CHAPTER V
IMPLEMENTATION AND TESTING

5.1. Implementation

The fingerprint recognition process will divided into following four major processes:

5.1.1. Preprocessing

First process in preprocessing will be extracting ridge structure of fingerprint from background image. The algorithm implemented here is Histogram Equalization. This algorithm will generate a new color range value then inverse it. Inversed color than applied to image. The extracted fingerprint will have low contrast, so Normalization will be executed to increase color value by some computed values.

The next step is Segmentation. This process done by computing each pixel block's mean and gray variance. These two variable will be used to decide whether this block will be processed in Gabor filtering or not. Blocks which will be used will have mask value 1 or 2 and 0 value for unused blocks.

Gabor filtering will refine ridge structure so it becomes line. There are two other variables needed to execute Gabor algorithm. First, compute orientation of each pixel block by averaging all pixel orientations inside this block.

```
public double getOrientation(int xx, int yy) {
    double ori;
    double oriA = 0;
    double oriB = 0;

    for (int y = (yy - (getBlockSize() / 2)); y < (yy + (getBlockSize() / 2)); y++) {
        for (int x = (xx - (getBlockSize() / 2)); x < (xx + (getBlockSize() / 2)); x++) {
            try {
                oriA += 2 * gradientX[y][x] * gradientY[y][x];
                oriB += Math.pow(gradientX[y][x], 2) * Math.pow(gradientY[y][x], 2);
            } catch (ArrayIndexOutOfBoundsException e) { }
        }
    }

    ori = 0.5 * Math.atan(oriA / oriB);

    return ori;
}
```

Figure 5.1. Function to get orientation

Second variable is ridge frequency. To get ridge frequency, generate array of X-Signature and find peaks which represent ridge on this pixel block. Then average the distance between each two peaks. This average will be ridge frequency.

```
public int[] getSignature(int xx, int yy, int zz) {
    int sig[] = new int[3]; // Store x, y, and signature
    int x, y;

    sig[2] = 0;

    for (int z = 0; z < getBlockSize(); z++) {
        x = (int) (xx
                + (((z - (getBlockSize() / 2)) * Math.cos(orientation[yy][xx]))
                + ((zz - 0.5) * Math.sin(orientation[yy][xx]))));
        y = (int) (yy
                + (((z - (getBlockSize() / 2)) * Math.sin(orientation[yy][xx]))
                + ((0.5 - zz) * Math.cos(orientation[yy][xx]))));

        try {
            sig[2] += data.getPixel(x, y).getValue();
        } catch (ArrayIndexOutOfBoundsException e) { }

        if (z == 0) {
            sig[0] = x;
            sig[1] = y;
        }
    }

    sig[2] = sig[2] / getBlockSize();

    return sig;
}
```

Figure 5.2. Function to get number of pixels height (signature)

By given orientation and frequency images, calculate Gabor function on each block inside mask. Block outside mask will have Gabor value 255.

```
public double getGabor(int xx, int yy) {
    int size = 11 / 2; // Half size of filter
    int x, y;
    double o, f;
    int dx = 4;
    int dy = 4;
    double gab;
    double gabA = 0;
    double gabB = 0;
    double val = 0;

    for (int j = -size; j <= size; j++) {
        for (int i = -size; i <= size; i++) {
            try {
                o = orientation[yy + (j * getBlockSize())][xx + (i * getBlockSize())];
                f = frequency[yy + (j * getBlockSize())][xx + (i * getBlockSize())];
                x = (int) (((xx + (i * getBlockSize())) * Math.cos(o))
                        + ((yy + (j * getBlockSize())) * Math.sin(o)));
                y = (int) (-((xx + (i * getBlockSize())) * Math.sin(o))
                        + ((yy + (j * getBlockSize())) * Math.cos(o)));
                gabA += Math.exp(-0.5 * (((x * x) / (dx * dx)) + ((y * y) / (dy * dy))));
                gabB += Math.cos(2 * Math.PI * f * x);
                val += data.getPixel((xx + (i * getBlockSize()))
                        , (yy + (j * getBlockSize()))).getType();

            } catch (ArrayIndexOutOfBoundsException e) { }
        }
    }

    gab = (gabA * gabB) * val;

    return gab;
}
```

Figure 5.3. Function to get Gabor value

This Gabor filter had fully coded and ran without error, but not giving any result. There might be a miss on algorithm implementation. On some other references, before applying Gabor filter, a filter bank must be defined. This filter bank contains a set of filter with various rotations. The most common Gabor filter contains 24 filters (8x3). This filter applied after finding Gabor value.

