**Problem Statement :**

Perform Distance Transform on the below-mentioned image.



Convert the image to binary and perform the distance transform. Share the distance matrix with neat documentation. From Scratch implementation with C++ is expected.

**Solution:**

***Aim:***

To covert the above image to binary and perform the distance transformation using c++ with opencv library.

***Explanation:***

A binary image is one that consists of pixels that can have one of exactly two colors, usually black and white. Binary images are also called bi-level or two-level. This means that each pixel is stored as a single bit—i.e., a 0 or 1. The names black-and-white, B&W, monochrome or monochromatic are often used for this concept, but may also designate any images that have only one sample per pixel, such as grayscale images. In Photoshop parlance, a binary image is the same as an image in "Bitmap" mode.

Binary images often arise in digital image processing as masks or thresholding, and dithering. Some input/output devices, such as laser printers, fax machines, and bilevel computer displays, can only handle bilevel images.

A binary image can be stored in memory as a bitmap, a packed array of bits. A 640×480 image requires 37.5 KiB of storage. Because of the small size of the image files, fax machine and document management solutions usually use this format. Most binary images also compress well with simple run-length compression schemes.

Binary images can be interpreted as subsets of the two-dimensional integer lattice Z2; the field of morphological image processing was largely inspired by this view.

* Use the OpenCV function **cv::filter2D** in order to perform some laplacian filtering for image sharpening
* Use the OpenCV function **cv::distancetransform** in order to obtain the derived representation of a binary image, where the value of each pixel is replaced by its distance to the nearest background pixel
* Use the OpenCV function **cv::watershed** in order to isolate objects in the image from the background

***C++ codes:***

#include <opencv2/core.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

#include <iostream>

using namespace std;

using namespace cv;

int main(int argc, char \*argv[])

{

// Load the image

CommandLineParser parser( argc, argv, "{@input | cards.png | input image}" );

Mat src = imread( samples::findFile( parser.get<String>( "@input" ) ) );

if( src.empty() )

{

cout << "Could not open or find the image!\n" << endl;

cout << "Usage: " << argv[0] << " <Input image>" << endl;

return -1;

}

// Show source image

imshow("Source Image", src);

// Change the background from white to black, since that will help later to extract

// better results during the use of Distance Transform

for ( int i = 0; i < src.rows; i++ ) {

for ( int j = 0; j < src.cols; j++ ) {

if ( src.at<Vec3b>(i, j) == Vec3b(255,255,255) )

{

src.at<Vec3b>(i, j)[0] = 0;

src.at<Vec3b>(i, j)[1] = 0;

src.at<Vec3b>(i, j)[2] = 0;

}

}

}

// Show output image

imshow("Black Background Image", src);

// Create a kernel that we will use to sharpen our image

Mat kernel = (Mat\_<float>(3,3) <<

1, 1, 1,

1, -8, 1,

1, 1, 1); // an approximation of second derivative, a quite strong kernel

// do the laplacian filtering as it is

// well, we need to convert everything in something more deeper then CV\_8U

// because the kernel has some negative values,

// and we can expect in general to have a Laplacian image with negative values

// BUT a 8bits unsigned int (the one we are working with) can contain values from 0 to 255

// so the possible negative number will be truncated

Mat imgLaplacian;

filter2D(src, imgLaplacian, CV\_32F, kernel);

Mat sharp;

src.convertTo(sharp, CV\_32F);

Mat imgResult = sharp - imgLaplacian;

// convert back to 8bits gray scale

imgResult.convertTo(imgResult, CV\_8UC3);

imgLaplacian.convertTo(imgLaplacian, CV\_8UC3);

// imshow( "Laplace Filtered Image", imgLaplacian );

imshow( "New Sharped Image", imgResult );

// Create binary image from source image

Mat bw;

cvtColor(imgResult, bw, COLOR\_BGR2GRAY);

threshold(bw, bw, 40, 255, THRESH\_BINARY | THRESH\_OTSU);

imshow("Binary Image", bw);

// Perform the distance transform algorithm

Mat dist;

distanceTransform(bw, dist, DIST\_L2, 3);

// Normalize the distance image for range = {0.0, 1.0}

// so we can visualize and threshold it

normalize(dist, dist, 0, 1.0, NORM\_MINMAX);

imshow("Distance Transform Image", dist);

// Threshold to obtain the peaks

// This will be the markers for the foreground objects

threshold(dist, dist, 0.4, 1.0, THRESH\_BINARY);

// Dilate a bit the dist image

Mat kernel1 = Mat::ones(3, 3, CV\_8U);

dilate(dist, dist, kernel1);

imshow("Peaks", dist);

// Create the CV\_8U version of the distance image

// It is needed for findContours()

Mat dist\_8u;

dist.convertTo(dist\_8u, CV\_8U);

// Find total markers

vector<vector<Point> > contours;

findContours(dist\_8u, contours, RETR\_EXTERNAL, CHAIN\_APPROX\_SIMPLE);

// Create the marker image for the watershed algorithm

Mat markers = Mat::zeros(dist.size(), CV\_32S);

// Draw the foreground markers

for (size\_t i = 0; i < contours.size(); i++)

{

drawContours(markers, contours, static\_cast<int>(i), Scalar(static\_cast<int>(i)+1), -1);

}

// Draw the background marker

circle(markers, Point(5,5), 3, Scalar(255), -1);

imshow("Markers", markers\*10000);

// Perform the watershed algorithm

watershed(imgResult, markers);

Mat mark;

markers.convertTo(mark, CV\_8U);

bitwise\_not(mark, mark);

// imshow("Markers\_v2", mark); // uncomment this if you want to see how the mark

// image looks like at that point

// Generate random colors

vector<Vec3b> colors;

for (size\_t i = 0; i < contours.size(); i++)

{

int b = theRNG().uniform(0, 256);

int g = theRNG().uniform(0, 256);

int r = theRNG().uniform(0, 256);

colors.push\_back(Vec3b((uchar)b, (uchar)g, (uchar)r));

}

// Create the result image

Mat dst = Mat::zeros(markers.size(), CV\_8UC3);

// Fill labeled objects with random colors

for (int i = 0; i < markers.rows; i++)

{

for (int j = 0; j < markers.cols; j++)

{

int index = markers.at<int>(i,j);

if (index > 0 && index <= static\_cast<int>(contours.size()))

{

dst.at<Vec3b>(i,j) = colors[index-1];

}

}

}

// Visualize the final image

imshow("Final Result", dst);

waitKey();

return 0;

}

***Output:***

***Binary image***

******

***Distance transform***

******