue;

if (y == GetHeight(ImageMatrix) - 1)

{

//boundary pixel.

for (int i = 0; i < 3; i++)

{

gradient.Y = 0;

}

}

else

{

RGBPixel downPixel = ImageMatrix[y + 1, x];

double downPixelGrayVal = 0.21 \* downPixel.red + 0.72 \* downPixel.green + 0.07 \* downPixel.blue;

gradient.Y = pixelGrayVal - downPixelGrayVal;

}

if (x == GetWidth(ImageMatrix) - 1)

{

//boundary pixel.

gradient.X = 0;

}

else

{

RGBPixel rightPixel = ImageMatrix[y, x + 1];

double rightPixelGrayVal = 0.21 \* rightPixel.red + 0.72 \* rightPixel.green + 0.07 \* rightPixel.blue;

gradient.X = pixelGrayVal - rightPixelGrayVal;

}

return gradient;

}

#endregion

}

}

using System;

using System.Collections.Generic;

namespace FibonacciHeap

{

/// <summary>

/// Fibonacci Heap realization. Uses generic type T for data storage and TKey as a key type.

/// </summary>

/// <typeparam name="T">Type of the stored objects.</typeparam>

/// <typeparam name="TKey">Type of the object key. Should implement IComparable.</typeparam>

public class FibonacciHeap<T, TKey> where TKey : IComparable<TKey>

{

private readonly TKey \_minKeyValue;

/// <summary>

/// Minimum (statring) node of the heap.

/// </summary>

private FibonacciHeapNode<T, TKey> \_minNode;

/// <summary>

/// The nodes quantity.

/// </summary>

private int \_nNodes;

/// <summary>

/// Initializes the new instance of the Heap.

/// </summary>

/// <param name="minKeyValue">Minimum value of the key - to be used for comparing.</param>

public FibonacciHeap(TKey minKeyValue)

{

\_minKeyValue = minKeyValue;

}

/// <summary>

/// Identifies whatever heap is empty.

/// </summary>

/// <returns>true if heap is empty - contains no elements.</returns>

public bool IsEmpty()

{

return \_minNode == null;

}

/// <summary>

/// Removes all the elements from the heap.

/// </summary>

public void Clear()

{

\_minNode = null;

\_nNodes = 0;

}

/// <summary>

/// Decreses the key of a node.

/// O(1) amortized.

/// </summary>

public void DecreaseKey(FibonacciHeapNode<T, TKey> x, TKey k)

{

if (k.CompareTo(x.Key) > 0)

{

throw new ArgumentException("decreaseKey() got larger key value");

}

x.Key = k;

FibonacciHeapNode<T, TKey> y = x.Parent;

if (y != null && x.Key.CompareTo(y.Key) < 0)

{

Cut(x, y);

CascadingCut(y);

}

if (x.Key.CompareTo(\_minNode.Key) < 0)

{

\_minNode = x;

}

}

/// <summary>

/// Deletes a node from the heap.

/// O(log n)

/// </summary>

public void Delete(FibonacciHeapNode<T, TKey> x)

{

// make newParent as small as possible

DecreaseKey(x, \_minKeyValue);

// remove the smallest, which decreases n also

RemoveMin();

}

/// <summary>

/// Inserts a new node with its key.

/// O(1)

/// </summary>

public void Insert(FibonacciHeapNode<T, TKey> node)

{

// concatenate node into min list

if (\_minNode != null)

{

node.Left = \_minNode;

node.Right = \_minNode.Right;

\_minNode.Right = node;

node.Right.Left = node;

if (node.Key.CompareTo(\_minNode.Key) < 0)

{

\_minNode = node;

}

}

else

{

\_minNode = node;

}

\_nNodes++;

}

/// <summary>

/// Returns the smalles node of the heap.

/// O(1)

/// </summary>

/// <returns></returns>

public FibonacciHeapNode<T, TKey> Min()

{

return \_minNode;

}

/// <summary>

/// Removes the smalles node of the heap.

