Generating, Classifying and Indexing Large Scale Fingerprints

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Jadhav Sachin Namdeo

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2012

TO

My parents

Acknowledgements

I owe my deepest gratitude to my advisor, Prof. Jayant Haritsa. I would like to thank him for his patient guidance, encouragement and advice. I am extremely fortunate to work with him.

I would like thank all my fellow labmates for their help and suggestions. Also I thank my friends who made my stay at IISc pleasant, and for all the fun we had together.

Abstract

Fingerprints are the most widely used biometric feature for identification and authentication. Fingerprint analysis algorithms are usually trained and tested with relatively small fingerprint database. Because of public security issues large fingerprint databases are not available. So synthetic fingerprints can be used as substitute to the real fingerprints. Implementation of SFinGe algorithm for generating synthetic fingerprints is described in [1]. In this implementation, I added some more features like generating fingerprints having distortion and scratches, utilizing all the cores of system using multiple threads and graphical and command line interface.

Matching a fingerprint with millions of fingerprints is expensive. To reduce candidate fingerprints for matching, [11] implemented two classification techniques, Directional Field and FingerCode. Directional Field and FingerCode give feature vectors of sizes 127 and 192 respectively. Sequential scan over data of 192 dimension, to find candidate fingerprints for matching, is expensive. Hence to reduce time to search candidate fingerprints for matching, [1] made changes in Locality-Sensitive Hashing (LSH) algorithm. Pyramid Indexing Technique [3] can also be used for biometric databases and is shown in [10]. My contribution is to implement Pyramid Indexing Technique in database to index FingerCode features of fingerprints and compare its performance with LSH implementation [1].

Another method to reduce candidate fingerprints for matching is indexing fingerprints using features extracted from minutiae triplets [4] [9]. My contribution is to implement this method and compare with Directional Field and FingerCode.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Biometric features are popular to identify and authenticate a person. Among all the biometrics, fingerprints are the most widely used due to their uniqueness and immutability. Many fingerprint recognition algorithms are trained and tested with relatively small fingerprint database. For the projects like UID where database size is an order of billion fingerprints, it is crucial to test the implementation against such large scale database. But because of the public security issues large fingerprint databases are not available. So the synthetic fingerprints can be used as an alternative to the real fingerprints.

SFinGe is a method for generating synthetic fingerprints on the basis of mathematical models. Implementation and validation of SFinGe algorithm is given in [1]. In this implementation, I added some more features like generating fingerprints having distortion and scratches, utilizing all the cores of system using multiple threads. From this implementation I developed a software, Anguli, with user friendly graphical and command line interface. Details are in Chapter 2.

A naive identification system would just match the query fingerprint with all fingerprints in database. Minutiae matching algorithm implemented in [11], to match a pair of fingerprints takes 20 milliseconds. Hence for the large scale fingerprint databases like UID, response time of such systems will be very poor. To reduce the number of fingerprints for matching, they are classified using some features extracted from them.

Two continuous classification techniques, Directional Field and FingerCode, are implemented in [11]. These methods return feature vectors of length 127 and 192 respectively. For these methods, matching score is calculated as Euclidean distance between corresponding feature vectors. To find the fingerprints which are within the hypersphere of radius R need a scan on the database and simple sequential scan on large database is expensive. To reduce time to search candidate fingerprints for matching, [1] made changes in Locality-Sensitive Hashing (LSH) algorithm. This modified LSH takes more time and returns large number of candidate fingerprints if the query fingerprint is very noisy. Pyramid Indexing Technique [3] can be efficiently used for biometric databases and is shown in [10]. Adding an indexing technique in a database system needs changes in system code base. However GiST [6] gave a novel approach to tackle this problem. GiST feature is available in PostgreSQL. I implemented Pyramid indexing technique using GiST in PostgreSQL.

Another promising approach to reduce the number of candidate fingerprints for matching is given in [4] [9]. In this approach, for each triangle formed by three non collinear minutiae of fingerprint, some features are extracted and these features are used to index fingerprints. There will be $O(n^3)$ triangles where n is the number of minutiae. Instead of exhausting all triplets of the minutiae set, [9] used low order Delaunay triangles. This saves computation cost and reduces the possibility of mismatch. My contribution is to implement this method and compare with Directional Field and FingerCode.

Chapter 2

Generating Large Scale Fingerprints

Algorithms for fingerprint recognition are trained and tested with relatively small fingerprint database and might not be scalable. Because of the public security issues large digital fingerprint databases are not available. So the synthetic fingerprints can be used as an alternative to the real fingerprints. SFinGe is a method for the generation of synthetic fingerprints on the basis of mathematical models. Implementation and validation of SFinGe method are described in [1]. In this section we will use the terms Fingerprint and Impressions and are shown in Figure 2.1. Impression is generated from a fingerprint by masking and adding noise, scratches and distortion to a masked fingerprint. Multiple impressions can be generated from a fingerprint by adding random noise, scratches and distortion.

2.1 Synthetic Fingerprint Generator: Anguli

From [1] implementation, I developed a software Anguli that is easy to use. Code is written in C++ using Qt, a cross-platform application framework, and used OpenCV library, an Open Source Computer Vision Library. Followings are the changes done in [1] implementation:

1. Algorithmic changes
   (a) Distorted impressions

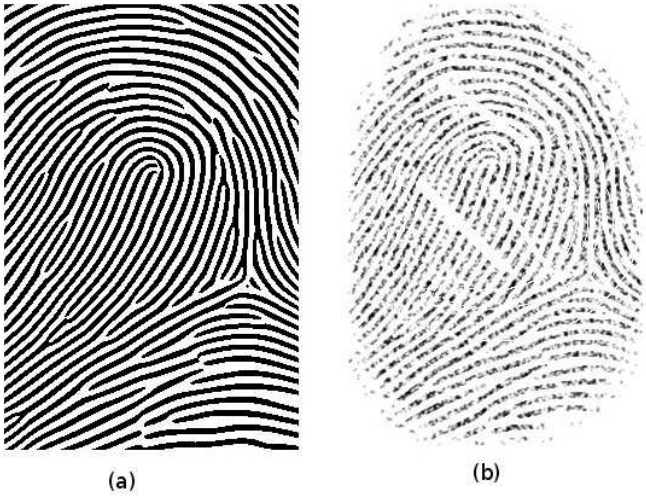

Figure 2.1: (a) Fingerprint generated by Anguli. (b) Masked impression of fingerprint (a) with noise, scratches and distortion.

   (b) Scratches in impressions

2. Performance changes

   (a) Removed memory leaks

   (b) Multi-threaded application to utilize the CPU cores of the machine

   (c) Generating multiple impressions in single run

3. Interface changes

   (a) Graphical User Interface

   (b) Command Line Interface

4. Maintenance changes

   (a) Modularized the code

   (b) Documented the code

   (c) Website

5. Porting

(a) Linux 32-Bit

(b) Linux 64-Bit

(c) Windows 32-Bit

Usage of Anguli is described in the following subsections.

2.2 Graphical User Interface

Figure 2.2 shows the main window of Anguli. From this window user can generate a new fingerprint database or impressions from the previously generated fingerprint database.

2.2.1 Generating Fingerprint Database

On clicking "Generate Fingerprints" button of main window, in Figure 2.2, window shown in Figure 2.3 opens.

