Biometrics
Laboratory Report

Jakub Ciecierski

December 11, 2015

Date Performed: October 30, 2015

Contents

1 KMM

1.1 Algorithm

1.1.1 KMM - Main

KMM works as follows

a. Convert input image into normalized and binary image

b. Initialize a Bit Map based on the image. Corresponding elements of black pixels have value 1, white pixels have 0.

c. Choose Sticking Pixel To background.

   (a) Pixels that stick with background directly (South, North, East, West) are given value 2.

   (b) Pixels that stick with background by corners (South-East, South-West, North-East, North-West) are given value 3.

d. Select Pixel for deletion by giving them values 4. Consider a 3×3 Matrix of weights (see Implementation for details). For each Pixel do:

   (a) Let the current Pixel be the anchor point of Weight Matrix which is placed in the middle of that Matrix. For all neighbours in that area, compute the sum of corresponding weights.

   (b) Refer to Deletion Array to see if Pixel with given sum of weights should be deleted or not. If it is in Deletion Array, set value of that Pixel to 4

e. Remove certain Pixels without Loss of Connectivity

   (a) Remove Pixels with value 2 without LoC.

   (b) Remove Pixels with value 3 without LoC.

f. Repeat $a.$ through $e.$ until no further changes have been made

1.1.2 KMM - Loss of Connectivity

The algorithm for checking if pixels can be removed without Loss of Connectivity is looks as follows:

a. For input Pixel create its local 8-point neighbourhood

b. Initialize each element in the neighbourhood:

   (a) If value of corresponding element in Bit Map is 0 then set value to 0

   (b) Otherwise set it to 1.

c. The element of neighbourhood corresponding to input Pixel is given value 0. As to imitate the situation where this pixel was removed.

d. If the neighbourhood is connected then input Pixel can be removed

1.1.3 KMM - Neighbourhood Connectivity

The algorithm for checking if the given neighbourhood is connected

a. Choose one of the elements with value 1 as source. If no such element exists, stop the algorithm

b. Consider a Reachability Flags. 0 if corresponding node is unreachable, 1 if it is reachable.

    (a) Initialize flags for all nodes to 0. Initialize flag for source node to 1.

c. Consider Current Nodes queue - holds nodes to be checked.

    (a) Insert source node to queue.

d. Dequeue the Current Node.

    (a) For all other nodes that are connected to Current Node

        i. If this node has Reachability Flag 0. Change flag value to 1 and add to Current Nodes queue.

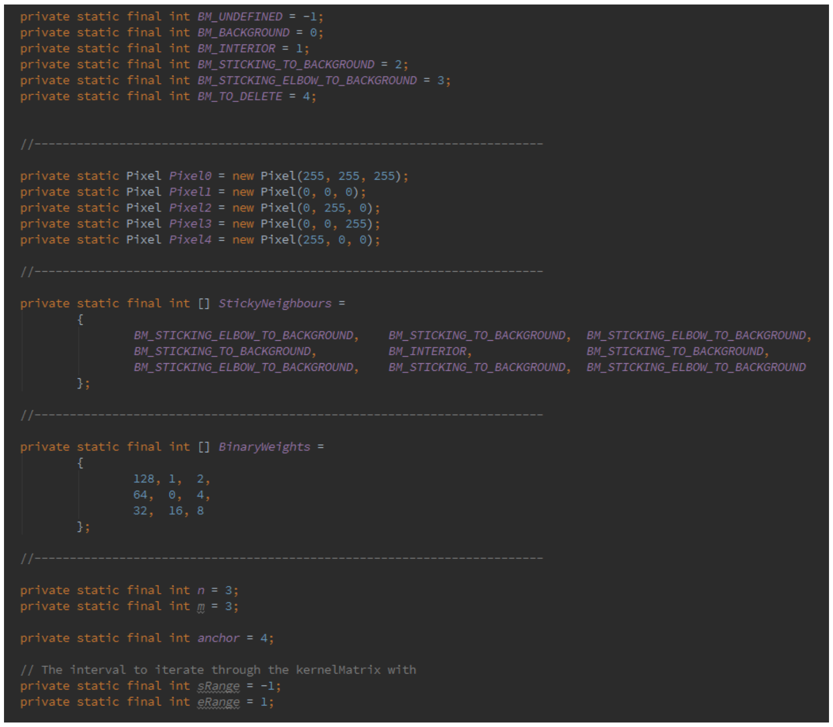
    (b) Repeat util the queue is empty

e. If all Nodes are reachable from source. Then neighbourhood is connected.

