**Липецкий государственный технический университет**  
Факультет автоматизации и информатики  
Кафедра автоматизированных систем управления

Лабораторная работа №2

по предмету «Организация графических систем и систем мультимедиа»

Реализация алгоритма выделения границ и контуров Canny

Максимов А.В.

Студент

Группа М-АС-21

Кургасов В.В.

Руководитель

Доцент

Липецк 2022 г.

Задание кафедры

Реализация алгоритма выделения границ и контуров Canny. Наличие графического интерфейса пользователя. Обработка на GPU должна присутствовать.

Содержание

[Теоретическая часть 4](#_Toc103102515)

[Ход работы 5](#_Toc103102516)

[Вывод 7](#_Toc103102517)

[Приложение А 8](#_Toc103102518)

[Приложение Б 10](#_Toc103102519)

Теоретическая часть

Оператор Кэнни (детектор границ Кэнни, алгоритм Кэнни) в дисциплине компьютерного зрения — оператор обнаружения границ изображения. Был разработан в 1986 году Джоном Кэнни (англ. John F. Canny) и использует многоступенчатый алгоритм для обнаружения широкого спектра границ в изображениях.

Кэнни изучил математическую проблему получения фильтра, оптимального по критериям выделения, локализации и минимизации нескольких откликов одного края. Он показал, что искомый фильтр является суммой четырёх экспонент. Он также показал, что этот фильтр может быть хорошо приближен первой производной Гауссианы. Кэнни ввёл понятие подавления немаксимумов (англ. Non-Maximum Suppression), которое означает, что пикселями границ объявляются пиксели, в которых достигается локальный максимум градиента в направлении вектора градиента.

Хотя его работа была проведена на заре компьютерного зрения, детектор границ Кэнни до сих пор является одним из лучших детекторов. Кроме особенных частных случаев трудно найти детектор, который бы работал существенно лучше, чем детектор Кэнни.

Основные этапы алгоритма:

1. Сглаживание

Размытие изображения для удаления шума. Оператор Кэнни использует фильтр, который может быть хорошо приближен к первой производной гауссианы. σ = 1.4:

1. Поиск градиентов

Границы отмечаются там, где градиент изображения приобретает максимальное значение. Они могут иметь различное направление, поэтому алгоритм Кэнни использует четыре фильтра для обнаружения горизонтальных, вертикальных и диагональных ребер в размытом изображении.

Угол направления вектора градиента округляется и может принимать такие значения: 0, 45, 90, 135.

1. Подавление немаксимумов.

Только локальные максимумы отмечаются как границы.

1. Двойная пороговая фильтрация

Потенциальные границы определяются порогами.

1. Трассировка области неоднозначности.

Итоговые границы определяются путём подавления всех краёв, не связанных с определенными (сильными) границами.

Ход работы

На рисунке 1 представлен результат выполнения программы.

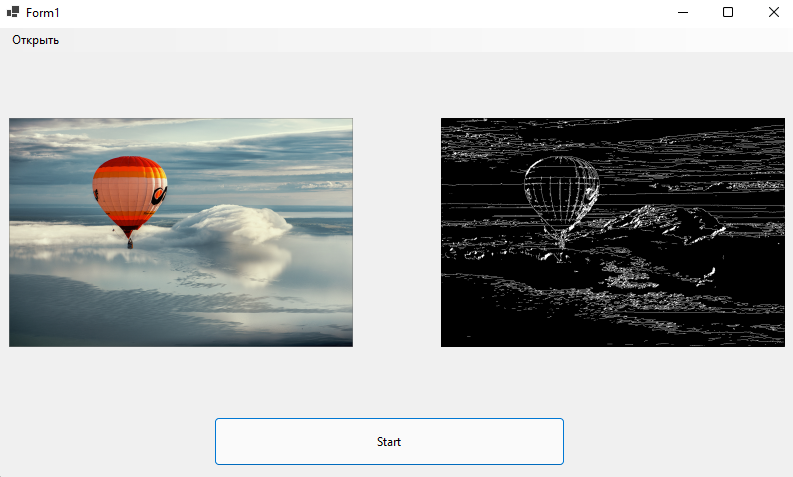


Рисунок 1 – Результат выполнения программы

Создадим инсталлятор для разработанного приложения.

