MTR 94B0000021V4

MITRE TECHNICAL REPORT

NCIC 2000 Image Compression Algorithms Volume IV: Flat Live-Scan Searchprint Compression

April 1994

David J. Braunegg
Eric J. Donaldson
Richard D. Forkert
Margaret A. Lepley
Sherry L. Olson

MITRE
Bedford, Massachusetts

MTR 94B0000021V4

MITRE TECHNICAL REPORT

NCIC 2000 Image Compression Algorithms Volume IV: Flat Live-Scan Searchprint Compression

April 1994

David J. Braunegg
Eric J. Donaldson
Richard D. Forkert
Margaret A. Lepley
Sherry L. Olson

Sponsor: FBI
Dept. No.: G034

Contract No.: J-FBI-93-039
Project No.: 3469M

MITRE

Bedford, Massachusetts

ABSTRACT

Under the National Crime Information Center (NCIC) 2000 program, there is a need to compress, transmit, and decompress flat live-scan, single finger searchprints. Due to the limited bandwidth of police radios and the need for responsive transmission times, the compressed file size goal on average for flat live-scan searchprints is 20,000 bits. In addition to meeting the high rate of compression needed, the final decompressed fingerprint representation must maintain a high degree of ridge positional accuracy, such as minutiae points and relative ridge locations, for matching. This report presents an algorithmic process for compressing and decompressing flat live-scan searchprints. The compression algorithms were developed to remove all extraneous information from the fingerprint, thin the ridges to a single pixel width, and mathematically encode the ridge information. The decompression algorithm reverses this process to reconstruct the thinned ridge representation of the fingerprint.

TABLE OF CONTENTS

SECTION		PAGE

LIST OF FIGURES

FIGURE PAGE

FIGURE		PAGE

FIGURE **PAGE**

LIST OF TABLES

SECTION 1

INTRODUCTION

The National Crime Information Center (NCIC) maintains a national database that includes information about wanted persons, missing persons, and identifiable stolen property. As part of the NCIC 2000 program, law enforcement officers in police cars will be able to access this database through their patrol car radio network. This will assist officers in verifying if a detainee is a wanted or missing person, or to identify a stolen item. To assist in this process, images maintained in the NCIC database can be transmitted to the patrol car. In addition, the officer in the car can transmit a detainee's fingerprint to the NCIC headquarters in Washington, D.C. for processing and positive identification. The large amount of data contained in these images and the limited data transmission capacity of police radio networks necessitates a substantial level of image compression to make this new capability responsive to law enforcement needs without adversely impacting critical radio communication.

This report describes compression and decompression algorithms developed for flat live-scan searchprints to meet the requirements of the NCIC 2000 program.

1.1 BACKGROUND INFORMATION

A substantial level of image compression is required to prepare flat live-scan searchprints for transmission under the NCIC 2000 program. The requirements for a high rate of compression and for an accurate representation of certain fingerprint information led to the development of compression and decompression algorithms designed specifically for this type of fingerprint. This section provides background information on flat live-scan searchprints, as well as size and accuracy requirements for the compression/decompression algorithms.

1.1.1 Flat Live-Scan Searchprints

Searchprints are the fingerprints of unidentified or suspect individuals, which are used for identification purposes. Minutiae points (ridge endpoints and bifurcations), and possibly other information, are extracted from the searchprint and automatically compared to the same types of information extracted from fileprints contained in a central database. If a significant amount of information from the searchprint matches the fileprint, a match is declared. In any case, the requestor is informed of the comparison results.

Two types of searchprints are utilized in the NCIC 2000 system: flat live-scan searchprints and non-live-scan (rolled, inked) searchprints. Non-live-scan searchprints are generated using standard methods with ink and paper. By contrast, flat live-scan searchprints will be obtained from a scanning device either in a patrol car or at a user workstation. The individual to be printed will place his or her right index finger on the scanning surface of the device, and a beam of light will be passed over the pressed finger to provide detailed friction ridge and valley information. The digital gray-scale image generated by this live-scan device is the searchprint that will be discussed in this document. Another document describes processing of non-live-scan searchprints.

1.1.2 Original Image Characteristics

The flat live-scan searchprints used to develop and test the compression algorithms described in this document were simulated from hardcopy examples provided by the FBI. The searchprints of the right index finger of 50 individuals were provided on 10-print cards, each card containing one high-quality imprint of a laser-scanned image. The searchprint was digitized from the card using either an Eikonix camera or a Truvel scanner at 500 dpi. The area scanned was 0.88 inches by 1.2 inches, simulating the area that might be obtained from a flat live-scan device. Gray-scale images were obtained with 256 shades of gray (eight bits). Each searchprint file contained 270,000 bytes, or 2,160,000 bits, of image data.

1.1.3 Compressed Data Goals

Due to the limited bandwidth of police radios and the need for responsive transmission times, the NCIC 2000 goal for the compressed bit stream of searchprint data was determined to be 20,000 bits on average [1]. Not only must a high rate of compression be achieved to reach this goal, but the final fingerprint representation must also maintain a high degree of accuracy for matching. That is, the compressed, decompressed, and reconstructed fingerprint must correctly maintain ridge positional data, such as minutiae points and relative ridge locations.

1.1.4 Reconstructed Image Characteristics

In order to achieve the levels of compression needed to reduce 270,000 bytes of data to 20,000 bits, a series of steps is performed to remove any extraneous information from the searchprint and reduce the data to only essential elements. The resulting reconstructed image is actually a two-valued representation of the searchprint with all important positional information preserved. The ridges are represented by single pixel-width curves that retain the general shape of the ridges and preserve minutiae locations.

1.2 OVERVIEW OF COMPRESSION/DECOMPRESSION ALGORITHMS

The process developed to compress and decompress flat live-scan searchprints, described in this report, actually consists of a suite of algorithms encompassing several stages of processing. Each step in the suite of algorithms is essential in preparing the data for the next processing step. Figure 1 illustrates pictorially the suite of algorithms, and figures 2 and 3 show flowcharts of the processes involved. Detailed descriptions of each of these algorithms, as well as pseudocode, are provided in the remaining sections of this document.

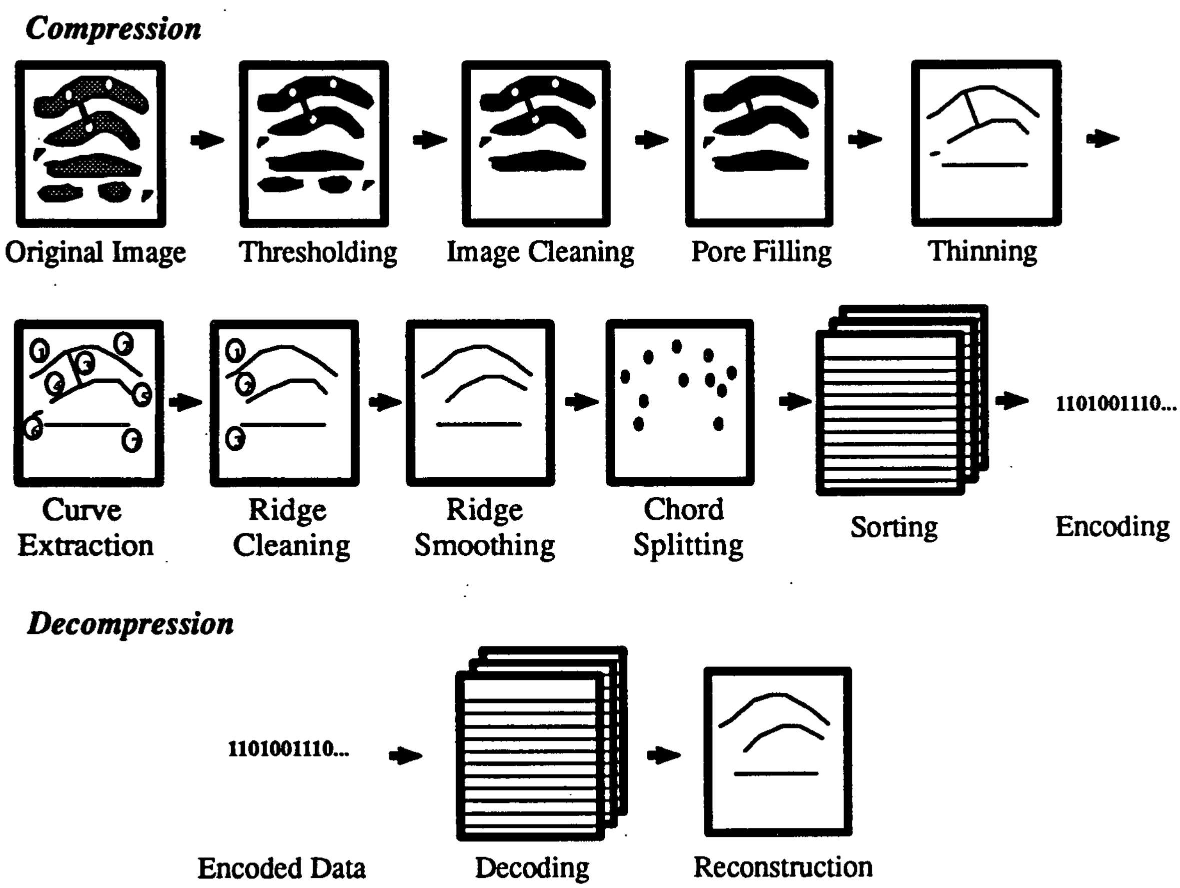


Figure 1. Compression/Decompression Algorithm Diagram

1.2.1 Compression Algorithms

The following paragraphs briefly describe each stage in the compression process. It is important to note that a gray-scale searchprint image is the input to the first stage of processing, and a bit stream is the final output. In the operational system, the final bit stream produced by compression will be transmitted and then decompressed upon receipt.

BHO Binarization: The original gray-scale image is reduced to a two-valued image using a modified version of the Home Office Automatic Fingerprint Recognition System (HOAFRS) Encoder (see Appendix A).

Image Cleaning: Only enough of the ridge area below the flexion crease is retained for context, the rest of this area is removed. Then, one pixel-wide ridge spurs, which are artifacts of the thresholding, are removed.

Pore Filling: Sweat pores are eliminated from the processed fingerprint image in two steps. First, small pores are eliminated based on their sizes. Then, certain large pores are detected by comparing them to the surrounding ridge and eliminated.

Ridge Thinning: The thresholded and cleaned ridges in the searchprints are thinned using a chamfering technique. This technique calculates the distance of every pixel in the ridge to the nearest edge of the ridge, and only retains pixels whose distances indicate that they are along the center of a ridge. This produces a thinned, single pixel-width representation of each ridge.

Curve Extraction: Ridges are detected by scanning a thinned fingerprint image. Each detected ridge is followed in both directions until it terminates or bifurcates. The halves are then combined into a single ridge curve and the bifurcations (if any) are also followed to create additional ridge curves.

Ridge Cleaning: Ridge disconnects that are less than a specified size are reconnected, and the majority of small, thin connections between ridges are removed.

Ridge Smoothing: The ridge curves are smoothed to remove unnecessary noise.

Chord Splitting: This process selects the fingerprint ridge points that will be used as control points by the B-spline algorithm to reconstruct the ridge. A residue, or error, input parameter determines the largest error from the original curve that the user is willing to allow upon reconstruction.

Sorting: Spline point curves are sorted to reduce the intercurve distances and to arrange the curves efficiently for encoding.

Data Encoding: Sorted curve points are encoded using differential encoding and several other encoding strategies to produce the compressed data.

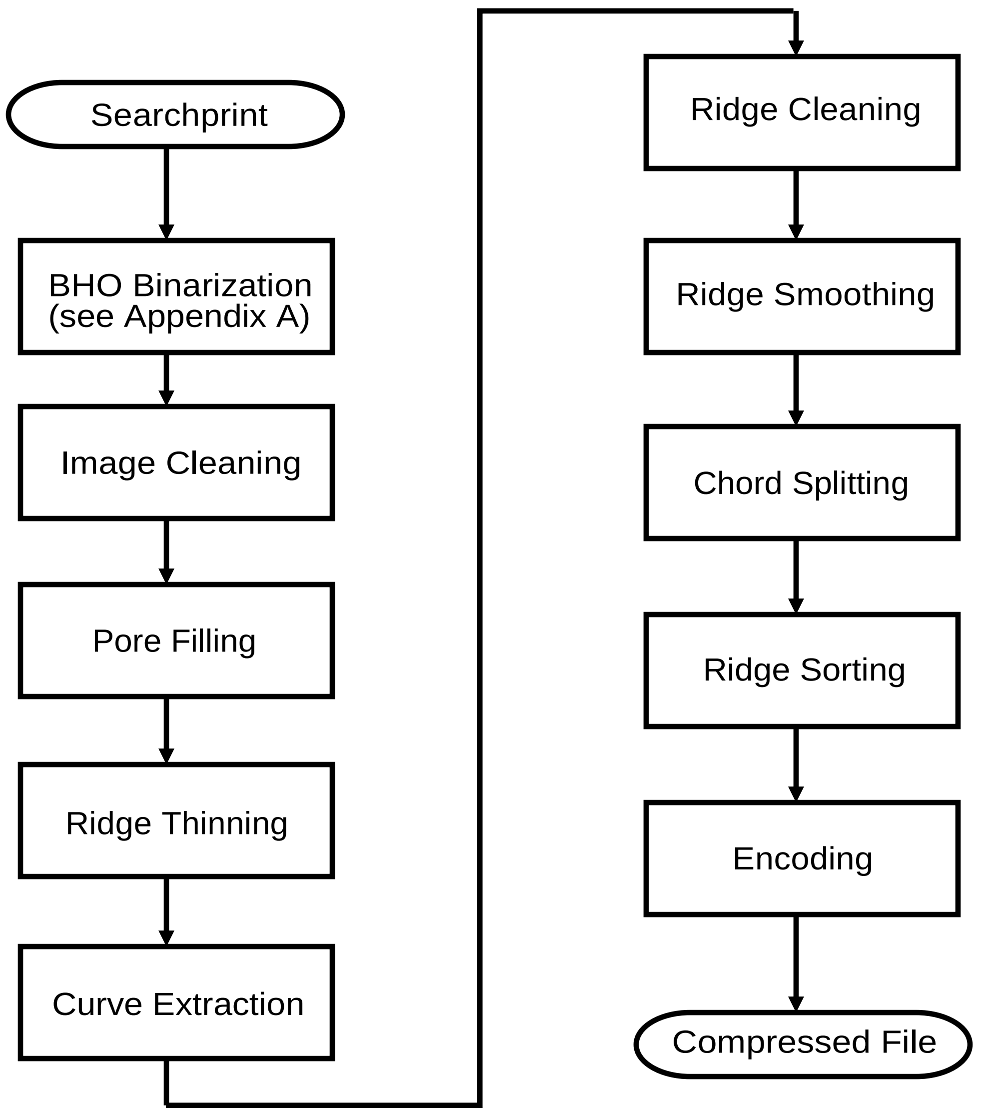


Figure 2. Compression Algorithm Flowchart

1.2.2 Decompression Algorithms

The transmitted bit stream is received and processed by the decompression algorithms. The bit stream representing the searchprint data is the input to the decompression algorithms and a reconstructed two-valued image is the final output.

Data Decoding: After transmission, decoding interprets and regenerates the spline points from the compressed data.

Ridge Reconstruction: B-splines are used to reconstruct the thinned ridges from the decoded control points. This process consists of a standard technique that constructs a smooth curve through a sequence of points.

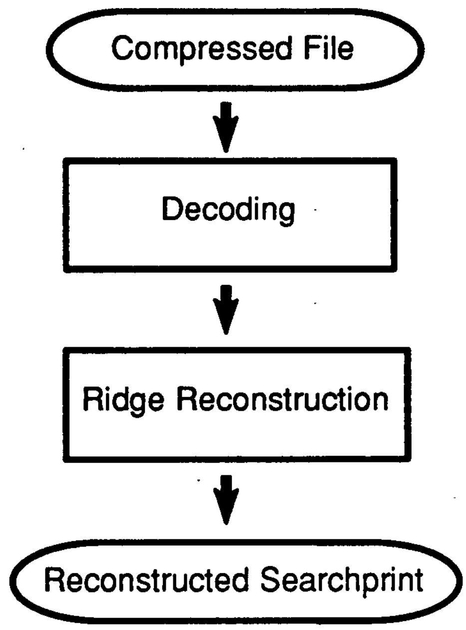


Figure 3. Decompression Algorithm Flowchart

1.2.3 Algorithm Tuning

Due to the high rate of compression, information in the original gray-scale flat live-scan images is lost in the compression/decompression process. However, at each stage of processing, the developers evaluated the information lost and modified the algorithms, if necessary, to prevent the removal of any critical information. In addition, since details of the matching algorithm were not known at the time of development, a very conservative approach was taken in each stage of processing to ensure compatibility with the final matching algorithm. Although the values of various input parameters to the routines were set during testing to reflect this conservative approach, these parameters may be changed to reflect a more liberal or an even more conservative approach when additional information about the matching algorithms and system operational characteristics becomes available. Descriptions and pseudocode in the following sections clearly indicate these input parameters.

1.3 NOTATION AND ASSUMPTIONS

Below are examples and descriptions of the standard notation used in the documents that describe the NCIC 2000 Image Compression Algorithms.

Example	Orthography	Description
P	Times, bold italic, uppercase	array
P(i, j)	Times, italic, uppercase	array element
p	Times, bold italic, lowercase	vector
p(i)	Times, italic, lowercase	vector element
$\boldsymbol{p_b}$	Times, bold italic, lowercase, subscript b	binary (bit) vector
$p_b(i)$	Times, italic, lowercase, subscript b	binary (bit) vector element
BLACK	Times, small caps	constant
MAX-DIST	Helvetica, uppercase	parameter
x	Times, italic, lowercase	variable
if	Times, bold, lowercase	reserved words (keywords)
**	Times, bold	begin comment
Func[<args>] **FUNC[<args>]**	Times, bold, capitalized small caps or Times, bold, normal small caps	defined routine
$x = 4$		"=" denotes assignment (except in conditionals, where "=" denotes an equality test)
(*a*, *b*, *c*)	Parenthesized list of variables	Multiple values returned from a function

1.3.1 Special Notation and Assumptions

The following special notation and assumptions are used throughout this document in addition to those shown above:

(1) All division is floating-point division, unless otherwise noted.
(2) All arrays are assumed to be one-based, i.e., the first row/column is indexed as 1.
(3) The first index into an image array is the row, and the second is the column. The upper left corner of the image array is indexed by (1, 1); row indices increase downward, and column indices increase to the right.
(4) Curve points are described by ordered pairs of the form (x, y). The x-coordinate corresponds to an image column index and the y-coordinate corresponds to an image row index.
(5) Parameters that may be changed to tune the algorithm are denoted in uppercase Helvetica throughout the text and pseudocode. For example, J, U_V, and R are selectable parameters, while j, u_v, and r are variables. The values these parameters were assigned during development are given at the ends of each section.
(6) Mathematical set notation and logical symbols are used throughout the pseudocode. The symbols used follow these conventions:

$\notin$	is not an element of
$\{i : q(i)\}$	set of all i such that $q(i)$ is true
$\lceil r \rceil$	the ceiling function (the closest integer $\geq r$)
$\lfloor r \rfloor$	the floor function (the closest integer $\leq r$)
$\lvert x \rvert$	the absolute value of x
$[a, b] \times [c, d]$	rectangle created by the intersection of two intervals

(7) Common functions, e.g., maximum or cosine, are used in the text and pseudocode without definition. They appear in roman typeface with their common function names, e.g., $\max(x, y, z)$ or $\cos(\theta)$.