1.2 Implementation

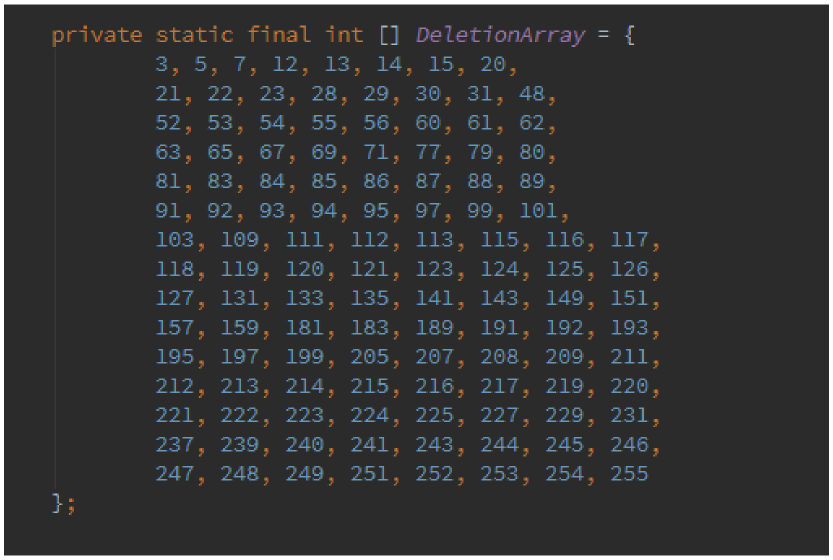
The implementation details of KMM algorithm are presented. Figure 1 shows the header of the KMM class. The Sticky Neighbours array defines which value should the pixels sticking to border pixels take. The Binary Weights array is used as template to compute the weight in the thinning phase.
The figure 2 shows the Deletion array used determine whether a pixel with given sum of weights should be deleted or not.
The main iterative procedure is presented in figure 3. The *bitMap* is used to represent the image.
The step 0 - the filter step - turns the image into a binary one. The step 1. is simply an initialization of the bit map, where black pixels get value 1, and white pixels get value 0. The *updateImageWithBitMap* simply updates the image pixels based on the bit map values. These methods are presented in figure 4.
Choosing Pixels sticking to background phase is presented in figure 5. The *StickyNeighbours* Look Up table is used to determine which value should the corresponding pixel get, sticking directly or sticking by the corner.
Further, step 3, chooses and deletes pixels based on the binary weights. *BinaryWeights* array is used to calculate the sum of corresponding neighbours. Then the *DeletionArray* checks whether a tested Pixel should be deleted. This is presented in figure 6.
The step 4. removes the pixels without loss of connectivity. The main method of that procedure is shown in figure 7. The idea is to create a neighbourhood for

each pixel that we are considering to delete. The neighbourhood is initialized based on the corresponding pixels in the image. The target Pixel is removed from the neighbourhood by setting the corresponding element to Background value. Source pixel representing a interior point is then chosen. The $isConnectedToAll$ method will check whether the source is connected to all other interior points in the neighbourhood. This method is presented in figure 8. The idea is to compute whether all interior points in the neighbourhood are connected to each other after temporally removing the tested Pixel. If this is true, then this Pixel can be safely removed.
Finally a few utility methods used through out the algorithm are presented in figure 9.



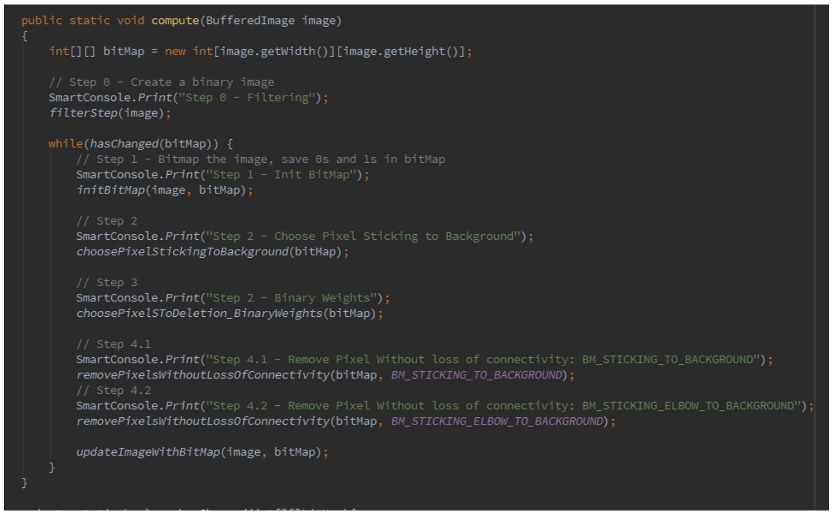
```java
private static final int BM_UNDEFINED = -1;
private static final int BM_BACKGROUND = 0;
private static final int BM_INTERIOR = 1;
private static final int BM_STICKING_TO_BACKGROUND = 2;
private static final int BM_STICKING_ELBOW_TO_BACKGROUND = 3;
private static final int BM_TO_DELETE = 4;


//----------------------------------------------------------------------

private static Pixel Pixel0 = new Pixel(255, 255, 255);
private static Pixel Pixel1 = new Pixel(0, 0, 0);
private static Pixel Pixel2 = new Pixel(0, 255, 0);
private static Pixel Pixel3 = new Pixel(0, 0, 255);
private static Pixel Pixel4 = new Pixel(255, 0, 0);

//----------------------------------------------------------------------

private static final int [] StickyNeighbours =
        {
                BM_STICKING_ELBOW_TO_BACKGROUND,    BM_STICKING_TO_BACKGROUND,  BM_STICKING_ELBOW_TO_BACKGROUND,
                BM_STICKING_TO_BACKGROUND,          BM_INTERIOR,                BM_STICKING_TO_BACKGROUND,
                BM_STICKING_ELBOW_TO_BACKGROUND,    BM_STICKING_TO_BACKGROUND,  BM_STICKING_ELBOW_TO_BACKGROUND
        };

//----------------------------------------------------------------------

private static final int [] BinaryWeights =
        {
                128, 1,  2,
                64,  0,  4,
                32,  16, 8
        };

//----------------------------------------------------------------------

private static final int n = 3;
private static final int m = 3;

private static final int anchor = 4;

// The interval to iterate through the kernelMatrix with
private static final int sRange = -1;
private static final int eRange = 1;
```

