8. Erosion, Dilation, Opening, Closing, and Connection

1. Erosion, Dilation, Opening, and Closing (noisy_fingerprint, noise_rectangle)

- **1.1 Erosion:**

Algorithm:

As the sets A and B in Z^2, the erosion of B to A denoted as $A \ominus B$ is defined as:

$$A \ominus B = \{z \mid (B)_z \subseteq A\}$$

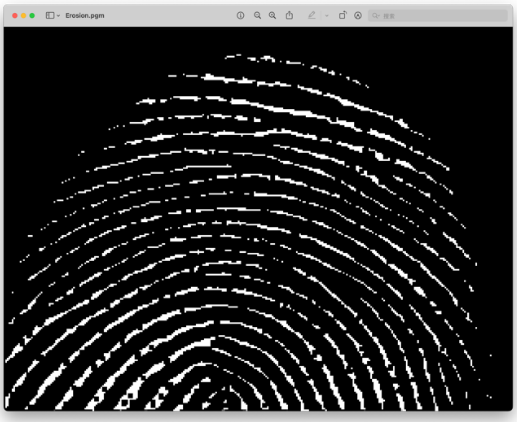
To put it another way, in my implementation, the algorithm will convolve image A with a kernel B of a square shape. Kernel B has a definable anchor point, the center point. And then swiping the kernel B across the image, extract the minimum pixel value of the area covered by the kernel B, and replace the pixel at the anchor point.

Results (including pictures):

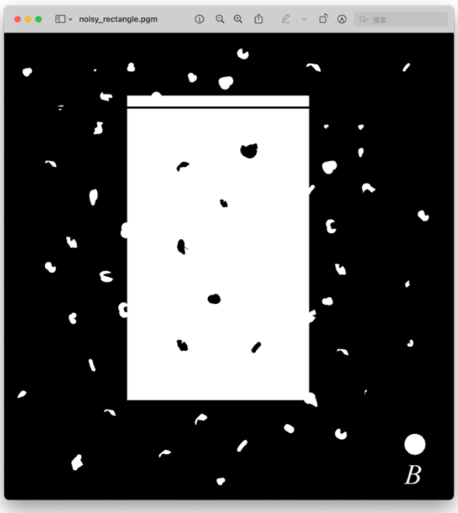
Process result of “noisy_fingerprint.pgm” and “noisy_rectangle.pgm”:

Source Image:

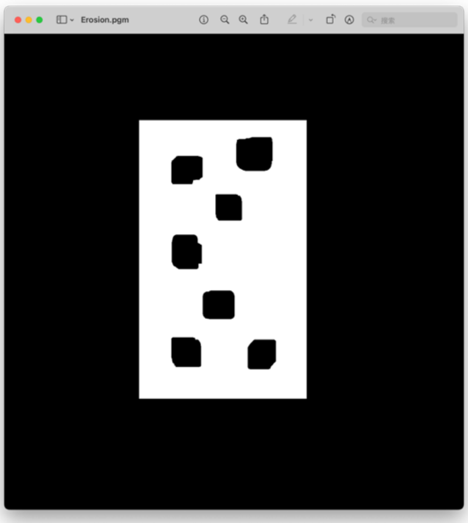
Result of erosion (element size: 3*3):



Source Image:



Result of erosion (element size: 50*50):



Discussion:

The erosion operation will reduce the highlight areas (**white** in our case) in the image. It essentially removes pixels along object boundaries and reduces the size of the object. It has the effect to reduce sporadic highlight noises and its level increases as the size of structural element increases, and the time consumption will increase dramatically at the same time.

Codes:

```
// Algorithms Code:
Image *Erosion(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int min = 255;
            // the size of structual element can be changed values of x and y:
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    int temp = tempin[(image->Width)*(i+x) + (j+y)];
                    if(temp < min) min = temp;
                }
            }
            tempout[image->Width * i + j] = min;
        }
    }
    return(outimage);
}
```

- **1.2 Dilation:**

Algorithm:

As the sets A and B in Z^2, the dilation of B to A denoted as $A \oplus B$ is defined as:

$$A \oplus B = \left\{ z \mid (\hat{B})_z \cap A \neq \varnothing \right\}$$

Similar to the erosion algorithm, it will also convolve image A with a kernel B of a square shape. And then swiping the kernel B across the image, extract the maximum pixel value of the area covered by the kernel B, and replace the pixel at the central point.