Figure 2.2: Main window of Anguli

In the Basic Settings, user can specify the number of fingerprints and number of different distorted impressions per fingerprint to be generated. Fingerprints are generated according to the Henry Classification. User can select the distribution of the fingerprint classes. As Anguli uses the pseudo random number generator, it allows user to give custom seed, an integer. So that user can ensure distinct fingerprint images on each

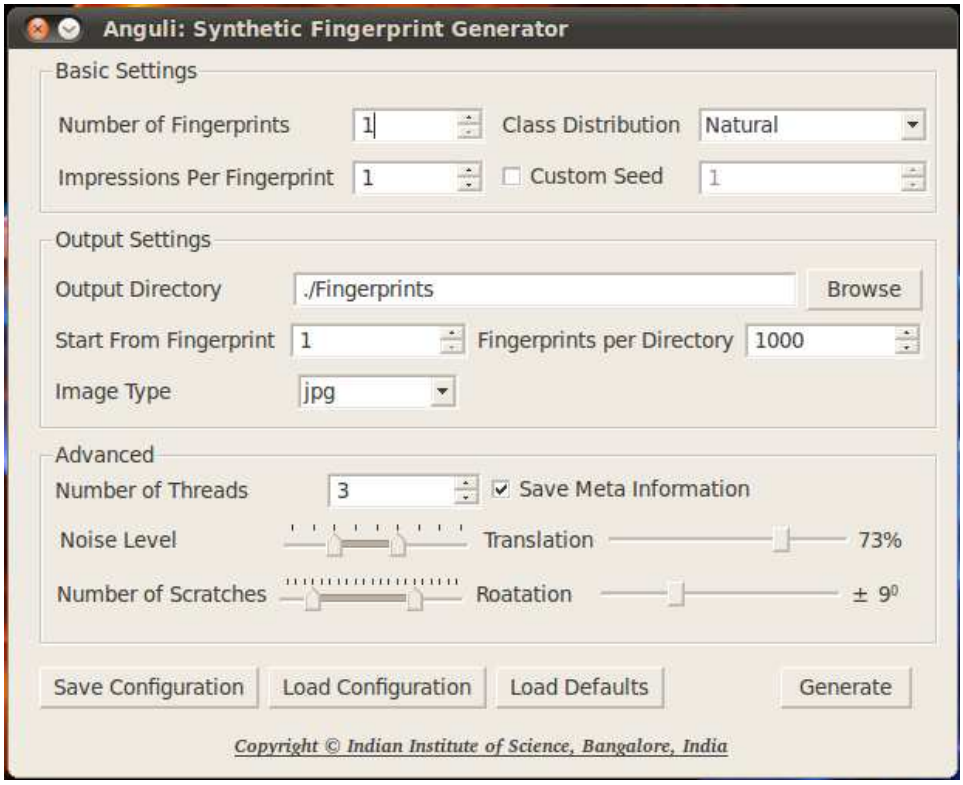

Figure 2.3: Window for generating fingerprints

run of fingerprint generation. By default Anguli takes the system date, time as seed for pseudo random number generator.

In the Output Settings, user can specify the directory in which fingerprints are to be stored, number from which naming to fingerprint images should start, number of fingerprints per directory and format in which the fingerprint images are to be stored. Some file system have restrictions on the number of files per directory, so the user has to give appropriate number for the number of fingerprints per directory. Anguli can save fingerprint images in *jpeg, jpg, png, bmp, tif, tiff* format.

In the Advanced Settings, user can specify the number of threads to be created to generate fingerprints. One fingerprint will be generated by only one thread. As the generation is CPU intensive operation, user should be careful about specifying the number of threads. It depends on the number of CPUs and their speed. In default settings it is one less than the number of CPU cores in the system. Anguli also saves the information about the generated fingerprints (meta information). This information

includes the class of fingerprint according to the Henry Classification, location of the core and delta points in the generated fingerprint. User can specify the ranges for noise level, number of scratches to be added to impressions and rotation and displacement of finger to produce distorted impressions.

Generation process will start on clicking "Generate" button. Anguli also provides the options to save the current settings and load previous settings. It also saves the settings in output directory before generation actually starts. When generation actually begins, it shows the progress bar with number of fingerprints generated out of total number of fingerprints and time to generate remaining fingerprints.

2.2.2 Generating Only Impressions From Previous Fingerprint Database

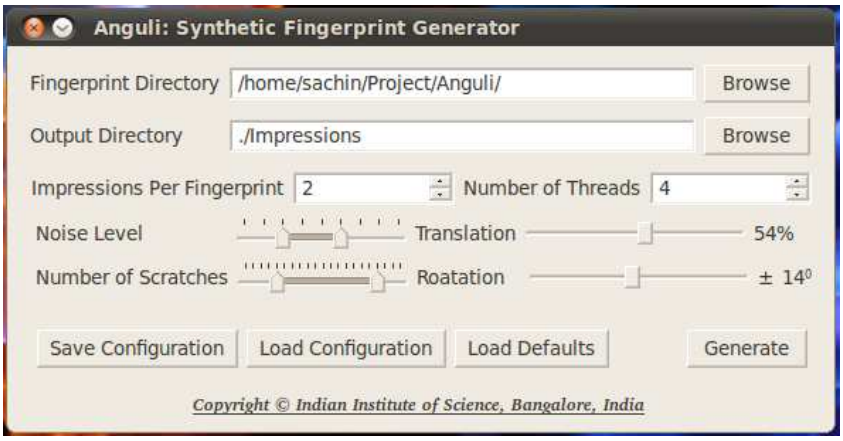

Figure 2.4: Window for generating impressions

On clicking "Generate Impressions" button of main window, in Figure 2.2, window shown in Figure 2.4 opens. It allows user to generate the impressions of fingerprints that has been generated previously. Input directory should contain either the fingerprint images or subdirectories containing the fingerprint images. Generated impressions will be stored in directory specified in output directory option. User can specify the number of distorted impressions to be produced per fingerprint and number of threads to be created to generate the impressions. Setting for noise levels, number of scratches, rotation and

displacement is same as of generating fingerprint database. Clicking generate button will start generating impressions.

2.2.3 Command Line Interface

Anguli can also be executed from remote console using the command line. Command "Anguli -h" displays all the options available with Anguli. Options are described in Appendix A. Anguli in command line mode is used as follows:

1. For generating new fingerprints:
   ./Anguli -num number [Options]

2. For generating impressions:
   ./Anguli -impr -indir <path> -ni number [Options]

2.3 Generation Speed

Anguli generates 1 Million fingerprints in about 4 days with 7 threads, on a system with 8 cores of 2 GHz and 16 GB RAM. Anguli took only 200 MB of memory for 7 threads. Required memory will increase if the number of threads are increased. With 1000 cores and 1000 threads, Anguli can generate 1 Billion fingerprints in 4 days.

Chapter 3

Pyramid Indexing for Directional Field and FingerCode

A naive fingerprint identification system would just match the query fingerprint with all fingerprints in database. But the response time of such system is large, as the fingerprint database contains more than several million fingerprints, which is unacceptable. To reduce such problem, fingerprints are assigned to classes in consistent and reliable way. In continuous classification, fingerprint is classified uniquely and independently summarizing its main features. Implementation of two classification techniques, Directional Field and FingerCode are described in [11].

Directional Field describes the coarse structure, or the basic shape of a fingerprint. It is a orientation of the ridge and valley structures which describes the shape of a fingerprint.

In FingerCode technique a circular area around core point is divided into several tracks and tracks are divided into several sectors. Circular area is filtered with eight different Gabor filters and standard deviation of gray values in all sectors are computed. Final feature vector consist of standard deviation in all sectors of filtered image.

For Directional Field and FingerCode, matching score is calculated as Euclidean distance between the feature vectors of two impressions. Implementations of FingerCode returns 192 dimensional feature vector and implementation of Directional Field returns

127 dimensional feature vector [11]. Normal sequential scan on a large database of 192 or 127 dimensional feature vector is expensive. Pyramid Indexing Technique [3] is efficient for biometric database and is shown in [10]. Adding an indexing technique in a database system needs changes in system code base. However GiST [6] gave a novel approach to tackle this problem. Details of Pyramid Indexing Technique and GiST are given in the Appendices B and C.

3.1 Implementation of Pyramid Technique in PostgreSQL Using GiST

I implemented the Pyramid Indexing Technique [3] in PostgreSQL using GiST [6]. To store a tree node, created a new variable length data type. For each internal node we store the minimum and maximum pyramid value of the pyramid values of all nodes contained in its sub-tree. Leaf node has minimum and maximum values and both are equal to pyramid value of data pointed by it. Along with minimum and maximum values store the original data point, which will be needed to compare with query points, in leaf node.

Pyramid Indexing Technique returns all the points which are inside the hypercube formed by range given by user. As the dimension of feature vector is high, small increase in the range will increase the number of valid data points by large number. Suppose $P = [p_1, p_2, ..., p_{192}]$ and $Q = [q_1, q_2, ..., q_{192}]$ be two feature vectors of two impressions of same finger. Let $x = max(abs(p_i - q_i))\ i = 1, ..., 192$ and $d = EuclideanDistance(P, Q)$. We need to create a hypercube of length $2 \times x$ with Q at center, so that P will be returned by query. If we return all the points in hypercube, then it will also return some data points having distance $x \times \sqrt{192}$. If $d \ll x \times \sqrt{192}$ then there will be many false positives. To remove such false positives calculate distance of each data point, inside the hypercube, from the query point and return data point if calculated distance is less than d. So the data points are firstly filtered by hypercube and then by distance.