Figure 1: Header of KMM Class



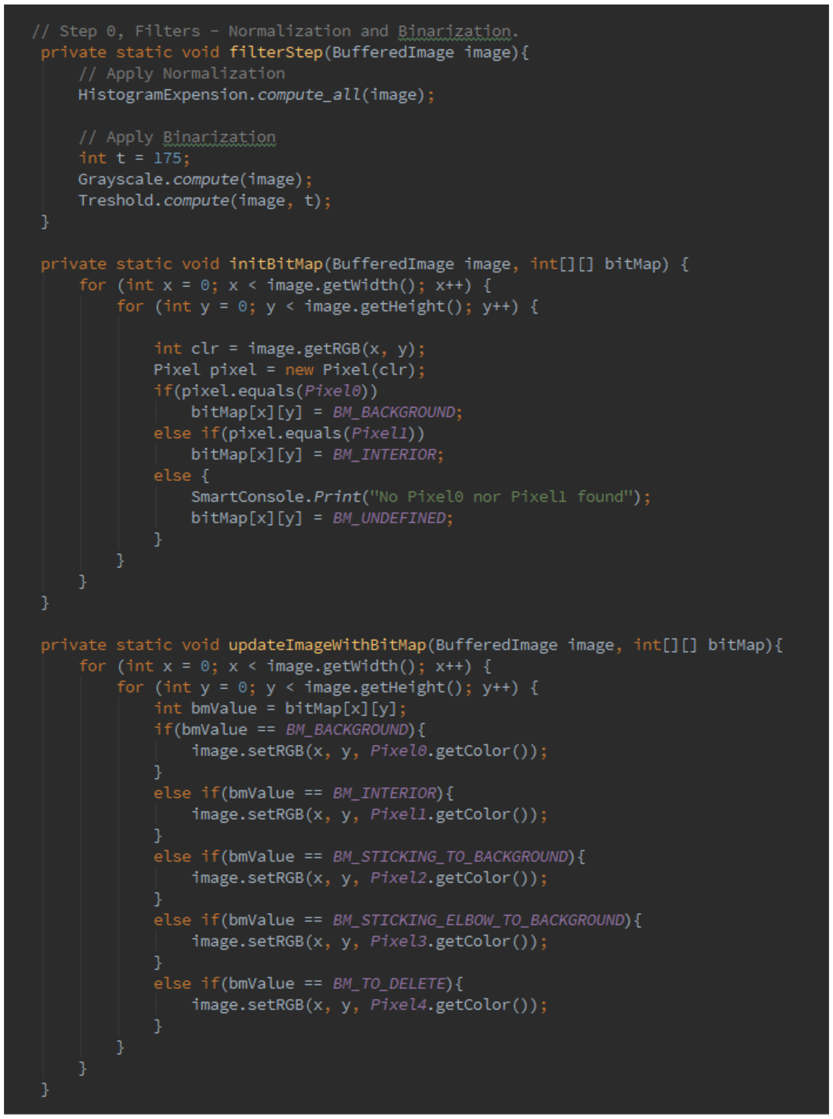
```java
private static final int [] DeletionArray = {
        3, 5, 7, 12, 13, 14, 15, 20,
        21, 22, 23, 28, 29, 30, 31, 48,
        52, 53, 54, 55, 56, 60, 61, 62,
        63, 65, 67, 69, 71, 77, 79, 80,
        81, 83, 84, 85, 86, 87, 88, 89,
        91, 92, 93, 94, 95, 97, 99, 101,
        103, 109, 111, 112, 113, 115, 116, 117,
        118, 119, 120, 121, 123, 124, 125, 126,
        127, 131, 133, 135, 141, 143, 149, 151,
        157, 159, 181, 183, 189, 191, 192, 193,
        195, 197, 199, 205, 207, 208, 209, 211,
        212, 213, 214, 215, 216, 217, 219, 220,
        221, 222, 223, 224, 225, 227, 229, 231,
        237, 239, 240, 241, 243, 244, 245, 246,
        247, 248, 249, 251, 252, 253, 254, 255
};
```

