**Lab 8**

Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. **erosion(noisy\_fingerprint, noise\_rectangle):**

**Algorithm:**

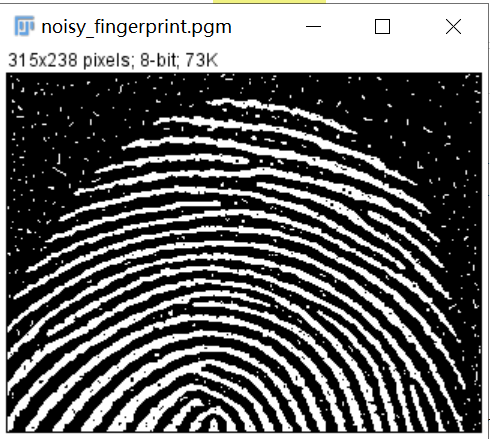
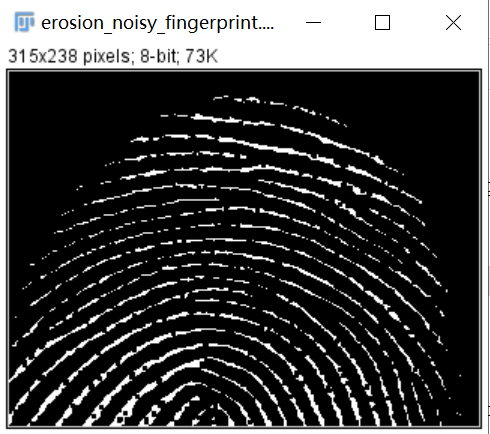
Define a 3×3 structural element and use it scan the image.

If there is a value that equals to 0 in the structural element, then the center of the structural element will be 0.

Only when all the values in the structural element equal to 255, the center of the structural element will be 255.

**Results (including pictures):**

Source: result:

Source:



Result:



**Discussion:**

In the noisy\_fingerprint.pgm, we can see that the noise has been removed because the erosion operation make the white pixel value in small region disappears.

However, in thenoise**\_**rectangle.pgm, the white noise gathers in the big area which is bigger than 3×3, so the noise becomes small after the erosion and not completely been removed. The white rectangle becomes smaller and the letter in the lower left corner becomes mumble. The black noise in the white rectangle becomes bigger.

**Codes:**

Image \*erosion(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "white connected component";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

int i, j, x, y,flag;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

flag = 1;

for (x = i - 1; x < i + 2; x++){

for (y = j - 1; y < j + 2; y++){

if (tempin[x\*width + y] == 0){

flag = 0;

break;

}

}

if (flag == 0){

break;

}

}

if (flag == 0){

tempout[i\*width + j] = 0;

}

else{

tempout[i\*width + j] = 255;

}

}

}return(outimage);

}

1. **dilation(noisy\_fingerprint,** **noise\_rectangle):**

**Algorithm:**

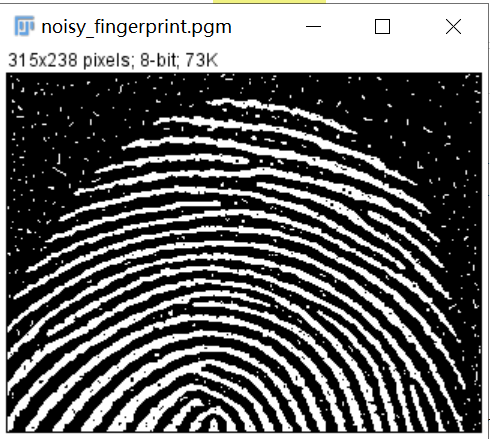
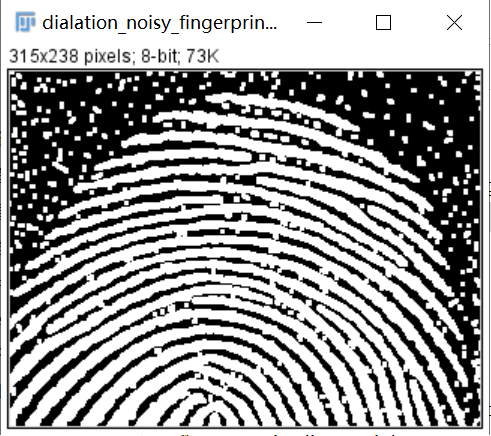
Define a 3×3 structural element and use it scan the image.

If there is a value that equals to 255 in the structural element, then the center of the structural element will be 255.

Only when all the values in the structural element equal to 0, the center of the structural element will be 0.

**Results (including pictures):**

Source: result:

Source:



Result:



**Discussion:**

In the noisy\_fingerprint.pgm, we can see that the noise becomes bigger because of the dilation operation.

In thenoise**\_**rectangle.pgm, the white noise becomes bigger after the dilation. The white rectangle becomes bigger and the lines of the letter become thicker. The black noise in the white rectangle becomes smaller.