Let feature vector be $(x_1, x_2, ..., x_d)$ and E be a tree node. If E is a leaf node then E

$= (min, max, x_1, \cdots, x_d)$ else $E = (min, max)$. Implementations of GiST key methods for Pyramid Indexing Technique are described below:

1. **Compress**$(x_1, x_2, ..., x_d)$: Calculate the pyramid value of $(x_1, x_2, ..., x_d)$. Create a leaf node, copy data point $(x_1, x_2, ..., x_d)$ in leaf node and set min and max value to the calculated pyramid value. Return leaf node.

2. **Penalty**$(E(min_1, max_1), F(min_2, max_2, ...))$: F is a new node which is to be inserted. Return $max(max_2 - max_1, 0) + max(min_1 - min_2, 0)$

3. **Picksplit**(P): P is a set of nodes E_i. Sort all entries in node P according to min values of E_i. First $\lfloor \frac{|P|}{2} \rfloor$ entries will go into left child node and $\lceil \frac{|P|}{2} \rceil$ will go into right child. Create two child nodes and their parent node, and return them.

4. **Union** $(E_1, E_2, ..., E_n)$: $E_i = (min_i, max_i, ...)$ Return a new node E with minimum value equal to $min(min_1, min_2, ..., min_n)$ and maximum value equal to $max(max_1, max_2, ..., ma$

5. **Decompress** (E): No need to change E, return E.

6. **Consistent**$(E(min, max, ...),\ Q(range, radius, x_1, x_2, .., x_d))$: For range queries two functions are called, one for "<" operator and one for ">" operator. To reduce the function call overhead implemented only "=" operator with user feed range and radius of hypersphere. Consistent function handles leaf and non-leaf node differently as follows:

   - Non-leaf node $E(min, max)$: Initialize $A = (x_1 - range, ..., x_d - range)$, $B = (x_1 + range, ..., x_d + range)$. Return true if $Intersect(E, A, B)$ returns true. Intersect() is given in Algorithm 1.
   - Leaf node $E(min, max, p_1, ..., p_d)$: Let $X = (x_1, ..., x_d)\ and\ P = (p_1, ..., p_d)$. If $abs(p_i - x_i) < range$ for $1 \leq i \leq d$ and $dist(P, X) \leq radius$ then return true else return false.

Algorithm 1 Intersect (E, A, B)

$E(min, max, ...)$ is a node of the tree
$A(a_1, a_2, ..., a_d)$ is a minimum value of the interval
$B(b_1, b_2, ..., b_d)$ is a maximum value of the interval

```
1  for i = ⌊min⌋ to ⌊max⌋ do
2    P = A
3    Q = B
4    if i ≥ d then
5      P = [1, 1, ..., 1] − B
6      Q = [1, 1, ..., 1] − A
7    end if
8    if Query rectangle represented by P and Q intersect pyramid i then
9      Calculate interval (h_low, h_high) of intersection
10     if intervals (i + h_low, i + h_high) and (min, max) intersect then
11       return true
12     end if
13   end if
14 end for
15 return false
```

3.2 Experimental Evaluation

Experiments were run on a system with 4 cores of 3 GHz and 8 GB RAM. Used dataset containing 192 dimensional feature vector of 1 Million fingerprints extracted using FingerCode method. Values in the feature vectors are floating points and are in the interval (0, 100). Built pyramid index on this dataset in PostgreSQL. Values of translation and rotation limits for all noise levels used in experiments are shown in Table 3.1.

First calculated values for range and radius empirically. For this experiment used above dataset and took random 5000 fingerprints. Added noise of level 1 to these 5000 fingerprints. Let $P = [p_1, p_2, ..., p_{192}]$ be feature vector of a fingerprint of noise level 1 and $Q = [q_1, q_2, ..., q_{192}]$ be feature vector of corresponding fingerprint in database. Computed $maximum(x)$ where $x = max(abs(p_i - q_i))\ where\ i = 1, ..., 192$; $average(d)$ and $maximum(d)$ where $d = EuclideanDistance(P, Q)$. This is repeated for noise levels 2 to 10. $maximum(x)$ is used as range for queries and radius for queries are changed based on $average(d)$ and $maximum(d)$ in following experiments. Graph in Figure 3.1 shows

Noise Level	Translation Limits in Pixels		Rotation Limits in Degrees	
	Min	Max	Min	Max
1	-2	1	-2	1
2	-3	3	-3	3
3	-5	4	-5	4
4	-6	6	-6	6
5	-8	7	-8	7
6	-9	9	-9	9
7	-11	10	-11	10
8	-12	12	-12	12
9	-14	13	-14	13
10	-15	15	-15	15

Table 3.1: Parameter values for different noise levels

the $maximum(x)$ and graph in Figure 3.2 shows $average(d)$ and $maximum(d)$ for noise levels 1 to 10.

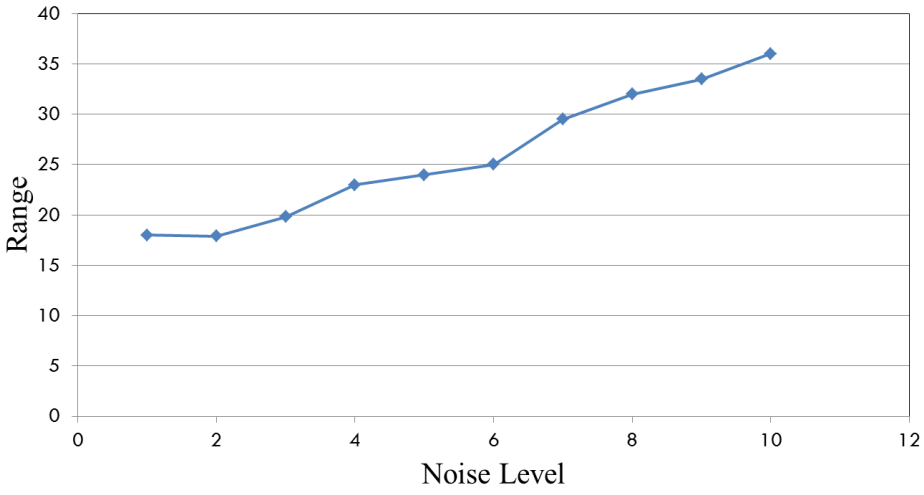

Figure 3.1: Range Vs. Noise Level

To evaluate the performance of Pyramid Indexing Technique implementation, chose random 1000 query fingerprints. Added noise of levels 2, 4, 8 and 10 to create 4 sets N2, N4, N8 and N10, each containing 1000 fingerprints. Noise levels and query fingerprints are same as of experiments in [1]. Features are extracted from query fingerprints using

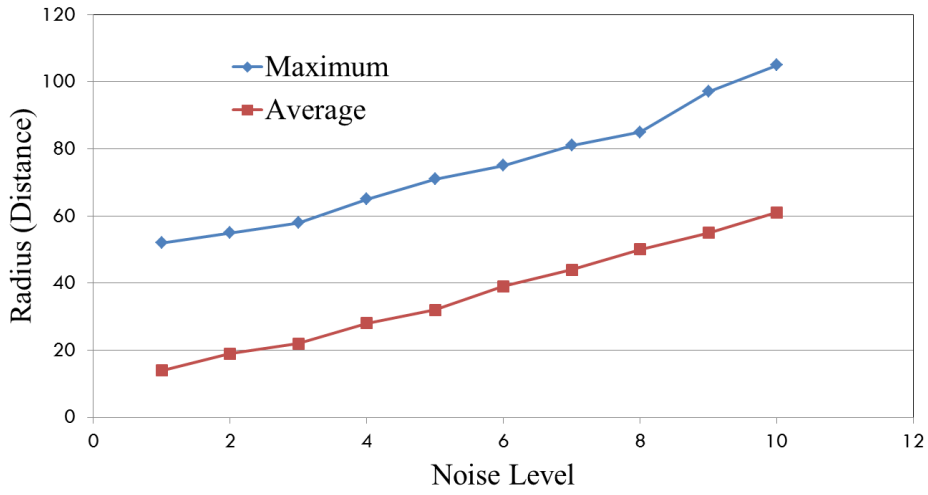

Figure 3.2: Radius (Distance) Vs. Noise Level

FingerCode method. For queries fixed the size of hypercube to $maximum(x)$ of respective noise level as per graph in Figure 3.1, and varied the radius of hypersphere from $average(d)$ to $maximum(d)$ of respective noise level as per graph in Figure 3.2.