/// O(log n) amortized

/// </summary>

/// <returns></returns>

public FibonacciHeapNode<T, TKey> RemoveMin()

{

FibonacciHeapNode<T, TKey> minNode = \_minNode;

if (minNode != null)

{

int numKids = minNode.Degree;

FibonacciHeapNode<T, TKey> oldMinChild = minNode.Child;

// for each child of minNode do...

while (numKids > 0)

{

FibonacciHeapNode<T, TKey> tempRight = oldMinChild.Right;

// remove oldMinChild from child list

oldMinChild.Left.Right = oldMinChild.Right;

oldMinChild.Right.Left = oldMinChild.Left;

// add oldMinChild to root list of heap

oldMinChild.Left = \_minNode;

oldMinChild.Right = \_minNode.Right;

\_minNode.Right = oldMinChild;

oldMinChild.Right.Left = oldMinChild;

// set parent[oldMinChild] to null

oldMinChild.Parent = null;

oldMinChild = tempRight;

numKids--;

}

// remove minNode from root list of heap

minNode.Left.Right = minNode.Right;

minNode.Right.Left = minNode.Left;

if (minNode == minNode.Right)

{

\_minNode = null;

}

else

{

\_minNode = minNode.Right;

Consolidate();

}

// decrement size of heap

\_nNodes--;

}

return minNode;

}

/// <summary>

/// The number of nodes. O(1)

/// </summary>

/// <returns></returns>

public int Size()

{

return \_nNodes;

}

/// <summary>

/// Joins two heaps. O(1)

/// </summary>

/// <param name="h1"></param>

/// <param name="h2"></param>

/// <returns></returns>

public static FibonacciHeap<T, TKey> Union(FibonacciHeap<T, TKey> h1, FibonacciHeap<T, TKey> h2)

{

var h = new FibonacciHeap<T, TKey>(h1.\_minKeyValue.CompareTo(h2.\_minKeyValue) < 0

? h1.\_minKeyValue

: h2.\_minKeyValue);

if (h1 != null && h2 != null)

{

h.\_minNode = h1.\_minNode;

if (h.\_minNode != null)

{

if (h2.\_minNode != null)

{

h.\_minNode.Right.Left = h2.\_minNode.Left;

h2.\_minNode.Left.Right = h.\_minNode.Right;

h.\_minNode.Right = h2.\_minNode;

h2.\_minNode.Left = h.\_minNode;

if (h2.\_minNode.Key.CompareTo(h1.\_minNode.Key) < 0)

{

h.\_minNode = h2.\_minNode;

}

}

}

else

{

h.\_minNode = h2.\_minNode;

}

h.\_nNodes = h1.\_nNodes + h2.\_nNodes;

}

return h;

}

/// <summary>

/// Performs a cascading cut operation. This cuts newChild from its parent and then

/// does the same for its parent, and so on up the tree.

/// </summary>

private void CascadingCut(FibonacciHeapNode<T, TKey> y)

{

FibonacciHeapNode<T, TKey> z = y.Parent;

// if there's a parent...

if (z != null)

{

// if newChild is unmarked, set it marked

if (!y.Mark)

{

y.Mark = true;

}

else

{

// it's marked, cut it from parent

Cut(y, z);

// cut its parent as well

CascadingCut(z);

}

}

}

private void Consolidate()

{

int arraySize = (int)Math.Floor(Math.Log(\_nNodes) \* Constants.OneOverLogPhi) + 1;

var array = new List<FibonacciHeapNode<T, TKey>>(arraySize);

// Initialize degree array

for (var i = 0; i < arraySize; i++)

{

array.Add(null);

}

// Find the number of root nodes.

var numRoots = 0;

FibonacciHeapNode<T, TKey> x = \_minNode;

if (x != null)

{

numRoots++;

x = x.Right;

while (x != \_minNode)

{

numRoots++;

x = x.Right;

}

}

// For each node in root list do...

while (numRoots > 0)

{

// Access this node's degree..

int d = x.Degree;

FibonacciHeapNode<T, TKey> next = x.Right;

// ..and see if there's another of the same degree.

for (; ; )

{

FibonacciHeapNode<T, TKey> y = array[d];

if (y == null)

{

// Nope.

break;

}

// There is, make one of the nodes a child of the other.