**Codes:**

Image \*dialation(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "dialation";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

int i, j, x, y, flag;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

flag = 1;

for (x = i - 1; x < i + 2; x++){

for (y = j - 1; y < j + 2; y++){

if (tempin[x\*width + y] == 255){

flag = 0;

break;

}

}

if (flag == 0){

break;

}

}

if (flag == 0){

tempout[i\*width + j] = 255;

}

else{

tempout[i\*width + j] = 0;

}

}

}return(outimage);

}

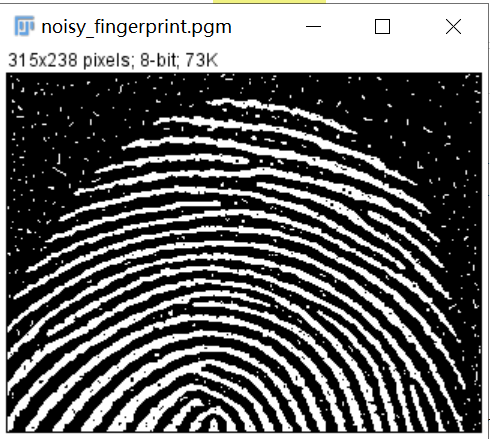
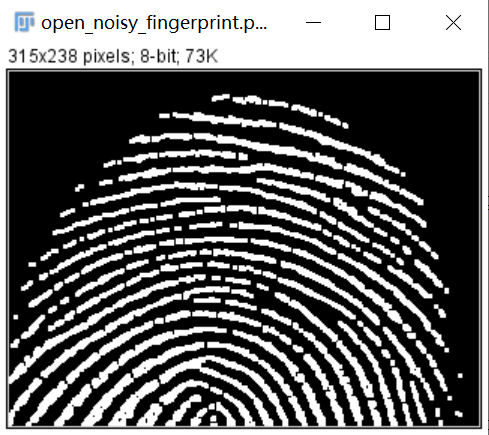
1. **opening (noisy\_fingerprint, noise\_rectangle):**

**Algorithm:**

First use the erosion to process the image, then use the dilation to process the image after erosion.

**Results (including pictures):**

Source: result:

Source:



Result:



**Discussion:**

After the erosion with appropriate structural elements, the white noise can be removed.

After the dilation, the detail of the image can be enlarged.

**Codes:**

Image \*open(Image\* image) {

return(dialation(erosion(image)));

}

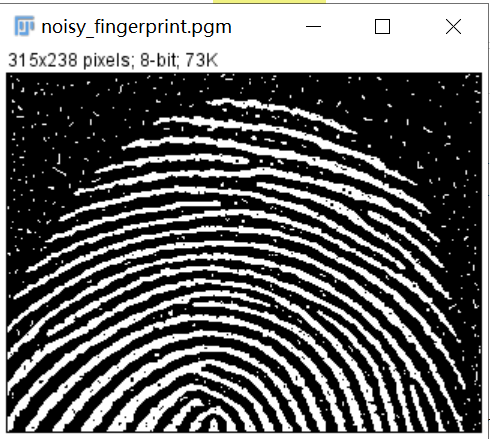
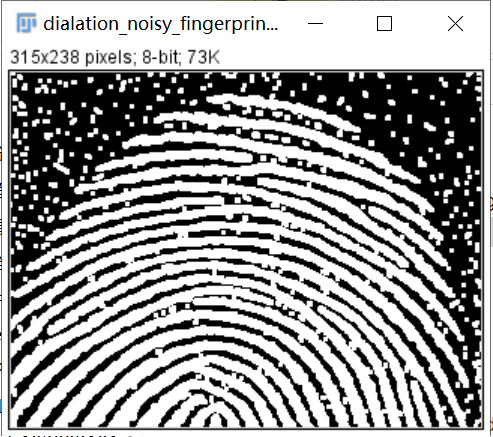
1. **closing(noisy\_fingerprint, noise\_rectangle):**

**Algorithm:**

First use the dilation to process the image, then use the erosion to process the image after erosion.

**Results (including pictures):**

Source: result:

Source:



Result:



**Discussion:**

After the dilation, the white noise will be enlarged and it will be more difficult to be removed.

After the erosion, the white noise has not been removed effectively because the dilation enlarges the white noise.

**Codes:**

Image \*close(Image\* image) {

return(erosion(dialation(image)));

}

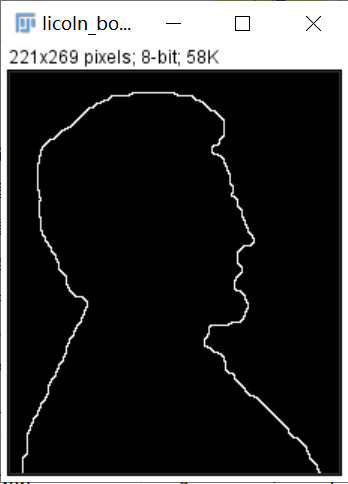
1. **extract the boundaries (licoln, U):**