Following parameters are used to evaluate the performance of Pyramid Indexing Technique implementation:

$$Penetration = \frac{Number\ of\ data\ points\ returned\ by\ successful\ query\ \times\ 100}{Number\ of\ data\ points\ in\ database}$$

$$Accuracy = \frac{Number\ of\ queries\ returning\ required\ data\ point\ \times 100}{Total\ number\ of\ queries}$$

Graphs in Figure 3.3 shows the penetration rate for different noise levels. For noise level 2 and 4 penetration is just 0.0001% for 100% accuracy. Considering 8 as maximum acceptable noise we need to match 0.06% of fingerprints for 100% accuracy. Even for noise level 10, penetration rate is just 0.6% for 100% accuracy which is much less than 1.93% of LSH scheme [1] for noise level 8. One advantage with Pyramid is that we can compute Euclidean distance between two feature vectors during scan of index tree. With LSH, we cannot compute the actual Euclidean distance from the hash values.

Graph in Figure 3.4 shows the time taken by a single query when the accuracy is 100% for respective noise level. For noise level 8 running time of single query is 5.2 seconds which is less than 10.7 seconds when LSH scheme is used for noise level 8, because with

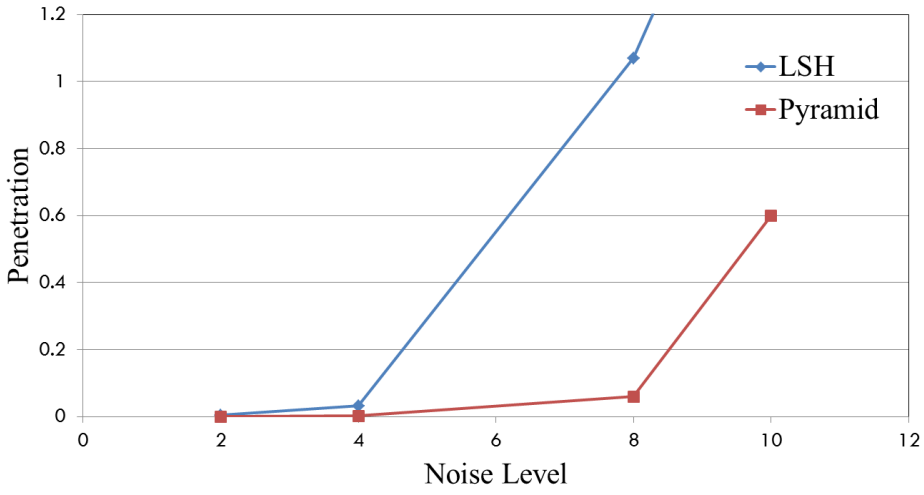

Figure 3.3: Penetration Vs. Noise level

LSH [1] we have to do sequential scan over the columns having hash values. For noise level 8 LSH implementation [1] had 144 columns. For noise level 2 and 4 LSH implementation [1] had 34 and 55 columns respectively, hence search time is less compared to search using Pyramid Tree on 192 columns. Query time for Pyramid Technique increases by small value with increase in noise level, but for LSH it increases by relatively large value. So even if the input fingerprints are worse than noise level 10, time taken by Pyramid Indexing will not increase by large value.

Graph in Figure 3.5 shows the effect of radius of hypersphere on penetration rate for noise level 8. For noise level 8, accuracy is 100% when 85 is used for radius. In the graph penetration increases by large value after 85.

If we know range and radius, computed empirically, then Pyramid Indexing Technique can be used efficiently for reducing penetration and running time.

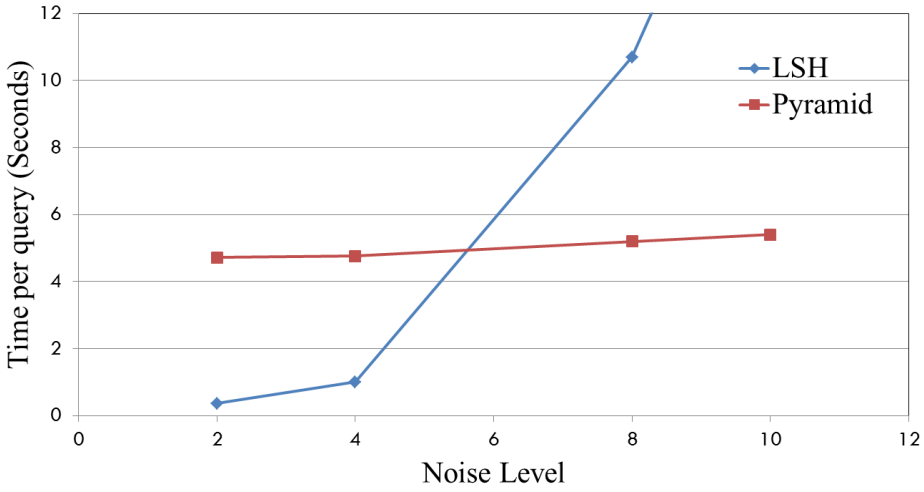

Figure 3.4: Retrieval Time Vs. Noise level

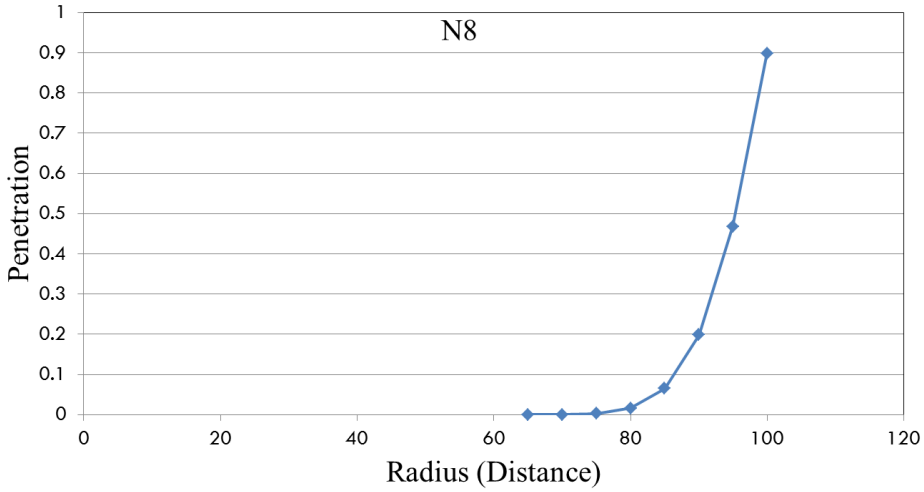

Figure 3.5: Penetration Vs. Radius

Chapter 4

Minutiae Triplets

Another promising approach to reduce the number of candidate fingerprints for matching is given in [4] [9]. In this approach, for each triangle formed by three non collinear minutiae of fingerprint, some features are extracted and these features are used to index fingerprints. There are various types of minutiae, among these ridge ending and ridge bifurcation are the most commonly used.

4.1 Minutiae Extraction

Procedure to extract minutia from Fingerprint has following steps and is depicted in Figure 4.1.

1. **Image Enhancement:** Success of minutiae detection algorithm depends on the quality of input fingerprint image. To minimize the number of false minutia, fingerprint image needs to be enhanced to suppress the noise and enhance the ridge-valley structure. I implemented the enhancement technique using Gabor filter [7].

2. **Binarization:** Gray-scale image is converted into binary image. Image is binarized by thresholding method. I have implemented local adaptive thresholding method where image is divided into blocks of size $16 * 16$ and mean intensity value is computed for the block and this value is used as a threshold for that block.