Results (including pictures):

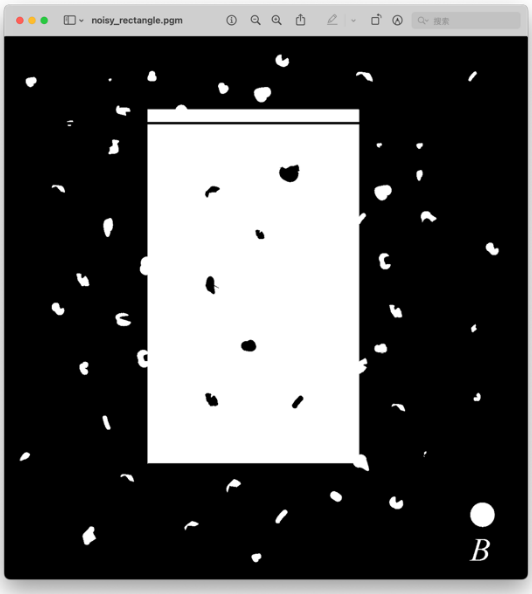
Process result of “noisy_fingerprint.pgm”:

Source Image:

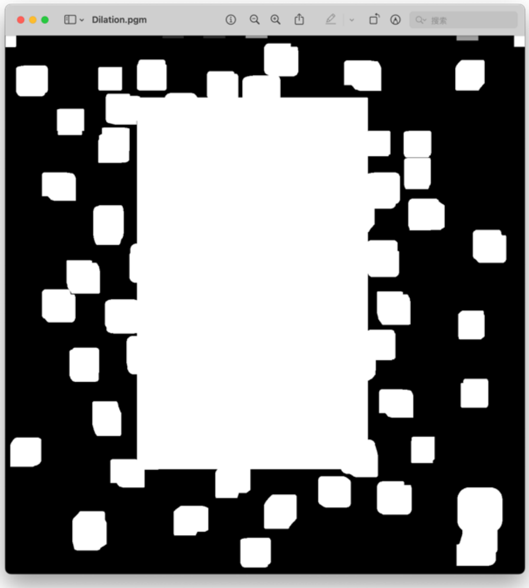
Result of dilation (element size: 3*3):

Process result of “noisy_rectangle.pgm”:

Source Image:



Result of dilation (element size: 50*50):



Discussion:

Dilation works opposite of erosion, it will enlarge the highlight areas (**white** in our case) in the image. It has the effect of magnifying details, and the degree of magnification also depends on the size of the structural elements. But dilation operation will amplify unwanted sporadic noises in the image.

Codes:

```
Image *Dilation(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int max = 0;
            // the size of structual element can be changed values of x and y:
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    int temp = tempin[(image->Width)*(i+x) + (j+y)];
                    if(temp > max) max = temp;
                }
            }
            tempout[image->Width * i + j] = max;
        }
    }
    return(outimage);
}
```

- **1.3 Opening:**

Algorithm:

The opening operation of the structure element B on the set A, denoted as $A \circ B$, is defined as:

$$A \circ B = (A \ominus B) \oplus B$$

So it will be achieved by first eroding the image and then dilating it:

$$\mathrm{dst} = \mathrm{open}(\mathrm{src}, \mathrm{element}) = \mathrm{dilate}(\mathrm{erode}(\mathrm{src}, \mathrm{element}))$$

Results (including pictures):

Process result of “noisy_fingerprint.pgm”:

