Homework #3

B05902120 / Yu-Ting, TSENG

Apr 4, 2019

Basic Execution

- Platform: Unix (MacBook)
- Language: C++

Problem1: Morphological Processing

Given a binary image I_1 as shown in Fig.1. Please follow the instructions below to create several new images along with discussions about the results.

(a) Perform boundary extraction on I_1 to extract the objects' boundaries and output the result as an image B.

The concept is to mark the inner pixels to be black. As a result, the difficulty would be deciding whether it is a inner pixel or not. The following is the corresponding code.

```
for (int i = 1; i < size - 1; i ++)
    for (int j = 1; j < size - 1; j ++){
        if (img[i][j] == 0) continue;

        if (img[i + 1][j + 1] != 255 ||
            img[i - 1][j + 1] != 255 ||
            img[i + 1][j - 1] != 255 ||
            img[i - 1][j - 1] != 255 ||
            img[i + 1][j] != 255 ||
            img[i - 1][j] != 255 ||
            img[i][j + 1] != 255 ||
            img[i][j - 1] != 255) tmp[i][j] = 255;
    }
```

To implement, I detect the neighbor pixels, if all of them were white means that they are inner pixel. If the pixels meet the above mentioning standard we give them new pixel values 0, which represents black. The resultant image is shown below.

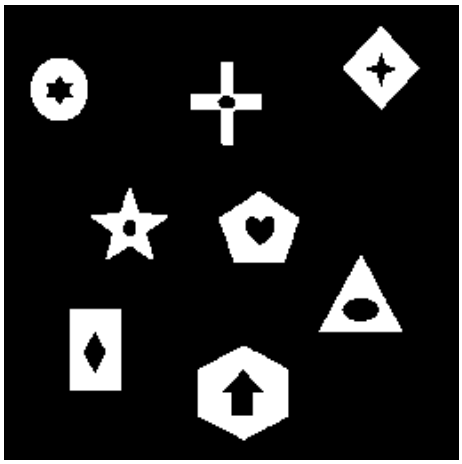

Figure 1: sample1.png

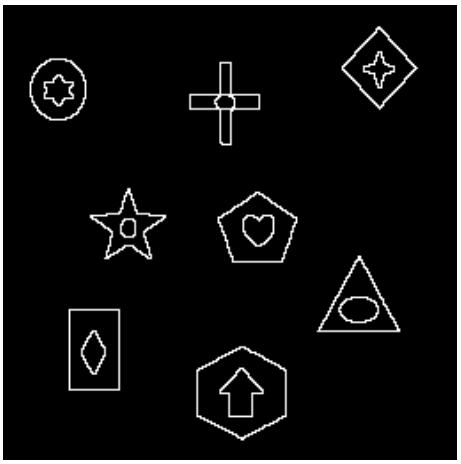

Figure 2: B.png

(b) Perform connected component labeling on I_1 to obtain an image C where different objects are labeled with different colors.

This problem is also easy to solve. However, we might meet some difficulties – how to output an image with RGB? According to the reference provided by TAs, we might find something has gone wrong. After searching on the Net, it is the 3 dimension arrays :"`size*size*3`" rather than "`3*size*size`" arrays.

```
void recursion(int lab, int i, int j){
    for (int m = -1; m <= 1; m ++)
        for (int n = -1; n <= 1; n ++){
            if (i + m < 0 || i + m > 255) continue;
            if (j + n < 0 || j + n > 255) continue;

            if (img[i + m][j + n] == 255 && tmp[i + m][j
                + n] == 0){
                tmp[i + m][j + n] = 255;
                res[i + m][j + n][0] = col[lab][0];
                res[i + m][j + n][1] = col[lab][1];
                res[i + m][j + n][2] = col[lab][2];
                recursion(lab, i + m, j + n);
            }
        }}
```

As we have learned the best way to find the connected component is scanning the whole image and implement bfs or dfs algorithm. As soon as we have labeled the pixel, we marked the pixels so that we won't repeatedly do the same thing. At last, we painted the pixels of different categorize by distinct color. Eventually, we could get the resultant image.