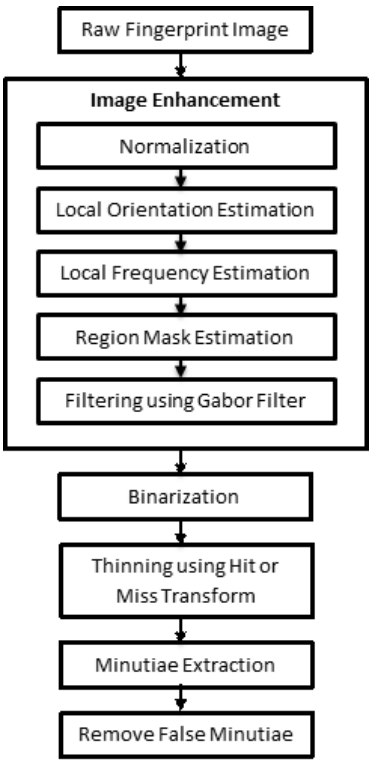

Figure 4.1: Minutiae extraction procedure

3. **Thinning:** This operation reduces the width of each ridge to a single pixel. I implemented Hit or Miss transform [8]. Hit or Miss transform uses prespecified templates of size 3×3 pixels and removes pixel at the center of 3×3 block if matched with any of the templates giving thinned ridges.

4. **Minutiae Detection:** For each black pixel count the number of black pixels in it's 3×3 neighborhood and also the number of black to white transitions in clock-wise direction. Pixel is ridge end if both counts are 1. Pixel is ridge bifurcation if both counts are 3.

5. **Removing Spurious Minutiae:** Even after enhancement some spurious minutiae exist. Common spurious minutiae are shown in Figure 4.2. Following rules are applied to remove such false minutiae:

   - If distance between two minutiae is less than a threshold and difference between ridge orientations at these points is less then both minutiae are false.

Threshold for distance is the average inter-ridge width. This will remove minutiae produced due to ridge breaks, multiple ridge breaks, merge, bridge.

- For end point follow the thinned ridge and check if 8^{th} point is present or not. If not then mark it as false. Remove pixels which are visited during traversal. Also if pixels visited during traversal has more than two black pixel in its 3×3 neighborhood then mark this minutia as false minutia. This will remove minutiae produced because of spur, island.
- For bifurcation point follow each of the 3 thinned ridges and check if they have 8^{th} point. Remove pixels which are visited during traversal. If one of them does not have 8^{th} point then mark this minutia as false minutia. Also if pixels visited during traversal has more than two black pixel in its 3×3 neighborhood then mark this minutia as false minutia. This will remove minutiae produced because of ladder, lake.

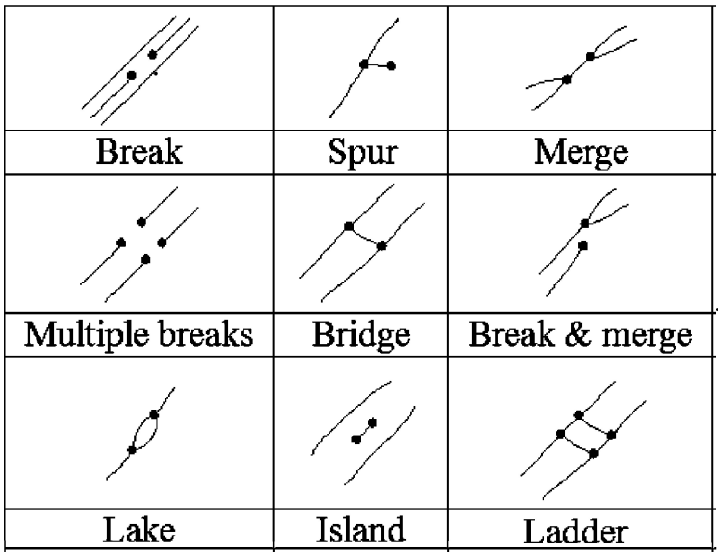

Figure 4.2: Examples of spurious minutiae (black dots)

4.2 Indexing Elements for Minutiae Triplets

Following features are computed from the triangle formed by each non-collinear triplets of minutiae to form index. Features and their definitions are same as [4] and [9].

1. $\mathbf{M_i}$ is minutiae detail for a vertex P_i of triangle. If vertex is bifurcation then, depending on the quadrants of two new ridges with respective to bifurcated ridge, minutia detail is identified as per Table 4.1. In Figure 4.3, r is the ridge before bifurcation and r_1 and r_2 are two ridges after bifurcation. From angles between the lines joining p and 8^{th} points along these ridges(ap, b_1p, b_2p), we can determine which is r, r_1 and r_2. Table 4.1 and Figure 4.3 are taken from [9].
   For ridge end point, quadrant of line joining end point and 8^{th} point with respect to line joining end point and centroid of the triangle is used as minutia detail.

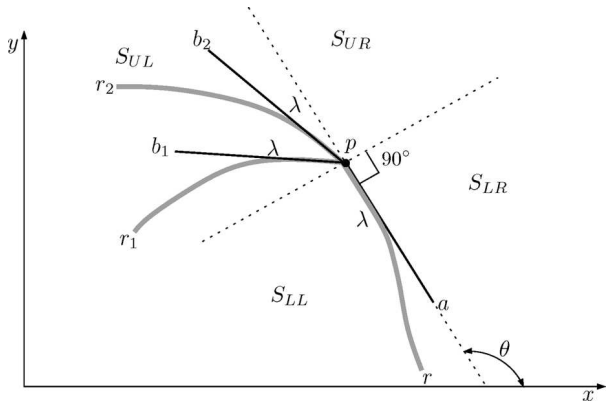

Figure 4.3: Minutia detail of bifurcation at p

2. $\alpha_{\mathbf{min}}$ and $\alpha_{\mathbf{med}}$ are the minimum and medium angles in the triangle. According to the magnitude of angles, the vertices of angles α_{max}, α_{med} and α_{min} are labeled as P_1, P_2 and P_3 respectively.

3. $\mathbf{H}$ is the triangle Handedness. Let $Z_i = x_i + y_i$ be a complex number corresponding to location (x_i, y_i) of point P_i. Define $Z_{21} = Z_2 - Z_1$, $Z_{32} = Z_3 - Z_2$ and $Z_{13} = Z_1 - Z_3$. Triangle Handedness is $H = sign(Z_{21} \times Z_{32})$, where $\times$ is the cross product.

4. $\mathbf{L_{max}}$ is the length of longest edge of the triangle.

Case	$b_1 \in$	$b_2 \in$	Condition
1	S_{LL}	S_{LL}	
2 (a)	S_{LL}	S_{UL}	$x[b_1] \leq x[b_2]$
2 (b)	S_{LL}	S_{UL}	$x[b_1] > x[b_2]$
3	S_{LL}	S_{UR}	
4	S_{LL}	S_{LR}	
5	S_{UL}	S_{UL}	
6 (a)	S_{UL}	S_{UR}	$y[b_1] \leq y[b_2]$
6 (b)	S_{LL}	S_{UR}	$y[b_1] > y[b_2]$
7	S_{UL}	S_{LR}	
8	S_{UR}	S_{UR}	
9 (a)	S_{UR}	S_{LR}	$x[b_1] \leq x[b_2]$
9 (b)	S_{UR}	S_{LR}	$x[b_1] > x[b_2]$
10	S_{LR}	S_{LR}	

Table 4.1: Different cases for bifurcation detail

5. $\phi_{\mathbf{i}}$ is the difference between angles of two edges of P_i and orientation field at P_i.

6. γ is triangle type where $\gamma = \gamma_1 + \gamma_2 + \gamma_3$ and γ_i is type of P_i. If P_i is an end-point then $\gamma_i = 1$ else $\gamma_i = 0$.

Conditions for Triangle Match

Following conditions are used to find the matching triangles among the triangles in database:

$$M_i = M_i^{'}$$
$$H = H^{'}$$
$$\gamma = \gamma^{'}$$
$$|\alpha_{min} - \alpha_{min}^{'}| < T_\alpha$$
$$|\alpha_{med} - \alpha_{med}^{'}| < T_\alpha$$
$$\frac{|L_{max} - L_{max}^{'}|}{L_{max}} < T_L$$
$$|\phi_i - \phi_i^{'}| < T_\phi$$

where T_α, T_L and T_ϕ are the thresholds and $i = 1, 2, 3$.