Для этого установим плагин Microsoft Visual Studio Installer Projects. После этого добавим к решению инсталлятор. На рисунке 4 показана структура проекта после добавления инсталлятора.

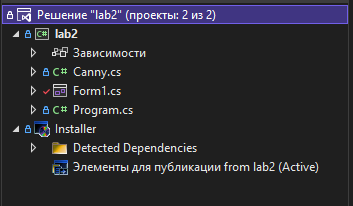


Рисунок 4 – Структура проекта

После сборки проекта получаем два файла: setup.exe и lab1.msi.

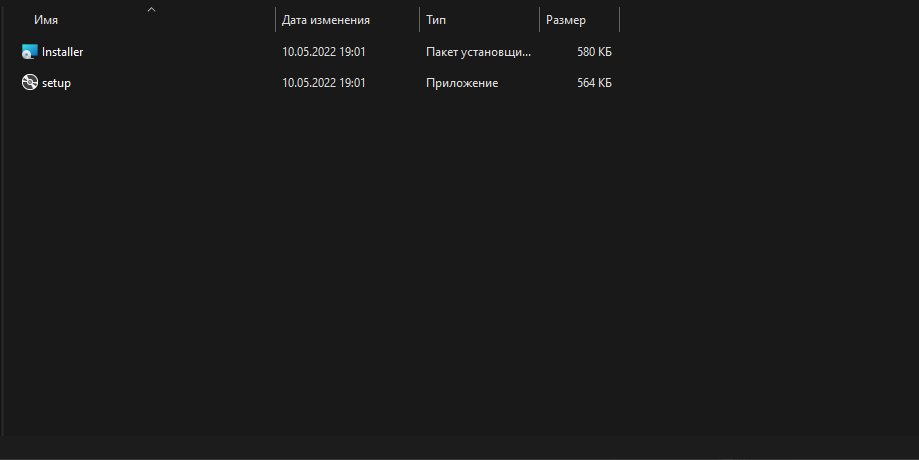


Рисунок 5 – Результат сборки проекта Installer.

Вывод

В ходе выполнения лабораторной работы было разработано приложение, определяющее контуры изображения с помощью алгоритма выделения границ и контуров Canny.

Для разработки использовался язык программирования C#, технологии Windows Form, а также библиотека для работы с изображениями EmguCV с модулем CUDA.

Приложение А

Исходный код главного компонента

using System.Drawing.Imaging;

using System.Runtime.InteropServices;

namespace lab2

{

public partial class Form1 : Form

{

private Bitmap inputImage;

public Form1()

{

InitializeComponent();

}

private void открытьToolStripMenuItem\_Click(object sender, EventArgs e)

{

try

{

DialogResult res = openFileDialog1.ShowDialog();

if (res == DialogResult.OK)

{

inputImage = new Bitmap(openFileDialog1.FileName);

pictureBox1.Image = inputImage;

}

else

{

MessageBox.Show("Файл не выбран", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

private void button1\_Click(object sender, EventArgs e)

{

Canny canny = new Canny(inputImage);

pictureBox2.Image = canny.DisplayImage(canny.EdgeMap);

}

}

}

Приложение Б

Исходный код алгоритма Canny

using System;

using System.Collections.Generic;

using System.Drawing.Imaging;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Emgu.CV.Cuda;

namespace lab2

{

internal class Canny

{

public int Width, Height;

public Bitmap Obj;

public int[,] GreyImage;

//Gaussian Kernel Data

int[,] GaussianKernel;

int KernelWeight;

int KernelSize = 5;

float Sigma = 1; // for N=2 Sigma =0.85 N=5 Sigma =1, N=9 Sigma = 2 2\*Sigma = (int)N/2