**Algorithm:**

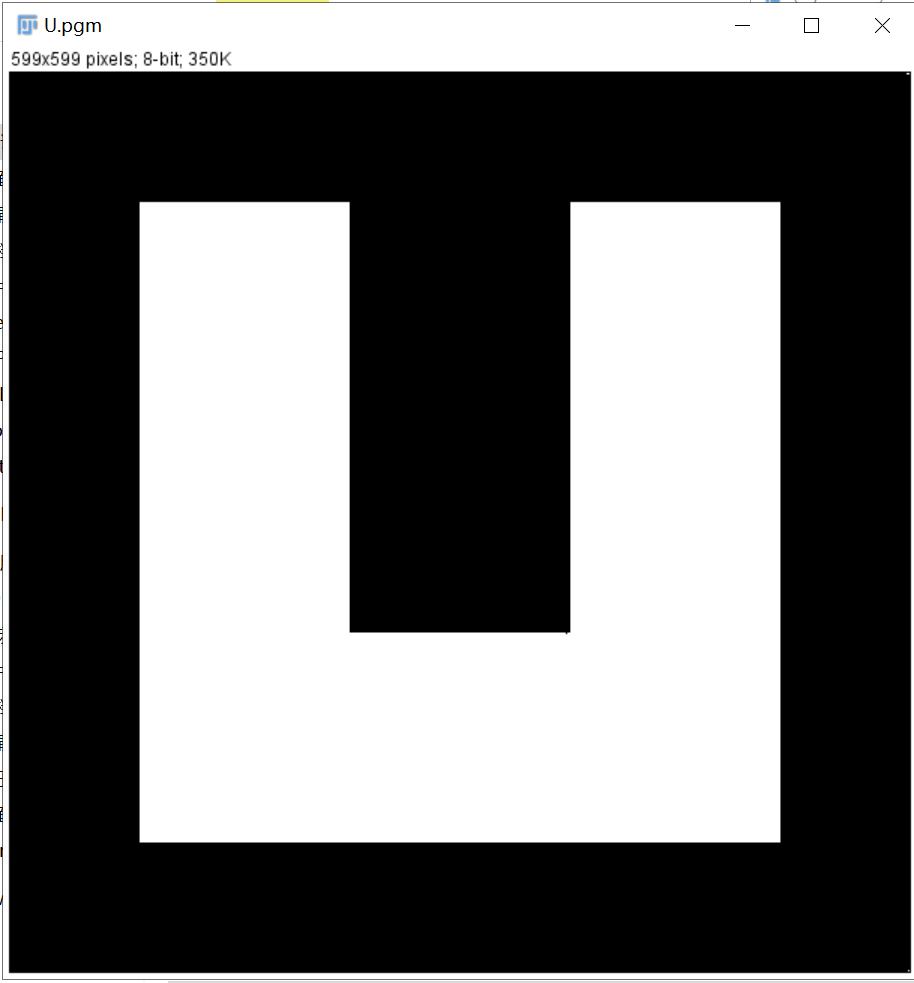
First use the erosion to process the image, then subtract the values in processed image from the original image.

**Results (including pictures):**

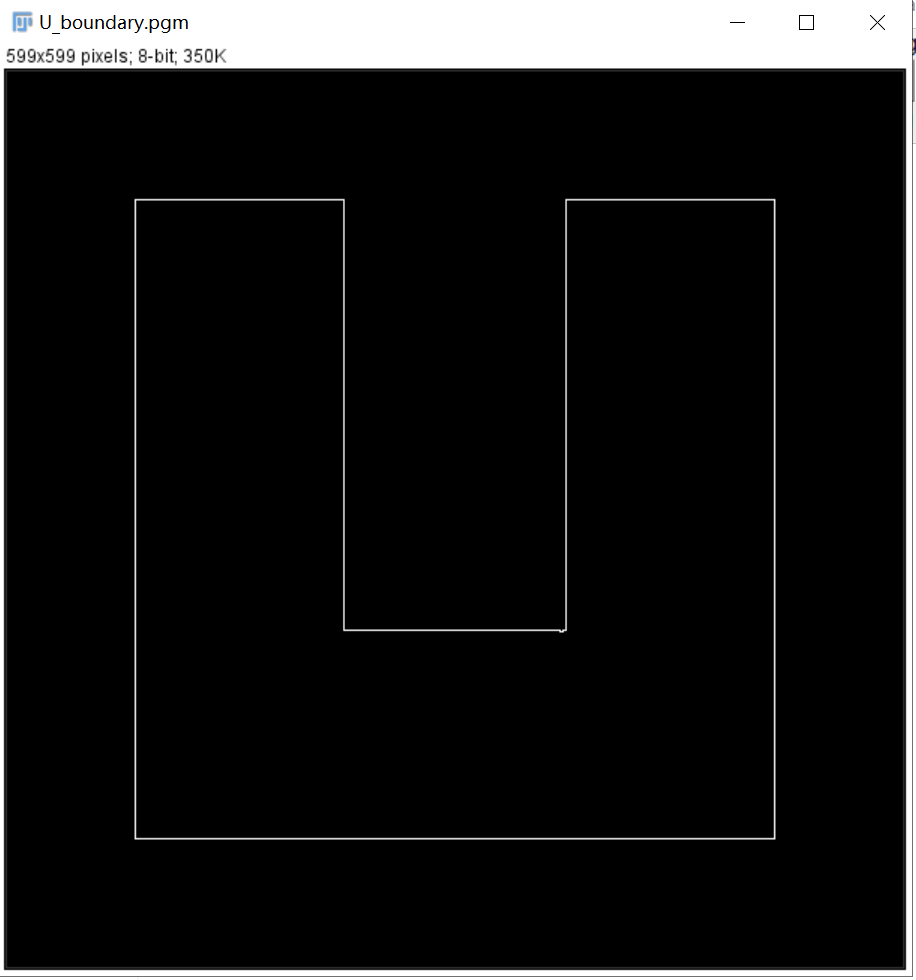
Source: result:

Source:



result:



**Discussion:**

We can see the boundaries has been extracted perfectly.