Figure 2: Deletion Array



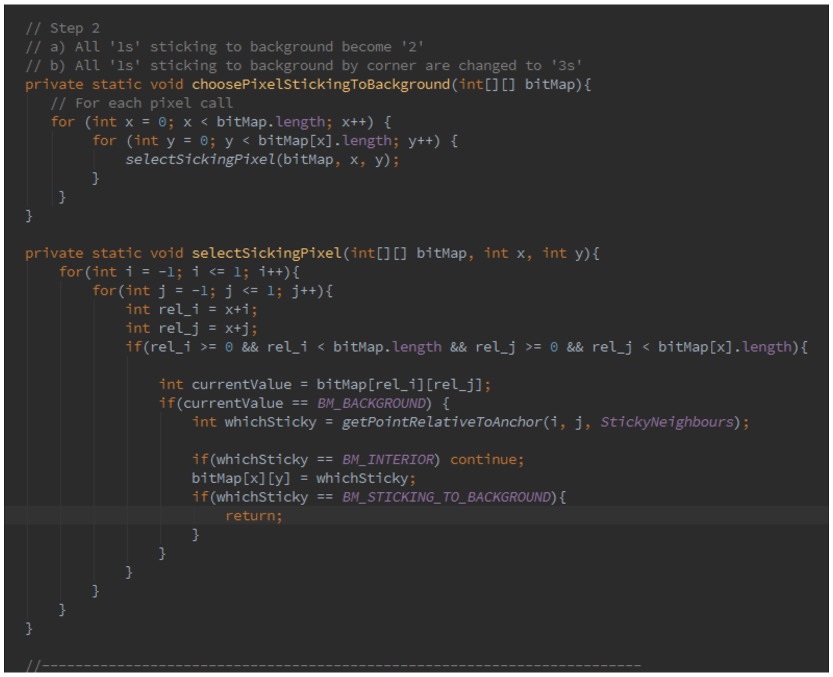
```java
public static void compute(BufferedImage image)
{
    int[][] bitMap = new int[image.getWidth()][image.getHeight()];

    // Step 0 - Create a binary image
    SmartConsole.Print("Step 0 - Filtering");
    filterStep(image);

    while(hasChanged(bitMap)) {
        // Step 1 - Bitmap the image, save 0s and 1s in bitMap
        SmartConsole.Print("Step 1 - Init BitMap");
        initBitMap(image, bitMap);

        // Step 2
        SmartConsole.Print("Step 2 - Choose Pixel Sticking to Background");
        choosePixelStickingToBackground(bitMap);

        // Step 3
        SmartConsole.Print("Step 2 - Binary Weights");
        choosePixelSToDeletion_BinaryWeights(bitMap);

        // Step 4.1
        SmartConsole.Print("Step 4.1 - Remove Pixel Without loss of connectivity: BM_STICKING_TO_BACKGROUND");
        removePixelsWithoutLossOfConnectivity(bitMap, BM_STICKING_TO_BACKGROUND);
        // Step 4.2
        SmartConsole.Print("Step 4.2 - Remove Pixel Without loss of connectivity: BM_STICKING_ELBOW_TO_BACKGROUND");
        removePixelsWithoutLossOfConnectivity(bitMap, BM_STICKING_ELBOW_TO_BACKGROUND);

        updateImageWithBitMap(image, bitMap);
    }
}
```