//Canny Edge Detection Parameters

float MaxHysteresisThresh, MinHysteresisThresh;

public float[,] DerivativeX;

public float[,] DerivativeY;

public int[,] FilteredImage;

public float[,] Gradient;

public float[,] NonMax;

public int[,] PostHysteresis;

int[,] EdgePoints;

public float[,] GNH;

public float[,] GNL;

public int[,] EdgeMap;

public int[,] VisitedMap;

public Canny(Bitmap Input)

{

// Gaussian and Canny Parameters

MaxHysteresisThresh = 20F;

MinHysteresisThresh = 10F;

Obj = Input;

Width = Obj.Width;

Height = Obj.Height;

EdgeMap = new int[Width, Height];

VisitedMap = new int[Width, Height];

ReadImage();

DetectCannyEdges();

return;

}

public Canny(Bitmap Input, float Th, float Tl)

{

// Gaussian and Canny Parameters

MaxHysteresisThresh = Th;

MinHysteresisThresh = Tl;

Obj = Input;

Width = Obj.Width;

Height = Obj.Height;

EdgeMap = new int[Width, Height];

VisitedMap = new int[Width, Height];

ReadImage();

DetectCannyEdges();

return;

}

public Canny(Bitmap Input, float Th, float Tl, int GaussianMaskSize, float SigmaforGaussianKernel)

{

// Gaussian and Canny Parameters

MaxHysteresisThresh = Th;

MinHysteresisThresh = Tl;

KernelSize = GaussianMaskSize;

Sigma = SigmaforGaussianKernel;

Obj = Input;

Width = Obj.Width;

Height = Obj.Height;

EdgeMap = new int[Width, Height];

VisitedMap = new int[Width, Height];

ReadImage();

DetectCannyEdges();

return;

}

public Bitmap DisplayImage()

{

int i, j;

Bitmap image = new Bitmap(Obj.Width, Obj.Height);

BitmapData bitmapData1 = image.LockBits(new Rectangle(0, 0, Obj.Width, Obj.Height),

ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

unsafe

{

byte\* imagePointer1 = (byte\*)bitmapData1.Scan0;

for (i = 0; i < bitmapData1.Height; i++)

{

for (j = 0; j < bitmapData1.Width; j++)

{

// write the logic implementation here

imagePointer1[0] = (byte)GreyImage[j, i];

imagePointer1[1] = (byte)GreyImage[j, i];

imagePointer1[2] = (byte)GreyImage[j, i];

imagePointer1[3] = (byte)255;

//4 bytes per pixel

imagePointer1 += 4;

}//end for j

//4 bytes per pixel

imagePointer1 += (bitmapData1.Stride - (bitmapData1.Width \* 4));

}//end for i

}//end unsafe

image.UnlockBits(bitmapData1);

return image;// col;

} // Display Grey Image

public Bitmap DisplayImage(float[,] GreyImage)

{

int i, j;

int W, H;

W = GreyImage.GetLength(0);

H = GreyImage.GetLength(1);

Bitmap image = new Bitmap(W, H);

BitmapData bitmapData1 = image.LockBits(new Rectangle(0, 0, W, H),

ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

unsafe

{

byte\* imagePointer1 = (byte\*)bitmapData1.Scan0;

for (i = 0; i < bitmapData1.Height; i++)

{

for (j = 0; j < bitmapData1.Width; j++)

{

// write the logic implementation here

imagePointer1[0] = (byte)((int)(GreyImage[j, i]));

imagePointer1[1] = (byte)((int)(GreyImage[j, i]));

imagePointer1[2] = (byte)((int)(GreyImage[j, i]));

imagePointer1[3] = (byte)255;

//4 bytes per pixel

imagePointer1 += 4;

} //end for j

//4 bytes per pixel

imagePointer1 += (bitmapData1.Stride - (bitmapData1.Width \* 4));

}//End for i

}//end unsafe

image.UnlockBits(bitmapData1);

return image;// col;