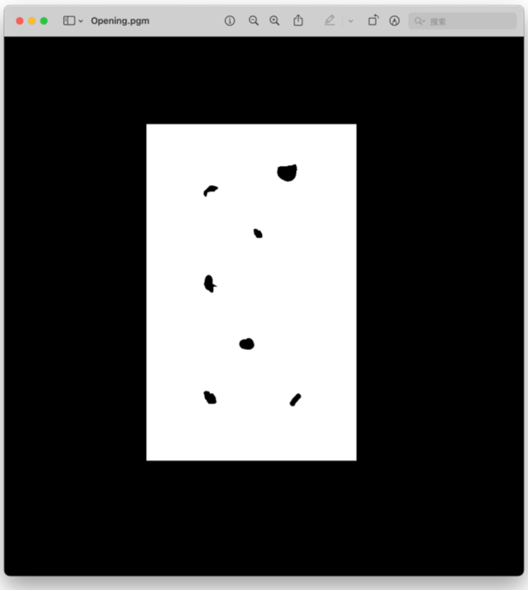
Source Image:

Result of opening (element size: 3*3):

Process result of “noisy_rectangle.pgm”:

Source Image:

Result of opening (element size: 50*50):



Discussion:

Opening operation can be used to eliminate small objects, separate objects at thin points, and smooth the boundaries of larger objects without significantly changing their area. It will suppress bright details smaller than the structuring elements. It can avoid the loss of the size of the objects in the image when only using the erosion operation.

Codes:

```
void Opening(Image *image) {
    unsigned char *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempout = outimage->data;

    outimage = Dilation(Erosion(image));
    SavePNMImage(outimage, (char*)"Opening.pgm");
}
```

- **1.4 Closing:**

Algorithm:

The closing operation of the structure element B on the set A, denoted as $A \cdot B$, is defined as:

$$A \bullet B = (A \oplus B) \ominus B$$

So it will be achieved by first dilating the image and then eroding it:

$$\mathrm{dst} = \mathrm{close}(\mathrm{src}, \mathrm{element}) = \mathrm{erode}(\mathrm{dilate}(\mathrm{src}, \mathrm{element}))$$

Results (including pictures):

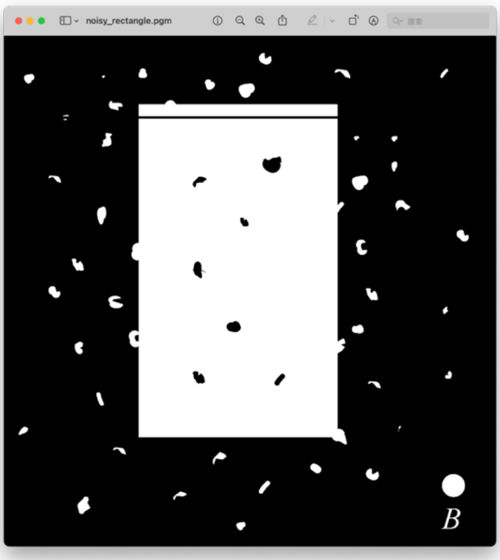
Process result of “noisy_fingerprint.pgm”:

Source Image:

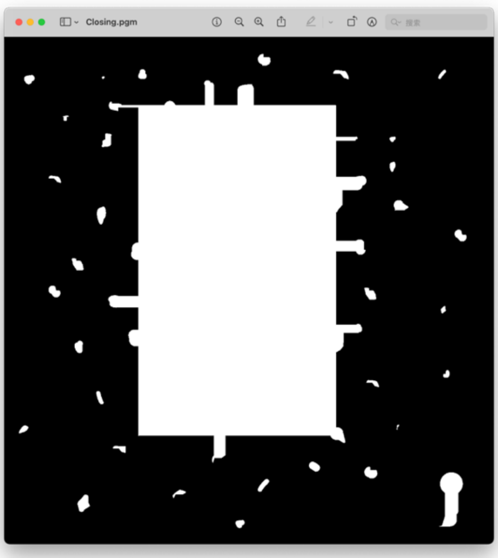
Result of closing (element size: 3*3):

Process result of “noisy_rectangle.pgm”:

Source Image:



Result of closing (element size: 50*50):



Discussion:

The closing operation can be used to fill small (dark) holes in objects, connect adjacent objects, smooth their boundaries without significantly changing their area, and suppress dark details smaller than structuring elements.