SECTION 2

DYNAMIC THRESHOLDING

The dynamic thresholding algorithm described in this section is no longer used to threshold the fingerprint image. Instead, a modified version of the HOAFRS Encoder (BHO binarization) is used. Details of the changes to the HOAFRS Encoder and references are given in Appendix A. **The remainder of this section should be ignored.**

Dynamic thresholding is a process that creates a two-valued fingerprint image from a gray-scale fingerprint image. In general, a thresholding process achieves this by assigning all the pixels above the threshold value to one value and all the pixels below the threshold to the other value. The image characteristics of the output from such a process are totally controlled by the selection of the threshold value. Because gray-scale fingerprint images vary in intensity levels between images and even within the same image, many thresholding strategies had to be considered. Straight thresholding uses the mean value of the entire image as its threshold. This responds to the difference in brightness between different images, but does not respond to the brightness variation across a single image. Dynamic thresholding responds to both of these brightness variations by using the mean value of the pixels in the neighborhood of each pixel as its threshold. Figure 4 compares straight thresholding and dynamic thresholding. On the left is shown a cross section of gray-scale fingerprint ridges that vary from high intensity to low intensity. The thresholds used by the thresholding techniques are shown as lines through these cross sections. The resulting two-valued cross section is shown on the right. Clearly the straight thresholding does not represent the fingerprint ridges as well as the dynamic thresholding which maintains the ridge size and spacing more accurately.

2.1 ALGORITHM DESCRIPTION

For each pixel $I(i,j)$ in the original gray-scale image, I, dynamic thresholding sets the corresponding pixel $T(i,j)$ in the thresholded image T to either BLACK or WHITE. This thresholding process bases the thresholding decision for each pixel on the mean value of the pixels in its neighborhood window (μ_{window}), an absolute upper limit (t_{upper}), and a lower limit (t_{lower}). The value of μ_{window} for each pixel $I(i,j)$ is calculated by finding the mean value of the pixels within the N × N neighborhood window centered on $I(i,j)$. For pixels within (N – 1)/2 pixels of an edge of the image I, the pixel values along the edge are repeated out into the border for the purposes of this calculation. The calculation of μ_{window} is illustrated in figure 5. The absolute upper and lower limits are calculated based on the *overall image* minimum value min_I, mean value μ_I, and maximum value max_I. The upper

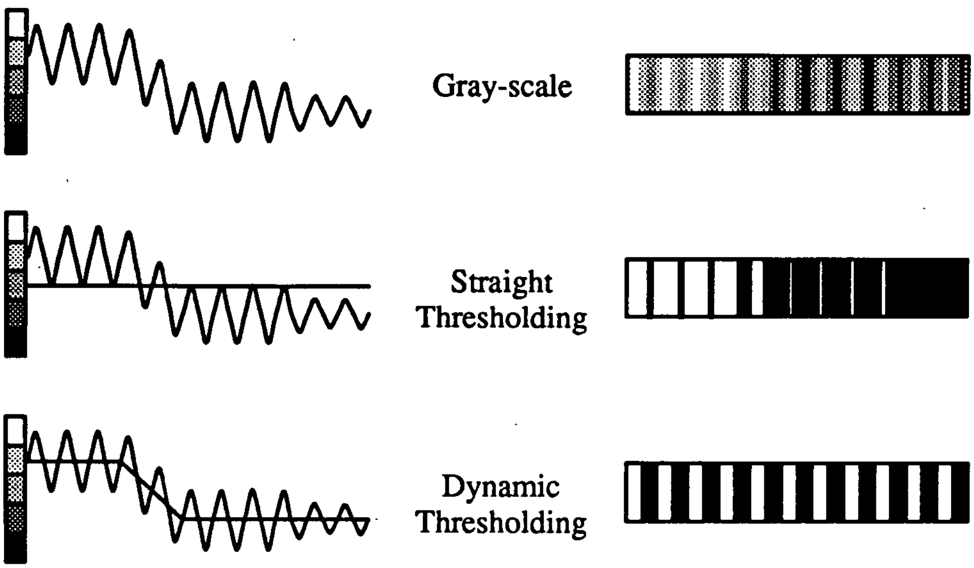


Figure 4. Comparison Between Straight Thresholding and Dynamic Thresholding

limit, t_{upper}, is set equal to $(\mu_I + max_I)/2$. The lower limit, t_{lower}, is set equal to $(\mu_I + 3\ min_I)/4$. These absolute limits prohibit the small variations in the brightness of the white background from being enhanced and also reduces computation for those pixels that are unquestionably black or white.

In processing the pixel (i, j), if $I(i, j)$ is greater than t_{upper}, the corresponding pixel in the thresholded image, $T(i, j)$, is set to WHITE and if $I(i, j)$ is less than t_{lower}, $T(i, j)$ is set to BLACK. Otherwise, $T(i, j)$ is set to WHITE if $I(i, j)$ is greater or equal to μ_{window} and to BLACK if $I(i, j)$ is less than μ_{window}.

2.2 SUMMARY

Parameters

N = 9	Height and width (in pixels) of the pixel neighborhood window

Input

I	Gray-scale fingerprint image (A pixel in ***I*** is referred to as $I(i,j)$.)

Output

T	Thresholded fingerprint image (A pixel in ***T*** is referred to as $T(i,j)$.)

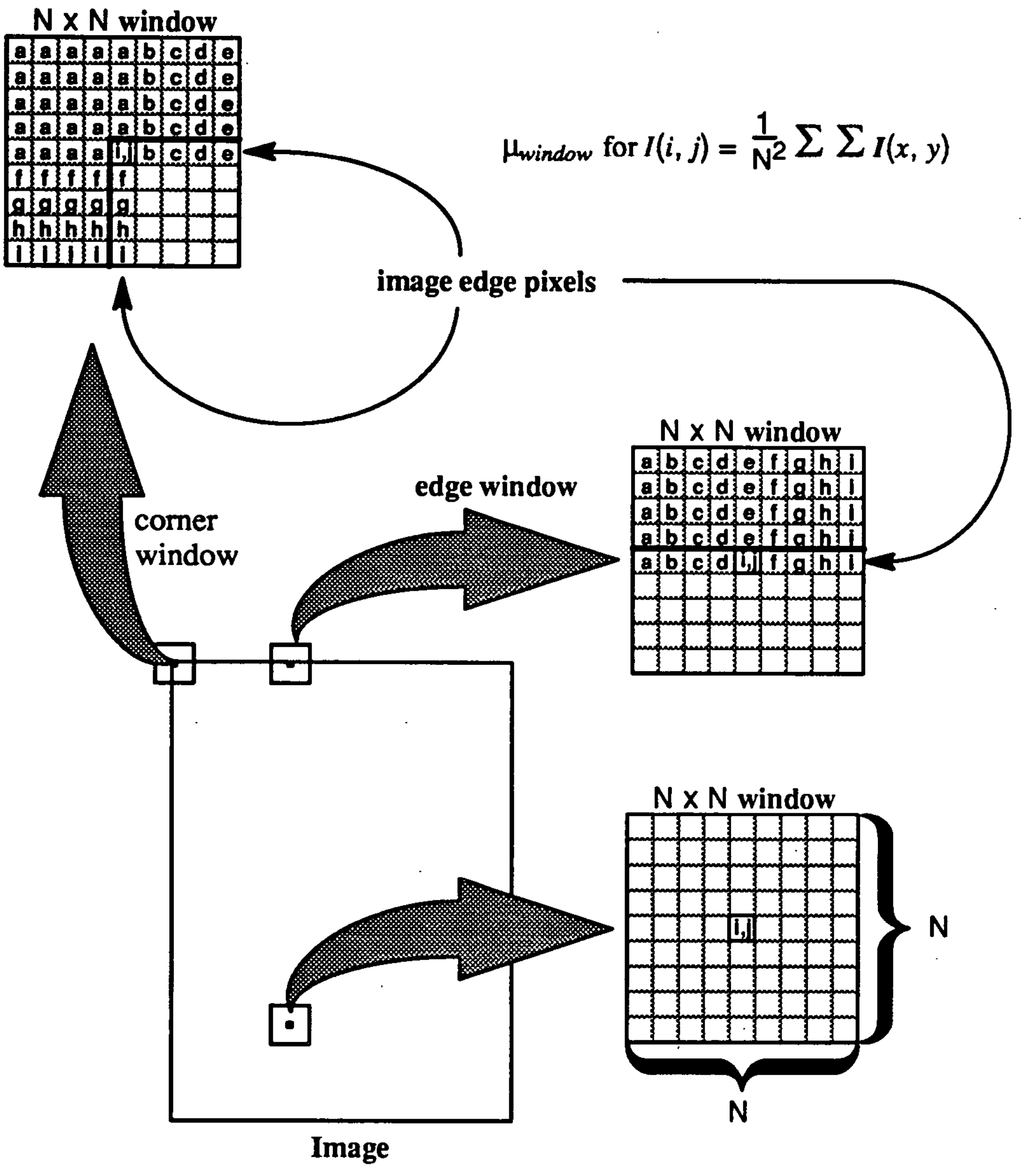


Figure 5. Windows for Calculating a Pixel's Neighborhood Mean Value, μ_{window}

Calculated values

t_{upper}	Absolute upper limit
t_{lower}	Absolute lower limit
μ_{window}	Mean pixel value of a pixel's neighborhood window
max_I	Image overall maximum pixel value
min_I	Image overall minimum pixel value
μ_I	Image overall mean pixel value

DYNAMIC_THRESHOLDING[*I*]

```
** The image I is thresholded to produce image T
min_I = minimum pixel value of I
max_I = maximum pixel value of I
μ_I = mean pixel value of I
t_upper = (max_I + μ_I) / 2
t_lower = (3 min_I + μ_I) / 4
for each pixel (i,j) in I
    if (I(i,j) > t_upper)
        T(i,j) = WHITE
    else if (I(i,j) < t_lower)
        T(i,j) = BLACK
    else
    {
        calculate μ_window for the N × N neighborhood window centered on I(i,j)
        if (I(i,j) < μ_window)
            T(i,j) = BLACK
        else
            T(i,j) = WHITE
    }
return T
```

SECTION 3

THRESHOLDED IMAGE CLEANING

Thresholded image cleaning is composed of a spur removal algorithm that detects and removes single-pixel-thin ridge spurs from the thresholded image. To remove a ridge spur, the spur removal algorithm finds the spur's end and removes the ridge spur until it intersects the ridge. Although crease trimming used to be part of thresholded image cleaning, it is no longer used with BHO binarization, as indicated in the pseudocode below.

```
IMAGE_CLEANING[ I ]
    ** This algorithm modifies I
    ** Crease trimming is not used with BHO binarization
    SPUR_REMOVAL[ I ]                                   ** Modifies I
    return
```

3.1 CREASE TRIMMING

The crease trimming process should not be used after BHO binarization. The remainder of section 3.1 is retained for historical reference, but **it should not be implemented.** Proceed to section 3.2 for a description of the spur removal algorithm.

Crease trimming removes ridges from the thresholded fingerprint image that are a fixed distance, which is a modifiable system parameter, below the flexion crease. The flexion crease in a fingerprint image is a large white area within the impression corresponding to the crease in the skin near the end joint of a finger. The crease trimming algorithm automatically detects this crease and erases all ridges a selectable distance below this crease. This allows the retention of the flexion crease for alignment purposes, while reducing the number of ridges to be encoded. The process first detects the crease as a large white area within the impression of the thresholded fingerprint image. Then the fingerprint is trimmed a fixed distance below the detected crease. This algorithm requires that the fingerprint impression be reasonably centered and large enough to cover the central portion of the fingerprint image.

3.1.1 Algorithm Description

The first step in crease trimming is to detect the fingerprint flexion crease, so that a portion of the fingerprint impression below the crease can be removed. In order to detect the crease in the thresholded fingerprint image, the algorithm must look for a large horizontal white area within the fingerprint impression. Care must be taken not to include the white border surrounding the fingerprint impression, as this would influence the definition of the large white areas within the fingerprint.

The algorithm considers only the bottom half of the thresholded fingerprint image since a crease is not likely to appear in the upper half of the image. As part of detecting the large horizontal white area defining the crease, the algorithm determines the largest vertical run of consecutive white pixels contained within this region for each column in the thresholded fingerprint image. A vertical run is defined to be a set of connected white pixels within a column. Note that a column may contain more than one vertical run. Given a column j, *vertical_run(j)* is defined to be the largest vertical run in the lower half of column j not touching the top or bottom of the lower half of the image. These restrictions prevent the white borders at the top and bottom of the fingerprint impression from being considered. The entire collection of largest vertical runs for all columns in the image is referred to as ***vertical_run***.

Next, the algorithm processes ***vertical_run*** to find the left and right edges of the fingerprint impression. First, it calculates some statistics on the central $S_{VERTICAL\_RUN}$ columns in ***vertical_run***. During development the value of $S_{VERTICAL\_RUN}$ was set to select the central half of the fingerprint image. The statistics are calculated on the *lengths* of the runs in this central section of ***vertical_run*** for each image: the mean ($\mu_{vertical\_run}$), maximum ($max_{vertical\_run}$), and standard deviation ($\sigma_{vertical\_run}$). These values are used to determine the usable columns of the runs data. Starting at the center column of the image and iterating towards the left edge of the image, the algorithm searches for the first *vertical_run* element whose length exceeds the threshold t_{max}, calculated as the maximum plus one standard deviation ($max_{vertical\_run} + \sigma_{vertical\_run}$). If such a column is found, the algorithm iterates from this column toward the right edge of the image, searching for the first column whose *vertical_run* length is less than the threshold t, calculated as the mean plus one standard deviation ($\mu_{vertical\_run} + \sigma_{vertical\_run}$). This column is the left edge of the fingerprint impression, $edge_{left}$. Otherwise, if a column outside the central $S_{VERTICAL\_RUN}$ columns is found whose *vertical_run* length is zero, then the following column is $edge_{left}$. This removes the border around the left side of fingerprint from consideration.

The algorithm then performs a similar process on the right side of the thresholded fingerprint image to find the right border. Starting at the center column of the image and iterating towards the right edge of the image, the algorithm searches for the first *vertical_run* whose length exceeds t_{max}. If such a column is found, the algorithm iterates from this column toward the left edge of the image, searching for the first column whose *vertical_run* length is less than t. This column is the right edge of the fingerprint impression, $edge_{right}$. Otherwise, if a column outside the central $S_{VERTICAL\_RUN}$ columns is found whose *vertical_run* length is zero, then the preceding column is $edge_{right}$. Figure 6 illustrates the determination of ***vertical_run***, $edge_{left}$, and $edge_{right}$.

Now that the left and right edges of the actual fingerprint impression have been determined, the algorithm finds the largest horizontal run of white pixels for each row in the

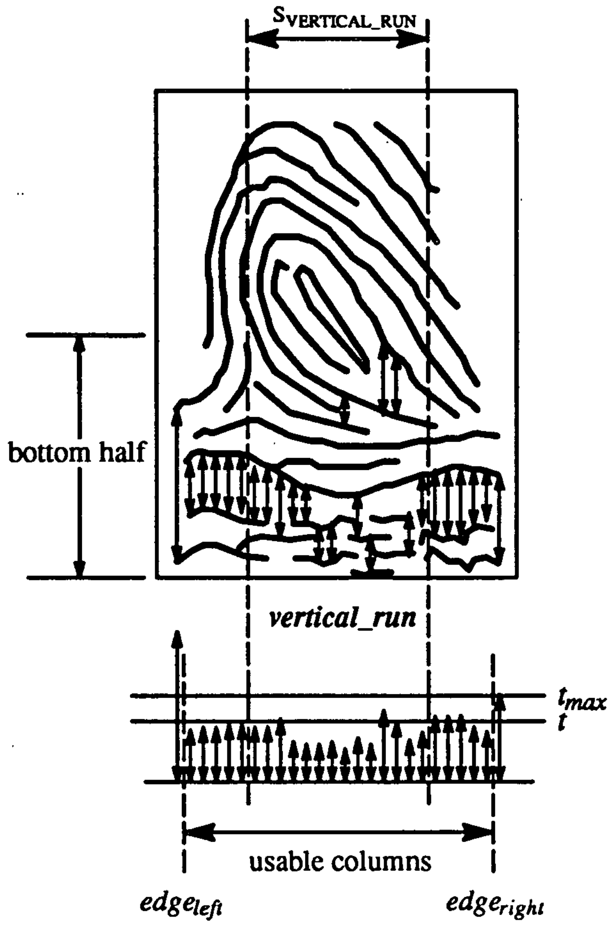


Figure 6. Determination of the Largest Vertical Runs and the Edges of the Fingerprint Impression

region under consideration. A horizontal run is defined to be a set of connected white pixels within a row. Note that a row may contain more than one horizontal run. Given a row i, *horizontal_run*(i) is defined to be the largest horizontal run in row i between $edge_{left}$ and $edge_{right}$. Note that *horizontal_run*(i) may contain a pixel from either column $edge_{left}$ or $edge_{right}$; the horizontal run simply can not extend past these limits. The entire collection of largest horizontal runs for all rows in the image is referred to as ***horizontal_run***. The determination of ***horizontal_run*** is illustrated in figure 7. At this point, the largest horizontal and vertical runs of consecutive white pixels within the fingerprint impression for each column and row have been determined.

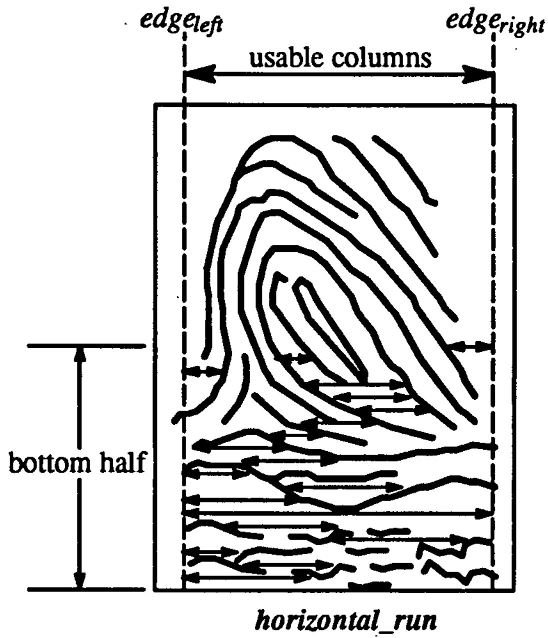


Figure 7. The Determination of the Largest Horizontal Runs

Once these largest runs are found, a score is associated with each row that approximates the area of the largest, thickest white portion touching that row. Given a row *i*, *row_score*(*i*) is determined by first finding the vertical runs of ***vertical_run*** that intersect *horizontal_run*(*i*), then multiplying the length of *horizontal_run*(*i*) by the sum of the lengths of the intersecting vertical runs. The entire collection of scores for all the rows is referred to as ***row_score***. Figure 8 illustrates this process of calculating ***row_score***.