Figure 3: Main Compute method



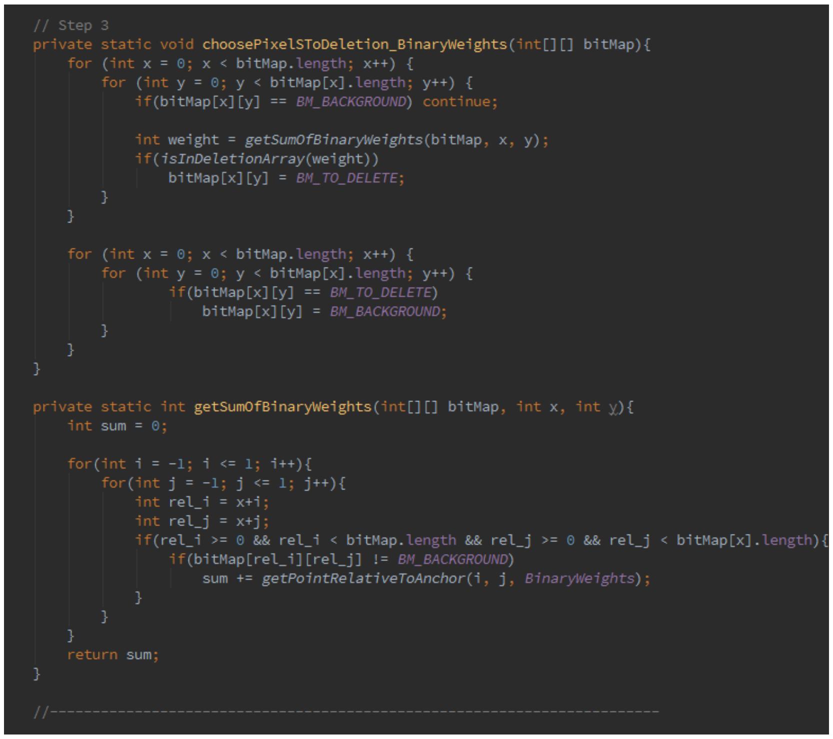
```java
// Step 0, Filters - Normalization and Binarization.
 private static void filterStep(BufferedImage image){
     // Apply Normalization
     HistogramExpension.compute_all(image);

     // Apply Binarization
     int t = 175;
     Grayscale.compute(image);
     Treshold.compute(image, t);
 }

 private static void initBitMap(BufferedImage image, int[][] bitMap) {
     for (int x = 0; x < image.getWidth(); x++) {
         for (int y = 0; y < image.getHeight(); y++) {

             int clr = image.getRGB(x, y);
             Pixel pixel = new Pixel(clr);
             if(pixel.equals(Pixel0))
                 bitMap[x][y] = BM_BACKGROUND;
             else if(pixel.equals(Pixel1))
                 bitMap[x][y] = BM_INTERIOR;
             else {
                 SmartConsole.Print("No Pixel0 nor Pixel1 found");
                 bitMap[x][y] = BM_UNDEFINED;
             }
         }
     }
 }

 private static void updateImageWithBitMap(BufferedImage image, int[][] bitMap){
     for (int x = 0; x < image.getWidth(); x++) {
         for (int y = 0; y < image.getHeight(); y++) {
             int bmValue = bitMap[x][y];
             if(bmValue == BM_BACKGROUND){
                 image.setRGB(x, y, Pixel0.getColor());
             }
             else if(bmValue == BM_INTERIOR){
                 image.setRGB(x, y, Pixel1.getColor());
             }
             else if(bmValue == BM_STICKING_TO_BACKGROUND){
                 image.setRGB(x, y, Pixel2.getColor());
             }
             else if(bmValue == BM_STICKING_ELBOW_TO_BACKGROUND){
                 image.setRGB(x, y, Pixel3.getColor());
             }
             else if(bmValue == BM_TO_DELETE){
                 image.setRGB(x, y, Pixel4.getColor());
             }
         }
     }
 }
```