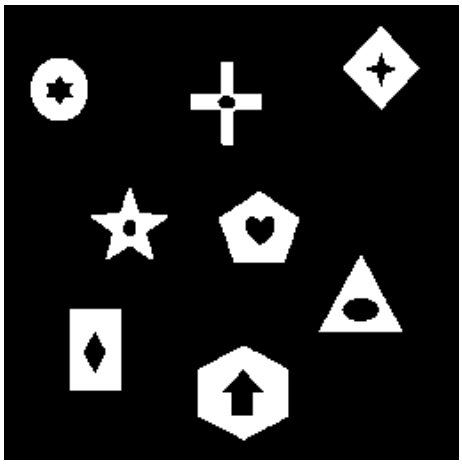

Figure 3: sample1.png

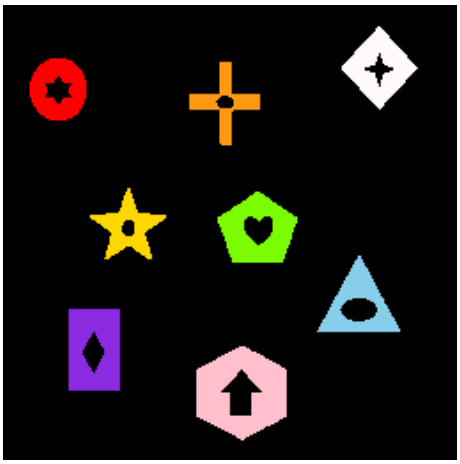

Figure 4: C.png

(c) Perform thinning and skeletonizing on I_1 and output the results as image D_1 and D_2. Originally, I don't really understand the difference skeletonizing and thinning thoroughly. Consulting the TAs, we know that skeletonize is to implement "thinning" until no more can be done. In other words, we can decide the extent of thinning; on the other hand, skeletonizing is extremely thinning. The most simple method to thin the image might be "Zhung and Suen Algorithm".

```
for (int i = 1; i < size - 1; i ++)
    for (int j = 1; j < size - 1; j ++){
        if (img[i][j] != 255) continue;

        int nei = 0;
        int num = 0;
        int pre = img[i + pos[7][0]][j + pos[7][1]];
        int fir[2] = {1, 1};
        int sec[2] = {1, 1};
        for (int k = 0; k < 8; k ++){
            int pix = img[i + pos[k][0]][j + pos[k][1]];
            if (pix == 255 && pre == 0) num ++;
            pre = pix;
```

```
            nei += pix / 255;
            if (k % 2 != 0) continue;
            if (k != 6) fir[0] *= pix / 255;
            if (k != 0) sec[0] *= pix / 255;
            if (k != 4) fir[1] *= pix / 255;
            if (k != 2) sec[1] *= pix / 255;
        }

        if (num != 1) continue;
        if (nei <= 2 || nei >= 7) continue;
        if (fir != 0 || sec != 0) continue;
        tmp[i][j] = 1;
        flg = chg = 1;
    }
```

There are 4 conditions to obey simultaneously. `nei` detected the number of the neighbor pixels that are white. If `nei` $= 0$, the pixel is isolated. If `nei` $= 1$, the pixel is the end of some object. If `nei` ≥ 7, the pixel is the center of an object. In these three situations, the pixels must not be disposed. `num` is the times that the surroundings pixel values change while considering in order. If `num` $\neq 1$, the pixel is the medium that connect different parts of the object, representing that it could not be deleted either. The other two rules are using to erase the side and corner pixels. We iterate these steps until nothing is changed and get the image of skeletonize. To discuss the difference them, we can input a number to decide how many iterations are going to be implemented.