**Codes:**

Image \*Extractboundaries(Image\* image){

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "Extractboundaries";

outimage = CreateNewImage(image, comment);

tempin = image->data;

int i, j;

outimage = erosion(image);

tempout = outimage->data;

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

tempout[i\*width + j] = tempin[i\*width + j] - tempout[i\*width + j];

}

}return(outimage);

}

1. **Count** **the number of pixels in each white connected component from and put them into a .txt file (connected):**

**Algorithm:**

Using the idea of regional growth:

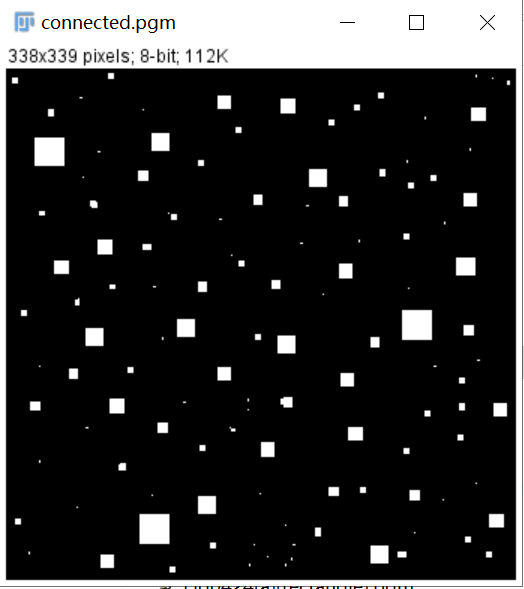
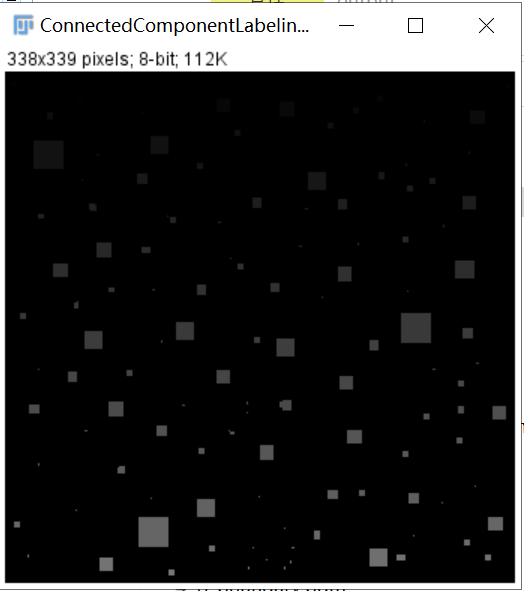
Input the image ***tempin*** to be marked, initialize a marking matrix ***tempout*** with the same size as the input image, a queue and a marking count ***labelindex***; Scan ***tempin*** from left to right and from top to bottom. When an unmarked foreground pixel P is scanned, ***labelindex*** is added by 1, and P is marked in ***tempout*** (the value of the corresponding point is assigned as ***labelindex***). At the same time, scan the eight neighborhood points of P. if there are unmarked foreground pixels, mark them in ***tempout*** and put them into queue as seeds for regional growth; When the queue is not empty, take out a growing seed point P1 from the queue and scan the eight neighborhood points of P1. If there are unmarked foreground pixels, mark them in ***tempout*** and put them into the queue; Repeat 3 until the queue is empty and a connected area is marked; Turn to 2 until the whole image is scanned, and get the marking matrix ***tempout*** and the number of connected areas ***labelindex***.

After the labeling, for example, the value in first connected region is 1, the value in second connected region is 2 and so on. Then count the number of the value and the result is the number of pixels in each white connected component.