} // Display Grey Imag

public Bitmap DisplayImage(int[,] GreyImage)

{

int i, j;

int W, H;

W = GreyImage.GetLength(0);

H = GreyImage.GetLength(1);

Bitmap image = new Bitmap(W, H);

BitmapData bitmapData1 = image.LockBits(new Rectangle(0, 0, W, H),

ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

unsafe

{

byte\* imagePointer1 = (byte\*)bitmapData1.Scan0;

for (i = 0; i < bitmapData1.Height; i++)

{

for (j = 0; j < bitmapData1.Width; j++)

{

// write the logic implementation here

imagePointer1[0] = (byte)GreyImage[j, i];

imagePointer1[1] = (byte)GreyImage[j, i];

imagePointer1[2] = (byte)GreyImage[j, i];

imagePointer1[3] = (byte)255;

//4 bytes per pixel

imagePointer1 += 4;

} //end for j

//4 bytes per pixel

imagePointer1 += (bitmapData1.Stride - (bitmapData1.Width \* 4));

}//End for i

}//end unsafe

image.UnlockBits(bitmapData1);

return image;// col;

} // Display Grey Image

private void ReadImage()

{

int i, j;

GreyImage = new int[Obj.Width, Obj.Height]; //[Row,Column]

Bitmap image = Obj;

BitmapData bitmapData1 = image.LockBits(new Rectangle(0, 0, image.Width, image.Height),

ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

unsafe

{

byte\* imagePointer1 = (byte\*)bitmapData1.Scan0;

for (i = 0; i < bitmapData1.Height; i++)

{

for (j = 0; j < bitmapData1.Width; j++)

{

GreyImage[j, i] = (int)((imagePointer1[0] + imagePointer1[1] + imagePointer1[2]) / 3.0);

//4 bytes per pixel

imagePointer1 += 4;

}//end for j

//4 bytes per pixel

imagePointer1 += bitmapData1.Stride - (bitmapData1.Width \* 4);

}//end for i

}//end unsafe

image.UnlockBits(bitmapData1);

return;

}

private void GenerateGaussianKernel(int N, float S, out int Weight)

{

float Sigma = S;

float pi;

pi = (float)Math.PI;

int i, j;

int SizeofKernel = N;

float[,] Kernel = new float[N, N];

GaussianKernel = new int[N, N];

float[,] OP = new float[N, N];

float D1, D2;

D1 = 1 / (2 \* pi \* Sigma \* Sigma);

D2 = 2 \* Sigma \* Sigma;

float min = 1000;

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = ((1 / D1) \* (float)Math.Exp(-(i \* i + j \* j) / D2));

if (Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] < min)

min = Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

int mult = (int)(1 / min);

int sum = 0;

if ((min > 0) && (min < 1))

{

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (float)Math.Round(Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] \* mult, 0);

GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (int)Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

sum = sum + GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

}

else

{

sum = 0;

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (float)Math.Round(Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j], 0);

GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (int)Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

sum = sum + GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

}

//Normalizing kernel Weight

Weight = sum;

return;

}

private int[,] GaussianFilter(int[,] Data)

{

GenerateGaussianKernel(KernelSize, Sigma, out KernelWeight);

int[,] Output = new int[Width, Height];

int i, j, k, l;

int Limit = KernelSize / 2;

float Sum = 0;

Output = Data; // Removes Unwanted Data Omission due to kernel bias while convolution

for (i = Limit; i <= ((Width - 1) - Limit); i++)

{

for (j = Limit; j <= ((Height - 1) - Limit); j++)

{

Sum = 0;

for (k = -Limit; k <= Limit; k++)

{

for (l = -Limit; l <= Limit; l++)

{

Sum = Sum + ((float)Data[i + k, j + l] \* GaussianKernel[Limit + k, Limit + l]);

}

}

Output[i, j] = (int)(Math.Round(Sum / (float)KernelWeight));

}

}

return Output;

}

private float[,] Differentiate(int[,] Data, int[,] Filter)