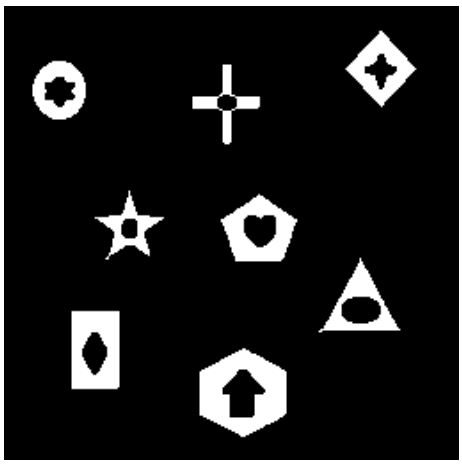

Figure 5: iteration= 1

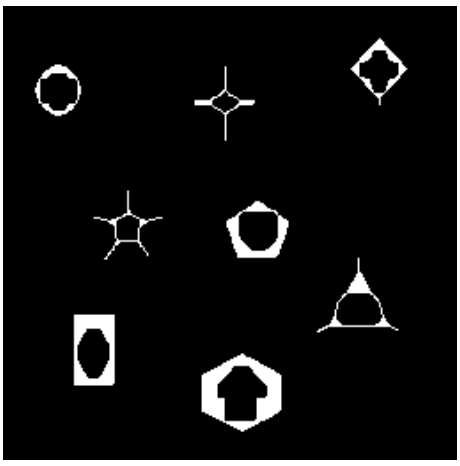

Figure 6: iteration= 3

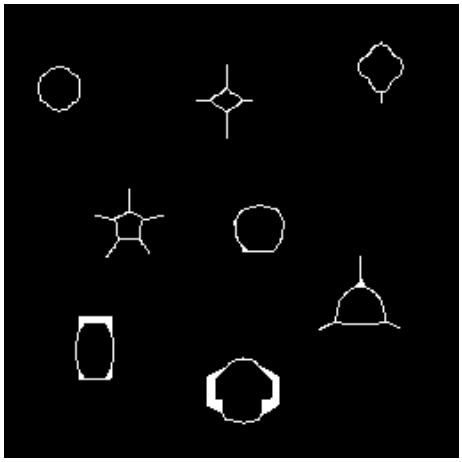

Figure 7: iteration= 5

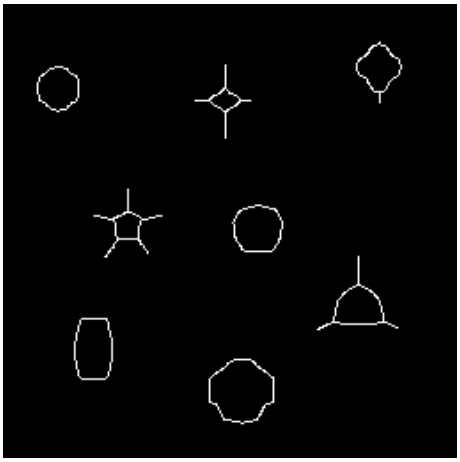

Figure 8: D_2.png

Problem2: Texture Analysis

As shown in Fig.2, image I_2 is composed of several different textures.

(a) Perform Law's method on I_2 to obtain the feature vector of each pixel.

There are two steps to analyze the texture, convolution and energy computation. We calculate the feature vectors of each pixels by certain masks. To do it simply, we might choose the basis set of impulse response arrays.

$$\frac{1}{36}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}, \frac{1}{12}\begin{bmatrix}1&0&-1\\2&0&-2\\1&0&-1\end{bmatrix}, \frac{1}{12}\begin{bmatrix}-1&2&-1\\-2&4&-2\\-1&2&-1\end{bmatrix} \tag{1}$$

$$\frac{1}{12}\begin{bmatrix}-1&2&-1\\0&0&0\\1&2&1\end{bmatrix}, \frac{1}{4}\begin{bmatrix}1&0&-1\\0&0&0\\-1&0&1\end{bmatrix}, \frac{1}{4}\begin{bmatrix}-1&2&-1\\0&0&0\\1&-2&1\end{bmatrix} \tag{2}$$

$$\frac{1}{12}\begin{bmatrix}-1&2&-1\\2&4&2\\-1&2&-1\end{bmatrix}, \frac{1}{4}\begin{bmatrix}-1&0&1\\2&0&-2\\-1&0&1\end{bmatrix}, \frac{1}{4}\begin{bmatrix}1&-2&1\\-2&4&-2\\1&-2&1\end{bmatrix} \tag{3}$$

After we get the microstructure array, we start doing step2, energy computation. Accumulated the square of the values just obtain in the window and fill into the feature vectors.