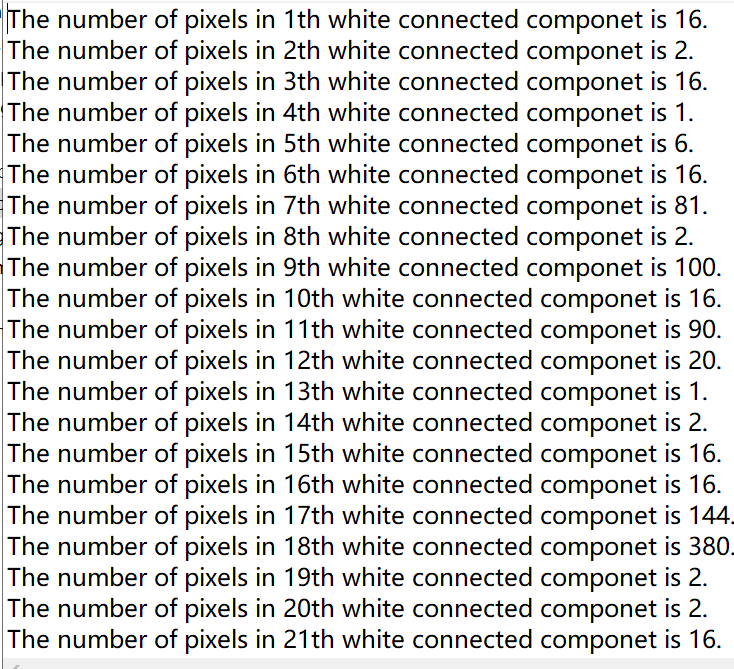
Finally, store these information into the .txt file.

**Results (including pictures):**

Source: result(after labeling):

Result(the screenshot of txt file):



**Discussion:**

The advantage of the algorithm is easy to understand, but the disadvantage is that it needs to scan the image twice and the efficiency is not high. However, the image data is not very large, so this is not a big problem.

In the worst case, the algorithm will perform an eight neighborhood search for each pixel, and the complexity of the algorithm is O (n).

Codes:

typedef struct QNode{

int data;

struct QNode \*next;

}QNode;

typedef struct Queue{

struct QNode\* first;

struct QNode\* last;

}Queue;

void PushQueue(Queue \*queue, int data){

QNode \*p = NULL;

p = (QNode\*)malloc(sizeof(QNode));

p->data = data;

if (queue->first == NULL){

queue->first = p;

queue->last = p;

p->next = NULL;

}

else{

p->next = NULL;

queue->last->next = p;

queue->last = p;

}

}

int PopQueue(Queue \*queue){

QNode \*p = NULL;

int data;

if (queue->first == NULL){

return -1;

}

p = queue->first;

data = p->data;

if (queue->first->next == NULL){

queue->first = NULL;

queue->last = NULL;

}

else{

queue->first = p->next;

}

free(p);

return data;

}

void SearchNeighbor(unsigned char \*tempin, int width, int height, unsigned char \*tempout, int labelIndex, int pixelIndex, Queue \*queue){

static int NeighborDirection[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 }, { 0, -1 }, { -1, -1 }, { -1, 0 }, { -1, 1 } };

int searchIndex, i, length;

tempout[pixelIndex] = labelIndex;

length = width \* height;

for (i = 0; i < 8; i++){

searchIndex = pixelIndex + NeighborDirection[i][0] \* width + NeighborDirection[i][1];

if (searchIndex > 0 && searchIndex < length &&

tempin[searchIndex] == 255 && tempout[searchIndex] == 0){

tempout[searchIndex] = labelIndex;

PushQueue(queue, searchIndex);

}

}

}

void countNumberOfPixelsInWhiteConnectedComponetAndPutIntoTXT(Image\* image){

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "countNumberOfPixelsInWhiteConnectedComponetAndPutIntoTXT";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

memset(tempout, 0, width \* height);

int i, j, popIndex, labelIndex = 0;

Queue \*queue = NULL;

queue = (Queue\*)malloc(sizeof(Queue));

queue->first = NULL;

queue->last = NULL;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

if (tempin[i \* width + j] == 255 && tempout[i \* width + j] == 0){

labelIndex++;

SearchNeighbor(tempin, width, height, tempout, labelIndex, i \* width + j, queue);

popIndex = PopQueue(queue);

while (popIndex > -1){

SearchNeighbor(tempin, width, height, tempout, labelIndex, popIndex, queue);

popIndex = PopQueue(queue);

}

}

}

}

free(queue);

SavePNMImage(outimage, "..\\output\\ConnectedComponentLabeling.pgm");

int k, count;

FILE\* fp = NULL;

fp = fopen("..\\output\\numberOfPixelsInWhiteConnectedComponet.txt", "w+");

for (k = 1; k <= labelIndex; k++){

count = 0;

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

count++;

}

}

}

fprintf(fp, "The number of pixels in %dth white connected componet is %d.\n", k, count);

}

fclose(fp);

}

1. **Seprdate Particles Merged With Boundries (bubbles\_on\_black\_background):**