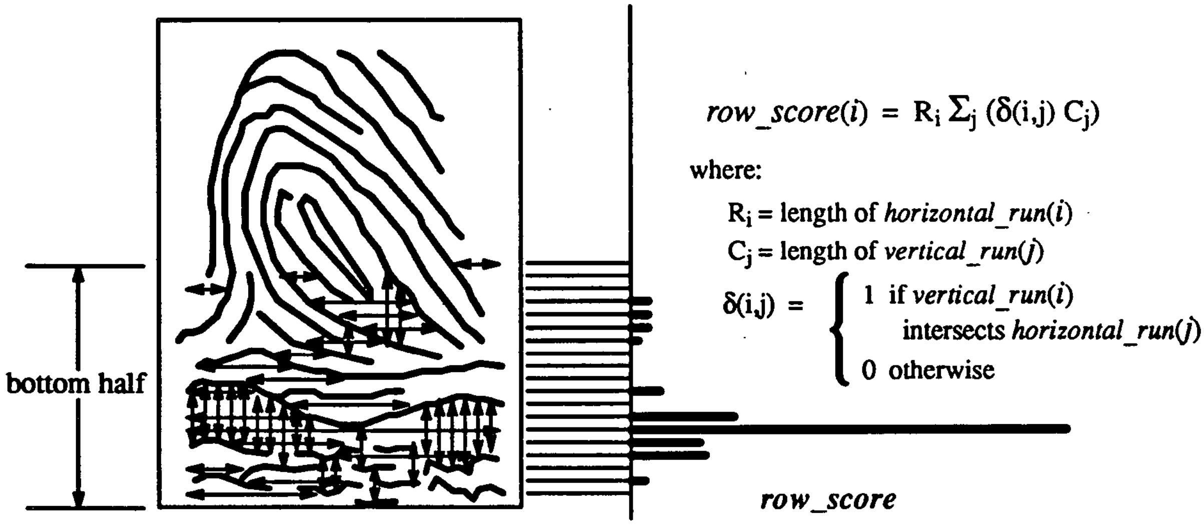
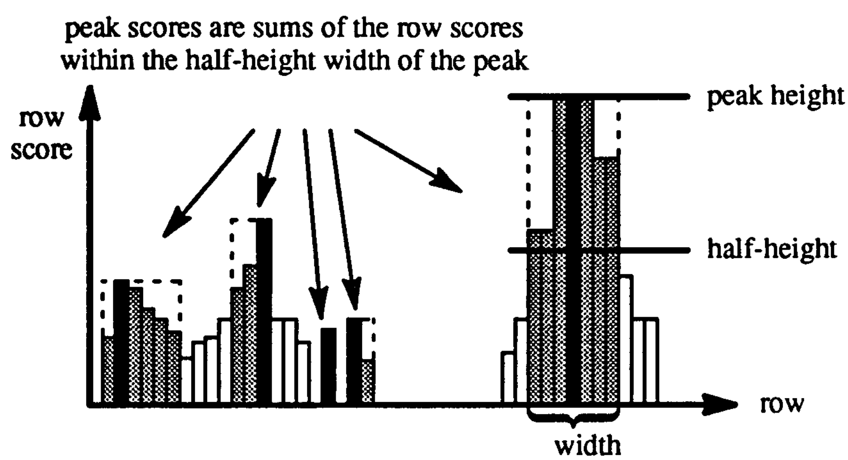


Figure 8. Combining ***horizontal_run*** and ***vertical_run*** to Produce the Row Scores

To detect the crease of the fingerprint, ***row_score*** is searched for the best broad high peak indicating the crease row, $crease_{center}$. The best broad high peak is selected by calculating a peak score for each row whose row score is a local maximum. Given such a row *i*, *peak_score*(*i*) is calculated as the sum of all the *row_score*(*k*) that are greater than half the peak value, *row_score*(*i*), where *k* is such that no row score less than half of this peak value exists between rows *i* and *k*. This calculation of peak score is illustrated in figure 9, which represents ***row_score*** as a bar graph. The best broad high peak is chosen as the peak with the largest peak score. In the unlikely event of a tie, the best broad high peak is selected to be the peak closest to the bottom of the thresholded fingerprint image. The row having the best peak score corresponds to the crease row, $crease_{center}$, of the fingerprint. The first row below the crease row whose peak score is less than half of *peak_score*($crease_{center}$) corresponds to the crease bottom $crease_{bottom}$. The trimming row is calculated as T_{OFFSET} rows below T_{bottom}. All the pixels in the rows of the thresholded fingerprint image below this trimming row are set to WHITE. Figure 10 illustrates calculation of the trimming row and trimming below the crease in the fingerprint image.



peak_score(*i*) = Σ { *row_score*(*k*) : row *k* is within the $peak_i$'s half-height width }

Figure 9. Calculation of the Peak Scores

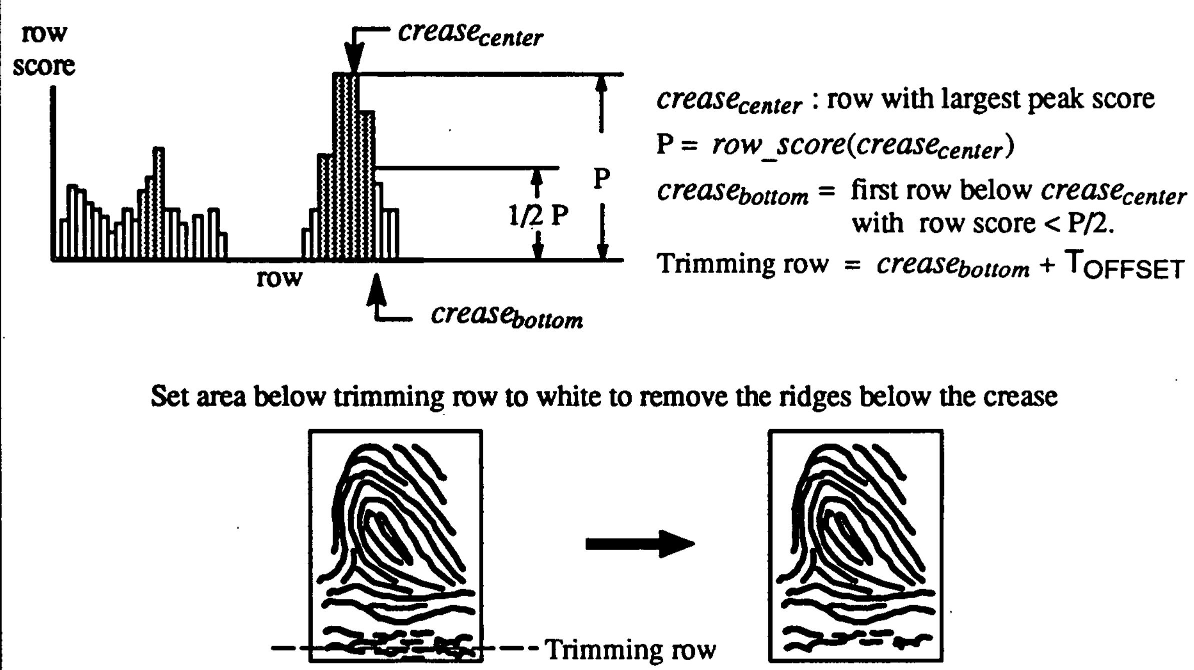


Figure 10. Trimming the Fingerprint Below the Crease

CREASE_TRIMMING[***I***]

** The fingerprint in ***I*** is modified by trimming T_{OFFSET} rows below the fingerprint crease

** Find the largest vertical runs of consecutive white pixels (***vertical_run***)

for each column *j* **in *I***

   $vertical\_run(j)$ = longest vertical run of white pixels in the lower half of column *j* not touching the top or bottom edge of the lower half of ***I***

** Calculate statistics on the sampled ***vertical_run*** lengths

$\mu_{vertical\_run}$ = mean{length of $vertical\_run(i)$: $(I_{width} - S_{VERTICAL\_RUN})/2 < i < (I_{width} + S_{VERTICAL\_RUN})/2$}

$\sigma_{vertical\_run}$ = standard deviation{length of $vertical\_run(i)$: $(I_{width} - S_{VERTICAL\_RUN})/2 < i < (I_{width} + S_{VERTICAL\_RUN})/2$}

$max_{vertical\_run}$ = max{length of $vertical\_run(i)$: $(I_{width} - S_{VERTICAL\_RUN})/2 < i < (I_{width} + S_{VERTICAL\_RUN})/2$}

```
** Set the threshold to be used in cleaning vertical_run of extrema
t = μ_vertical_run + σ_vertical_run
t_max = max_vertical_run + σ_vertical_run

** Find left edge of fingerprint impression (edge_left) and remove extreme vertical runs
   from left side of image
edge_left = 1                                  ** Initialize edge_left to left side of image
for each column j from I_width / 2 down to 1
    if(length of vertical_run(j) > t_max)
    {
        for each column edge_left from column j to I_width / 2
            if (length of vertical_run(edge_left) < t)
                exit from loop
        exit from loop
    }
    else if(length of vertical_run(j) = 0 and i ≤ (I_width−S_VERTICAL_RUN)/2)
    {
        edge_left = j + 1
        exit from loop
    }

** Find right edge of fingerprint impression (edge_right) and remove extreme
   vertical runs from right side of image
edge_right = I_width                           ** Initialize edge_right to right side of image
for each j from I_width / 2 to I_width
    if (length of vertical_run(j) > t_max)
    {
        for each column edge_right from column j to I_width / 2
            if (length of vertical_run(edge_right) < t)
                exit from loop
        exit from loop
    }
    else if(length of vertical_run(j) = 0 and i ≥ (I_width+S_VERTICAL_RUN)/2))
    {
        edge_right = j − 1
        exit from loop
    }
```

```
** Find the largest horizontal runs of consecutive white pixels (horizontal_run)
for each row i in I
    horizontal_run(i) = longest run of consecutive white pixels in row i
                                  between edge_left and edge_right

** Calculate the row scores for each row
for each row i from I_height/2 to I_height
{
    sum = 0
    for each column j from edge_left to edge_right
        if (vertical_run(j) intersects with horizontal_run(i))
            sum = sum + length of vertical_run(j)
    row_score(i) = sum × length of horizontal_run(i)
}

** Find the largest, broadest peak in row_score
best_score = 0
C_bottom = I_height
for each row i from I_height down to I_height/2 + 1
    if (row_score(i) ≥ row_score(i–1))
    {
        sum = row_score(i)
        k = i – 1
        while ((row_score(k) > row_score(i)/2) and (k ≥ I_height/2))
            if (row_score(k) ≤ row_score(i))
                sum = sum + row_score(k)
                k = k – 1
            else
                sum = 0
                exit from loop
        if (sum = 0)
            continue loop for next row
        k = i + 1
        while ((row_score(k) > row_score(i)/2) and (k ≤ I_height))
            if (row_score(k) ≤ row_score(i))
                sum = sum + row_score(k)
                k = k + 1
            else
                sum = 0
                exit from loop
```

```
        if (sum > best_score)
            best_score = sum
            C_center = i
            C_bottom = k
    }

** Trim the fingerprint below C_bottom + T_OFFSET
for each row i from C_bottom + T_OFFSET to I_height
    for each column j in I
        I(i, j) = WHITE
return
```

3.1.2 Summary

Parameters

$T_{OFFSET} = 40$ Number of rows below the crease where trimming begins

$S_{VERTICAL\_RUN} = 0.5\, I_{width}$ Width of image central region used in collecting statistics on ***vertical_run***

Input

I Dynamically thresholded fingerprint image (see section 2)

Output

I Crease trimmed fingerprint image

Calculated Values

vertical_run The longest run of consecutive white pixels for every column

$\mu_{vertical\_run}$ Mean of the sampled ***vertical_run*** lengths

$max_{vertical\_run}$ Maximum of the sampled ***vertical_run*** lengths

$\sigma_{vertical\_run}$ Standard deviation of the sampled ***vertical_run*** lengths

t_{max} Maximum threshold of ***vertical_run*** lengths

t Threshold of ***vertical_run*** lengths

$edge_{left}$ The left edge of the fingerprint impression

$edge_{right}$ The right edge of the fingerprint impression

horizontal_run The longest run of consecutive white pixels for every row

row_score The score proportional to the area of the white region around each *horizontal_run*(*i*)

peak_score The score proportional to the area of each peak in the ***row_score***

$crease_{center}$ Row with the largest peak score closest to the bottom of the thresholded fingerprint image

$crease_{bottom}$ First row below the Crease_row whose peak score is half the C_{center} peak score

3.2 SPUR REMOVAL

Spur removal removes the small, single-pixel ridge spurs and isolated BLACK pixels which may occasionally occur in the thresholded image due to either the fingerprint scanning or the dynamic thresholding process. These thin ridge spurs must be removed for correct ridge processing. An example of several ridge spurs is illustrated in figure 11. The ridge spurs are removed by detecting single black pixel spur ends, then removing these single pixel spurs down to the fingerprint ridge. The algorithm removes BLACK pixels starting from the end of a thin ridge spur in order not to remove the single-pixel borders between some fingerprint pores and their neighboring fingerprint valley. If a single-pixel border is removed, the associated pore will open onto the neighboring fingerprint valley.

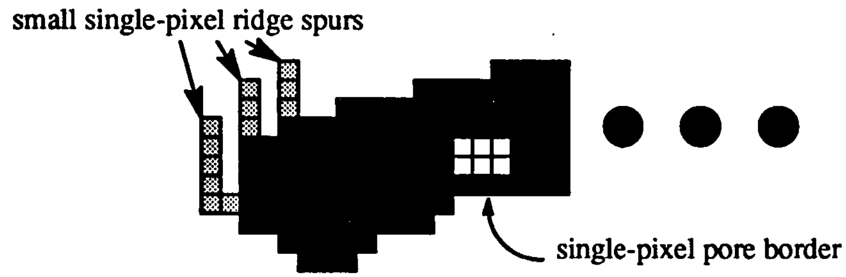


Figure 11. Examples of Small Single-Pixel Ridge Spurs

3.2.1 Algorithm Description

Spur removal operates on a thresholded fingerprint image that has been processed by crease trimming. The algorithm scans the entire image, checking each pixel for the possibility of being the end of a single-pixel ridge spur. If such a spur is found, it is immediately removed down to the actual ridge. Once the spur is removed, the algorithm returns to the point at which the removal of the spur began and continues the image scan in a manner such that all remaining pixels in the image are considered.

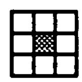
As the algorithm scans through the image, it checks if the current pixel is a ridge pixel (BLACK). If so, the algorithm considers its eight neighboring pixels and determines the number of these neighboring pixels that are ridge pixels. If, in the process of counting, the number of neighboring ridge pixels exceeds three, the count terminates and the algorithm continues by examining the next pixel of the image scan. This early termination decreases the processing needed for ridge pixels that are definitely not part of a thin ridge spur. The spur removal process considers several alternatives:

- If the current pixel has three neighboring ridge pixels that are all touching and are in a straight four-connected line, the current pixel is erased and the algorithm continues by examining the next pixel of the image scan.

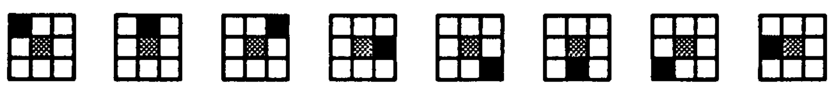
- If three or more neighboring pixels are ridge pixels and they are not in a straight four-connected line, the current pixel is not part of a thin ridge spur, hence the algorithm continues by examining the next pixel of the image scan.

for example: 

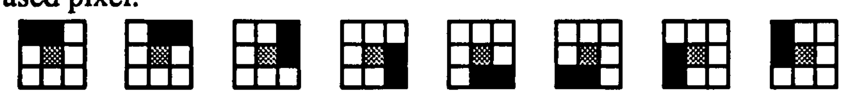
- If the current pixel has no ridge neighbors (i.e., is an isolated pixel), the current pixel is erased and the algorithm continues by examining the next pixel of the image scan.



- If the current pixel has only one ridge neighbor, the current pixel is erased and the algorithm continues by examining the current pixel's neighboring ridge pixel.



- If the current pixel has only two neighboring ridge pixels that are four-connected to each other, the current pixel is erased and the algorithm continues by examining the neighboring ridge pixel that is four-connected to this erased pixel.



The spur removal process is illustrated by a flowchart in figure 12.

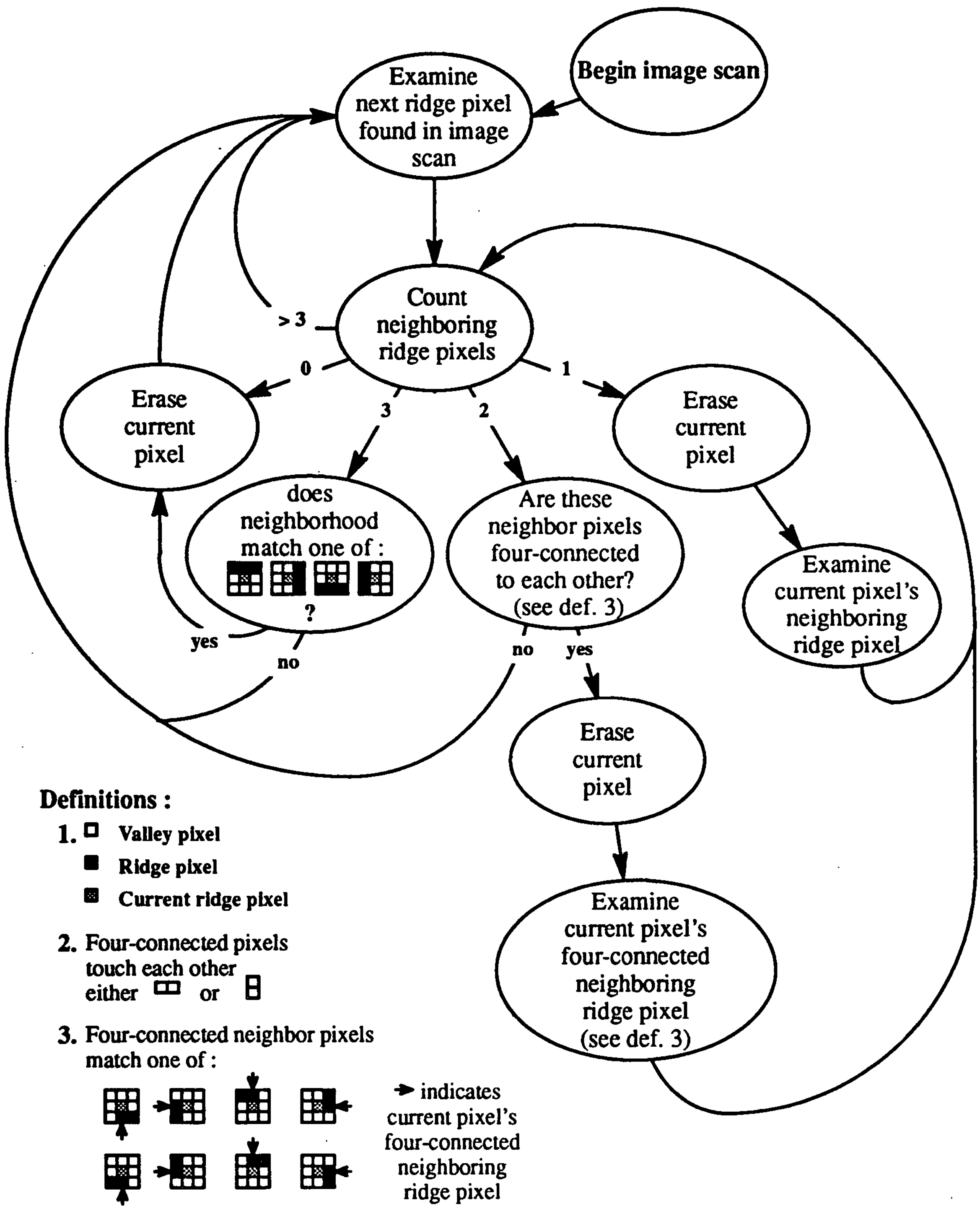


Figure 12. Flowchart for the Spur Removal Algorithm

Spur_Removal[*I*]

```
    ** This algorithm modifies I
    ** Every BLACK pixel in image I is checked for being the end of a ridge spur by a call to
    Process_Candidate_Spur_Pixel[]. If a pixel satisfies the conditions of being part of a
    spur, Process_Candidate_Spur_Pixel[] is recursively called until the spur is
    completely removed. At that point, the image scan proceeds from the pixel that began
    the ridge spur.
for each pixel (i, j) in I
    if (I(i, j) = BLACK)
        Process_Candidate_Spur_Pixel[ i, j ]        ** Modifies I
return
```

Process_Candidate_Spur_Pixel[*i, j*]

```
    ** To keep the data stack for this recursive process from growing larger than necessary,
    image I is not explicitly passed to this routine, instead, image I is considered to be a
    localized global value accessible and modifiable from within this procedure.
n = number of pixels neighboring the current pixel (i, j) whose value equals BLACK
if (n = 0)
    I(i, j) = WHITE                          ** Erase the current pixel
else if (n = 1)
    I(i, j) = WHITE                          ** Erase the current pixel
    set (i, j) to the coordinates of the neighboring ridge pixel
    Process_Candidate_Spur_Pixel[ i, j ]
else if (n = 2)
    if the neighboring ridge pixels are four-connected to each other
        I(i, j) = WHITE                      ** Erase the current pixel
        set (i, j) to the coordinates of the neighboring ridge pixel that is four-connected
                to the current pixel (See figure 12)
        Process_Candidate_Spur_Pixel[ i, j ]
else if (n = 3)
    if all three neighboring pixels are touching and in a straight line
        I(i, j) = WHITE                      ** Erase the current pixel
return
```

3.2.2 Summary

Input

I	Dynamically thresholded, crease-trimmed fingerprint image

Output

I	Clean thresholded fingerprint image

SECTION 4

PORE FILLING

Sweat pores in fingerprints are naturally occurring features that result from sweat glands breaking through the skin surface. However, pores are not reliably present in fingerprints and they can be obliterated or altered by pressure or other factors [2].