During class, professor mentioned that we usually set the window size $13 * 13$ or $15 * 15$. However, we might find that the result isn't as well as expected. Therefore, we can give tries to variety of size, 3, 11, 21, 37 respectively. The results would be shown within Problem.2(b).

(b) Use k-means to classify each pixel and label same kind of texture with same gray-level intensity. Please output the result as E.

K-means algorithm is an important classify method in machine learning. Supposed we are required to classify whole data into k categories. The basic idea is to choose k points as the initial center vectors. Next, we decide which groups those points belong to by the Euclidean distance between each center vectors and the points. The points belong to the groups to which the distance is minimum.

```
int distant(unsigned char fea[9], int avg[9], int wei
   [9]){
    int tot = 0;
    for (int k = 0; k < 9; k ++)
        tot += (fea[k] - avg[k]) * (fea[k] - avg[k]) *
           wei[k];
    return int(sqrt(tot / 18));
}
```

Subsequently, recompute the center vectors by calculating the average values of the points in each groups. Continuously do the same thing until things converge.

```
for (int i = 0; i < size; i ++)
    for (int j = 0; j < size; j ++){
        // "ind" is the group num
        for (int k = 0; k < 9; k ++)
            cal[ind][k] += res[i][j][k];
        ara[ind] += 1;
    }


for (int n = 0; n < 4; n ++)
        for (int k = 0; k < 9; k ++)
            avg[n][k] = cal[n][k] / ara[n];
```

At last, we paint the area that in the same category with the same gray-scale intensity. With the different window size (discussed in Problem.2(a)), we can get entirely different images. While implementing, we can find the consuming time obviously increases as the window size become larger in the sake of great computation in the second step of obtaining the feature vectors.

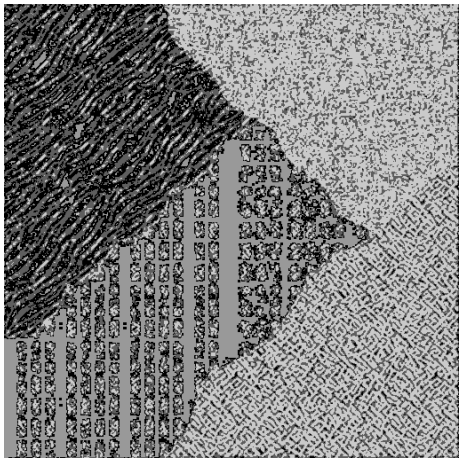

Figure 9: window size= 3

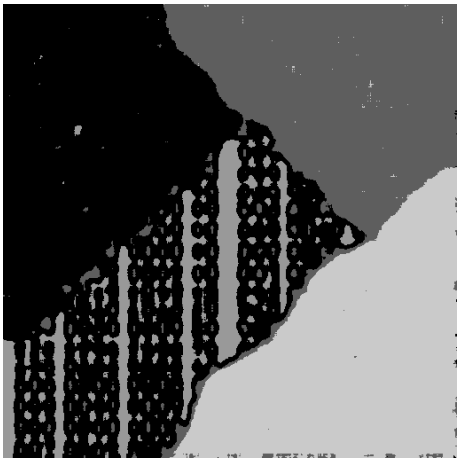

Figure 10: window size= 11

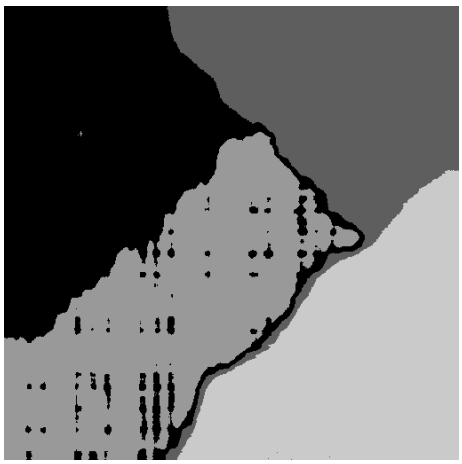

Figure 11: window size= 21

Figure 12: E.png window size= 37

We could observe that while the window is small, the results is full of variety within a small area, whereas the window is large, it separate more apparently. It makes sense, because the size of the window has an influence on the feature vectors. We identify the texture by considering each shown appearance in the window. If the window is large enough to included all basic pixels composed the texture, it will be considered to be the same texture.