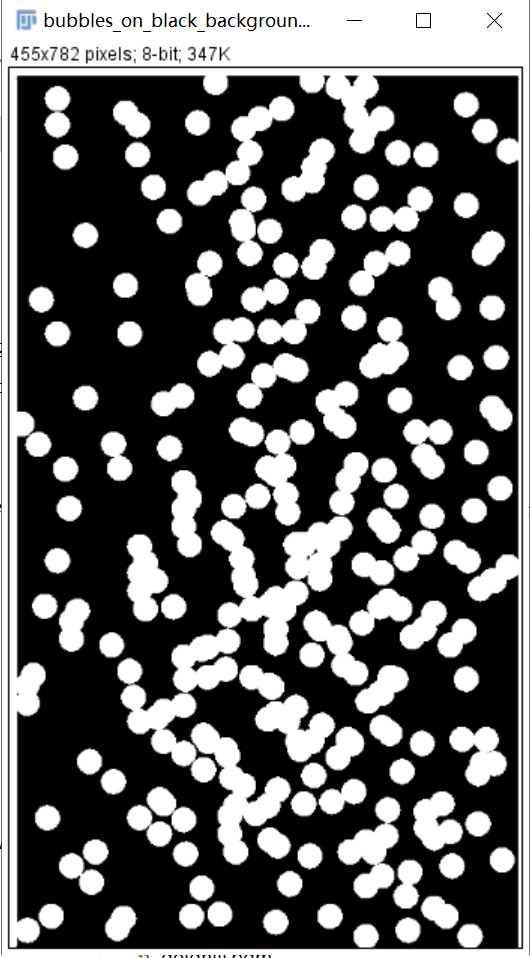
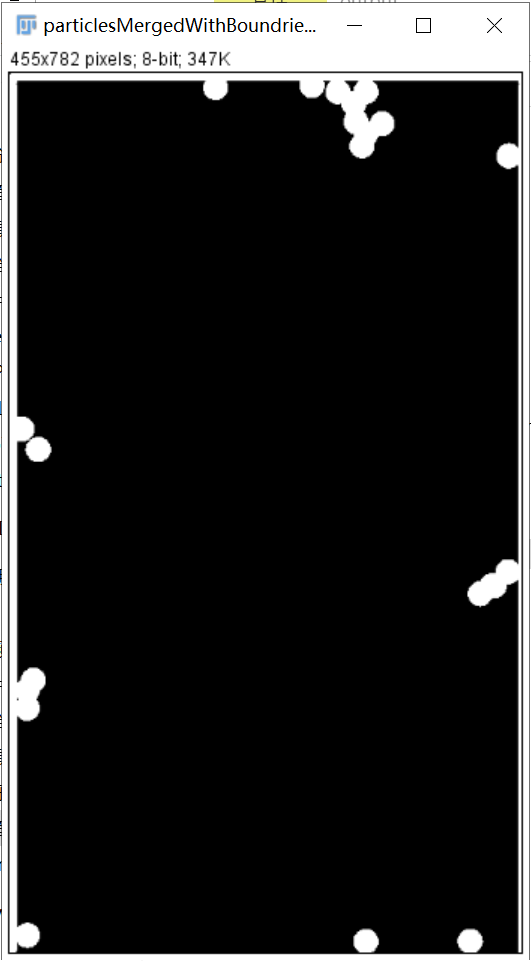
**Algorithm:**

Set all boundary values to 255 to make the boundary and particles merged with boundaries form a white connected area.

Then label the connected area. In this situation, the boundary and particles merged with boundaries is the first white connected area and the all the values are one. Then set all these values to 255 and others to 0.

**Results (including pictures):**

Source: result:

**Discussion:**

We can see the particles merged with boundaries have been separated.

**Codes:**

Image \*seprdateParticlesMergedWithBoundries(Image\* image){

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "splitParticlesMergedWithBoundries";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

memset(tempout, 0, width \* height);

int i, j, popIndex, labelIndex = 0;

for (j = 0; j < width; j++){

tempin[j] = 255;

}

for (j = 0; j < width; j++){

tempin[(height - 1) \* width + j] = 255;

}

for (i = 0; i < height; i++){

tempin[i \* width] = 255;

}

for (i = 0; i < height; i++){

tempin[i \* width + width - 1] = 255;

}

Queue \*queue = NULL;

queue = (Queue\*)malloc(sizeof(Queue));

queue->first = NULL;

queue->last = NULL;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

if (tempin[i \* width + j] == 255 && tempout[i \* width + j] == 0){

labelIndex++;

SearchNeighbor(tempin, width, height, tempout, labelIndex, i \* width + j, queue);

popIndex = PopQueue(queue);

while (popIndex > -1){

SearchNeighbor(tempin, width, height, tempout, labelIndex, popIndex, queue);

popIndex = PopQueue(queue);

}

}

}

}

free(queue);

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (tempout[i \* width + j] == 1){

tempout[i \* width + j] = 255;

}

else{

tempout[i \* width + j] = 0;

}

}

}

return(outimage);