4.3 Low Order Delaunay Triangle

There will be $O(n^3)$ triangles if we take all possible triplets of minutiae set of size n. This will increase the computation and storage costs. Instead of exhausting all triplets of the minutiae set, [9] [2] used Delaunay triangulation. For a set 'P' of points in a plane, Delaunay triangulation is a triangulation such that no point in 'P' is inside the circumcircle of any triangle in triangulation set DT(P). Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation. As per [9], I used Low Order Delaunay Triangles which is a union of Delaunay triangles and order 1 Delaunay triangles. For each possible convex quadrilateral formed by a pair of Delaunay triangles, order 1 Delaunay triangles are obtained by edge flipping of shared edge of two Delaunay triangles and making sure that both new triangles have only one point inside their circumcircle.

Complexity of the implemented algorithm to construct Delaunay triangulation is $O(nlogn)$ where n is the number of minutiae and 1 order Delaunay triangles are computed in linear time. Low Order Delaunay triangulation has advantages: 1) Effect of insertion or deletion of a point is local. 2) Creates only $O(n)$ triangles instead of $O(n^3)$.

4.4 Indexing Score

Suppose F is a query fingerprint image and F_i are fingerprints in database, $i = 1, 2, ...D$ where D is the number of fingerprint images in database. Let M and M_i be set of minutiae in F and F_i respectively. Suppose there are n potential corresponding minutia m_j in a pair of fingerprints F and F_i, $m_j \in M \bigcap M_i$ and $j = 1, 2, ...n$. Let r_j be the number of matched triangles which includes m_j. Indexing score for F_i is calculated as $S_i = \sum_{j=0}^{n} r_j$. The indexing algorithm is as Algorithm 2.

Algorithm 2 Minutiae Triplet Indexing

Input: Minutiae set M of input fingerprint F
Output: Top N possible matching fingerprints

```
1  Compute set of Low Order Delaunay Triangles DT(M) from M
2  for all triangle T in DT(M) do
3    Extract features (M_j, H, α_min, α_med, L_max, φ_i, γ) of triangle T
4    Get the matching triangles from the database using conditions in Section 4.2
5    for all fingerprints F_j having matching triangle do
6      if F_j is in candidate fingerprint set U then
7        Increment counter CT_j of F_j
8      else
9        Add F_j into U
10     end if
11   end for
12 end for
13 for all F_j in U do
14   if CT_j < T_CT then
15     Remove F_j from U, where T_CT is a threshold.
16   end if
17 end for
18 if candidate fingerprint list is empty
19   Reject the input fingerprint
20 end if
21
22 Compute indexing score S_j based on M_j for all F_j in U
23 Sort S_1, ...., S_u in descending order
24 return Top N matching fingerprints
```

4.5 Experimental Evaluation

Average number of triplets per fingerprint is 112. Hence used 0.1 Million fingerprints from Pyramid Indexing Experiments. Extracted features from all triplets of fingerprints. Features of one triplet are stored as one tuple $(id, H, \gamma, M_1, M_2, M_3, \phi_1, \phi_2, \phi_3, \alpha_{min}, \alpha_{med}, L_{max})$ in database where id is an identification number of the fingerprint. So resulting database has 11.2 Million tuples. Used 4 sets of query fingerprints of noise levels 2, 4, 8 and 10. Each set contains 500 fingerprints of respective noise level.

I implemented a simple B+-tree indexing technique using GiST in PostgreSQL. This B+-tree is same as multi-column B+-tree index of PostgreSQL except leaf node contains all the feature values so that we can verify whether it is within the thresholds of queried

Algorithm 3 Query on GiST B+-Tree

Input:
$Q(0,\ 0,\ 0,\ 0,\ 0,\ T_\phi,\ T_\phi,\ T_\phi,\ T_\alpha,\ T_\alpha,\ T_L, H^i, \gamma^i, M_1^i, M_2^i, M_3^i, \phi_1^i, \phi_2^i, \phi_3^i, \alpha_{min}^i, \alpha_{med}^i, L_{max}^i)$
$i = 1, ..., n$ where n is the number of low order Delaunay triangles of query fingerprint.

$N(H^{min}, \gamma^{min}, M_1^{min}, M_2^{min}, M_3^{min}, \phi_1^{min}, \phi_2^{min}, \phi_3^{min},$
$H^{max}, \gamma^{max}, M_1^{max}, M_2^{max}, M_3^{max}, \phi_1^{max}, \phi_2^{max}, \phi_3^{max},$
$[H, \gamma, M_1, M_2, M_3, \phi_1, \phi_2, \phi_3, \alpha_{min}, \alpha_{med}, L_{max}])$

```
1  if N is Non−Leaf then
2    for all A(H, γ, M_j, φ_j) in Q j = 1, 2, 3 do
3      if A is inside N then
4        return true
5      end if
6    end for
7    return false
8  end if
9  if N is Leaf then
10   for all A(H, γ, M_j, φ_j, α_min, α_med, L_max) in Q; j = 1, 2, 3 do
11     B(H, γ, M_j, φ_j, α_min, α_med, L_max) of N
12     C(0, 0, 0, 0, 0, T_φ, T_φ, T_φ, T_α, T_α, T_L) of Q
13     if abs(A − B) ≤ C then
14       return true
15     end if
16   end for
17   return false
18 end if
```

feature values. Also we can do bulk matching for all the triplet features from query fingerprint. Created a B+-tree index on $(H, \gamma, M_1, M_2, M_3, \phi_1, \phi_2, \phi_3)$. Thresholds for H, γ, M_1, M_2 and M_3 are 0. So filtering with these features will reduce the number of nodes to be traversed by large extent. As database size is large, even with initial filtering there will be large number of nodes. These nodes are again filtered using ϕ_1, ϕ_2, ϕ_3 with threshold T_ϕ. ϕ_1, ϕ_2, ϕ_3 have values between -360 to +360. In the query user gives thresholds for each feature followed by list of feature values from all triplets of minutiae of query fingerprint. Algorithm 3 describes handling of query at each node N of B+-tree.

Table 4.2 compares FingerCode and Minutiae Triplets with respect to storage space usage.

	FingerCode	Minutiae Triplets
Number of Tuples	0.1 million	11.2 million
Table Size(MB)	156	1177
Index Size(MB)	269	2472

Table 4.2: Storage space usage of FingerCode and Minutiae Triplets

Distance is invariant under translation and rotation. However minutiae locations are uncertain as they depend on enhancement and thinning algorithms. Hence the location of each vertex of triplet changes independently in small local area in random manner. Angle is ratio of distances, so angles α_{med}, α_{min} and ϕ_i change by very small value. Thresholds used for experiments: $T_\alpha = 5^o$, $T_L = 10\%\, pixels$ and $T_\phi = 5^o$.

The output of the indexing algorithm is a set of top N fingerprints. If the fingerprint corresponding to the query fingerprint is in the list of top N fingerprints, then it considered as correct result.

With the above thresholds, accuracies of N2, N4, N8 and N10 are 100% when top 1 fingerprint is considered. As the noise level increases some minutiae will get added or removed. This affects the triangulation and can be seen from the graph in Figure 4.4. As the noise level increases the average and minimum percentage of triplets matched between a query fingerprint impression and corresponding fingerprint impression in database decreases because of the affect of change in minutiae set on triangulation. Also less than 8% triplets are matched between a query fingerprint impression and impression of different fingerprint in database. So if we use 8% as threshold T_{CT} in indexing Algorithm 2, then algorithm will return only one fingerprint if query fingerprint corresponds to registered fingerprint.