(c) Based on E, try to generate another texture image by exchanging the types of different texture patterns. Please output the result as G.

The third problem is much easier than the frontier problem. What we need to do is identify the different colors and design some texture. I choose some simple texture, grid, crisscross, strip and black.

```
for (int i = 0; i < size; i ++)
    for (int j = 0; j < size; j ++){
        if (img[i][j] == 60){
```

```
            tmp[i][j] = 40;
            if (i % 16 == 0) tmp[i][j] = 80;
            if (j % 16 == 1) tmp[i][j] = 80;
        }

        if (img[i][j] == 120){
            tmp[i][j] = 100;
            if (i/16 % 2 != j/16 % 2) tmp[i][j] = 135;
        }

        if (img[i][j] == 180){
            if ((i + j) / 16 % 2 == 0) tmp[i][j] = 160;
            if ((i + j) / 16 % 2 == 1) tmp[i][j] = 225;
        }
    }
```

The result image is displayed below.

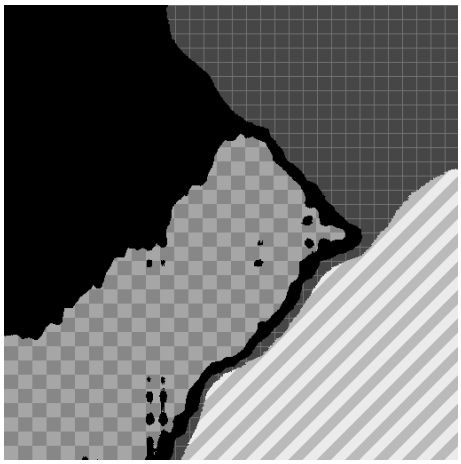

Figure 13: *G*.png

Compared to the original image, we might find some area is not judging well, as well as the image is relative simple and easy to distinguish. I am convinced that it is hard to create a complex texture by small computation. However, if I have time I would like to try something unique.

Bonus

(a) Fig.3 shows a gray-level image I_3. Please design an algorithm to count the number of berries in the image. Please describe the proposed method in detail.

We have gained a great variety of knowledge from the course. The first method come in mind is edge detection. Due to the computation, we detect the edge by 1^{st} Order Gradient, and get the image.

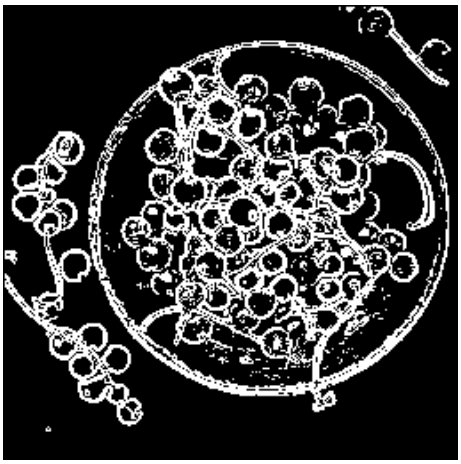

Figure 14: H_1.png

It seems that the edge isn't clear enough, therefore we could adopt closing operation, implementing Dilation and Erosion in turn. The reason why we need to do this is that we can connect gap between the edges which are actually the same. However, we might also make some mistakes, eroding the wrong edges. The resultant image ia as follow:

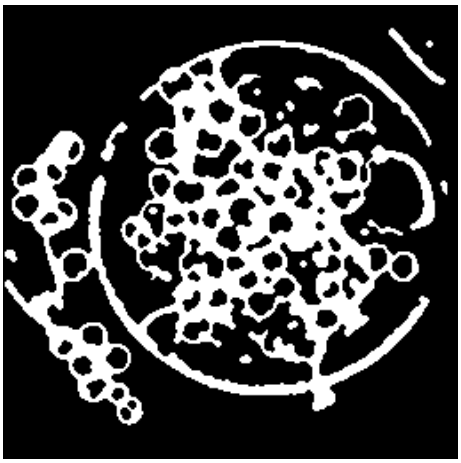

Figure 15: H_2.png

What we do next is to skeletonize the edges, so that the lines will become more clearly and no unnecessary lines appears in the image. The result is the image on the left side below. We can find that we can easily identify the different district by our eyes. To make things more simple, we colored the background white, so each black area would represent a berry.