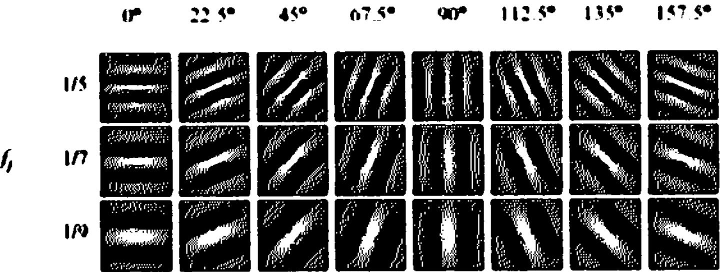

Figure 5.4. A graphical representation of a bank of 24 (n o = 8 and n f = 3) Gabor filters where $\sigma x = \sigma y = 4$

Because the Gabor filter failed to be implemented, further implementation only can be applied to enhanced fingerprint images. After the image enhanced, a thresholding operation will be performed. This step will change pixel value into 0 (black) or 255 (white) only. Pixel value below 128 will change to 0 and pixel type 1. Other pixel values will change into 255 and type 0.

5.1.2. Feature Extraction

Feature extraction is a process to extract core point and minutia points. Core point will performed on normalized image, not enhanced image. For minutia extraction, the ridge structures need to be thinned. Thinning will be applied to reduce ridge width to one pixel. This algorithm will process only pixels with type 1. Thinning will iteratively remove pixel on ridge until only skeleton pixels remain. Thinning will process only pixels inside mask (mask = 1 or mask = 2).

Minutiae extraction will check crossing number (CN) value on each pixel with type = 1 and mask = 1. This algorithm will calculate 3x3 pixel neighbors to find CN. If CN = 1 or CN = 3, mark it as minutiae by changing pixel type to 2.

```java
public PixelMatrix extract(PixelMatrix d) {
    int cn; // Crossing number

    data = d;
    minutia = new int[getHeight()][getWidth()];

    for (int y = 1; y < (getHeight() - 1); y++) {
        for (int x = 1; x < (getWidth() - 1); x++) {
            if ((data.getPixel(x, y).getType() == 1) && (data.getPixel(x, y).getMask() == 2)) {
                cn = Math.abs(data.getPixel(x+1, y).getType()
                        - data.getPixel(x+1, y-1).getType()) // P1 - P2
                        + Math.abs(data.getPixel(x+1, y-1).getType()
                        - data.getPixel(x, y-1).getType()) // P2 - P3
                        + Math.abs(data.getPixel(x, y-1).getType()
                        - data.getPixel(x-1, y-1).getType()) // P3 - P4
                        + Math.abs(data.getPixel(x-1, y-1).getType()
                        - data.getPixel(x-1, y).getType()) // P4 - P5
                        + Math.abs(data.getPixel(x-1, y).getType()
                        - data.getPixel(x-1, y+1).getType()) // P5 - P6
                        + Math.abs(data.getPixel(x-1, y+1).getType()
                        - data.getPixel(x, y+1).getType()) // P6 - P7
                        + Math.abs(data.getPixel(x, y+1).getType()
                        - data.getPixel(x+1, y+1).getType()) // P7 - P8
                        + Math.abs(data.getPixel(x+1, y+1).getType()
                        - data.getPixel(x+1, y).getType()); // P8 - P1
                cn = cn / 2;

                // 1 - Ridge termination
                // 3 - Ridge bifurcation
                if ((cn == 1) || (cn >= 3)) {
                    for (int j = -10; j <= 10; j++) {
                        for (int i = -10; i <= 10; i++) {
                            try {
                                // Check if there is another minutiae in 7x7 neighborhood
                                if (minutia[y + i][x + j] == 2) {
                                    // Delete minutiae
                                    minutia[y + i][x + j] = 1;
                                    data.getPixel(x + i, y + j).setValue(225);
                                }
                            } catch (ArrayIndexOutOfBoundsException e) { }
                        }
                    }

                    // Mark pixel as minutiae
                    minutia[y][x] = 2;
                    data.getPixel(x, y).setValue(0);
                }
            }
        }
    }

    return data;
}
```