{

int i, j, k, l, Fh, Fw;

Fw = Filter.GetLength(0);

Fh = Filter.GetLength(1);

float sum = 0;

float[,] Output = new float[Width, Height];

for (i = Fw / 2; i <= (Width - Fw / 2) - 1; i++)

{

for (j = Fh / 2; j <= (Height - Fh / 2) - 1; j++)

{

sum = 0;

for (k = -Fw / 2; k <= Fw / 2; k++)

{

for (l = -Fh / 2; l <= Fh / 2; l++)

{

sum = sum + Data[i + k, j + l] \* Filter[Fw / 2 + k, Fh / 2 + l];

}

}

Output[i, j] = sum;

}

}

return Output;

}

private void DetectCannyEdges()

{

Gradient = new float[Width, Height];

NonMax = new float[Width, Height];

PostHysteresis = new int[Width, Height];

DerivativeX = new float[Width, Height];

DerivativeY = new float[Width, Height];

//Gaussian Filter Input Image

FilteredImage = GaussianFilter(GreyImage);

//Sobel Masks

int[,] Dx = {{1,0,-1},

{1,0,-1},

{1,0,-1}};

int[,] Dy = {{1,1,1},

{0,0,0},

{-1,-1,-1}};

DerivativeX = Differentiate(FilteredImage, Dx);

DerivativeY = Differentiate(FilteredImage, Dy);

int i, j;

//Compute the gradient magnitude based on derivatives in x and y:

for (i = 0; i <= (Width - 1); i++)

{

for (j = 0; j <= (Height - 1); j++)

{

Gradient[i, j] = (float)Math.Sqrt((DerivativeX[i, j] \* DerivativeX[i, j]) + (DerivativeY[i, j] \* DerivativeY[i, j]));

}

}

// Perform Non maximum suppression:

// NonMax = Gradient;

for (i = 0; i <= (Width - 1); i++)

{

for (j = 0; j <= (Height - 1); j++)

{

NonMax[i, j] = Gradient[i, j];

}

}

int Limit = KernelSize / 2;

int r, c;

float Tangent;

for (i = Limit; i <= (Width - Limit) - 1; i++)

{

for (j = Limit; j <= (Height - Limit) - 1; j++)

{

if (DerivativeX[i, j] == 0)

Tangent = 90F;

else

Tangent = (float)(Math.Atan(DerivativeY[i, j] / DerivativeX[i, j]) \* 180 / Math.PI); //rad to degree

//Horizontal Edge

if (((-22.5 < Tangent) && (Tangent <= 22.5)) || ((157.5 < Tangent) && (Tangent <= -157.5)))

{

if ((Gradient[i, j] < Gradient[i, j + 1]) || (Gradient[i, j] < Gradient[i, j - 1]))

NonMax[i, j] = 0;

}

//Vertical Edge

if (((-112.5 < Tangent) && (Tangent <= -67.5)) || ((67.5 < Tangent) && (Tangent <= 112.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j]) || (Gradient[i, j] < Gradient[i - 1, j]))

NonMax[i, j] = 0;

}

//+45 Degree Edge

if (((-67.5 < Tangent) && (Tangent <= -22.5)) || ((112.5 < Tangent) && (Tangent <= 157.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j - 1]) || (Gradient[i, j] < Gradient[i - 1, j + 1]))

NonMax[i, j] = 0;

}

//-45 Degree Edge

if (((-157.5 < Tangent) && (Tangent <= -112.5)) || ((67.5 < Tangent) && (Tangent <= 22.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j + 1]) || (Gradient[i, j] < Gradient[i - 1, j - 1]))

NonMax[i, j] = 0;

}

}

}

//PostHysteresis = NonMax;

for (r = Limit; r <= (Width - Limit) - 1; r++)

{

for (c = Limit; c <= (Height - Limit) - 1; c++)

{

PostHysteresis[r, c] = (int)NonMax[r, c];