Codes:

```
void Closing(Image *image) {
    unsigned char *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempout = outimage->data;

    outimage = Erosion(Dilation(image));
    SavePNMImage(outimage, (char*)"Closing.pgm");
}
```

2. Extract boundaries (licoln, U):

Algorithm:

The boundary extraction of the image is denoted as the set of $\beta(A)$ (the boundary of A) can be obtained by first eroding A with B, and then performing the set difference between A and the result of the erosion, that is:

$$\beta(A) = A - (A \ominus B)$$

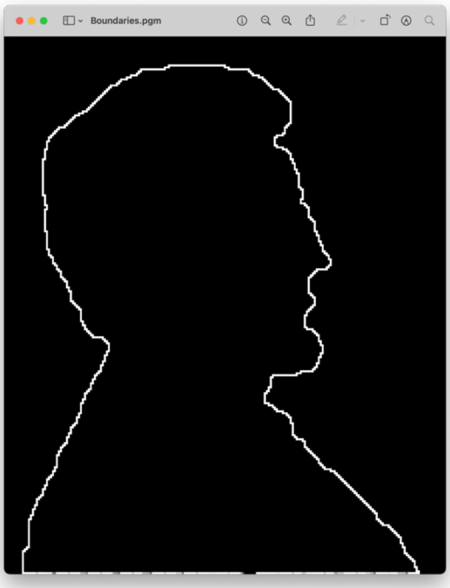
Where B is a proper structural element.

Results (including pictures):

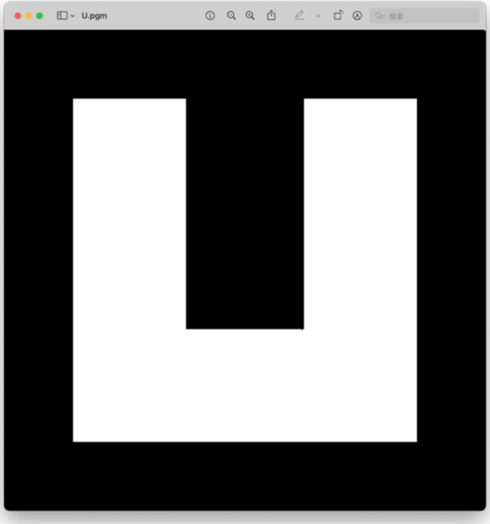
Process result of "licoln.pgm" and "U.pgm":

Source Image:

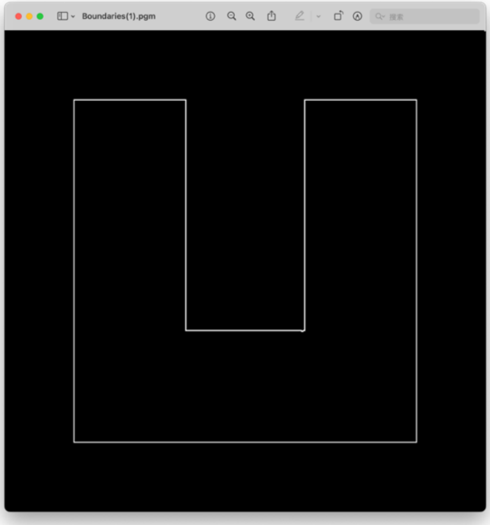
Extracting boundary:



Source Image:



Extracting boundary:



Discussion:

The boundaries results are obtained from subtracting the image eroded by the structuring element from the original image. And the width of the boundaries increases when the size of structural element increases.

Codes:

```
void ExtractBoundaries(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    int size = image->Width * image->Height;

    outimage = Erosion(image);
    tempin = image->data;
    tempout = outimage->data;
    for(int i = 0; i < size; i++) {
        tempout[i] = tempin[i] - tempout[i];
    }
    SavePNMImage(outimage, (char*)"Boundaries.pgm");
}
```

3. Count the number of connected component, and output (connected):

Algorithm:

The algorithm on the textbook is:

$$X_k = (X_{k-1} \oplus B) \cap A \quad k = 1, 2, 3, \cdots$$

Where B is a proper structural element, and the iterate operation will stop when $X_k = X_{k-1}$. Nevertheless, after understanding the principle of the algorithm, it's found that the method of **Deep First Search** (Backtracking) and its recursive algorithm is very suitable and efficient for solving this question. Its simplify pseudocode is shown below:

If (this pixel is black || its position is beyond boundary || this pixel has been traversed)

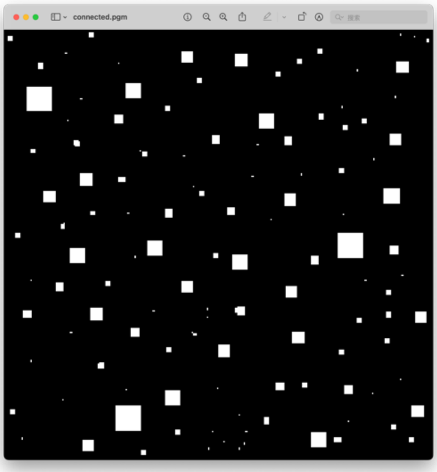
Then return 0

ELSE return 1 + sum_of_its_surrounding_white_components

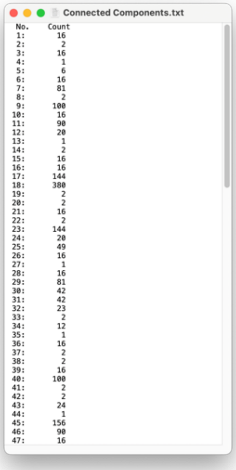
The algorithm is that, if a pixel is detected to be **white**, use recursion to calculate the sum of all **white** components connected to this pixel (using 8-adjacent), and save sum value of this area. Also need to establish a truth table to record whether each white component has been traversed and skip the area that has been traversed or is **0 (black)** or beyond the image boundaries.

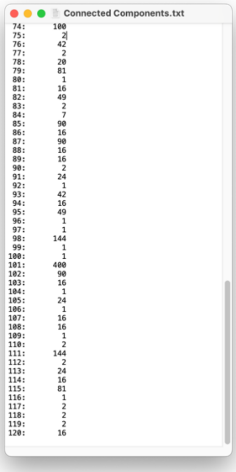
Results (Please refer to the attaching txt file “Connected Components.txt”):

Source image:



Output txt file (head and tail):





Discussion:

The algorithm of Deep First Search (Backtracking) turns out to be very efficient, since it only traverses all the pixels in the image once using the recursion. So it has a time complexity of $O(MN)$, where M, N are the side lengths. And the output contains pixel counts of each white component in the image (refer to the file “Connected Components.txt”).

Codes:

(1) The main function of counting:

```
void connectedComponent(Image *image) {
    int size = image->Width * image->Height;
    int checkBoard[size];// mark the traversed pixels
    FILE *fp;
    fp = fopen("Connected Components.txt", "w");
    int index = 1;

    for(int i = 0; i < size; i++) {
        checkBoard[i] = 0;// initialize the checkboard
    }

    fprintf(fp, "  No.    Count\n");
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int count = DFS(image, checkBoard, j, i);// use Deep First Search here
            if(count != 0) fprintf(fp, "%3d: %8d\n", index++, count);
        }
    }
    fclose(fp);
}
```

(2) Deep First Search(Recursion) to traverse all the nearby white components:

```c
int DFS(Image * image, int *checkBoard, int x, int y) {
    unsigned char *tempin;
    tempin = image->data;
    int currPosition = image->Width * y + x;
    // Base Case:
    if(x < 0 || y < 0 || x >= image->Width || y >= image->Height ||
       checkBoard[currPosition] == 1 || tempin[currPosition] == 0) {
        return 0;
    }
    // Recursive Steps:
    checkBoard[currPosition] = 1;
    int tempSum = 0;
    // use 8-adjacent checking:
    for(int m = -1; m <= 1; m++) {
        for(int n = -1; n <= 1; n++) {
            tempSum += DFS(image, checkBoard, x+m, y+n);
        }
    }
    return (1 + tempSum);
}
```

4. Separate the 3 sets of white bubbles (bubbles_on_black_background):

Algorithm:

Implementation steps:

(1) Use the same method as in question 3 to find all white bubbles that **connected with the image boundaries**. It just needs to start with a white pixel located on the boundary. And use a label table to mark all traversed pixels as **1**, which will be the first result image.