Figure 5.5. Function to detect minutia points

Reference point needed to generate hash. It is the core of fingerprint. Reference point located on pixel block which have minimum orientation consistency.

```java
public double getConsistency(int xx, int yy, int s) {
    int m = 8 * s;
    double con;
    double conA = 0;
    double conB = 0;

    for (int y = yy; y <= (yy + getW()); y++) {
        for (int x = xx; x <= (xx + getW()); x++) {
            try {
                conA += Math.cos(2 * orientation[y][x]);
                conB += Math.sin(2 * orientation[y][x]);
            } catch (ArrayIndexOutOfBoundsException e) { }
        }
    }

    con = Math.sqrt(Math.pow(conA, 2) + Math.pow(conB, 2)) / m;

    return con;
}
```

Figure 5.6. Function to get orientation consistency

Implementation of reference point detection was not giving a good result. Detected reference point always located outside mask, or even on pixel block without ridge. Like Gabor filter, application will not execute this algorithm.

5.1.3. Minimum Distance Graph (MDG) based Hashing

To generate fingerprint hash, detected minutia points' coordinates (x and y) need to be stored into a new array. These minutia points will be used as graph nodes. Because the implementation failed to detect reference point, the first detected minutiae will be the first node of graph.

For each node, calculate its distance with every other nodes and store it to distance matrix. Then, construct minimum distance graph by sorting all stored distance. The source node will be the first node (id 0). Next, store the distance of graph's nodes to new array. This array of distances is the fingerprint hash which will be stored to database file.

```java
public int[][] calculate(int[][] n) {
    nodes = n;
    distance = new int[getLength()][getLength()];

    // Calculate distance between minutiae points
    for (int j = 0; j < getLength(); j++) {
        for (int i = 0; i < getLength(); i++) {
            distance[j][i] = getDistance(nodes[i], nodes[j]);
        }
    }

    return distance;
}

public int getLength() { return nodes.length; }

public int getDistance(int[] src, int[] dst) {
    return (int) Math.sqrt(
            Math.pow(dst[0] - src[0], 2) +
            Math.pow(dst[1] - src[1], 2));
}

public int[] generate(int[][] d) {
    distance = d;
    index = new int[getLength()];
    hash = new int[getLength()];

    int min;
    int next = 0;

    for (int i = 1; i < getLength(); i++) {
        min = -1;
        for (int j = 0; j < getLength(); j++) {
            if (!isExist(j)) {
                if ((min == -1) || (distance[index[i - 1]][j] < min)) {
                    min = distance[(index[i - 1])][j];
                    next = j;
                }
            }
        }
        index[i] = next;
        hash[i - 1] = min;
    }

    return hash;
}
```

Figure 5.7. Functions to construct distance matrix and fingerprint hash

5.1.4. Correspondence Search Algorithm (CSA)

This step contains matching query hash against stored hashes using CSA. CSA will find maximum matching pairs and calculate its score. The stored hash with maximum matching score and matching score above threshold will be the same fingerprint as query hash.

The matching process needs two hashes; query hash and stored hash. Stored hash can be retrieved from "data.txt". Higher length hash denoted as hashA, and the other as hashB.