// Do this based on the key value.

if (x.Key.CompareTo(y.Key) > 0)

{

FibonacciHeapNode<T, TKey> temp = y;

y = x;

x = temp;

}

// FibonacciHeapNode<T> newChild disappears from root list.

Link(y, x);

// We've handled this degree, go to next one.

array[d] = null;

d++;

}

// Save this node for later when we might encounter another

// of the same degree.

array[d] = x;

// Move forward through list.

x = next;

numRoots--;

}

// Set min to null (effectively losing the root list) and

// reconstruct the root list from the array entries in array[].

\_minNode = null;

for (var i = 0; i < arraySize; i++)

{

FibonacciHeapNode<T, TKey> y = array[i];

if (y == null)

{

continue;

}

// We've got a live one, add it to root list.

if (\_minNode != null)

{

// First remove node from root list.

y.Left.Right = y.Right;

y.Right.Left = y.Left;

// Now add to root list, again.

y.Left = \_minNode;

y.Right = \_minNode.Right;

\_minNode.Right = y;

y.Right.Left = y;

// Check if this is a new min.

if (y.Key.CompareTo(\_minNode.Key) < 0)

{

\_minNode = y;

}

}

else

{

\_minNode = y;

}

}

}

/// <summary>

/// The reverse of the link operation: removes newParent from the child list of newChild.

/// This method assumes that min is non-null.

/// Running time: O(1)

/// </summary>

private void Cut(FibonacciHeapNode<T, TKey> x, FibonacciHeapNode<T, TKey> y)

{

// remove newParent from childlist of newChild and decrement degree[newChild]

x.Left.Right = x.Right;

x.Right.Left = x.Left;

y.Degree--;

// reset newChild.child if necessary

if (y.Child == x)

{

y.Child = x.Right;

}

if (y.Degree == 0)

{

y.Child = null;

}

// add newParent to root list of heap

x.Left = \_minNode;

x.Right = \_minNode.Right;

\_minNode.Right = x;

x.Right.Left = x;

// set parent[newParent] to nil

x.Parent = null;

// set mark[newParent] to false

x.Mark = false;

}

/// <summary>

/// Makes newChild a child of Node newParent.

/// O(1)

/// </summary>

private static void Link(FibonacciHeapNode<T, TKey> newChild, FibonacciHeapNode<T, TKey> newParent)

{

// remove newChild from root list of heap

newChild.Left.Right = newChild.Right;

newChild.Right.Left = newChild.Left;

// make newChild a child of newParent

newChild.Parent = newParent;

if (newParent.Child == null)

{

newParent.Child = newChild;

newChild.Right = newChild;

newChild.Left = newChild;

}

else

{

newChild.Left = newParent.Child;

newChild.Right = newParent.Child.Right;

newParent.Child.Right = newChild;

newChild.Right.Left = newChild;

}

// increase degree[newParent]

newParent.Degree++;

// set mark[newChild] false

newChild.Mark = false;

}

}

}

**Functions Analysis**

Btnopen\_Click(): O(V)

DisplayImage(): O(V)

InitAdjacencyList(): O(V)

Testgraph(): O(E)

BtnGaussClick(): O(E)

PictureBox1\_Click(): O(V)

PictureBox1\_MouseMove(): O(VLog(V)+E)

PictureBox1\_DoubleClick(): O(VLog(V)+E)

Dijkstra(): O(VLog(V)+E)

Check(): O(1)

OpenImage(): O(V)

GetHeight(): O(1)

GetWidth(): O(1)

CalculatePixelEnergies(): O(1)

CalculateGradientAtPixel(): O(1)

**Graph Construction Description**

The main goal of the graph construction is to construct a weighted graph adjacency list. The main idea is to Iterate over each pixel in the image and calculate energies between its right and bottom neighbors, save that value in an array to be used to calculate energies of the top or left neighbors of another pixel. After that save indexes of neighboring pixels and their weights.