After the fingerprint images are thresholded into binary images, the pores that are internal to the ridges become apparent. These pores do not need to be retained for the automated matching task. Therefore, the algorithm attempts to remove as many as possible without changing the important fingerprint characteristics, i.e., without changing the fingerprint topology as represented by the ridges and ridge minutiae (terminations and bifurcations). Because the algorithm must be conservative when removing the pores in order not to change the fingerprint characteristics, a small number of pores may remain in a fingerprint after pore removal. However, the criteria for pore retention can be varied by adjusting certain input parameters.

4.1 ALGORITHM DESCRIPTION

Pore removal proceeds in two phases: small pore removal and large pore removal. Small pores are first identified in a binary fingerprint image based on consideration of the widths of the ridges in neighborhoods around the pore candidates. After the small pores are removed, the large pores are identified based on a comparison of the widths of the ridges across the pores with the widths of the ridges on the sides of the pores. Finally, identified large pores are also removed.

For purposes of the following discussion, the distinction between ridge pixels and pore and valley pixels must be defined. Following the standard for inked fingerprints, black pixels are taken to be ridge pixels and white pixels are taken to be pore or valley pixels. Ridge pixels are considered to be connected if they are adjacent horizontally, vertically, or diagonally, i.e., if they are eight-connected. Pore and valley pixels are considered to be connected if they are adjacent horizontally or vertically, i.e., if they are four-connected. This distinction is important since the algorithm may deal with ridges that are one pixel wide.

Parameters associated with the pore filling algorithm are described in section 4.2. In the algorithm descriptions, "distance" refers to the Euclidean distance between two pixels in the image, using the pixel coordinates as the point locations. In the text and pseudocode, image pixels are referred to as P_{sub}, where $_{sub}$ is an identifying subscript.

```
PORE_FILLING[ IMAGE ]
    ** This function has the side effect of modifying IMAGE

if (REMOVE_SMALL_PORES[IMAGE] = TRUE)                ** Section 4.1.1
    return
REMOVE_LARGE_PORES[IMAGE]                            ** Section 4.1.2
return
```

4.1.1 Small Pore Filling

The goal of small pore filling is to fill in the white spaces in a fingerprint image that can be reliably and quickly identified as pores. The smaller a white space is, the more likely it is to be a pore. Based on this fact, the algorithm developers have created a canonical definition of a "small pore" that will be reliable for most fingerprints to be processed. The magnitude of "small" is determined relative to the width of the ridges in the region surrounding the pore candidate. The canonical small pore is defined to be a circular white space inside a ridge, where the diameter of the pore is one pixel less than the average ridge width in the surrounding neighborhood (see figure 13). Use of the average neighborhood ridge width (see section 4.1.3) ensures that small pore filling is sensitive to the ridge width variations across the fingerprint without being overly sensitive to individual ridge width behavior. If the area of a small pore candidate is less than the area of the canonical small pore in its neighborhood, that candidate is declared to be a small pore and is filled. Note that the candidate need not be a circular region; the circular canonical pore was only defined in order to provide a reliable maximum area for a small pore.

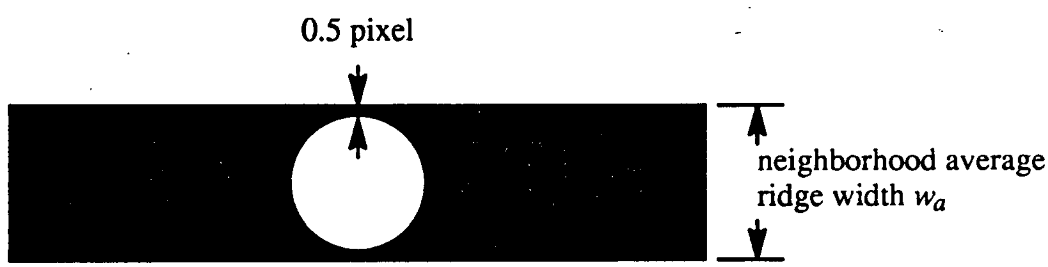


Figure 13. Canonical Small Pore

Given the above definition of a canonical small pore, the algorithm for identifying and filling small pores is straightforward. First, a connected-components analysis is performed on the fingerprint image and the connected white regions are identified and labeled. Using the labels, the area (number of pixels) of each region can be found in a lookup table (see Appendix A of [3]). Second, the fingerprint is broken into fixed-sized regions and the average ridge width is determined for each region (see section 4.1.3). Each of these regions serves as the neighborhood for every small pore candidate that is contained in them. If analysis of these regions shows that there are no pores or no ridges, pore filling is complete. Otherwise, the image is scanned (left-to-right, top-to-bottom) and the labeled white regions are selected in turn as candidate small pores. Given P_o, the first pixel of the candidate small pore encountered by the scan, the average ridge width w_a in the neighborhood containing the candidate is found through the methods of section 4.1.3 by using P_o as the location of the candidate. If the area of a candidate pore, i.e., the number of pixels it contains, is less than $\pi((w_a - 1)/2)^2$, then the candidate is identified as a small pore and is filled in.

```
REMOVE_SMALL_PORES[ IMAGE ]

    ** This function has the side effect of modifying IMAGE

    ** Label white regions (connected components), as discussed in Appendix A of [3]
    (num_regions, LABEL_IMAGE, area_vector)
            = FOUR-CONNECTED_COMPONENTS[IMAGE, BLACK, LABEL_IMAGE]
    if (num_regions = 0)
        exit                                        ** Error: IMAGE is all black
    else if (num_regions = 1)
        if (area_vector[1] = width * height)
            exit                                    ** Error: IMAGE is all white
        else
            return TRUE  ** A single white area not covering the whole image, so no pores

    PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS[IMAGE]          ** Section 4.1.3
    if (all average neighborhood ridge widths are 0)
        return TRUE         ** There are no ridges with width between WMIN and WMAX

    for i from 1 to height
        for j from 1 to width
            if (IMAGE(i, j) is WHITE and IMAGE(i, j – 1) is BLACK)
                wa = AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH[i, j]
                if (area_vector(LABEL_IMAGE(i, j)) < ⌈π ((wa – 1) / 2)²⌉)
                    fill white region containing IMAGE(i, j)     ** Fill the pore
    return FALSE                                  ** Proceed with large pore filling
```

4.1.2 Large Pore Filling

The goal of large pore filling is to fill in as many of the pores as possible that were not filled in by the small pore filling algorithm, without filling in white spaces that are not pores. This process is more difficult than small pore filling because some large pores are similar to valleys and vice versa. To identify large pores, the algorithm compares candidates to the model of a large pore that was developed for this task (see below). If the candidate pore matches the pore model, it is further checked to verify that it is not a small valley. If the verification succeeds, the pore has been identified and is filled in.

4.1.2.1 Large Pore Model

To identify large pores, the algorithm first needs a model of a large pore. A large pore is identified based on its width and the width of the ridge containing it. If the width of the ridge across the pore candidate is sufficiently small with respect to the minimum ridge width to either side of the candidate (figure 14), then the candidate matches the large pore model. If the candidate matches the model, it is further checked to ensure that it is a pore and not a valley.

Because the ridge width calculation can inadvertently span more than one ridge (figure 15), the ridge widths measured to the sides of the candidate are compared to the average ridge width in the neighborhood around the candidate. If the minimum side ridge width is sufficiently greater than the neighborhood average ridge width, the algorithm assumes that the measurement of the side ridge width spanned more than one ridge and is thus invalid. In this case, the candidate is declared to be a valley. Otherwise, the candidate is declared to be a pore and is filled in. This process is designed to be conservative to avoid filling in valleys at the expense of not filling in questionable pores.

4.1.2.2 Candidate Selection

The algorithm for identifying and filling large pores once again uses the average ridge width w_a for regions of the image (section 4.1.3), but these widths must first be recalculated because the small pores have now been filled in. After this recalculation, the large pore candidates are selected by considering each white region that still remains in the fingerprint image after small pore elimination. Given the parameter L_{MAX} (see section 4.2), if the area of a white region (the number of pixels it contains) is greater than $w_a L_{MAX}$, that candidate is assumed to be a valley; otherwise, it identified as a large pore candidate. (This size consideration implies that a white space larger than one average ridge width wide and L_{MAX} average ridge widths long is taken to be a valley.) To fill large pores, the image is scanned (left-to-right, top-to-bottom) until a black-to-white pixel transition is encountered. If the region that contains the white pixel of this transition meets the criterion for a large pore candidate, the pixel is labeled P_o and identifies the pore candidate; P_o and the pixels in the

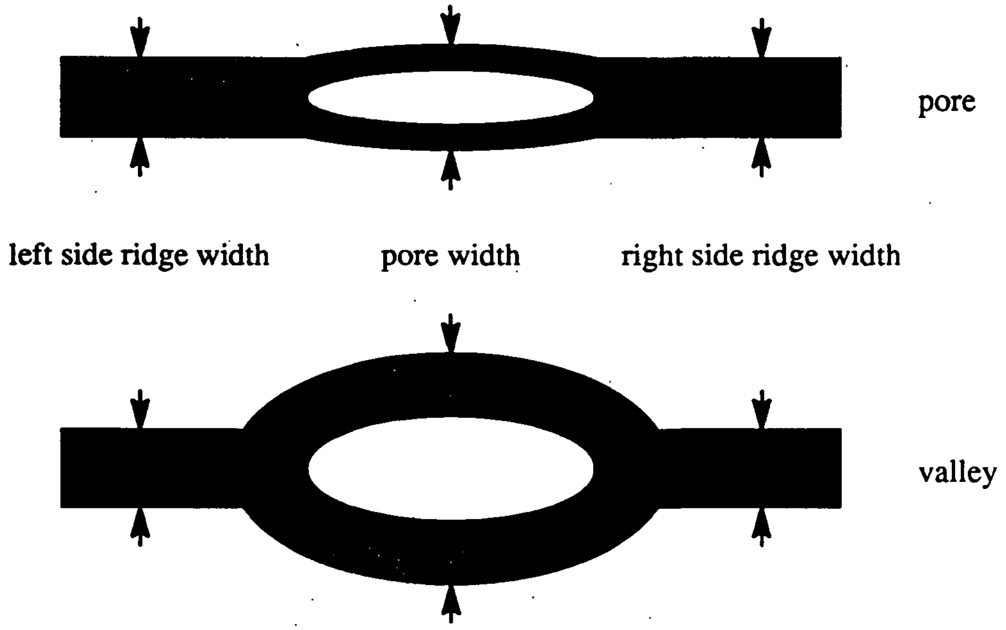


Figure 14. Large Pore Model

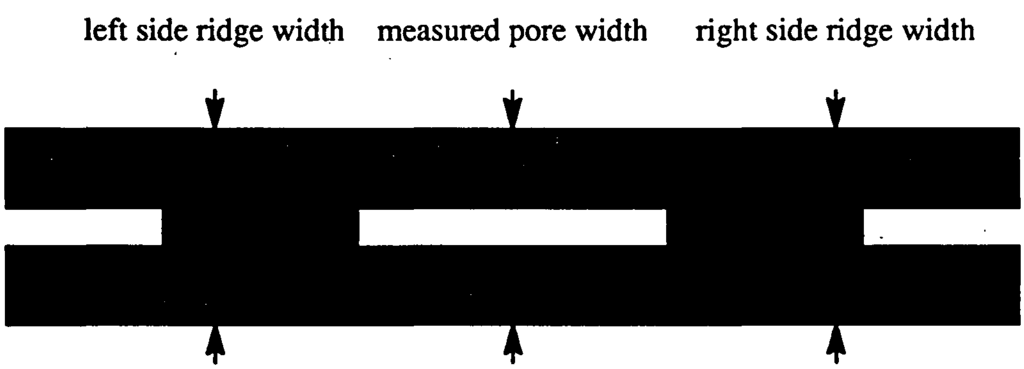


Figure 15. A Valley that is Similar to a Large Pore

white region containing it are then labeled LARGE_PORE_CANDIDATE. (The labels could be implemented, for example, by maintaining a separate label array of the same size as the image.) Each pore candidate found is checked against the large pore model and filled in if it matches. After the check, the scan resumes until all the pore candidates have been tested.

```
REMOVE_LARGE_PORES[ IMAGE ]
    ** This function has the side effect of modifying IMAGE
    ** The array IMAGE should be available globally to the subroutines
       under REMOVE_LARGE_PORES

PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS[IMAGE]          ** Section 4.1.3
for i from 1 to height
    for j from 1 to width
        if (IMAGE(i, j) is WHITE and IMAGE(i, j − 1) is BLACK)
            w_a = AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH[i, j]
            if (area of white region containing IMAGE(i, j) < w_a L_MAX)
            {
                for pixel in white region containing IMAGE(i, j)
                    label pixel as LARGE_PORE_CANDIDATE
                LARGE_PORE_TEST[i, j]        ** Removes large pores, section 4.1.2.3
            }
return
```

4.1.2.3 Large Pore Model Test

To compare a candidate pore against the large pore model, the width of the ridge must be calculated both across the white space and at the sides of the white space (figure 14). (Because a ridge edge actually is a black-pixel-to-white-pixel transition, one side of this transition must be selected to represent the ridge edge. For this algorithm, the white pixels of the transition are chosen to define the ridge edge.) First, the edges of the enclosing ridges are located. Because the candidate was found using a raster scan of the image (left-to-right, top-to-bottom), it is guaranteed that black (ridge) pixels are to the left of and above the initial candidate pixel, P_o (figure 16). To find the edges of the enclosing ridges, the image pixels are searched to the left of, and searched up from, P_o until the first black-to-white (ridge-to-valley) transition found in each direction. If the white pixel of either transition has been labeled LARGE_PORE_CANDIDATE, (i.e., it is contained in a large pore or large pore candidate, see figure 17), or if the distance from P_o to the white pixel exceeds LU_{MAX}, the transition is invalid and is discarded. (Note that, because of the order of the raster scan of the image, all large pore candidates to the left of P_o and all large pore candidates above P_o have already been labeled.) If both transitions are valid, the white pixel of the transition closest to P_o is selected as P_e, the edge of the ridge surrounding the pore candidate. If only one transition is valid, it is used to select P_e. If neither transition is valid, no decision can be made about this large pore candidate, so it is not filled in and the scan continues for the next candidate.

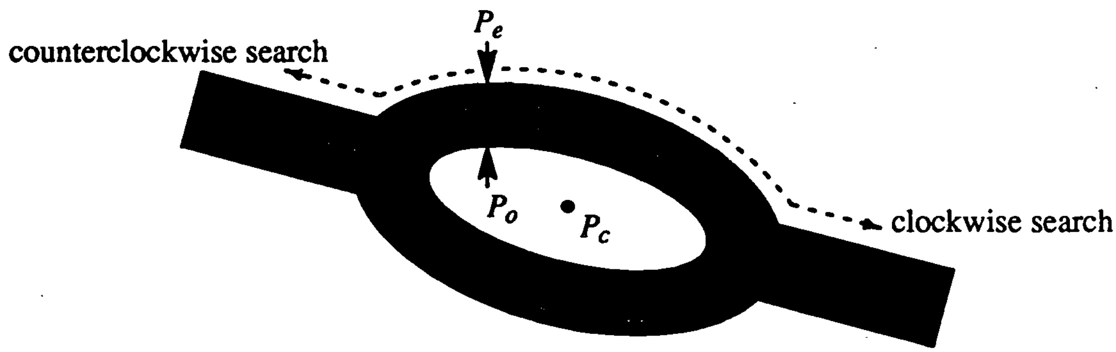


Figure 16. P_o, P_e, and the Search for P_{pl} in a Large Pore Candidate

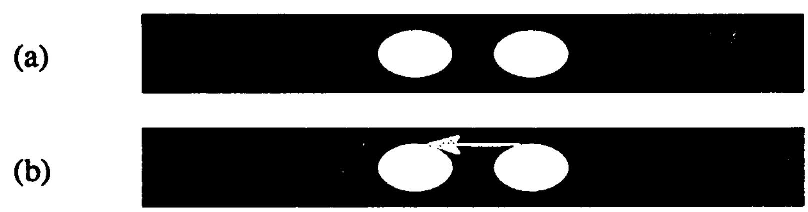


Figure 17. Search Failure. This figure shows an example of a failure in the search for the black-to-white transition of the enclosing ridge edge of a large pore candidate. (a) A ridge containing a large pore on the left and a large pore candidate on the right. (b). The search to the left from the candidate finds a black-to-white transition, but the transition is invalid because the white pixel is contained in a large pore.

After finding the enclosing ridge, the point on the ridge closest to the center of the pore candidate must be found. First, the center of area of the candidate's (white) pixels, P_c, is found. Then, given P_e and P_c, the algorithm finds the pixel P_{pl} on the ridge edge that is closest to the candidate's center of area. The search used to find the minimizing pixel P_{pl} is described in section 4.1.2.4. This search is conducted in both the clockwise and counterclockwise directions along the ridge edge from P_e (see figure 16). If no point P_{pl} can be found, no decision can be made about this large pore candidate, so it is not filled in and the scan continues for the next candidate.

The local tangent is calculated at P_{pl} using P_{pl} and two edge pixels to either side of it. (By fitting a line in rho-theta form to these points using the least squares technique described by Horn [4] instead of using the more common slope-intercept formulation, lines that are vertical or near vertical pose no special problem.) The perpendicular to this tangent is then searched across the ridge to find the other side of the ridge. The first black-to-white crossing should be at the pore candidate. If it is not, the ridge width across the candidate cannot be measured and the candidate is declared (by default) to be a valley. Otherwise, the search along the perpendicular is continued until the next black-to-white transition is found, ignoring any white regions encountered that are also part of the pore candidate. (The shape of the pore candidate may cause its white region to be encountered more than once.) The white pixel of this black-to-white edge transition across the pore from P_{p1} is labeled P_{p2} (see figure 18). The distance from P_{p1} to P_{p2} is the width across the pore candidate, w_p.

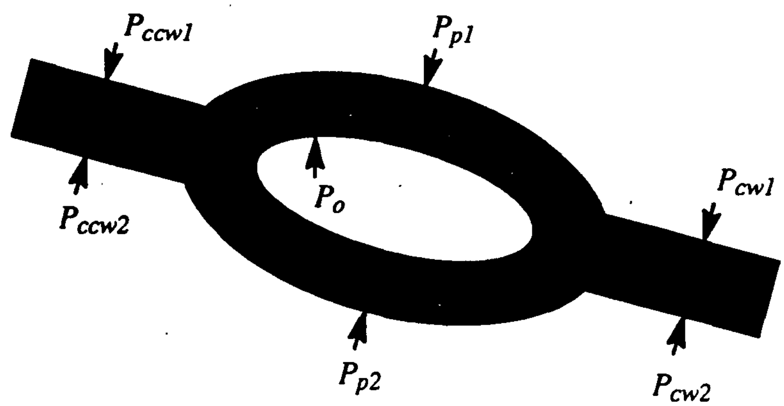


Figure 18. Comparison of Pore Candidate to Model

After the width across the pore candidate is determined, the width of the ridge to either side of the candidate must be found. First, the minimum and maximum row and column ($i_{min}, i_{max}, j_{min}$, and j_{max}) are found for the white candidate region. Then, given the average ridge width w_a in the neighborhood of the candidate pixel P_o, the ridge is traced from P_{p1} in both directions (clockwise and counterclockwise) until the row and column are outside the rectangle $[i_{min} - w_a, i_{max} + w_a] \times [j_{min} - w_a, j_{max} + w_a]$. These (white) ridge points are labeled P_{cw1} and P_{ccw1} (see figure 18). If tracing the ridge in the clockwise (counterclockwise) direction does not yield a ridge point outside the bounds stated above within E_{MAX} pixels of P_{p1}, the search is abandoned and P_{cw1} (P_{ccw1}) is not defined. Given P_{cw1} (P_{ccw1}), the point P_{cw2} (P_{ccw2}) directly across the ridge is found by starting at P_{p2} and tracing along the ridge edge in the counterclockwise (clockwise) direction until the distance between the white ridge pixel and P_{cw1} (P_{ccw1}) is minimized. (See section 4.1.2.4 for the minimizing procedure.) The pixel that minimizes this distance is labeled P_{cw2} (P_{ccw2}). If P_{cw2} (P_{ccw2}) is found to be the same as P_{cw1} (P_{ccw1}), then the search has wrapped around the ridge and P_{cw2} (P_{ccw2}) is not valid. The ridge width w_{cw} (w_{ccw}) is calculated as the distance between the points P_{cw1} and P_{cw2} (P_{ccw1} and P_{ccw2}). The minimum of w_{cw} and w_{ccw} is taken to be the ridge width w_r of the ridge containing the pore candidate. If any of P_{cw1}, P_{ccw1}, P_{cw2}, or P_{ccw2} cannot be found, the corresponding ridge width is not used and w_r is set to the other ridge width. If neither ridge width w_{cw} nor w_{ccw} can be found, the candidate is declared to be a valley.