}

1. **Seprdate overlapping Particles (bubbles\_on\_black\_background):**

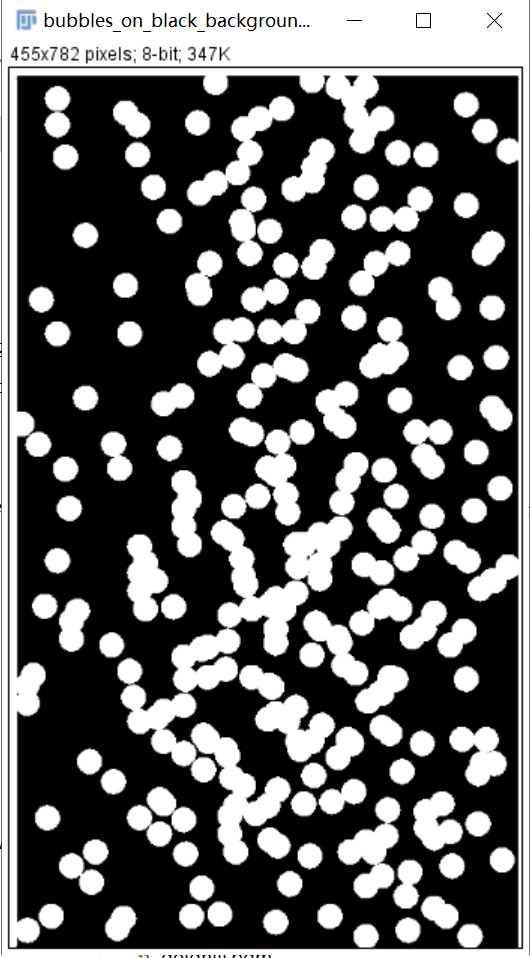
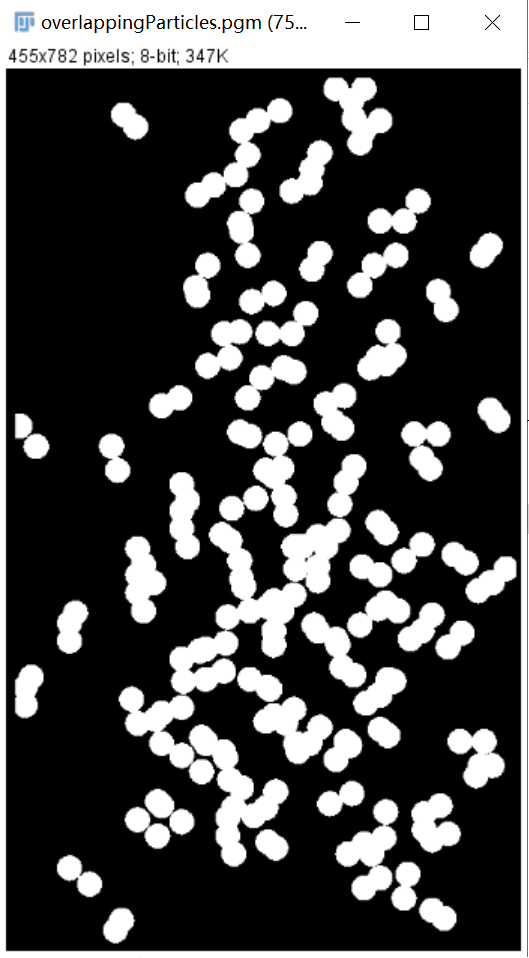
**Algorithm:**

I observed that the single particle occupies less than 22×22.

In this situation, we first label the image. If the number of pixel values in the white connected area is bigger than 22×22, then this white connected area contains overlapping particles.

**Results (including pictures):**

Source: result:

**Discussion:**

My method is a little bit lazy, but actually it works in this situation and I can’t find a better way.

**Codes:**

Image \*seprdateOverlappingParticles(Image\* image){

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "seprdateOverpallingParticles";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

memset(tempout, 0, width \* height);

int i, j, popIndex, labelIndex = 0;

for (i = 0; i < 7; i++){

for (j = 0; j < width; j++){

tempin[i \* width + j] = 0;

}

}

for (j = 0; j < width; j++){

tempin[(height - 1) \* width + j] = 0;

}

for (i = 0; i < height; i++){

for (j = 0; j < 7; j++){

tempin[i \* width + j] = 0;

}

}

for (i = 0; i < height; i++){

for (j = 452; j < width; j++){

tempin[i \* width + j] = 0;

}

}

Queue \*queue = NULL;

queue = (Queue\*)malloc(sizeof(Queue));

queue->first = NULL;

queue->last = NULL;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

if (tempin[i \* width + j] == 255 && tempout[i \* width + j] == 0){

labelIndex++;

SearchNeighbor(tempin, width, height, tempout, labelIndex, i \* width + j, queue);

popIndex = PopQueue(queue);

while (popIndex > -1){

SearchNeighbor(tempin, width, height, tempout, labelIndex, popIndex, queue);

popIndex = PopQueue(queue);

}

}

}

}

free(queue);

int k, count;

for (k = 1; k <= labelIndex; k++){

count = 0;

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

count++;

}

}

}

if (count < 22 \* 22){

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

outimage->data[i \* width + j] = 0;

}

}

}

}

else{

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

outimage->data[i \* width + j] = 255;

}

}

}

}

}

return(outimage);