Figure 4: Filter and BitMap stage



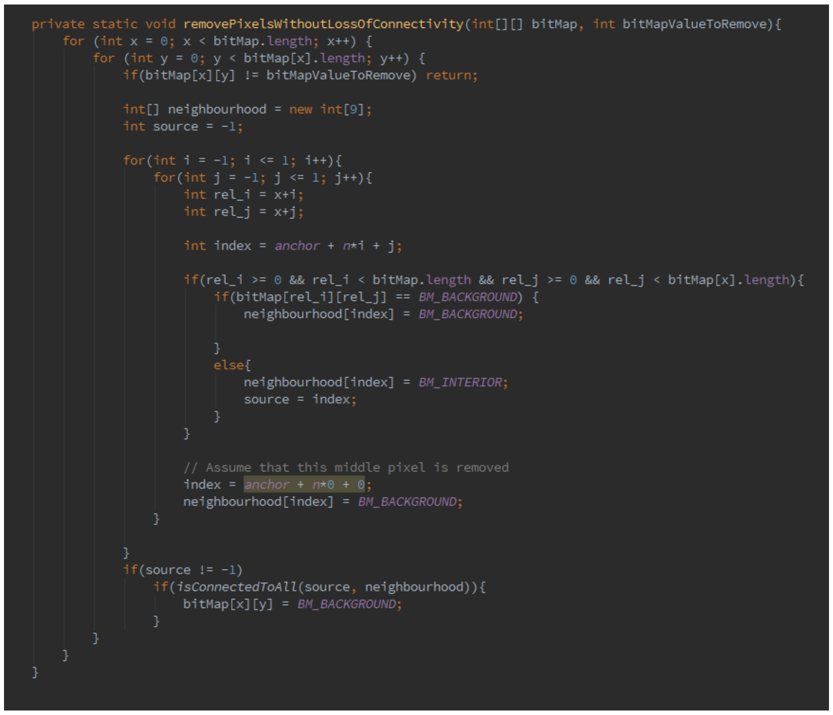
```java
// Step 2
// a) All '1s' sticking to background become '2'
// b) All '1s' sticking to background by corner are changed to '3s'
private static void choosePixelStickingToBackground(int[][] bitMap){
   // For each pixel call
   for (int x = 0; x < bitMap.length; x++) {
        for (int y = 0; y < bitMap[x].length; y++) {
            selectSickingPixel(bitMap, x, y);
        }
    }
}

private static void selectSickingPixel(int[][] bitMap, int x, int y){
    for(int i = -1; i <= 1; i++){
        for(int j = -1; j <= 1; j++){
            int rel_i = x+i;
            int rel_j = x+j;
            if(rel_i >= 0 && rel_i < bitMap.length && rel_j >= 0 && rel_j < bitMap[x].length){

                int currentValue = bitMap[rel_i][rel_j];
                if(currentValue == BM_BACKGROUND) {
                    int whichSticky = getPointRelativeToAnchor(i, j, StickyNeighbours);

                    if(whichSticky == BM_INTERIOR) continue;
                    bitMap[x][y] = whichSticky;
                    if(whichSticky == BM_STICKING_TO_BACKGROUND){
                        return;
                    }
                }
            }
        }
    }
}

//---------------------------------------------------------------------
```

Figure 5: Choosing Pixels Sticking to Background



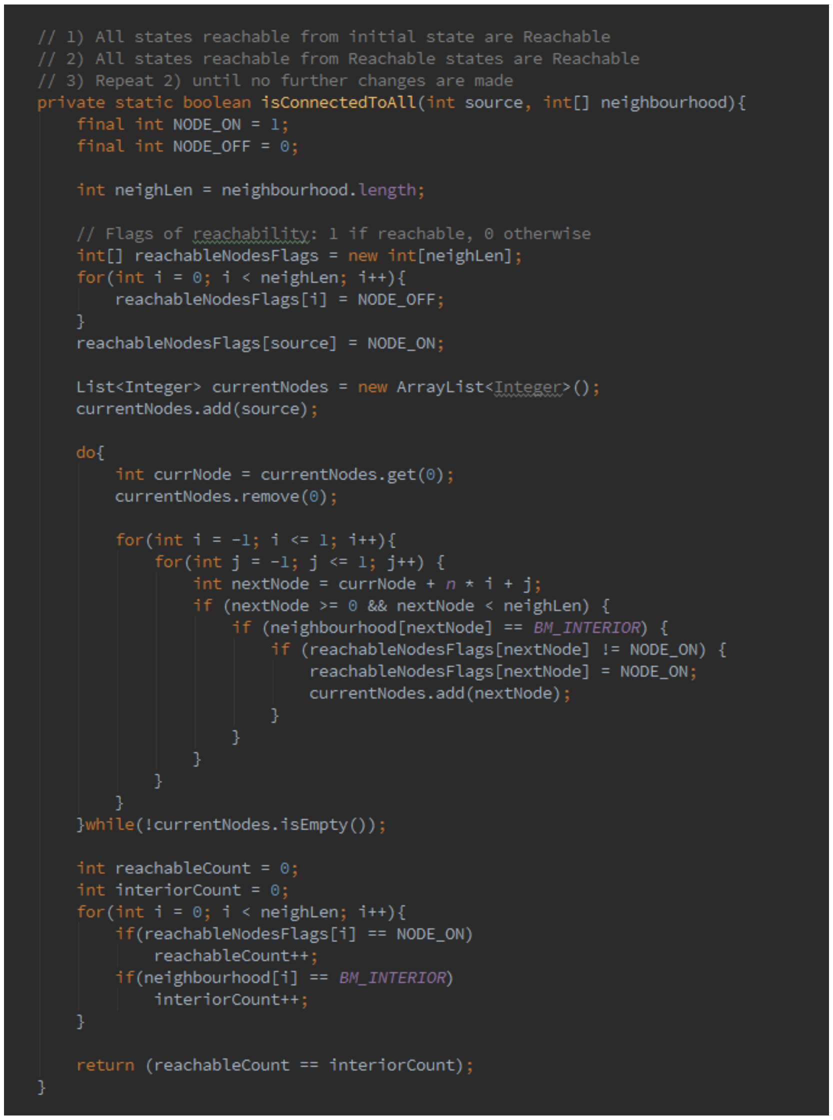
```java
// Step 3
private static void choosePixelSToDeletion_BinaryWeights(int[][] bitMap){
    for (int x = 0; x < bitMap.length; x++) {
        for (int y = 0; y < bitMap[x].length; y++) {
            if(bitMap[x][y] == BM_BACKGROUND) continue;

            int weight = getSumOfBinaryWeights(bitMap, x, y);
            if(isInDeletionArray(weight))
                bitMap[x][y] = BM_TO_DELETE;
        }
    }

    for (int x = 0; x < bitMap.length; x++) {
        for (int y = 0; y < bitMap[x].length; y++) {
                if(bitMap[x][y] == BM_TO_DELETE)
                    bitMap[x][y] = BM_BACKGROUND;
        }
    }
}

private static int getSumOfBinaryWeights(int[][] bitMap, int x, int y){
    int sum = 0;

    for(int i = -1; i <= 1; i++){
        for(int j = -1; j <= 1; j++){
            int rel_i = x+i;
            int rel_j = x+j;
            if(rel_i >= 0 && rel_i < bitMap.length && rel_j >= 0 && rel_j < bitMap[x].length){
                if(bitMap[rel_i][rel_j] != BM_BACKGROUND)
                    sum += getPointRelativeToAnchor(i, j, BinaryWeights);
            }
        }
    }
    return sum;
}

//------------------------------------------------------------------------
```