Total time comparison of FingerCode and Minutiae Triplet for 0.1 million data fingerprint and single query fingerprint of N8 is given in Table 4.3 (only one CPU core and one disk is used). Minutiae matching algorithm implemented in [11] takes 0.02 seconds to match a pair of fingerprints. For FingerCode query time is less and Matching time is more compared to Minutiae Triplet. Matching Time for FingerCode can be reduced by increasing the number of CPU cores and doing matching in parallel. As per [5] if

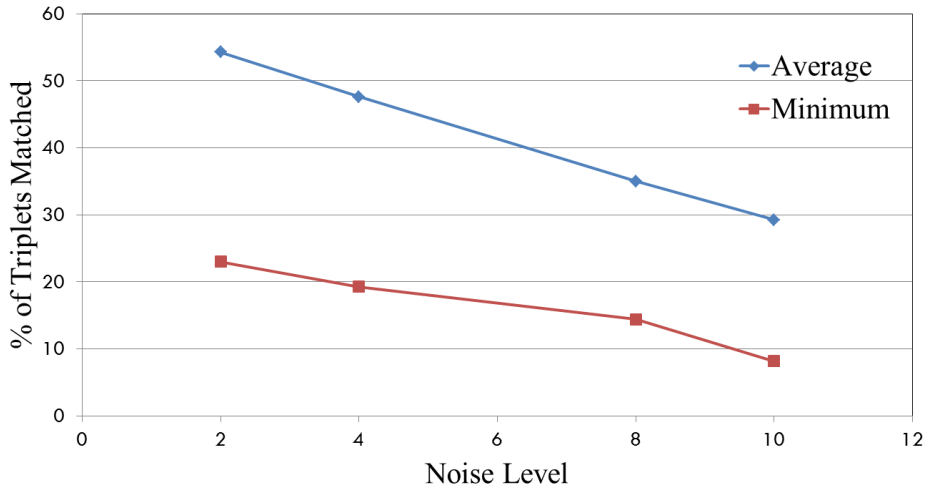

Figure 4.4: Percentage of triplets matched per true pair of fingerprints

we distribute IO operations over large number of disks so that disks parallelism can be used, then we could reduce the time for minutiae triplets by large extent.

Method	**Retrieval Time** (Seconds)	**Matching Time** (Seconds)	**Total Time** (Seconds)
FingerCode	0.6	1.4	2
Minutiae Triplet	3.98	0.02	4

Table 4.3: Total time comparison of FingerCode and Minutiae Triplets

Chapter 5

Conclusions

With Anguli we can generate very large number of fingerprints. Anguli takes 400 milliseconds to generate a fingerprint and 300 milliseconds to add noise of level 8. Time to add noise increases with increase in the noise level.

Pyramid Indexing Technique can be used to index Directional Field or FingerCode feature vectors of fingerprints to reduce the penetration rate as well as search time. For noise level 8 penetration is 0.06% and time is 5.2 seconds which are much less than LSH implementation [1] where penetration was 1.93% and time was 10.7 seconds. For Directional Field and FingerCode methods matching time is more as the penetration is more. Matching time can be reduced by increasing the number of CPU cores. Graph in Figure 5.1 shows that time to retrieve candidate fingerprint increases linearly with increase in the size of database for a query fingerprint of noise level 8. It increases by fraction by which database size is increased, but penetration is constant.

Minutiae Triplet reduces the number of candidate fingerprint for matching to 1 with the accuracy of 100%. But query time is more which can be reduced by increasing the number of disks and distributing the data across these disks such that disk parallelism can be used. One strategy could be distributing the triplets according to type and handedness. Graph in Figure 5.2 shows that time to retrieve candidate fingerprint increases linearly with increase in the size of database for a query fingerprint of noise level 8. It increases by fraction by which database size is increased, but accuracy is 100% when top

1 fingerprint is considered.

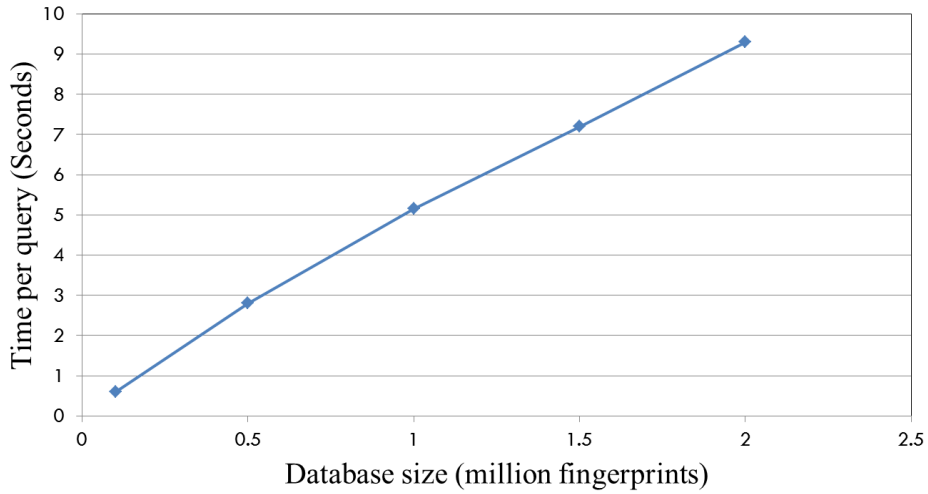

Figure 5.1: Time for one query using pyramid indexing technique on different database sizes

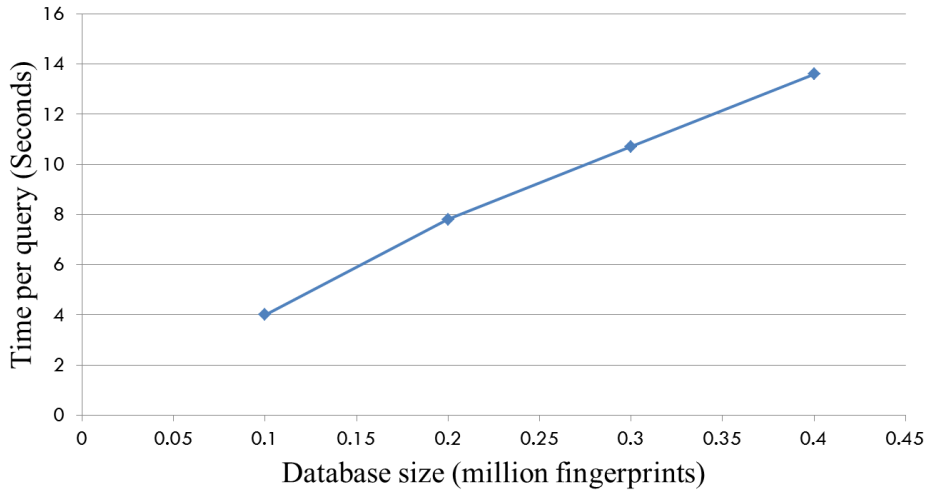

Figure 5.2: Time for one query using minutiae triplets on different database sizes

References

[1] A. H. Ansari, "Generation and storage of large synthetic fingerprint database", *M.E. Thesis*, Jul. 2011.

[2] G. Bebis, T. Deaconu, and M. Georgiopoulos, "Fingerprint identification using delaunay triangulation", *Information Intelligence and Systems*, 1999.

[3] S. Berchtold, C. Böhm, and H. Kriegel, "The pyramid-technique: Towards breaking the curse of dimensionality", *SIGMOD International Conference on Management of Data*, vol. 27, 1998.

[4] B. Bhanu and X. Tan, "Fingerprint indexing based on novel features of minutiae triplets", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 5, May 2003.

[5] R. S. Germain, A. Califano, and S. Colville, "Fingerprint matching using transformation parameter clustering", *IEEE Computational Science and Engineering*, Oct. 1997.

[6] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems", *VLDB*, 1995.

[7] L. Hong, Y. Wan, and A. Jain, "Fingerprint image enhancement: algorithm and performance evaluation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, Aug. 1998.

[8] B. K. Jang and R. T. Chin, "Analysis of thinning algorithms using mathematical morphology", *Pattern Analysis and Machine Intelligence*, vol. 12, Jun. 1990.

[9] X. Liang, A. Bishnu, and T. Asano, "A robust fingerprint indexing scheme using minutia neighborhood structure and low-order delaunay triangles", *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 4, Dec. 2007.

[10] A. Mhatre, S. Chikkerur, and V. Govindaraju, "Indexing biometric databases using pyramid technique", *5th International Conference on Audio- and Video-Based Biometric Person Authentication*, 2005.

[11] K. Wadhwani, "Large-scale fingerprint identification systems", *M.E. Thesis*, Jul. 2011.