(2) Use **DFS** to calculate the pixels number of connected components in the image. Since it's found that the area of each bubble is around 350~450pixels. So all the connected components within this size are the **single bubbles**, and label them as **2**. Moreover, the components with size larger than 450 will be the **overlapping bubbles**, mark them as **3**.

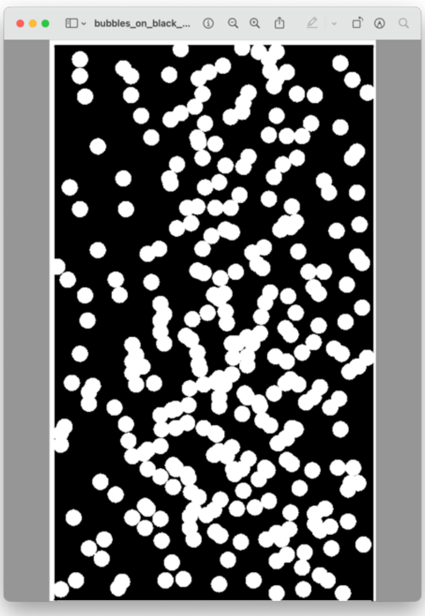
(3) Output the images containing the pixels with different kinds of labels.

Note that the algorithm is using **8-adjacent** to determine connected components.

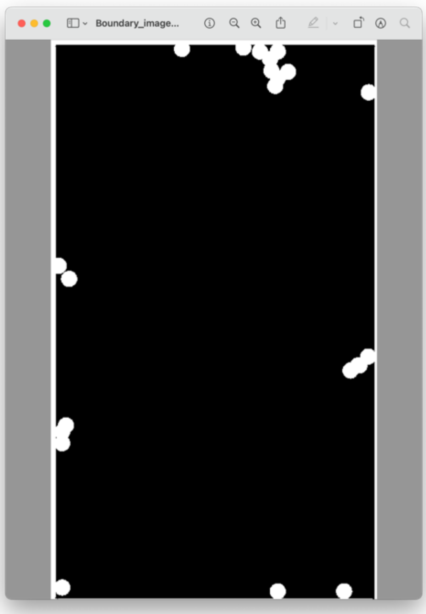
Results (including pictures):

Process result of "bubbles_on_black_background.pgm":

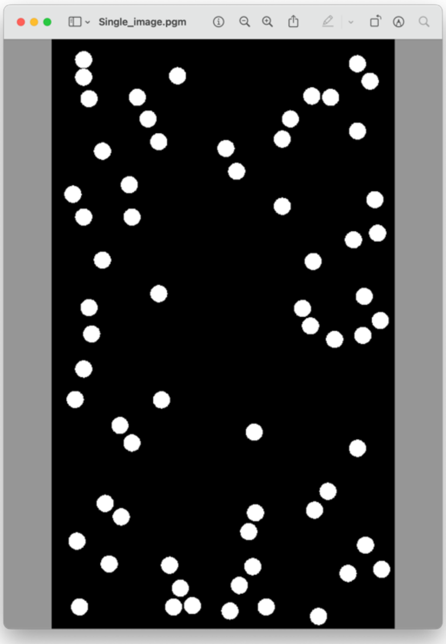
Source image:



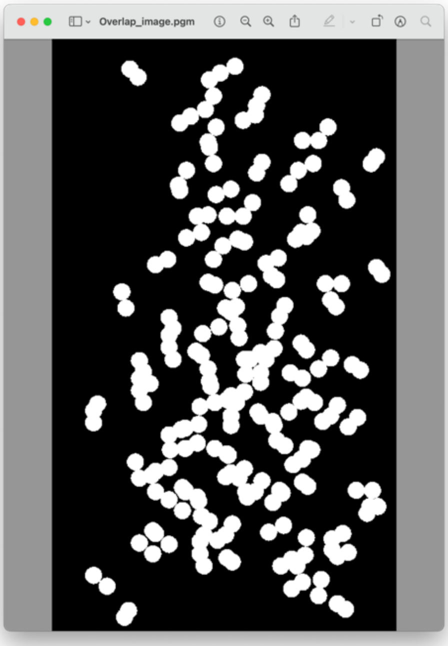
The particles merging with boundaries:



The extracted single particles:



The extracted overlapping particles:



Discussion:

This program achieved a one-time separation and output of three bubble sets by labeling different kinds of connected components using Deep First Search algorithm. The time complexity of this algorithm is also $O(MN)$.