Now that w_p and w_r have been calculated, the pore candidate can be compared to the pore model. If w_p is less than $w_r \mathrm{P_{MIN}}$, then the candidate matches the pore model. Otherwise, it is declared to be a valley. If the candidate matches the pore model, the next step is to verify that the candidate is a pore and not a small valley (figure 15). If w_r is greater than $w_a \mathrm{P_{MAX}}$, then w_r is assumed to have inadvertently spanned more than one ridge and the candidate is declared to be a valley. If w_p is greater than $w_a \mathrm{P_{MAX}}$, then the pore candidate is too wide and w_p is assumed to have been measured across the valley between two ridges; the candidate is declared to be a valley. Otherwise, the match of the candidate to the large pore model is accepted and the pore is filled in.

LARGE_PORE_TEST[*i*, *j*]

```
P_o = (i, j)
candidate = white region containing P_o

** Find enclosing ridge edge P_e
P_e,left = NOT_VALID
P_e,up = NOT_VALID
search left from P_o for black-to-white transition
P_temp = white pixel of transition
if (distance(P_temp, P_o) ≤ LU_MAX and P_temp is not labeled LARGE_PORE_CANDIDATE)
    P_e,left = P_temp
search up from P_o for black-to-white transition
P_temp = white pixel of transition
if (distance(P_temp, P_o) ≤ LU_MAX and P_temp is not labeled LARGE_PORE_CANDIDATE)
    P_e,up = P_temp
if (P_e,left ≠ NOT_VALID and P_e,up ≠ NOT_VALID)
    P_e = closer of (P_e,left, P_e,up) to P_o
else if (P_e,left ≠ NOT_VALID and P_e,up = NOT_VALID)
    P_e = P_e,left
else if (P_e,left = NOT_VALID and P_e,up ≠ NOT_VALID)
    P_e = P_e,up
else if (P_e,left = NOT_VALID and P_e,up = NOT_VALID)
    ** Not a pore, so return without filling the candidate
    return
```

** Find point P_{p1} on enclosing ridge closest to candidate center P_c
P_c = pixel at center of candidate
$P_{p1,cw}$ = SEARCH_EDGE_FOR_MINIMIZING_PIXEL[P_c, P_e, clockwise]
$P_{p1,ccw}$ = SEARCH_EDGE_FOR_MINIMIZING_PIXEL[P_c, P_e, counterclockwise]
if ($P_{p1,cw} \neq$ NOT_VALID **and** $P_{p1,ccw} \neq$ NOT_VALID)
    P_{p1} = closer of ($P_{p1,cw}$, $P_{p1,ccw}$) to P_c
else if ($P_{p1,cw} \neq$ NOT_VALID **and** $P_{p1,ccw} =$ NOT_VALID)
    $P_{p1} = P_{p1,cw}$
else if ($P_{p1,cw} =$ NOT_VALID **and** $P_{p1,ccw} \neq$ NOT_VALID)
    $P_{p1} = P_{p1,ccw}$
else if ($P_{p1,cw} =$ NOT_VALID **and** $P_{p1,ccw} =$ NOT_VALID)
    ** Not a pore, so return without filling the candidate
    return

** Find point P_{p2} on enclosing ridge edge opposite from P_{p1} and pore width w_p
find local tangent to P_{p1}
search from P_{p1} across ridge perpendicular to tangent until first black-to-white transition
if (white pixel of transition $\notin$ *candidate*)
    ** Not a pore, so return without filling the candidate
    return
else
    continue search until first black-to-white transition where white pixel $\notin$ *candidate*
    P_{p2} = white pixel of transition
w_p = distance(P_{p1}, P_{p2})

** Find ridge edge pixels to either side of candidate
P_{cw1} = NOT_VALID
P_{cw2} = NOT_VALID
$i_{min}, i_{max}, j_{min}, j_{max}$ = minimum and maximum rows and columns of *candidate*
trace at most E_{MAX} pixels clockwise along ridge edge from P_{p1} until outside the
        rectangle $[i_{min} - w_a, i_{max} + w_a] \times [j_{min} - w_a, j_{max} + w_a]$
if (trace succeeded)
    P_{cw1} = final white pixel of trace
trace at most E_{MAX} pixels counterclockwise along ridge edge from P_{p1} until outside the
        rectangle $[i_{min} - w_a, i_{max} + w_a] \times [j_{min} - w_a, j_{max} + w_a]$
if (trace succeeded)
    P_{cw2} = final white pixel of trace

** Find opposite ridge edge pixels to either side of candidate

P_{cw1} = NOT_VALID
P_{cw2} = NOT_VALID
P_{temp} = SEARCH_EDGE_FOR_MINIMIZING_PIXEL[P_{cw1}, P_{p2}, counterclockwise]
if ($P_{temp} \neq P_{cw1}$)
    $P_{cw2} = P_{temp}$
P_{temp} = SEARCH_EDGE_FOR_MINIMIZING_PIXEL[P_{ccw1}, P_{p2}, clockwise]
if ($P_{temp} \neq P_{ccw1}$)
    $P_{ccw2} = P_{temp}$

** Find ridge widths to sides of candidate

w_{cw} = NOT_VALID
w_{ccw} = NOT_VALID
if ($P_{cw1} \neq$ NOT_VALID **and** $P_{cw2} \neq$ NOT_VALID)
    w_{cw} = distance(P_{cw1}, P_{cw2})
if ($P_{ccw1} \neq$ NOT_VALID **and** $P_{ccw2} \neq$ NOT_VALID)
    w_{ccw} = distance(P_{ccw1}, P_{ccw2})

** Find ridge width to side of candidate

if ($w_{cw} \neq$ NOT_VALID **and** $w_{ccw} \neq$ NOT_VALID)
    w_r = min(w_{cw}, w_{ccw})
else if ($w_{cw} \neq$ NOT_VALID **and** w_{ccw} = NOT_VALID)
    $w_r = w_{cw}$
else if (w_{cw} = NOT_VALID **and** $w_{ccw} \neq$ NOT_VALID)
    $w_r = w_{ccw}$
else if (w_{cw} = NOT_VALID **and** w_{ccw} = NOT_VALID)
    ** Not a pore, so return without filling the candidate
    return

```
** Compare candidate to pore model
if (w_p < w_r P_MIN)
    if (w_r > w_a P_MAX or w_p > w_a P_MAX)
        ** Not a pore, so return without filling the candidate
        return
    else
        ** A pore
        fill in candidate
        return
else
    ** Not a pore, so return without filling the candidate
    return
end
```

4.1.2.4 Searching a Ridge Edge to Minimize the Distance between the Edge and Another Point

Given a point P and a white pixel Q on a ridge edge, various steps of the algorithm need to find the pixel Q_{min} on the ridge edge that minimizes the distance between the edge and P. Depending on the step in the algorithm, Q_{min} must be found in the clockwise or counterclockwise direction along the ridge edge from Q. To find Q_{min}, first let the current minimum m be the distance PQ and the minimizing pixel Q_{min} be Q. Then, choose the next neighboring white pixel Q' of Q_{min} in the clockwise (counterclockwise) direction and compare the distance PQ' to m. If PQ' is less than m, it becomes the new minimum distance m and Q' becomes the new minimizing pixel Q_{min}. Otherwise, the search continues in the same direction to the next neighboring white pixel of Q' and the process is repeated. If a new minimizing pixel is not found within H pixels along the edge from the current minimizing pixel Q_{min}, then the search ends and Q_{min} is the minimizing pixel. (This hysteresis H allows for small variations in the smoothness of the ridge edge.) To limit the search, if E_{MAX} edge pixels have been examined and the last pixel examined is less than H pixels from the current Q_{min}, the search has failed and no minimizing pixel is found.

```
SEARCH_EDGE_FOR_MINIMIZING_PIXEL[ P, Q, direction ]
    ** P is the fixed pixel to which this routine minimizes the distance along a ridge edge
    ** Q is a white pixel on a ridge edge and serves as a starting point for the search
    ** direction is the search direction: either clockwise or counterclockwise
    m = distance(P, Q)
    Q_min = Q
    n = 0
    n_past_min = 0
    while (n_past_min < H and n < E_MAX)
        Q' = neighboring white pixel of Q_min in direction
        increment n
        if (distance(P, Q') < m)
            m = distance(P, Q')
            Q' = Q_min
            n_past_min = 0
        else
            increment n_past_min
    if (n ≥ E_MAX)
        return NOT_VALID
    else
        return Q_min
    end
```

4.1.3 Neighborhood Average Ridge Width

The algorithms for identifying large and small pores use the average ridge width in the neighborhood of each pore candidate. Rather than calculate the average ridge width in neighborhoods centered on each candidate, which would be computationally expensive, the average ridge width is found for fixed regions across the fingerprint image. The average ridge width in the neighborhood of a pore candidate is then approximated by the average ridge width in the fixed region in which it lies.

The $R \times C$ (rows $\times$ columns) fingerprint image is partitioned into R_P sections vertically and C_P sections horizontally (figure 19). Each resulting $R/R_P \times C/C_P$ rectangle is used as a neighborhood for the average ridge width calculation. (The parameter values used during development and testing of the Pore Filling algorithms are given in section 4.2. The values of R_P and C_P were chosen to evenly partition the image so that the resulting neighborhoods were roughly 60×60, thus covering large enough portions of the fingerprint to yield meaningful average ridge widths. See Appendix D.) The widths for all ridges within each rectangle are calculated and the average ridge width is stored for each rectangle. To calculate the average ridge widths, a raw thinned image and a chamfered image are created from the binary fingerprint image (see section 5). Then, for each rectangle, the pixels in the raw thinned image are scanned. When a black (ridge) pixel is encountered, the corresponding value v_c from the chamfered image is found. The ridge width at this pixel is then calculated as $w = 2v_c / 1000$. (The algorithm for calculating the ridge width at a pixel is described fully in section 7.1.2. Note that although section 7.1.2 addresses the calculation of the average ridge width along a fingerprint curve, the part of the calculation that determines the ridge width *at a pixel* is used here in determining the average ridge width in a rectangle.) The sum of all the ridge widths w in a rectangle, divided by the number of raw thinned image ridge pixels in that rectangle, yields the average ridge width w_a for that rectangle. If the ridge width at any pixel falls outside of the inclusive bounds [W_{MIN}, W_{MAX}], however, the width is assumed to be in error and is not used. For development and testing of the algorithm, W_{MIN} was chosen to prevent the inclusion of one- and two-pixel wide ridges, which typically correspond to pore edges. W_{MAX} was chosen so that large "smudge" regions, which do not correspond to valid ridges, are not included in the average ridge width calculation.

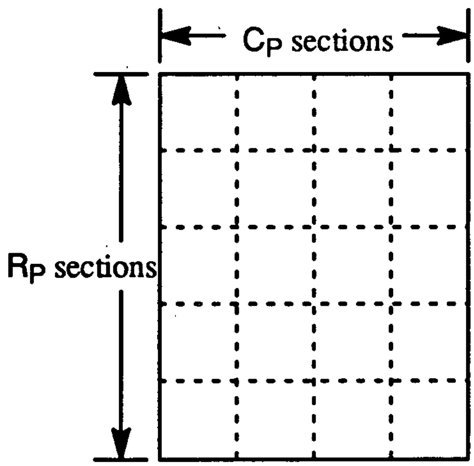


Figure 19. Partitioning of Fingerprint Image for Neighborhood Average Ridge Width Calculation

Given a point in the fingerprint image, the average neighborhood ridge width algorithm returns the average ridge width w_a for the rectangle containing that point. One possible implementation of the average ridge width routines is to store the average ridge widths for the rectangles of the partitioned image in an array and to access the array based on the given point's coordinates, the size $R \times C$ of the fingerprint image, and the number of sections R_P and C_P of the image partition.

```
PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS[ IMAGE ]
    ** This function has the side effect of modifying IMAGE

    ** rows_per_section and columns_per_section should be available globally to the
       subroutines dealing with average ridge widths
    ** See Appendix D for information on setting the parameters RP and CP
    rows_per_section = ⌈R / RP⌉
    columns_per_section = ⌈C / CP⌉
    create RIDGE_WIDTH_ARRAY with RP rows and CP columns
    initialize RIDGE_WIDTH_ARRAY with zeros
    (CHAMFER, RAW_THIN) = RIDGE_THINNING[IMAGE]  ** Section 5
    ** Store the average ridge widths of the sections of the fingerprint image
    for row from 1 to RP
        for column from 1 to CP
            ** Determine the upper-left corner and extent of the current section
            i_low = ((row – 1) * rows_per_section) + 1
            j_low = ((column – 1) * columns_per_section) + 1
            i_size = min(rows_per_section, height – i_low + 1)
            j_size = min(columns_per_section, width – j_low + 1)
            RIDGE_WIDTH_ARRAY(row, column)
                    = AVERAGE_SECTION_RIDGE_WIDTH[i_low, j_low, i_size, j_size]
    return

AVERAGE_SECTION_RIDGE_WIDTH[ i_low, j_low, i_size, j_size ]
    count = 0
    sum = 0
    ** Sum the ridge widths in this section of the fingerprint image
    for i from i_low to i_low + i_size – 1
        for j from j_low to j_low + j_size – 1
            if (RAW_THIN(i, j) is BLACK)
                w = 2 * CHAMFER(i, j) / 1000
                if (w ≥ WMIN and w ≤ WMAX)
                    sum = sum + w
                    increment count
    if (count = 0)
        return 0
    else
        return sum / count
    end
```

AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH[*i, j*]

```
row = ⌊(i – 1) / rows_per_section⌋ + 1
column = ⌊(j – 1) / columns_per_section⌋ + 1
   ** Return the average ridge width for the section containing (i, j)
return RIDGE_WIDTH_ARRAY(row, column)
```

4.2 SUMMARY

The parameter values used during development and testing of the algorithms described in this section, as well as the input and output variables, are listed below.

Parameters

$C = 450$	Number of columns in the fingerprint image
$C_P = 9$	Number of horizontal sections in the partition of the fingerprint image used to calculate average ridge widths (see Appendix D)
$E_{MAX} = 50$	The maximum distance for a search along a ridge edge, in pixels
$H = 5$	When choosing a ridge edge pixel to minimize the distance to a point, a pixel is considered to minimize this distance if no ridge edge pixel within H pixels yields a smaller distance.
$L_{MAX} = 10$	Maximum ratio between the white area of a large pore candidate and the average ridge width in its neighborhood
$LU_{MAX} = 15$	Maximum distance to the left of, or up from, an initial pore pixel to its enclosing ridge edge, in pixels
$P_{MAX} = 2.5$	Maximum ratio between the pore and ridge widths of a candidate and the average neighborhood ridge width in the large pore model
$P_{MIN} = 3.0$	Minimum ratio between the width of a pore candidate and the ridges to either side of it in the large pore model
$R = 600$	Number of rows in the fingerprint image
$R_P = 10$	Number of vertical sections in the partition of the fingerprint image used to calculate average ridge widths (see Appendix D)
$W_{MAX} = 8.0$	Maximum width of a ridge for the average ridge width calculation, in pixels
$W_{MIN} = 1.4$	Minimum width of a ridge for the average ridge width calculation, in pixels

Input

IMAGE	Binary fingerprint image

Output

IMAGE	Pore-filled binary fingerprint image

SECTION 5

RIDGE THINNING

Ridge thinning processes the thick fingerprint ridges of the trimmed, thresholded image to produce a raw thinned image containing mostly single-pixel lines that represent the fingerprint ridges. This ridge thinning algorithm is used twice in the flat live-scan searchprint compression process. It was used previously by the pore filling process to generate the chamfered and thinned images required for calculating average ridge widths (see section 4.1.3). The ridge thinning algorithm is now applied to the pore-filled image to produce a raw thinned image. A further processing step described in curve extraction (see section 6) will process this raw thinned image before extracting the curves. This processed image will be referred to as the thinned image and will be free of the artifacts that remain in the raw thinned image after the thinning process described in this section.

5.1 ALGORITHM DESCRIPTION

Three major steps characterize the ridge thinning process: chamfering, local maxima detection, and recursive ridge following. The products of these steps are represented in figure 20. Chamfering generates an image whose pixel values represent approximate distances to fingerprint ridge edges. The chamfered image is used extensively, not only in the other two steps of this process, but also for calculation of average ridge widths in the pore filling (section 4) and ridge cleaning (section 7) processes, and must be retained until no further needed. Local maxima detection finds local maxima points within the chamfered image that serve as seed points for the recursive ridge following step. These local maxima points are placed in the final raw thinned image as part of the raw thinned ridges. The recursive ridge following step fills in the gaps between local maxima points. The recursive nature of the ridge following algorithm allows the trimming of unwanted spurs that may be generated by other methods of thinning.

```
RIDGE_THINNING[ I ]
  C = CHAMFER[ I ]
  T = DETECT_LOCAL_MAXIMA[ C ]
  for each pixel (i, j ) in T marked as a LOCAL_MAXIMUM pixel
      FOLLOW_RIDGE[ i, j, UNDEFINED_DIRECTION ]     ** Refers to C & T and modifies T
  return (C, T)
```

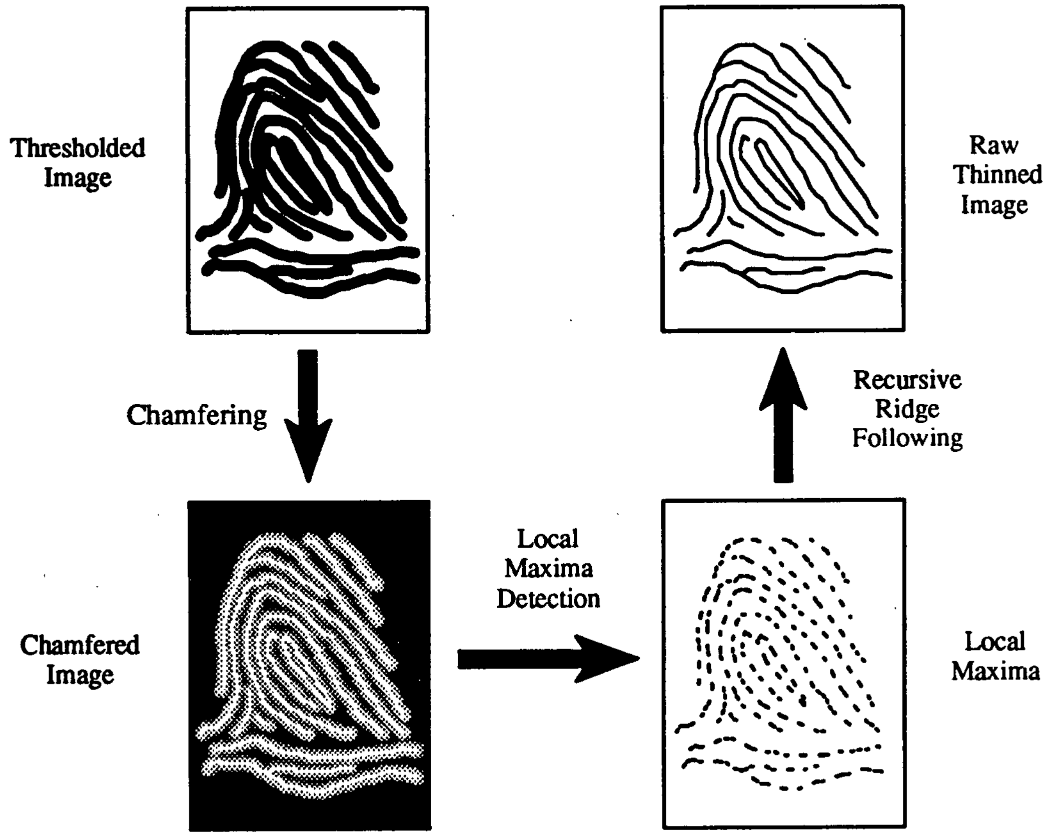


Figure 20. Intermediate Products of the Ridge Thinning Algorithm Steps

Inputs

I	Thresholded, cleaned, fingerprint image

Outputs

C	Chamfered image
T	Thinned image

5.1.1 Chamfering

The chamfering algorithm processes a binary image to produce an image in which each non-zero pixel value represents the shortest path distance to the closest edge pixel (i.e., the shortest path distance from each BLACK pixel to its nearest black pixel of a BLACK to WHITE transition). This shortest path distance was defined as the sum of diagonal pixel jumps and the rectilinear pixel jumps between two pixels. The chamfering algorithm is originally described in a paper by Barrow et al. [5]. In the chamfering algorithm used here, the shortest path distances are calculated for the pixels within the ridges, providing the basis for a fast algorithm to thin the fingerprint ridges to single-pixel widths. The resulting chamfered image also provides the capability to calculate the average ridge widths which is used in pore filling (section 4) and ridge cleaning (section 7).