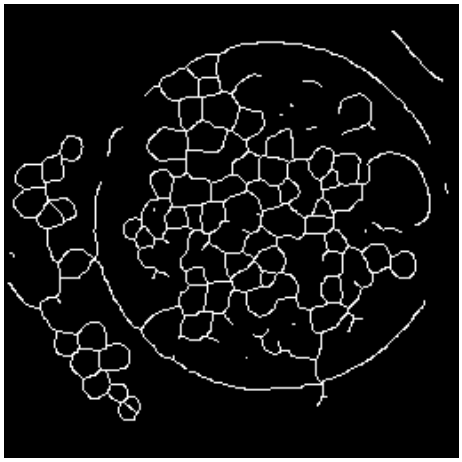

Figure 16: H_3.png

Figure 17: H_4.png

Eventually, we shrink the black part until a pixel symbolize an area; we can calculate the number of the berries by counting the black pixels left. The result is displayed as follow. Due to the unideal result, I come up with another idea.

```
prodigy1026 at TSENG-MacBook in ~/Documents/NationalTaiwanUniversity/Sub27-Digit
alImageProcessing/Assignment/Assign3 executing
> ./a.out
There are 65 berries in the image!
```

Figure 18: Result

The other possible way is to identify the background and berries part by texture analysis. After edge detection, if the area of the black pixels is within a certain range and the it is the texture of berries, accumulate the summation by 1.

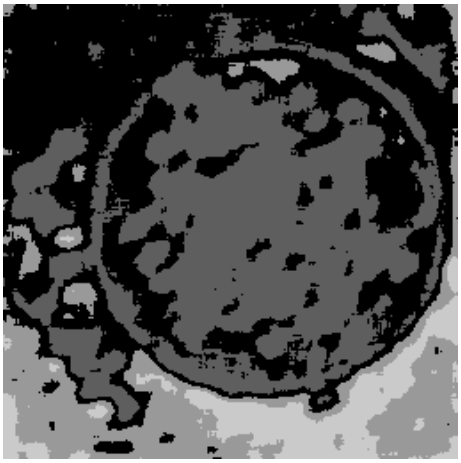

Figure 19: H_5.png

Appendix

- `Problem1_a.cpp` to execute boundary extraction.

Compile by `g++ Problem1_a.cpp -o Problem1_a`;
execute by `./Problem1_a` and get the output image `B.raw`.

- `Problem1_b.cpp` to execute connected component labeling.
  Compile by `g++ Problem1_b.cpp -o Problem1_b`;
  execute by `./Problem1_b` get the output image `C.raw`.

- `Problem1_c1.cpp` to execute thinning.
  Compile by `g++ Problem1_c1.cpp -o Problem1_c1`;
  execute by `./Problem1_c1` and input the iterating times then get the output image `D_1.raw`.

- `Problem1_c2.cpp` to execute skeletonizing.
  Compile by `g++ Problem1_c2.cpp -o Problem1_c2`;
  execute by `./Problem1_c2` and get the output image `D_2.raw`.

- `Problem2_ab.cpp` to execute edge crispening.
  Compile by `g++ Problem2_ab.cpp -o Problem2_ab`;
  execute by `./Problem2_ab` and input the size of the window then get the output image `E.raw`.

- `Problem3_1.cpp` to execute edge detection by 1^{st} order gradient.
  Compile by `g++ Problem3_1.cpp -o Problem3_1`;
  execute by `./Problem3_1` and get the output image `H_1.raw`.

- `Problem3_2.cpp` to execute closing by dilation first and erosion subsequently.
  Compile by `g++ Problem3_2.cpp -o Problem3_2`;
  execute by `./Problem3_2` and get the output image `H_2.raw`.

- `Bonus_3.cpp` to erase background and execute shrinking.
  Compile by `g++ Bonus_3.cpp -o Bonus_3`;
  execute by `./Bonus_3` and get the output image `H_4.raw`.