}

1. **Seprdate nonlapping Particles (bubbles\_on\_black\_background):**

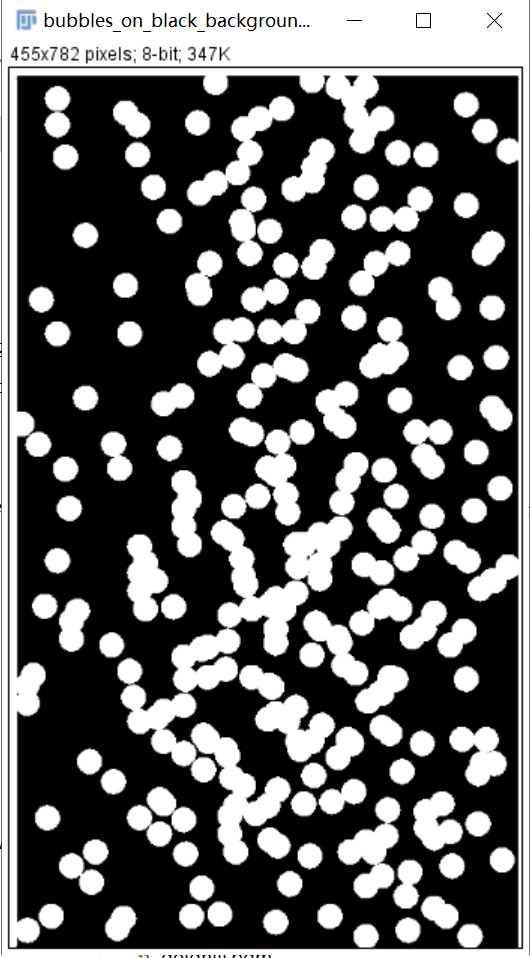
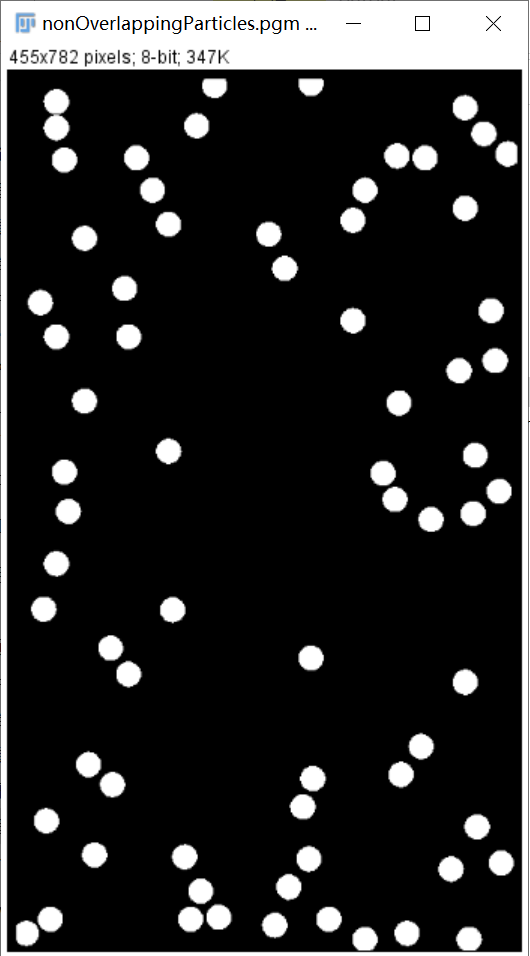
**Algorithm:**

I observed that the single particle occupies less than 22×22.

In this situation, we first label the image. If the number of pixel values in the white connected area is bigger than 22×22, then this white connected area contains overlapping particles.

**Results (including pictures):**

Source: result:

**Discussion:**

My method is a little bit lazy, but actually it works in this situation and I can’t find a better way.

**Codes:**

Image \*seprdateNonOverlappingParticles(Image\* image){

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "seprdateNonOverlappingParticles";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

memset(tempout, 0, width \* height);

int i, j, popIndex, labelIndex = 0;

for (i = 0; i < 7; i++){

for (j = 0; j < width; j++){

tempin[i \* width + j] = 0;

}

}

for (j = 0; j < width; j++){

tempin[(height - 1) \* width + j] = 0;

}

for (i = 0; i < height; i++){

for (j = 0; j < 7; j++){

tempin[i \* width + j] = 0;

}

}

for (i = 0; i < height; i++){

for (j = 452; j < width; j++){

tempin[i \* width + j] = 0;

}

}

Queue \*queue = NULL;

queue = (Queue\*)malloc(sizeof(Queue));

queue->first = NULL;

queue->last = NULL;

for (i = 1; i < height - 1; i++){

for (j = 1; j < width - 1; j++){

if (tempin[i \* width + j] == 255 && tempout[i \* width + j] == 0){

labelIndex++;

SearchNeighbor(tempin, width, height, tempout, labelIndex, i \* width + j, queue);

popIndex = PopQueue(queue);

while (popIndex > -1){

SearchNeighbor(tempin, width, height, tempout, labelIndex, popIndex, queue);

popIndex = PopQueue(queue);

}

}

}

}

free(queue);

int k, count;

for (k = 1; k <= labelIndex; k++){

count = 0;

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

count++;

}

}

}

if (count < 22 \* 22){

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

outimage->data[i \* width + j] = 255;

}

}

}

}

else{

for (i = 0; i < height; i++){

for (j = 0; j < width; j++){

if (outimage->data[i \* width + j] == k){

outimage->data[i \* width + j] = 0;

}

}

}

}

}

return(outimage);

}