The chamfering algorithm consists of an initialization pass and two chamfering passes. First, a new integer-typed image, the chamfered image, is created and initialized to zero. Then, initialization is completed by setting every chamfered image pixel corresponding to a thresholded image ridge pixel to a very large integer (see below). The very large integer used in the initialization must be larger than the largest possible chamfer value, c_{max}, of the final chamfered image, which can be calculated from the size of the image and the scaling factor as follows:

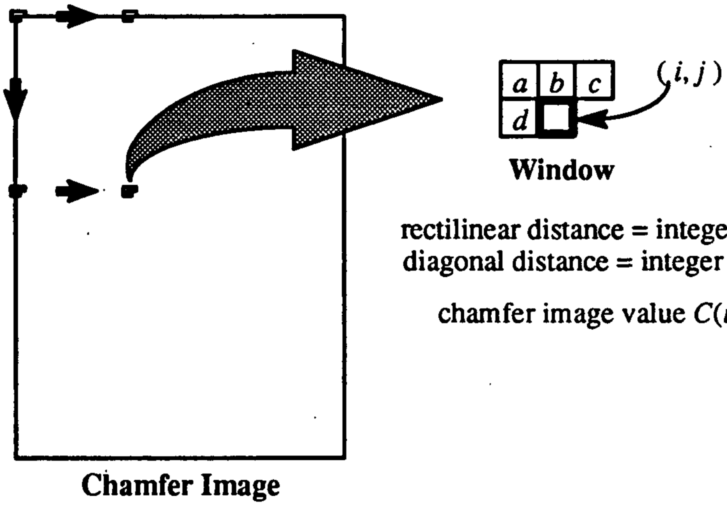
$$c_{max} = \text{floor}\left(\left(min\_size \times (\sqrt{2} - 1.0) + max\_size\right) \times scaling\_factor + 0.5\right)$$

Where:

min_size = the minimum of $height_I$ and $width_I$
max_size = the maximum of $height_I$ and $width_I$
scaling_factor is an integer larger than *min_size*.

The scaling factor specifies the precision retained in the integer arithmetic. Because the integer values of the square root of two and of one are both one, all numbers must be scaled by the scaling factor in order to preserve enough precision to differentiate between these two values. In the case of the 450×600 pixel live-scan fingerprint images, the scaling factor is set to 1000. Hence, all rectilinear jumps between pixels have a distance of 1000, and all diagonal jumps between pixels had a distance of 1414. The value of c_{max} computes to 782,254, requiring the chamfer image to have at least 20 bits per pixel.

Once the chamfered image is initialized, two passes of a similar operation are iterated over the image. The first chamfering operation is applied to the image from the top-left corner to the bottom-right corner of the image; scanning from left to right and from top to bottom. As this operation is applied to each pixel, the chamfered image values of the pixel and its neighboring pixels to the top-left, top, top-right, and left are considered. The chamfer value of the pixel is replaced with the minimum of the following values: its original chamfer value, the top-left value plus the diagonal jump distance, the top value plus the rectilinear jump distance, the top-right value plus the diagonal jump distance, and the left value plus the rectilinear jump distance. When a pixel under consideration is at the border of the image, only those neighboring pixels that are contained within the image are considered. This first pass, illustrated in figure 21, finds the shortest path distances from each ridge pixel to its nearest top-left ridge-edge pixel. The efficiency of this operation can be dramatically improved by first checking if the pixel being operated on has a value of zero before



rectilinear distance = integer of (1×1000) = 1000
diagonal distance = integer of ($\sqrt{2} \times 1000$) = 1414

chamfer image value $C(i, j)$ = minimum of :
a+1414,
b+1000,
c+1414,
d+1000,
$C(i, j)$

Figure 21. The First of Two Passes of the Chamfering Algorithm

calculating the above minimum. Approximately half of the pixels in the chamfered image have been initialized to zero (fingerprint valley pixels) and will continue to be zero.

The second chamfering operation on the image is identical to the first chamfering operation, except it is applied to the image as if it were rotated by 180 degrees. This second operation is applied from the bottom-right corner to the top-left corner of the image; scanning the image from right to left and from bottom to top. As this operation is applied to each pixel, the chamfer image values of the pixel and its neighboring pixels to the bottom-left, bottom, bottom-right, and right are considered. The chamfer value of the pixel is replaced with the minimum of the following values: its original chamfer value, the bottom-left value plus the diagonal jump distance, the bottom value plus the rectilinear jump distance, the bottom-right value plus the diagonal jump distance, and the right value plus the rectilinear jump distance. Again, when a pixel under consideration is at the border of the image, only those neighboring pixels that are contained within the image are considered. This second pass, illustrated in figure 22, finds the shortest path distances from each ridge pixel to its nearest ridge-edge pixel by completing the consideration of the bottom-right ridge-edge pixels. An example of the steps in generating the final chamfered image is shown in figure 23.

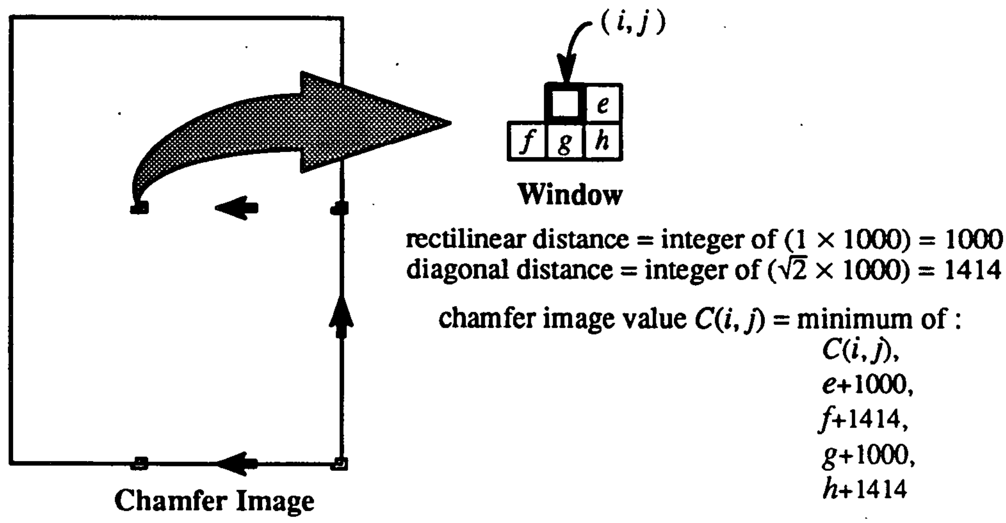


Figure 22. The Second of Two Passes of the Chamfering Algorithm

0	0	0	0	0	0	0	0	0
0	0	0	∞	∞	∞	∞	∞	∞
0	0	∞	∞	∞	∞	∞	∞	∞
0	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0
∞	∞	∞	∞	∞	∞	∞	∞	0
∞	∞	∞	∞	0	0	0	0	0
∞	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Initialized

∞ is the largest possible chamfer value, c_{max}

0	0	0	0	0	0	0	0	0
0	0	0	1000	1000	1000	1000	1000	1000
0	0	1000	1414	2000	2000	2000	2000	2000
0	1000	1414	2414	3000	3000	3000	3000	3000
1000	1414	2414	2828	3828	4000	4000	4000	0
2000	2414	2828	3828	4242	5000	5000	5000	0
3000	3414	3828	4242	0	0	0	0	0
4000	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

First Pass

0	0	0	0	0	0	0	0	0
0	0	0	1000	1000	1000	1000	1000	1000
0	0	1000	1414	2000	2000	2000	2000	2000
0	1000	1414	2414	3000	3000	2414	1414	1000
1000	1414	2414	2414	2000	2000	2000	1000	0
2000	2000	2000	1414	1000	1000	1000	1000	0
1414	1000	1000	1000	0	0	0	0	0
1000	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Second Pass

☐ RIDGE pixels

■ VALLEY pixels

Figure 23. An Example Portion of a Chamfered Image

CHAMFER[*I*]

```
** Initialization of the chamfered image C
for each pixel (i, j) in image I
    if (I(i, j) = RIDGE)
        C(i, j) = c_max
    else
        C(i, j) = 0

** First pass of the Chamfering algorithm
for each row i in C from 1 to height_I
    for each column j in C from 1 to width_I
        a = C(i-1, j-1) + 1414
        b = C(i-1, j) + 1000
        c = C(i-1, j+1) + 1414
        d = C(i , j-1) + 1000
        C(i, j) = minimum of a, b, c, d, C(i, j)

** Second pass of the Chamfering algorithm
for each row i in C from height_I to 1 step -1
    for each column j in C from width_I to 1 step -1
        e = C(i, j+1) + 1000
        f = C(i+1, j-1) + 1414
        g = C(i+1, j) + 1000
        h = C(i+1, j+1) + 1414
        C(i, j) = minimum of e, f, g, h, C(i, j)
return C
```

5.1.2 Local Maxima Detection

The local maxima detection algorithm generates a local maxima image in which the pixels are marked as either BACKGROUND pixels or LOCAL_MAXIMUM pixels. The LOCAL_MAXIMUM pixels are part of the thinned ridge and serve as seed pixels to the recursive ridge following algorithm. To generate the local maxima image, the algorithm scans the chamfer image from left to right and from top to bottom, applying the local maximum test to each pixel. If a pixel passes the local maximum test, its corresponding location in the output image is marked as a LOCAL_MAXIMUM pixel. Otherwise the pixel is marked as a BACKGROUND pixel.

A pixel must pass at least one of two following tests to be declared a LOCAL_MAXIMUM pixel. The first test has two conditions: (1) the pixel's chamfer value must be strictly greater than the chamfer values of the two pixels above that pixel and the pixel two rows below that pixel, and (2) the pixel's chamfer value must be greater than or equal to the chamfer values of the neighboring pixels to the left, right, and bottom. The second test has three conditions: (1) the pixel's chamfer value must be strictly greater than the chamfer values of the two pixels toward the left and the pixel two columns toward the right, (2) the pixel's chamfer value must be greater than or equal to the chamfer values of the neighboring pixels above, below, and to the right, and (3) the neighboring pixel above the pixel has not already been declared to be a LOCAL_MAXIMUM pixel in the output image. This local maximum detection algorithm is illustrated in figure 24. Notice that second condition is the reason that the rows of the chamfered image must be scanned from top to bottom. A pixel's neighbor toward the top must have already been considered as possibly being a LOCAL_MAXIMUM pixel before the second condition can be applied to the pixel.

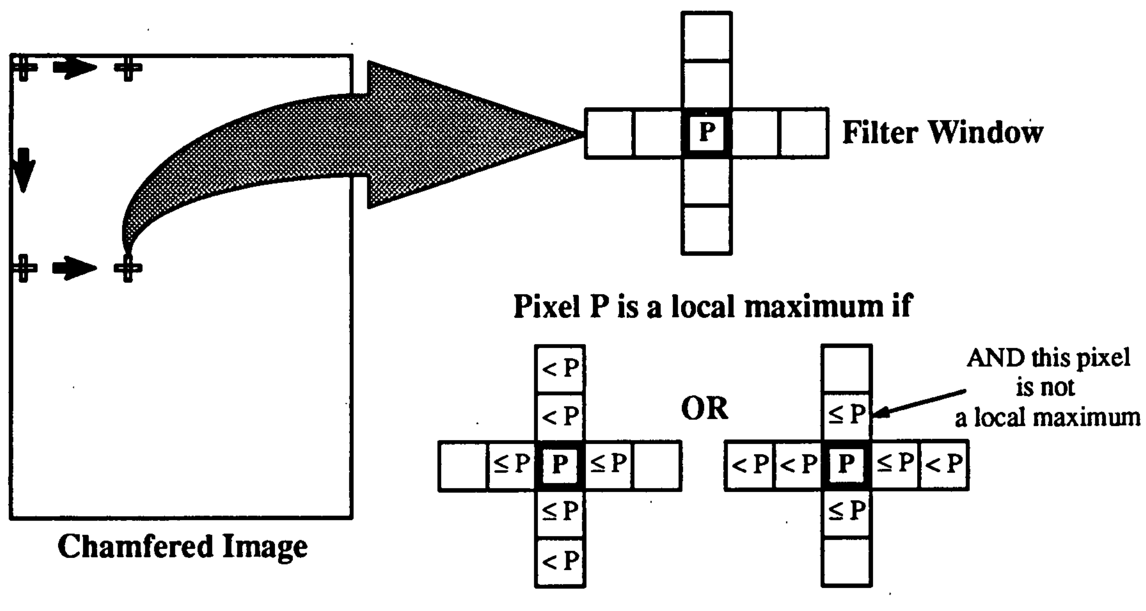


Figure 24. Filter for Detecting the Local Maxima Locations in a Chamfered Image

```
DETECT_LOCAL_MAXIMA[ C ]
for each row i in C from 3 to height_C – 3
    for each column j in C from 3 to width_C – 3
        if ((C(i–1, j) < C(i, j)) and (C(i+1, j) ≤ C(i, j))
            and (C(i, j–1) ≤ C(i, j)) and (C(i, j+1) ≤ C(i, j))
            and (C(i–2, j) < C(i, j)) and (C(i+2, j) < C(i, j)))
        {
            mark T(i, j) as a LOCAL_MAXIMUM
        }
        else if ((C(i–1, j) ≤ C(i, j)) and (C(i+1, j) ≤ C(i, j))
            and (C(i, j–1) < C(i, j)) and (C(i, j+1) ≤ C(i, j))
            and (C(i, j–2) < C(i, j)) and (C(i, j+2) < C(i, j))
            and T(i–1, j) is not marked as a LOCAL_MAXIMUM)
        {
            mark T(i, j) as a LOCAL_MAXIMUM
        }
        else
            mark T(i, j) as BACKGROUND

return T
```

5.1.3 Recursive Ridge Following

Recursive ridge following fills in the missing thin ridge pixels between the local maxima, using the local maxima pixels as starting pixels for the recursive algorithm. To find these starting pixels, the output image generated from local maxima detection is scanned to find pixels that are marked as LOCAL_MAXIMUM. As each local maximum pixel is found, it is processed by the recursive ridge following algorithm. Given a local maximum pixel, the recursive ridge following algorithm considers each neighboring pixel to check if that neighbor meets the conditions of being a candidate ridge pixel. If these conditions are met, a recursive call to the ridge following algorithm is made using that candidate pixel. A pixel that produces a candidate ridge pixel is referred to as the spawning pixel of that candidate (e.g., the local maximum pixel is the spawning pixel for any candidate ridge pixel found in searching its neighboring pixels). This recursion allows the exploration of candidate segments before committing to their inclusion as thin ridge segments. A thin ridge segment ends either with a local maximum pixel or with an intersection with another thin ridge. An example of recursive ridge following is illustrated in figure 25.

A call to the recursive ridge following algorithm must pass the position of the candidate ridge pixel being considered and the direction toward its spawning pixel. The candidate's

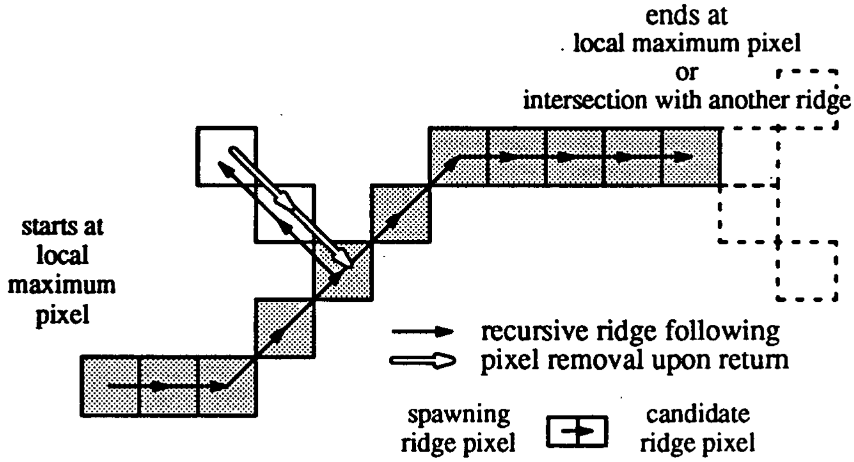


Figure 25. An Example of Recursive Ridge Following

position must include the image coordinate to allow for image boundary checking, and may include pixel pointers into the raw thinned image and the chamfered image to improve implementation efficiency. The pixel direction of the spawning pixel refers to the direction from which the current candidate pixel was discovered and is necessary to check for termination caused by intersecting another ridge. When this algorithm is first called, the candidate pixel is a local maximum and does not have a spawning direction. In this case, a null direction is passed in.

Upon entering the algorithm, the value of the corresponding pixel in the chamfer image is examined. If it is zero or greater than 14140, the algorithm returns a value of FALSE to indicate that the ridge has terminated and did not end on a local maximum or an intersection of ridges or to indicate that the ridge is too wide. These types of terminations will cause this pixel and the candidate pixels that are on this branch of recursion to be removed (in reverse order from that in which they were found) until a local maximum pixel is encountered.

Termination also occurs if the candidate pixel intersects another existing thin ridge. This terminating condition is tested by considering all eight neighboring pixels. If a neighboring pixel is marked as a RIDGE_PIXEL in the raw thinned image, further conditions are checked. These conditions ensure that the neighboring ridge pixel found is not part of a local section of ridge currently being followed. This is verified by considering the direction to the spawning pixel of this candidate pixel. If the direction to the neighboring ridge pixel is not closer than 90 degrees to the direction of the spawning pixel, the neighboring ridge pixel is considered to be from another ridge; hence the terminating condition of intersection with another ridge has been satisfied. These conditions are illustrated in figure 26. By ending in an intersection to another ridge, this branch of the recursion ended as an actual raw thinned ridge causing, the algorithm to return the Boolean value of TRUE to indicate that the spawning pixels of this ridge should be kept.

At this point in the algorithm, the Boolean value that keeps track of the validity of the candidate ridge is initialized. If this candidate thin ridge pixel is a local maximum the validity value is set to TRUE, otherwise it is set to FALSE. If the candidate pixel is not a local maximum, it is marked in the raw thinned image as a RIDGE pixel. This marking will be removed in the recursion if this branch of the recursion is determined not to be an actual thin ridge.