Figure 6: Choosing Pixels to Delete



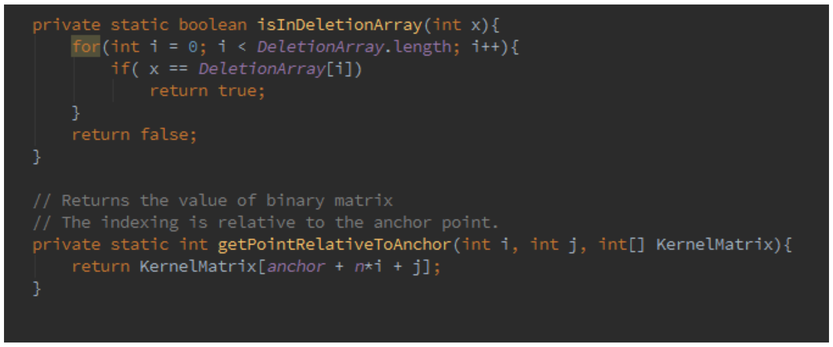
```java
private static void removePixelsWithoutLossOfConnectivity(int[][] bitMap, int bitMapValueToRemove){
    for (int x = 0; x < bitMap.length; x++) {
        for (int y = 0; y < bitMap[x].length; y++) {
            if(bitMap[x][y] != bitMapValueToRemove) return;

            int[] neighbourhood = new int[9];
            int source = -1;

            for(int i = -1; i <= 1; i++){
                for(int j = -1; j <= 1; j++){
                    int rel_i = x+i;
                    int rel_j = x+j;

                    int index = anchor + n*i + j;

                    if(rel_i >= 0 && rel_i < bitMap.length && rel_j >= 0 && rel_j < bitMap[x].length){
                        if(bitMap[rel_i][rel_j] == BM_BACKGROUND) {
                            neighbourhood[index] = BM_BACKGROUND;

                        }
                        else{
                            neighbourhood[index] = BM_INTERIOR;
                            source = index;
                        }
                    }

                    // Assume that this middle pixel is removed
                    index = anchor + n*0 + 0;
                    neighbourhood[index] = BM_BACKGROUND;
                }

            }
            if(source != -1)
                if(isConnectedToAll(source, neighbourhood)){
                    bitMap[x][y] = BM_BACKGROUND;
                }
        }
    }
}
```