CSA, which divided into two part, will first try finding exact matching pairs using CSA1. CSA1 takes every three hash nodes to be compared against another hash's three nodes. This matching will stop when there are no exact match found.

```
public void checkMatch(int i, int j) {
    // Step 2
    int ii = i;
    int jj = j;
    int s = 0;

    while ((i < (ii + 3)) && (j < (jj + 3)) && (i < (getLengthA() - 2))
            && (j < (getLengthB() - 2))) {
        if (Math.abs(hashA[i] - hashB[j]) <= thresholdDistance)
            s++;

        i++;
        j++;
    }

    flagMatch(i, j, s, ii, jj);
}
```

Figure 5.8. Matching function on CSA1

CSA2A will start immediately after CSA1 stopped. CSA2A will try to find matching pairs from the node which CSA1 stopped. CSA2A will assume that there is minutia insertion or deletion so the node id for hashA and hashB may not be the same. CSA2A can detect up to 4 insertion or deletion. CSA2B's algorithm is the same with CSA2A, but CSA2B can detect up to 9 insertion or deletion.

```java
public void checkCase1(int i, int j) {
    // Case 1
    if (Math.abs(hashA[i] - hashB[j]) <= thresholdDistance) {
        if (isExist('i', new int[]{(i + 1)}) && isExist('j', new int[]{(j + 1)})) {
            if (Math.abs(hashA[i + 1] - hashB[j + 1]) <= thresholdDistance) {
                flagA[i] = 1;
                flagB[j] = 1;
                checkCase(i + 1, j + 1);
            } else
                checkCase2(i, j);
        } else if (isExist('i', new int[]{(i - 1)}) && isExist('j', new int[]{(j - 1)})) {
            if (Math.abs(hashA[i - 1] - hashB[j - 1]) <= thresholdDistance) {
                flagA[i] = 1;
                flagB[j] = 1;
                checkCase(i + 1, j + 1);
            } else
                checkCase2(i, j);
        } else
            checkCase2(i, j);
    } else
        checkCase2(i, j);
}

public void checkCase2(int i, int j) {
    // Case 2
    if ((Math.abs(hashA[i] - hashB[j]) > thresholdDistance) &&
            isExist('i', new int[]{ (i - 1), (i + 1), (i + 2) }) &&
            isExist('j', new int[]{(j - 1), (j + 1), (j + 2)})) {
        if ((Math.abs(hashA[i + 1] - hashB[j + 1]) <= thresholdDistance) &&
                (Math.abs(hashA[i + 2] - hashB[j + 2]) <= thresholdDistance) &&
                (Math.abs(hashA[i - 1] - hashB[j - 1]) <= thresholdDistance))
            checkCase(i + 1, j + 1);
        else
            checkCase3(i, j, 2);
    } else
        checkCase3(i, j, 2);
}

public void checkCase3(int i, int j, int x) {
    // Case 3
    if ((x <= 5) && isExist('i', new int[]{(i + x + 1)})
            && isExist('j', new int[]{(j + x + 1)})) {
        // Case 3A
        if ((Math.abs(hashA[i + 1] - hashB[j + x]) <= thresholdDistance) &&
                (Math.abs(hashA[i + 2] - hashB[j + x + 1]) <= thresholdDistance)) {
            flagA[i + 1] = 1;
            flagB[j + x] = 1;
            insert = x - 1;
            checkCase(i + 2, j + x + 1);
        }
        // Case 3B
        else if ((Math.abs(hashA[i + x] - hashB[j + 1]) <= thresholdDistance) &&
                (Math.abs(hashA[i + x + 1] - hashB[j + 2]) <= thresholdDistance)) {
            flagA[i + x] = 1;
            flagB[j + 1] = 1;
            insert = x - 1;
            checkCase(i + x + 1, j + 2);
        }
        // Case 3C
        else
            checkCase3(i, j, x + 1);
    } else if (insert == 0)
        checkCase(i + 1, j + 1);
}
```

Figure 5.9. Matching functions on CSA2A

After executing all CSA algorithms, system will add matching pairs from CSA1 and CSA2A as scoreA. scoreB will be addition of matching pairs from CSA1 and CSA2B. The higher

score will then be determined as the matching score of this stored hash. This score will be compared to other stored hash's matching score to find the maximum one. If the maximum matching score is higher than given threshold, this hash is identified as the same fingerprint.

5.2. Testing

5.2.1. Graphical User Interface

The first interface that come up when user start this application is Launcher. Launcher has one button to browse image through gallery.