Next, the four diagonal neighbor pixels are considered as candidate ridge pixels. In order to be a candidate ridge pixel, a pixel's chamfer value must be larger than the chamfer values of neighboring pixels on either side of the direction of travel from the spawning pixel. The actual conditions for being a diagonal candidate ridge pixel are illustrated in figure 27. If the condition is TRUE, a recursive call to the ridge following algorithm is made, passing in the position of this candidate ridge pixel and the direction to its spawning pixel. When the recursive call returns to this point in the algorithm, the current ridge validity value is updated by a "logical or" with the returned Boolean value. This is done so that the algorithm knows if any of the recursive branches from this candidate pixel ended as an actual thin ridge.

After the four diagonal neighboring pixels are considered, the rectilinear neighbor pixels are considered as candidate ridge pixels. As with the diagonal neighbor pixels, a rectilinear neighbor pixel is a candidate ridge pixel if the chamfer value of that pixel is larger than the chamfer values of the neighboring pixels on either side of the direction from the spawning pixel. The actual conditions for being a rectilinear candidate ridge pixel are illustrated in

If there exists a neighboring pixel marked as a RIDGE or a LOCAL MAXIMUM
and the direction toward the pixel which spawned this candidate pixel
is any one of the directions indicated by the arrows
then the candidate pixel has intersected with another ridge. (return TRUE)
otherwise the candidate pixel has <u>not</u> intersected with another ridge (return FALSE)

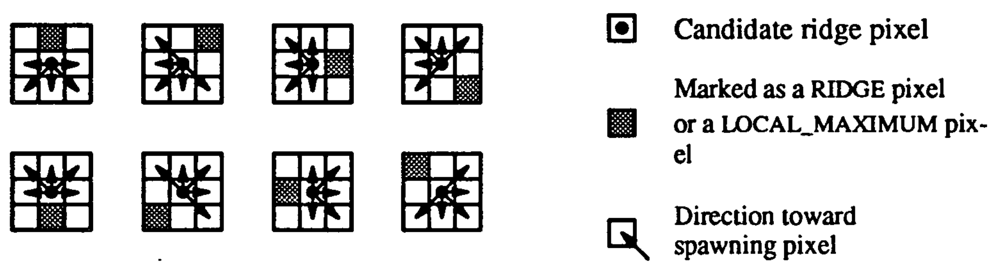
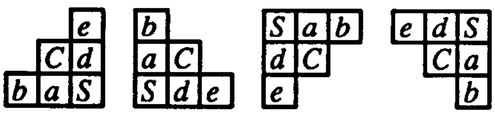


Figure 26. The Eight Conditions for Recursion Termination by Ridge Intersection

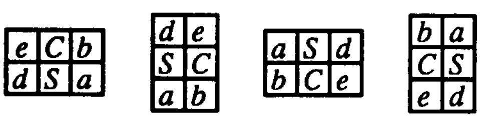


chamfered image values

if
pixels C, a, & d are marked as BACKGROUND pixels
and
$2C \geq (a + b)$ **and** $2C \geq (d + e)$
then
Pixel C is a candidate ridge pixel spawned by pixel S

Figure 27. Diagonal Candidate Ridge Pixels

figure 28. Again, as with the diagonal neighboring pixels, if the condition is to TRUE, a recursive call to the ridge following algorithm is made, passing in the position of this candidate ridge pixel and the direction to its spawning pixel. When the recursive call returns to this point in the algorithm, the current ridge validity value is updated by a "logical **or**" with the returned Boolean value. This is done so that the algorithm knows if any of the recursive branches from this candidate pixel ended as an actual thin ridge.



chamfered image values

if
pixels C, b & e are marked as BACKGROUND pixels
and
$(C + S) \geq (a + b)$ **and** $(C + S) \geq (d + e)$
then
Pixel C is a candidate ridge pixel spawned by pixel S

Figure 28. Rectilinear Candidate Ridge Pixels

At this point in the recursive ridge following algorithm, appropriate clean-up is done. If the above processing for the candidate pixel has resulted in the ridge validity value being set to TRUE, this candidate pixel is part of an actual ridge and will be kept in the raw thinned image. Otherwise, it will be removed by resetting the pixel in the raw thinned image to BACKGROUND. If the candidate pixel is an actual ridge pixel and was marked as a LOCAL_MAXIMUM, it is now downgraded to be simply a ridge pixel by marking it as RIDGE in the raw thinned image.

```
FOLLOW_RIDGE[ i, j, direction ]
    ** This process returns a Boolean value indicating the status of the followed ridge
    ** The chamfered image C and the thinned image T must be globally addressable from
       within this process
    if ((C(i, j) = 0)                       ** No longer on a ridge or
        or (C(i, j) >14140))                ** ridge is too wide, remove the candidate ridge pixel
        return FALSE

    if (T(i, j) intersects with another ridge)            ** Keep this ridge (see figure 26)
        mark T(i, j) as a RIDGE pixel
        return TRUE

    ** If the candidate pixel is a local maximum, keep it labeled as such, for now
    if (T(i, j) is marked as a LOCAL_MAXIMUM pixel)
        status = TRUE
    else                                    ** Otherwise label it as a potential ridge pixel
        mark T(i, j) as a RIDGE pixel
        status = FALSE
    ** Consider whether the diagonal neighbors are ridge pixels
    if (pixel (i–1, j+1) is a candidate ridge pixel)      ** Top right (see figure 27)
        if (FOLLOW_RIDGE[ i–1, j+1, BOTTOM_LEFT ] = TRUE)
            status = TRUE
    if (pixel (i+1, j–1) is a candidate ridge pixel)      ** Bottom right (see figure 27)
        if (FOLLOW_RIDGE[ i+1, j–1, TOP_LEFT ] = TRUE)
            status = TRUE
    if (pixel (i+1, j–1) is a candidate ridge pixel)      ** Bottom left (see figure 27)
        if (FOLLOW_RIDGE[ i+1, j–1, TOP_RIGHT ] = TRUE)
            status = TRUE
    if (pixel (i–1, j–1) is a candidate ridge pixel)      ** Top left (see figure 27)
        if (FOLLOW_RIDGE[ i–1, j–1, BOTTOM_RIGHT ] = TRUE)
            status = TRUE
```

```
** Consider if the rectilinear neighbors are ridge pixels
if (pixel (i–1, j) is a candidate ridge pixel)                ** Top (see figure 28)
    if (FOLLOW_RIDGE[ i–1, j, BOTTOM ] = TRUE)
        status = TRUE
if (pixel (i, j+1) is a candidate ridge pixel)                ** Right (see figure 28)
    if (FOLLOW_RIDGE[ i, j+1, LEFT ] = TRUE)
        status = TRUE
if (pixel (i+1, j) is a candidate ridge pixel)                ** Bottom (see figure 28)
    if (FOLLOW_RIDGE[ i+1, j, TOP ] = TRUE)
        status = TRUE
if (pixel (i, j–1) is a candidate ridge pixel)                ** Left (see figure 28)
    if (FOLLOW_RIDGE[ i, j–1, RIGHT ] = TRUE)
        status = TRUE

if (T(i, j) is marked as a LOCAL_MAXIMUM pixel)
    mark T(i, j) as a RIDGE pixel
else if (status = FALSE)          ** Otherwise if a ridge wasn't found above, remove it
    mark T(i, j) as a BACKGROUND pixel

return status
```

5.1.4 Summary

Input

I	Cleaned thresholded fingerprint image

Output

C	Chamfered image
T	Thinned image

SECTION 6

CURVE EXTRACTION

After the fingerprint ridges have been thinned by the previous procedure, they must be represented by abstract data structures. These data structures, called "curves," are used to encode the ridges efficiently for transmission. Curve extraction derives curves from the ridges in a thinned fingerprint image.

The thinning process may leave behind certain artifacts that are extraneous to the thinned ridges. For example, the ridges may have areas that are more than one pixel wide (figure 29) or there may be single-pixel "nubs" that do not represent true ridge structures. These artifacts are removed to convert the thinned ridges to single-pixel wide ridges prior to curve extraction.

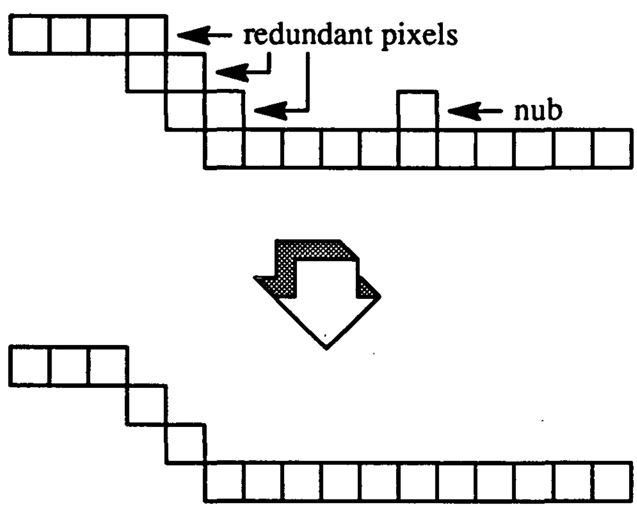


Figure 29. Conversion to Single-Pixel Wide Ridges

After all artifacts have been removed and the ridges are guaranteed to be one pixel wide, the individual curves are extracted from the ridges. A curve is an ordered list (or other structure) of points that correspond to the pixels along a thinned ridge. An individual curve must be a simple curve that extends between an endpoint or bifurcation at each end, with no intervening bifurcation. The curves must preserve the minutiae, i.e., the terminations and bifurcations, of the thinned ridges from which they are extracted. When a ridge terminates, the curve extracted from it must contain the termination point as an endpoint. On the other hand, when multiple ridges meet at a bifurcation, the extracted curves must all contain the bifurcation point as an endpoint (figure 30). This shared endpoint ensures that the reconstructed fingerprint ridges based on these curves will intersect at the same point.

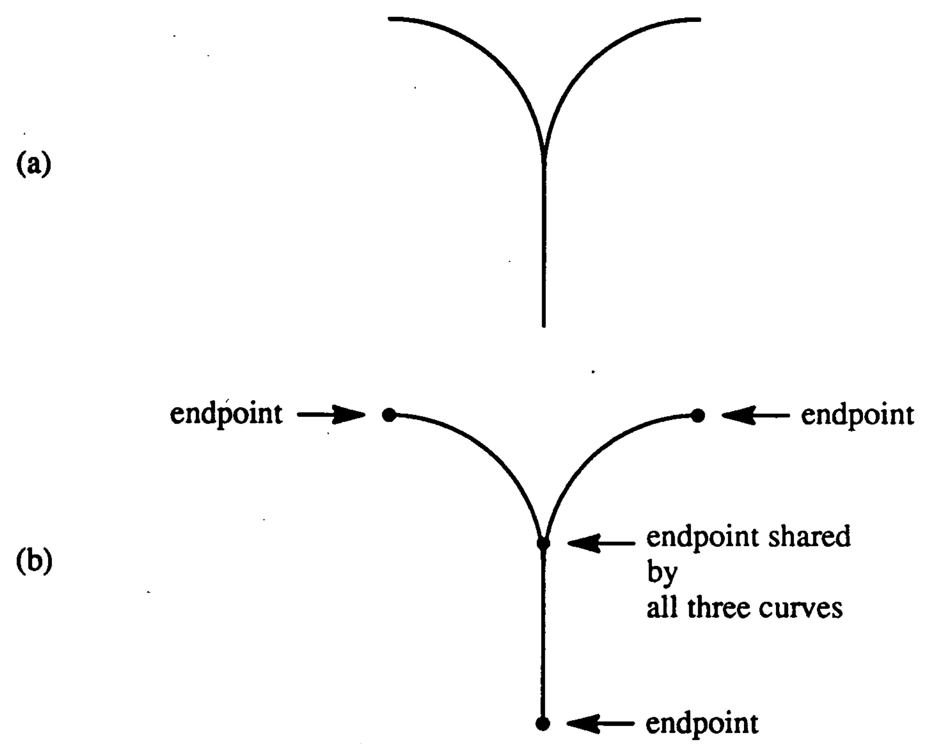


Figure 30. Curve Extraction at a Bifurcation. (a) A ridge bifurcation. (b) The three curves extracted from the ridge bifurcation, showing the endpoints and shared endpoint.

Throughout this section, the term *neighbors* refers to the eight-neighbors of a pixel, i.e., the eight adjacent pixels above, below, to the left of, to the right of, and to the diagonals of the pixel. A *neighbor* of a ridge (black) pixel is another ridge pixel that is one of its neighbors.

In the pseudocode contained in this section, the construct "**switch on** n" is used. Such a construct is followed by several blocks of code, each headed by a statement of the form "**case** m_i." If one of the m_i matches n, then the block of code headed by that matching case statement is executed. In the event that none of the m_i match n, none of the case blocks is executed.

6.1 ALGORITHM DESCRIPTION

Curve extraction proceeds in two phases: a pre-processing conversion of the raw thinned fingerprint image to a thinned fingerprint image guaranteed to contain only single-pixel wide ridges, followed by curve extraction. In the pre-processing stage, the locations on the ridges in the raw thinned fingerprint image that are not one pixel wide or that are inconsequential protrusions, or nubs, are first detected using masks and removed. The individual curves are then extracted from the resulting thinned fingerprint image.

CURVE_EXTRACTION[*IMAGE*]

    ** This function has the side effect of modifying *IMAGE*

CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES[*IMAGE*] ** Section 6.1.1
    ** *IMAGE* now contains a thinned fingerprint
curve_set = EXTRACT_CURVES[*IMAGE*] ** Section 6.1.2
return *curve_set*

6.1.1 Conversion to Single-Pixel Wide Ridges

Conversion of the thinned ridges to single-pixel wide ridges ensures that the curve extraction algorithm can make certain assumptions about the connectivity of ridge pixels. Thus, these assumptions simplify the curve extraction algorithm. Once the conversion algorithm has ensured that only single-pixel wide curves exist in an image, the assumptions for any given ridge pixel can be enumerated based on the number of neighbors of that pixel.

1. The pixel has no neighbor. Assume that the ridge consists of only one pixel.
2. The pixel has one neighbor. Assume that the pixel is a ridge endpoint.
3. The pixel has two neighbors. Assume that the ridge passes from one neighbor, through the pixel, and then through the other neighbor.
4. The pixel has more than two neighbors. Assume that the pixel is an intersection (bifurcation) point and that there is a ridge intersecting this pixel through each neighbor.

The above assumptions dictate which pixels must be removed from the raw thinned fingerprint image before curve extraction can take place (figure 29). First, any single pixel that protrudes from a natural line of pixels (a nub) must be removed so that it does not form a false bifurcation. Second, any ridge pixel that can be removed from the raw thinned fingerprint image without changing the topology (connectivity) of the fingerprint ridges must be removed. (Note that this implies that a pixel that has neighbors directly above, below, to the left, and to the right cannot be removed; the removed pixel would constitute a one-pixel valley.) The pixels to be removed are identified through the use of a set of masks.

Before the mask sets can be used, one other artifact of thinning must be removed. The thinning process will occasionally create a white pixel whose four-connected neighbors (top, bottom, left, and right) are all black. The image is scanned (left-to-right, top-to-bottom) and all such isolated white pixels are changed to black. The mask sets are then applied to the image.

6.1.1.1 Application of the Mask Sets

A set of masks (figure 31) has been defined that identify pixels that are nubs and that therefore should be removed from a thinned fingerprint image. The thinned fingerprint image is scanned (left-to-right, top-to-bottom) and at each ridge pixel, every nub mask is applied. If a mask matches the black and white configuration of a pixel and its surrounding pixels, then that pixel is removed from the thinned fingerprint image and the scan moves to the next pixel. A set of masks has also been defined (figure 32) that identify pixels that are not nubs but that can be removed from a ridge without changing its topology. The thinned fingerprint image is again scanned (left-to-right, top-to-bottom) and this time at each ridge pixel every topology mask is applied. Again, if a mask matches a pixel and its surrounding pixels, then that pixel is removed from the thinned fingerprint image and the scan moves on. (As an optimization, a mask set need not be applied at a ridge pixel if the number of neighbors of that pixel is not consistent with any mask in the set.)

The nub masks and the topology masks are applied in turn to the entire image and this process is repeated until no further pixels are removed in a complete application of all the nub and topology masks. At this point, the remaining ridges are one pixel wide, with no extraneous pixels, and the assumptions in section 6.1.1 about them are valid.

Although, conceptually, the mask sets are applied across the entire image, other implementations can be used to improve the algorithm's efficiency. One possible improvement is to partition the image into blocks and then to apply the masks to the image pixels on a per-block basis. Note that the blocks are used only to select the pixels to be tested and do not restrict the pixels to which the masks are applied. If, during any pass, no pixels in a particular block are removed by the application of either mask set, then that block need not be considered again. The overall algorithm for conversion to single-pixel wide ridges would then terminate when no blocks are left to consider.

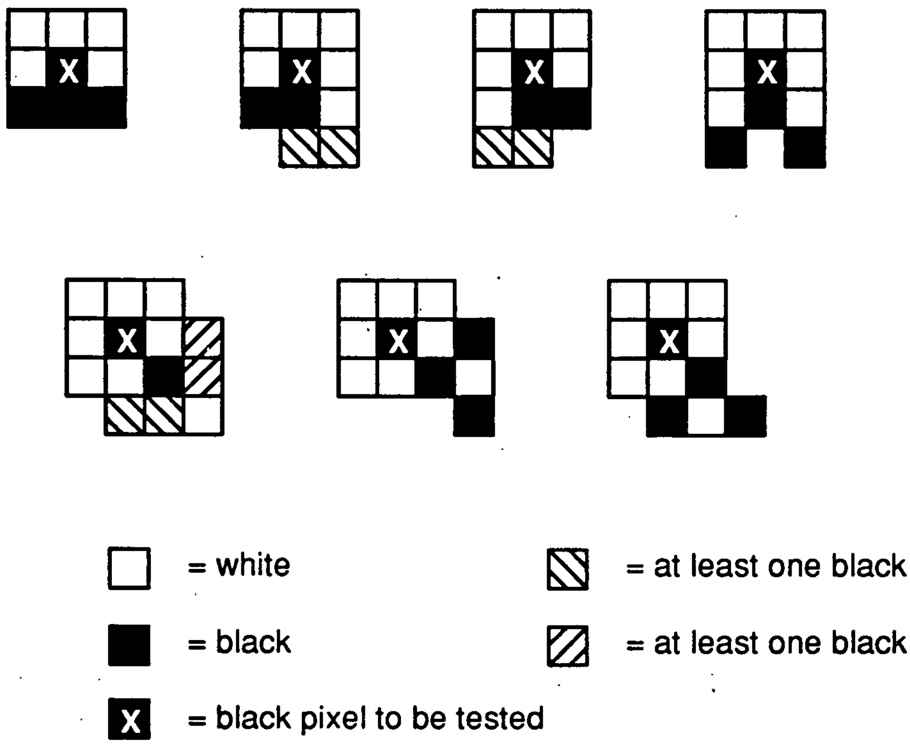


Figure 31. Masks Used to Remove Nubs from Ridges

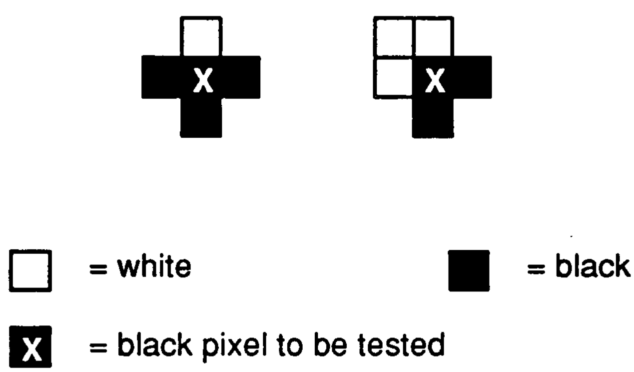


Figure 32. Masks Used to Remove Non-Topology-Changing Pixels from Ridges

```
CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES[ IMAGE ]

    ** This function has the side effect of modifying IMAGE

    ** Remove isolated white pixels
    for each pixel (i, j) in IMAGE
        if (IMAGE(i, j) = WHITE
                and IMAGE(i+1, j) = BLACK and IMAGE(i–1, j) = BLACK
                and IMAGE(i, j+1) = BLACK and IMAGE(i, j–1) = BLACK)
            IMAGE(i, j) = BLACK

    ** Initialize the flag that indicates whether any pixels were removed in the current pass
    pixel_set_to_white = TRUE
    ** Loop until no pixels are removed in a pass over the image
    while (pixel_set_to_white = TRUE) do
    {
        ** Reset flag to show that no pixels have yet been removed in this pass
        pixel_set_to_white = FALSE
        ** Apply the nub masks (figure 31)
        for each pixel (i, j) in IMAGE
            if (IMAGE(i, j) = BLACK)
                if (APPLY_MASKS[i, j, nub_mask_set, IMAGE] = TRUE)
                {
                    ** The current mask matched, so remove this pixel
                    IMAGE(i, j) = WHITE
                    pixel_set_to_white = TRUE          ** Flag that a pixel was removed
                }
        ** Apply the non-topology-changing masks (figure 32)
        for each pixel (i, j) in IMAGE
            if (IMAGE(i, j) = BLACK)
                if (APPLY_MASKS[i, j, topology_mask_set, IMAGE] = TRUE)
                {
                    ** The current mask matched, so remove this pixel
                    IMAGE(i, j) = WHITE
                    pixel_set_to_white = TRUE          ** Flag that a pixel was removed
                }
    }
    return
```

6.1.1.2 Mask Application

To apply a mask to a ridge pixel, the mask position labeled "**X**" is aligned with the ridge (black) pixel being tested (figures 31 and 32). The surrounding pixels are then compared to the mask pixels. For the surrounding pixels to match the mask, a black mask position must correspond to a ridge pixel and a white mask position must correspond to a background or valley pixel. Any pixel corresponding to a position not existing in the mask may be black or white. (If a portion of the mask falls outside of the image, those mask positions must be white for a valid match.) Finally, if left-to-right crosshatch mask positions occur, at least one of them must correspond to a black pixel. Also, at least one right-to-left crosshatch mask position must match a black pixel, if such mask elements occur. For each ridge pixel that will be tested, every mask must be applied in each of its four possible orientations (90 degree rotations). If any mask in any orientation matches the pixel and its surrounding pixels, that pixel is removed (set to white). The scan then proceeds to the next pixel.