Appendix A

Anguli Command Line Options

Option	Description
-h	Displays the help.
-num <number>	Number of fingerprints to be generated. Cannot be used with -impr.
-npd <number>	Number of fingerprints per directory. Cannot be used with -impr. Default value is 1000.
-cdist <distribution>	Fingerprint class Distribution. Class distributions are in Table A.2. Default distribution is natural. Cannot be used with -impr.
-meta	Enables saving of meta information, like class of fingerprint, in a text file of corresponding finger. Default is disabled. Cannot be used with -impr.
-ni <number>	Number of impressions per fingerprints. Default value is 1.
-outdir <path>	Path of directory in which fingerprints and impressions are to be stored. Can be used for generating fingerprints and impressions. Default path is "./Fingerprints" when generating fingerprints and "./Impressions" when "-impr" is used for generating impressions.

continued on next page

continued from previous page	
Option	**Description**
-numT <number>	Number of threads to be created for generating fingerprints and impressions. Default value is 1.
-seed <number>	Seed value for random number generator. Cannot be used with -impr.
-impr	Generate impressions only from a fingerprint database generated previously. Should be used with '-indir' option.
-indir <path>	Path of directory with fingerprint images or subdirectories containing fingerprint images. Default path is "./Fingerprints".
-scratch '<num><num>'	Minimum and maximum number of scratches to be added to impressions. Default value is 0.
-noise '<num><num>'	Minimum and maximum number of noise levels[0, 8] to be applied to impressions. Default value is 0.
-trans <num>	Percentage by which finger are to be translated to generate impressions. Fingerprints are translated in the range of [-num, +num]
-rot <num>	Degree by which fingers are to be rotated to generate impressions. Fingerprints are rotated in the range of [-num, +num].

Table A.1: Options for Anguli command line

natural	Generate fingerprints according to natural distribution of classes.
arch	Generate fingerprints of Arch class only.
tarch	Generate fingerprints of Tarch class only.
right	Generate fingerprints of Right Loop class only.
left	Generate fingerprints of Left Loop class only.
dloop	Generate fingerprints of Double loop class only.
whirl	Generate fingerprints of Whirl class only.

Table A.2: Fingerprint class distribution

Appendix B

Pyramid Technique

The pyramid technique [3] was especially designed for building index of data points in higher-dimensional spaces. Pyramid technique builds a height balanced tree structure which is dynamic to the frequent insertions and deletions.

Pyramid technique maps a d-dimensional point into a one-dimensional space and uses a B+-tree to index the one-dimensional space. It is not possible to get d-dimension point from one-dimensional key, hence d-dimensional point is also stored in data pages of B+-tree. Mapping in pyramid technique is based on a special partitioning strategy. It partitions the d-dimensional space into 2 * d pyramids having the center point of the space as their top and a (d-1) dimensional base. These single pyramids are then cut into slices parallel to the basis of the pyramid forming the data pages. Figure B.1 depicts this partitioning technique. For 2-dimensional case, we have 2 * 2 = 4 pyramids, as shown in the Figure B.1, with each of those pyramids $(P_0, ..., P_3)$ having a common tip-point and a 1-dimensional base. Since we work in 192-dimensional data space, the numbers of pyramids formed are 2 * 192 = 384. Figures B.1 and B.2 are taken from [3].

The Pyramid technique requires normalizing the data values to lie between 0 and 1. Pyramids are numbered in logical manner as shown in Figure B.2(a). A pyramid is numbered i if all the points within the pyramid are farthest from the tip-point along dimension i than any other dimension. If the points have their i^{th} coordinate greater than 0.5, the pyramid is labeled as $(i + d)$. In the Figure B.2, all points within pyramids

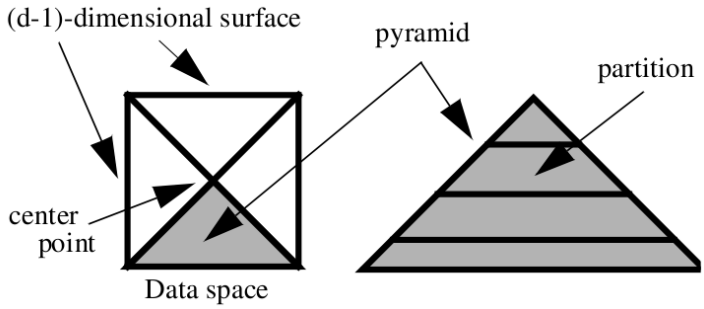

Figure B.1: Partitioning the data space into pyramids

p_1 and p_3 are farthest from the tip point along dimension d_1 than along dimension d_0 and points in p_3 have d_1 values greater than 0.5.

Given a point, the height of the point within its pyramid is the orthogonal distance of the point to the center-point of the data space as shown in Figure B.2(b). The height of a point inside pyramid i is defined as the distance from the tip-point in the (i *mod* d) dimension. For example, all points lying within pyramids p_1 and p_3 have the height defined as the distance from the tip-point along dimension d_1. In order to map a d-dimensional point into a one-dimensional value, the number of the pyramid in which the point is located, and the height of the point within this pyramid are added. This one dimensional value is used as a key in the B+-tree.

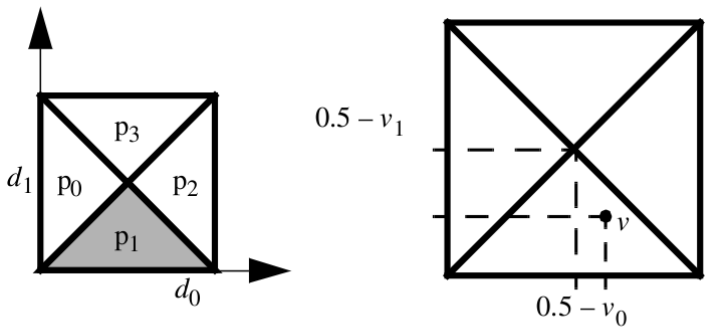

Figure B.2: Properties of pyramids

Appendix C

GiST

Generalized Search Tree (GiST) [6] is an index structure supporting an extensible set of queries and data types. The GiST allows new data types to be indexed in a manner supporting queries natural to the types. In a single data structure, the GiST provides all the basic search tree logic required by a database system, thereby unifying disparate structures such as B+-trees and R-trees in a single piece of code, and opening the application of search trees to general extensibility.

A GiST is a balanced tree of variable fanout between kM and M, $\frac{2}{M} \leq k \geq \frac{1}{2}$ with the exception of the root node, which may have fanout between 2 and M. The constant k is termed the minimum fill factor of the tree. Leaf nodes contain (p, ptr) pairs, where ptr is the identifier of some tuple in the database and p is a predicate that is used as a search key (p is true when instantiated with the values from the indicated tuple). Non-leaf nodes contain (p, ptr) pairs, where ptr is a pointer to another tree node and p is a predicate used as a search key (p is true when instantiated with the values of any tuple reachable from ptr). Predicates can contain any number of free variables, as long as any single tuple referenced by the leaves of the tree can instantiate all the variables. All leaves of the tree appear on the same level.

Key Methods

The keys of a GiST are arbitrary predicates and they come from a user-implemented objectclass, which provides a particular set of methods required by the GiST. Examples of key structures include ranges of integers for data from Z (as in B+-trees). The key class is open to redefinition by the user, with the following set of six methods required by the GiST (taken from [6]):

1. Compress (E): given an entry $E = (p, ptr)$ returns an entry (π, ptr) where π is a compressed representation of p.

2. Penalty (E_1, E_2): given two entries $E_1 = (p_1, ptr_1), E_2 = (p_2, ptr_2)$, returns a domain-specific penalty for inserting E_2 into the sub-tree rooted at E_1. This is used to aid the Split and Insert algorithms. Typically the penalty metric is some representation of the increase of size from $E_1.p_1$ to Union (E_1, E_2).

3. PickSplit (P): given a set P of $M + 1$ entries (p, ptr), splits P into two sets of entries P_1, P_2, each of size at least kM. The choice of the minimum fill factor for a tree is controlled here. Typically, it is desirable to split in such a way as to minimize some badness metric akin to a multi-way Penalty, but this is left open for the user.

4. Union (P): given a set P of entries $(p_1, ptr_1), ..., (p_n, ptr_n)$, returns some predicate r that holds for all tuples stored below ptr_1 through ptr_n. This can be done by finding an r such that $(p_1 \vee ... \vee p_n) \rightarrow r$.

5. Decompress (E): given a compressed representation $E = (\pi, ptr)$, where $\pi = Compress(p)$, returns an entry (r, ptr) such that $p \rightarrow r$. Note that this is a potentially "lossy" compression, since we do not require that $p \leftrightarrow r$.

6. Consistent (E, q): given an entry $E = (p, ptr)$, and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise. Note that an

accurate test for satisfiability is not required here: Consistent may return true incorrectly without affecting the correctness of the tree algorithms. The penalty for such errors is in performance, since they may result in exploration of irrelevant sub-trees during search.

The above are the only methods a GiST user needs to supply.