Figure 7: Removing Pixels without loss of connectivity: Main loop



```java
// 1) All states reachable from initial state are Reachable
// 2) All states reachable from Reachable states are Reachable
// 3) Repeat 2) until no further changes are made
private static boolean isConnectedToAll(int source, int[] neighbourhood){
    final int NODE_ON = 1;
    final int NODE_OFF = 0;

    int neighLen = neighbourhood.length;

    // Flags of reachability: 1 if reachable, 0 otherwise
    int[] reachableNodesFlags = new int[neighLen];
    for(int i = 0; i < neighLen; i++){
        reachableNodesFlags[i] = NODE_OFF;
    }
    reachableNodesFlags[source] = NODE_ON;

    List<Integer> currentNodes = new ArrayList<Integer>();
    currentNodes.add(source);

    do{
        int currNode = currentNodes.get(0);
        currentNodes.remove(0);

        for(int i = -1; i <= 1; i++){
            for(int j = -1; j <= 1; j++) {
                int nextNode = currNode + n * i + j;
                if (nextNode >= 0 && nextNode < neighLen) {
                    if (neighbourhood[nextNode] == BM_INTERIOR) {
                        if (reachableNodesFlags[nextNode] != NODE_ON) {
                            reachableNodesFlags[nextNode] = NODE_ON;
                            currentNodes.add(nextNode);
                        }
                    }
                }
            }
        }
    }while(!currentNodes.isEmpty());

    int reachableCount = 0;
    int interiorCount = 0;
    for(int i = 0; i < neighLen; i++){
        if(reachableNodesFlags[i] == NODE_ON)
            reachableCount++;
        if(neighbourhood[i] == BM_INTERIOR)
            interiorCount++;
    }

    return (reachableCount == interiorCount);
}
```

Figure 8: Finding if given sub neighbourhood is connected



```
private static boolean isInDeletionArray(int x){
    for(int i = 0; i < DeletionArray.length; i++){
        if( x == DeletionArray[i])
            return true;
    }
    return false;
}

// Returns the value of binary matrix
// The indexing is relative to the anchor point.
private static int getPointRelativeToAnchor(int i, int j, int[] KernelMatrix){
    return KernelMatrix[anchor + n*i + j];
}
```

Figure 9: Utility Methods

2 K3M

2.1 Algorithm

Input to K3M is again a binary image where interior pixels (black) are coded in the Bit Map with value 1 and background pixels (white) are 0 A single iteration of K3M consists of 7 phases. The algorithm iterates until no further modifications of the input image is done. Each phase decides which pixel to be deleted based on a specific to this phase Deletion Array. The last phase of the algorithm is supposed to create a 1-pixel width skeleton.
Let us consider the 7 phases of a single iteration. Firstly, defined a neighbourhood of a given Pixel as a 8-point neighbourhood which defines neighbours in the local area of the Pixel. Border pixels are pixels that stick in some way to the background. The border pixels are to be decided whether they should be removed or not. The Thinning decision is based on a 8-point neighbourhood template which defines a weight of the corresponding pixels in the neighbourhood. Phase 0 chooses the border pixels which then are potentially deleted in Phases 1 through 5. The neighbourhood Bit Matrix looks as follows:

$$N = \begin{bmatrix} 128 & 1 & 2 \\ 64 & 0 & 4 \\ 32 & 16 & 8 \end{bmatrix}$$

The weight of the neighbourhood is calculated as follows:

$$w(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} N(i+1, j+1) * I(x+i, y+j)$$

Where I is the input image and N is the Bit Matrix.
Each Phase contains its own Look Up table. The Look Up table for Phase 0 is used to determine the border pixels. Phases 1 through 5 determine whether these border pixels should be removed based on their own Look Up tables.
Now the flow of Phase 0 is presented

a. For each Pixel in the input image, do:

   (a) Let (x, y) be the coordinates of tested Pixel
   (b) Calculate Neighbourhood weight $w(x, y)$
   (c) If the weight $w(x, y)$ is contained in the Look Up table A_0 then add the tested Pixel to border pixels

The Phases i for $i = 1, 2, 3, 4, 5$ look as follows:

a. For each Pixel in the border pixels, do:
   (a) Let (x, y) be the coordinates of tested Pixel
   (b) Calculate Neighbourhood weight $w(x, y)$
   (c) If the weight $w(x, y)$ is contained in the Look Up table A_i then set the tested Pixel to background color.

The Look Up tables for each Phase:

a. Phase 0: A_0 - Marking borders
b. Phase 1: A_1 - Deleting pixels having 3 sticking neighbours
c. Phase 2: A_2 - Deleting pixels having 3 or 4 sticking neighbours
d. Phase 3: A_3 - Deleting pixels having 3, 4 or 5 sticking neighbours
e. Phase 4: A_4 - Deleting pixels having 3, 4, 5 or 6 sticking neighbours
f. Phase 5: A_5 - Deleting pixels having 3, 4, 5, 6 or 7 sticking neighbours
g. Phase 6: A_6 - Used for thinning the one-pixel width skeleton.

Finally the flow of entire K3M algorithm looks as follows.

a. For input binary image, iterate through Phases $1, 2, 3, 4, 5$
b. Repeat iterations until no further modification to image have been computed.
c. Compute the 1-pixel width phase.

3 Comparison

KMM used only one Look Up table where K3M uses plenty of them through out all of its phases. This leads to better quality results in the modified image. Both K3M and KMM produce a one-pixel wide skeletons but K3M leads to better angle preservation.
Both K3M and KMM are sequential and iterative algorithms which defines their high computational complexities.