}

}

//Find Max and Min in Post Hysterisis

float min, max;

min = 100;

max = 0;

for (r = Limit; r <= (Width - Limit) - 1; r++)

for (c = Limit; c <= (Height - Limit) - 1; c++)

{

if (PostHysteresis[r, c] > max)

{

max = PostHysteresis[r, c];

}

if ((PostHysteresis[r, c] < min) && (PostHysteresis[r, c] > 0))

{

min = PostHysteresis[r, c];

}

}

GNH = new float[Width, Height];

GNL = new float[Width, Height]; ;

EdgePoints = new int[Width, Height];

for (r = Limit; r <= (Width - Limit) - 1; r++)

{

for (c = Limit; c <= (Height - Limit) - 1; c++)

{

if (PostHysteresis[r, c] >= MaxHysteresisThresh)

{

EdgePoints[r, c] = 1;

GNH[r, c] = 255;

}

if ((PostHysteresis[r, c] < MaxHysteresisThresh) && (PostHysteresis[r, c] >= MinHysteresisThresh))

{

EdgePoints[r, c] = 2;

GNL[r, c] = 255;

}

}

}

HysterisisThresholding(EdgePoints);

for (i = 0; i <= (Width - 1); i++)

for (j = 0; j <= (Height - 1); j++)

{

EdgeMap[i, j] = EdgeMap[i, j] \* 255;

}

return;

}

private void HysterisisThresholding(int[,] Edges)

{

int i, j;

int Limit = KernelSize / 2;

for (i = Limit; i <= (Width - 1) - Limit; i++)

for (j = Limit; j <= (Height - 1) - Limit; j++)

{

if (Edges[i, j] == 1)

{

EdgeMap[i, j] = 1;

}

}

for (i = Limit; i <= (Width - 1) - Limit; i++)

{

for (j = Limit; j <= (Height - 1) - Limit; j++)

{

if (Edges[i, j] == 1)

{

EdgeMap[i, j] = 1;

Travers(i, j);

VisitedMap[i, j] = 1;

}

}

}

return;

}

private void Travers(int X, int Y)

{

if (VisitedMap[X, Y] == 1)

{

return;

}

//1

if (EdgePoints[X + 1, Y] == 2)

{

EdgeMap[X + 1, Y] = 1;

VisitedMap[X + 1, Y] = 1;

Travers(X + 1, Y);

return;

}

//2

if (EdgePoints[X + 1, Y - 1] == 2)

{

EdgeMap[X + 1, Y - 1] = 1;

VisitedMap[X + 1, Y - 1] = 1;

Travers(X + 1, Y - 1);

return;

}

//3

if (EdgePoints[X, Y - 1] == 2)

{

EdgeMap[X, Y - 1] = 1;

VisitedMap[X, Y - 1] = 1;

Travers(X, Y - 1);

return;

}

//4

if (EdgePoints[X - 1, Y - 1] == 2)

{

EdgeMap[X - 1, Y - 1] = 1;

VisitedMap[X - 1, Y - 1] = 1;

Travers(X - 1, Y - 1);

return;

}

//5

if (EdgePoints[X - 1, Y] == 2)

{

EdgeMap[X - 1, Y] = 1;

VisitedMap[X - 1, Y] = 1;

Travers(X - 1, Y);

return;

}

//6

if (EdgePoints[X - 1, Y + 1] == 2)

{

EdgeMap[X - 1, Y + 1] = 1;

VisitedMap[X - 1, Y + 1] = 1;

Travers(X - 1, Y + 1);

return;

}

//7

if (EdgePoints[X, Y + 1] == 2)

{

EdgeMap[X, Y + 1] = 1;

VisitedMap[X, Y + 1] = 1;

Travers(X, Y + 1);

return;

}

//8

if (EdgePoints[X + 1, Y + 1] == 2)

{

EdgeMap[X + 1, Y + 1] = 1;

VisitedMap[X + 1, Y + 1] = 1;

Travers(X + 1, Y + 1);

return;

}

//VisitedMap[X, Y] = 1;

return;

}

}

}