Codes:

(1) The main function of separating the bubbles:

```
void SeparateBubbles(Image *image) {
    unsigned char *tempout_boundary, *tempout_single, *tempout_overlap;
    Image *boundary_image, *single_image, *overlap_image;
    boundary_image = CreateNewImage(image, (char*)"#testing function");
    single_image = CreateNewImage(image, (char*)"#testing function");
    overlap_image = CreateNewImage(image, (char*)"#testing function");
    tempout_boundary = boundary_image->data;
    tempout_single = single_image->data;
    tempout_overlap = overlap_image->data;

    int size = image->Width * image->Height;
    int checkBoard[size], labelBoard[size];
    // initialize the checkboard and labelboard,
    // and set the background of the output images to black:
    for(int i = 0; i < size; i++) {
        tempout_boundary[i] = 0;
        tempout_single[i] = 0;
        tempout_overlap[i] = 0;
        labelBoard[i] = 0;
        checkBoard[i] = 0;
    }

    // 1: Extract all the particles that merged with boundaries:
    DFS_MarkPixels(image, labelBoard, 1, 1, 1);
    DFS_MarkPixels(image, labelBoard, 200, 781, 1);

    // 2 & 3: Extract all the single, and overlapping particles:
    for(int i = 7; i < image->Height-7-1; i++) {
        for(int j = 7; j < image->Width-7-3; j++) {
            int area = DFS(image, checkBoard, j, i);// record the size of a connected area
            if(area > 350 && area < 450) DFS_MarkPixels(image, labelBoard, j, i, 2);
            else DFS_MarkPixels(image, labelBoard, j, i, 3);
        }
    }
    // output the images:
```

```
    for(int i = 0; i < size; i++) {
        if(labelBoard[i] == 1) tempout_boundary[i] = 255;
        if(labelBoard[i] == 2) tempout_single[i] = 255;
        if(labelBoard[i] == 3) tempout_overlap[i] = 255;
    }
    SavePNMImage(boundary_image, (char*)"Boundary_image.pgm");
    SavePNMImage(single_image, (char*)"Single_image.pgm");
    SavePNMImage(overlap_image, (char*)"Overlap_image.pgm");
}
```

(2) The function to label all the pixels in the image:

```
void DFS_MarkPixels(Image *image, int *labelBoard, int x, int y, int label) {
    unsigned char *tempin;
    tempin = image->data;
    int currPosition = image->Width * y + x;
    // Base Case:
    if(x < 0 || y < 0 || x >= image->Width || y >= image->Height ||
       labelBoard[currPosition] != 0 || tempin[currPosition] == 0) {
        return;
    }
    // Recursive Steps:
    labelBoard[currPosition] = label;
     // use 8-adjacent marking:
    for(int m = -1; m <= 1; m++) {
        for(int n = -1; n <= 1; n++) {
            DFS_MarkPixels(image, labelBoard, x+m, y+n, label);;
        }
    }
}
// Algorithms End.
```

(3) DFS function (same as the question3)

```
int DFS(Image * image, int *checkBoard, int x, int y) {
    unsigned char *tempin;
    tempin = image->data;
    int currPosition = image->Width * y + x;
    // Base Case:
    if(x < 0 || y < 0 || x >= image->Width || y >= image->Height ||
       checkBoard[currPosition] == 1 || tempin[currPosition] == 0) {
        return 0;
    }
    // Recursive Steps:
    checkBoard[currPosition] = 1;
    int tempSum = 0;
    // use 8-adjacent checking:
    for(int m = -1; m <= 1; m++) {
        for(int n = -1; n <= 1; n++) {
            tempSum += DFS(image, checkBoard, x+m, y+n);
        }
    }
    return (1 + tempSum);
}
```