Figure 5.10. Launcher UI

After selecting fingerprint image, application will start Generator immediately. Each progress result will be displayed on screen.

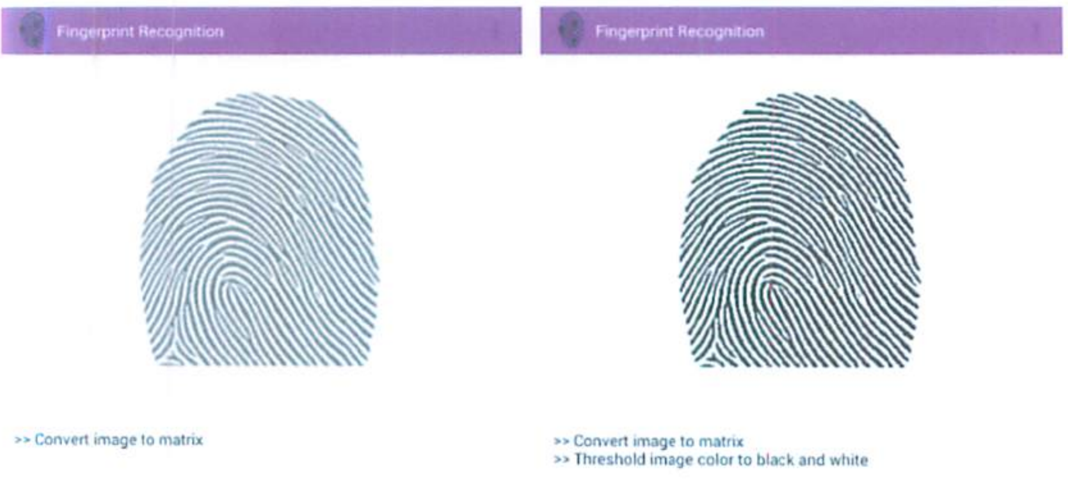

Figure 5.11. Generator UI on first run and after thresholding

Two buttons will show up when hashing complete. Registration will lead user to registration form. User have to enter name to create new identity record.

Identification process will first check whether data exist on "data.txt" file or not. If there is at least one data, application will execute CSA to match query hash with stored ones. Identification interface will then show maximum matching score.

5.2.2. Application

The application testing was done by using already enhanced images as input images. The images was taken from several journals which discuss about fingerprint preprocessing using Gabor Filter. This solution was taken because the implementation of Gabor Filter failed.

Reference point detection, which was also failed to extract true core point position, is not executed in testing. The first node on minimum distance graph that should be reference point will be replaced by first detected minutiae point. This replacement will affects the fingerprint matching score, especially on rotated query image.

This test divided into two parts; algorithm implementation and fingerprint matching. The algorithm implementation testing has maximum 2 points. If algorithm works as expected, the point will be 2 and 1 if it's not. Not working algorithm will have 0 point.

Table 5.1. Algorithm implementation test

No.	Item	Expected	Reality	Point
1	Fingerprint extraction	Extract ridge from background	Extract ridge from background	2
2	Fingerprint enhancement	Refine ridge into line	Cannot obtain refine ridge	0
3	Thinning	Reduce ridge with to one pixel	Reduce ridge with to one pixel	2
4	Minutia Points Extraction	Extract ridge termination and bifurcation	Extract ridge termination and bifurcation	2
5	Reference Point Extraction	Extract fingerprint core point	False core point position	1
6	Generate MDG based hash	Get MDG nodes distances	Get MDG nodes distances	2
7	Fingerprint detection using CSA	Recognize one's fingerprint (up to 40^0 rotation, any translation)	Failed to recognize rotated fingerprint	1
Total				10

The fingerprint matching test will list the matching scores of one original stored image against its rotated images. Given threshold score 80%, fingerprint which matching score below this threshold would not be identified.

Table 5.2. Fingerprint matching test

No.	Query Image	Rotation	Matching Score
1		0^0	93.10%
2		5^0	92.31%
3		-5^0	30.77%
4		10^0	28.57%

5		-10^0	22.22%
6		15^0	32.14%
7		-15^0	16.00%