APPLY_MASKS[*i*, *j*, *mask_set*, ***IMAGE***]

** This function has the side effect of modifying ***IMAGE***

```
for mask in mask_set
    for rotation in (0, 90, 180, 270) degrees
        if (mask at rotation matches IMAGE(i, j) and its surrounding pixels)
            return TRUE
return FALSE
```

Although, conceptually, each mask in the mask set is applied to a given ridge pixel, this need not be done in practice. The application of the set of masks to a ridge pixel can be made more efficient by first checking the number of neighbors of that pixel and only applying those masks that have the same number of neighbors for the pixel to be tested (figures 31 and 32). For example, when applying the nub mask set to a pixel that has exactly three neighbors, only the top left mask of figure 31 need be applied.

6.1.2 Curve Extraction

Conversion of the thinned ridges to single-pixel wide ridges ensures that the curve extraction algorithm can make certain assumptions about the connectivity of ridge pixels. The most important of these assumptions is that, given a ridge pixel, a curve exists that connects that ridge pixel to each of its eight-neighbors that is also a ridge pixel, if such neighbors exist (figure 33). A consequence of this connectivity is that a ridge can be followed from any of its pixels. If a ridge pixel has no neighbors, then it forms its own (one-pixel) ridge. If a ridge pixel has only one neighbor, then the pixel is a ridge endpoint; the ridge can be followed from this endpoint. If a ridge pixel has two neighbors, then the two halves of the ridge can be followed, one through each neighbor pixel, and the halves then connected to form the full ridge. If a ridge pixel has more than two neighbors, then it is a bifurcation point. In this case, although the algorithm could follow all the intersecting ridges from this bifurcation point, the algorithm scan instead skips this point. Because the algorithm scans the image searching for ridge points, it is guaranteed that it will find another point on every ridge that intersects the bifurcation point. Using these other points, the intersecting ridges can be followed using either singly connected endpoint processing or doubly connected midpoint processing, as described above. Therefore, bifurcation points can be skipped safely when encountered by the scan (figure 34).

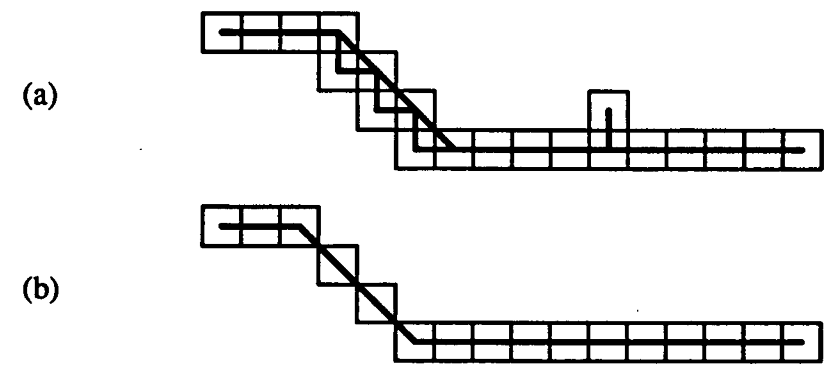


Figure 33. Curve Following for a Non-Thinned Ridge and a Thinned Ridge Based on Connectivity Assumptions. (a) The non-thinned ridge has many extraneous curves. (b) The thinned ridge has a single curve.

As a ridge is followed to extract it from the thinned fingerprint image, a bifurcation point may be reached. A bifurcation point is assigned the label BIFURCATION, and the curves meeting (branching) there are initialized as two-pixel "seed" curves (see figure 35). Each seed curve consists of the BIFURCATION point where the curves meet and the next point on the curve, which is assigned the label SEED. As the seed curves are created, they are stored on a "to-do" list. After the original curve is completely extracted from the image, these seed curves are taken from the to-do list and are also extracted before the scan continues across the image for the next initial curve pixel. Each of these processing steps is explained in more detail in the following sections.

To extract the individual ridge curves from the thinned fingerprint image (consisting of single-pixel wide ridges) the algorithm scans the image left-to-right, top-to-bottom until it encounters a ridge pixel that has not been labeled BIFURCATION. (If the pixel were labeled BIFURCATION, this would imply that the pixel had been processed previously, but had been left in the image because multiple curves branch from it. See section 6.1.2.5.) If the ridge pixel

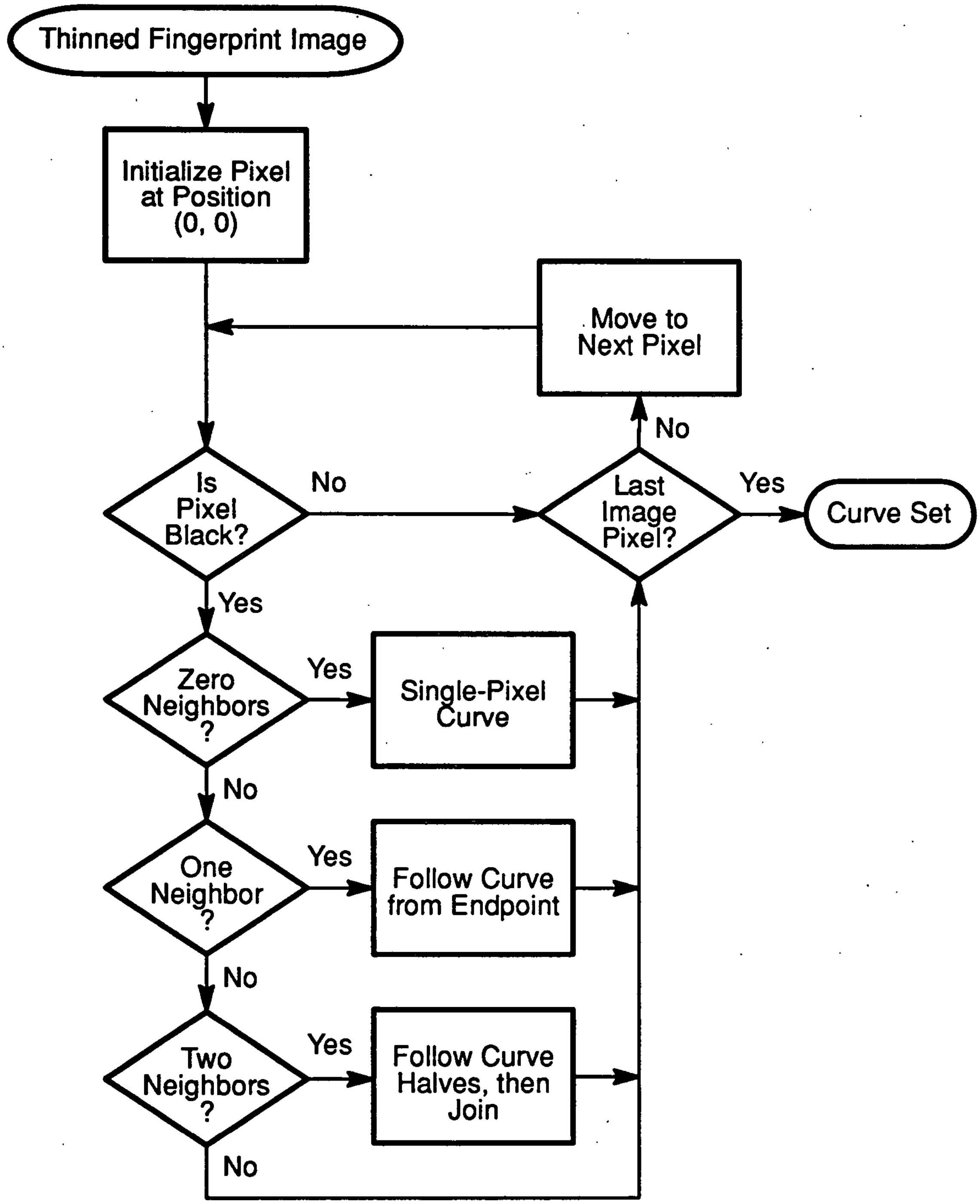


Figure 34. Flowchart of Overall Control of Curve Extraction

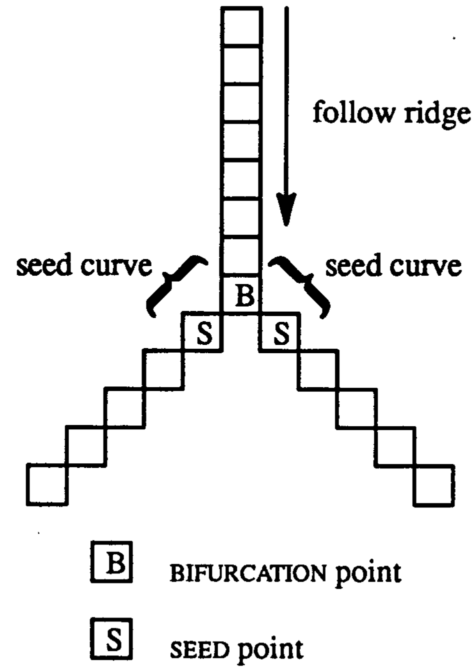


Figure 35. Seed Curves at a BIFURCATION Point

has zero, one, or two neighbors, a curve is initialized at that pixel and followed, thereby extracting it from the image. If the pixel has more than two neighbors, it is a bifurcation pixel and is skipped by the scan.

In the algorithms for curve extraction, various labels are used on the ridge pixels. These labels are associated with the image pixels themselves and not with the representations of the pixels that are stored in the curve structures.

```
EXTRACT_CURVES[ IMAGE ]
    ** This function has the side effect of modifying IMAGE

    ** IMAGE, seed_index, curve_set, and to_do are available globally to the subroutines
       under EXTRACT_CURVES
seed_index = 0
curve_set = EMPTY
to_do = EMPTY
for i from 1 to height
    for j from 1 to width
        ** INITIALIZE_BRANCHES (described in section 6.1.2.5) may have labeled this
           pixel BIFURCATION
        if (IMAGE(i, j) = BLACK and IMAGE(i, j) is not labeled BIFURCATION)
            if (number of neighbors of IMAGE(i, j) < 3)          ** See page 60
                curve = INITIALIZE_AND_FOLLOW_CURVE[i, j]        ** Section 6.1.2.1
                put curve into curve_set
                FOLLOW_TO_DO_LIST[]                              ** Section 6.1.2.6
if (curve_set = EMPTY)
    exit                                                  ** Error: No curves found
return curve_set
```

6.1.2.1 Curve Initialization

For each curve to be extracted, a list (or other structure) is created to hold the curve points. Given a ridge pixel found in the scan with zero, one, or two neighbors, the curve extraction is initialized by putting that pixel on the point list and removing it from the fingerprint image. If the pixel has no neighbors, the extraction of the curve is complete. If the initial pixel has one neighbor, the extraction continues by following the curve as described in section 6.1.2.2. Otherwise, the curve is initialized (and extracted) in two pieces, which are then joined (see figure 36).

Given a curve initialized with a ridge pixel that has two neighbors, one neighbor is arbitrarily chosen and the initialization for the first half of the curve continues by adding that neighbor to the point list. If the chosen neighbor pixel has been labeled BIFURCATION, the extraction of the first half of the curve is finished. Otherwise, the chosen neighbor pixel is removed from the image and extraction of the first half of the curve continues by following the curve as described in section 6.1.2.2. After the first half of the curve has been extracted, the second half is initialized and extracted.

Because the first half of the curve may have looped back to the initial pixel (see figure 37), before initializing and extracting the second half of the curve the algorithm first

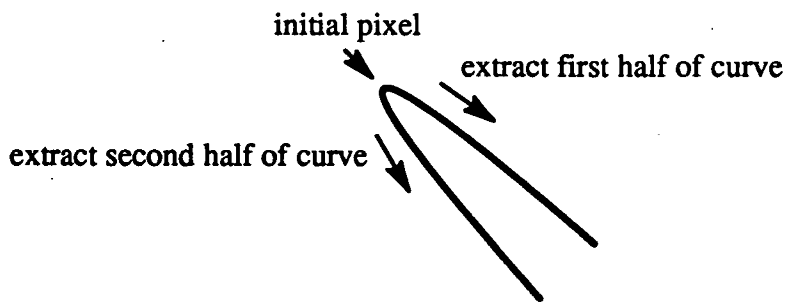


Figure 36. An Example of an Extraction of a Curve in Two Halves

checks that there still is one remaining neighbor of the initial pixel (that was not the first neighbor selected). If there is no remaining neighbor, the first half of the curve must have looped back to the initial pixel. In this case, the initial pixel is added once again to the curve (this time it appears at the end of the curve) to complete the loop, and the extraction of the full curve is complete. Otherwise, the extraction for the second half of the curve is initialized by putting the neighbor pixel on a new point list. If the neighbor pixel has been labeled BIFURCATION, the extraction of the second half of the curve is finished. Otherwise, the neighbor pixel is removed from the image and the extraction continues by following the curve as described in section 6.1.2.2. If the second half of the curve exists, the curve is completed by joining the halves together to form a single curve through the initial pixel. To join them, care must be taken so that the order of the points in the joined curve is the same as the order of the pixels along the curve in the image. Typically, the points in one half of the curve must be reversed and the halves then joined at the ends that were adjacent in the image.

INITIALIZE_AND_FOLLOW_CURVE[*i*, *j*]

```
    ** This function has the side effect of modifying IMAGE

curve = EMPTY
curve_2 = EMPTY
put IMAGE(i, j) onto curve
IMAGE(i, j) = WHITE                      ** Remove this pixel from IMAGE
switch on number of neighbors of IMAGE(i, j)
    case 0
        ** No neighbors, so curve ends here
        return curve
```

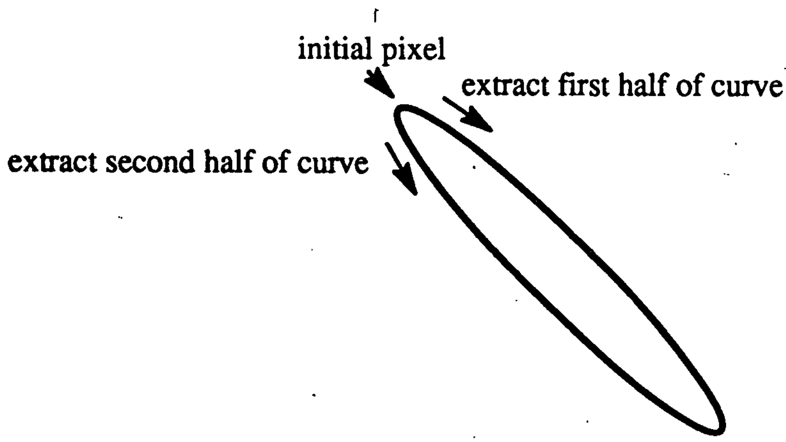


Figure 37. Looped Curve. The figure shows an example of an extraction of the first half of a curve that loops back to the initial pixel so that the second half extraction need not be done.

```
    case 1
        ** One neighbor, so add it to curve and continue following curve
        curve = FOLLOW[curve]                                   ** Section 6.1.2.2
        return curve
    case 2
        ** Two neighbors, so curve has two halves. Follow first half of curve
        neighbor_1 = a neighbor of IMAGE(i, j)
        put neighbor_1 onto curve
        if (neighbor_1 is not labeled BIFURCATION)
            neighbor_1 = WHITE                  ** Remove this pixel from IMAGE
            curve = FOLLOW[curve]                               ** Section 6.1.2.2
        ** Check for second half of curve
        if (no neighbors of IMAGE(i, j) exist
                or only neighbor of IMAGE(i, j) is neighbor_1)
            ** curve looped back on itself
            put IMAGE(i, j) onto curve
            return curve
        else
            ** Follow second half of curve
            neighbor_2 = neighbor of IMAGE(i, j) that is not neighbor_1
            put neighbor_2 onto curve_2
            if (neighbor_2 is not labeled BIFURCATION)
                neighbor_2 = WHITE              ** Remove this pixel from IMAGE
                curve_2 = FOLLOW[curve_2]                       ** Section 6.1.2.2
            reverse curve_2
            curve = append(curve, curve_2)
            return curve
end
```

6.1.2.2 Curve Following

Given a curve that has been initialized with one or more points, the ridge that the curve describes must be followed in the image to a termination or bifurcation and the pixels of the ridge added to the curve. Each time a point is added to the curve, the curve following routine is called again on the updated curve until the end of the curve is reached. Note that although this process is conceptually recursive, non-recursive implementations are also possible. The action taken at each invocation of the curve following routine depends on the number of neighbors of the last point on the curve (the point most recently added to the curve). First, the neighbors of the last curve point are counted in the thinned fingerprint image. This count of the neighbors should: (a) include all neighbors that are ridge pixels, whether or not labeled BIFURCATION, (b) not include (if it exists) the point before the last point in the curve, and (c) not include any neighbor labeled SEED if the last curve point is also labeled SEED and if the seed index of the neighbor matches that of the last curve point. (If the pixel were labeled SEED, this would imply that the pixel had been processed previously, but had been left in the image because it is part of an initialized, or "seeded," curve. See section 6.1.2.5.) (For Condition b, note that the point before the last point may still be in the image if it was previously labeled BIFURCATION.) If there are no neighbor points that match these conditions, the curve is complete. If there is one such neighbor point, it is added to the curve. If this neighbor point has been previously labeled BIFURCATION, the curve is complete. Otherwise, the neighbor point is removed from the image and the curve following routine is invoked on the updated curve. If there are two or more neighbor points that match these conditions, the action of the curve following routine depends on the number of possible branches from the current point.

FOLLOW[*curve*]

    ** This function has the side effect of modifying ***IMAGE***

last_point = last point on *curve*
previous_point = point before last point on *curve*
n_neighbors = COUNT_NEIGHBORS_FOR_FOLLOWING[*curve*]
switch on *n_neighbors*
    case 0
        ** No neighbors, so *curve* ends here
        return *curve*
    case 1
    {
        ** One neighbor, so add it to *curve* and continue following *curve*
        neighbor = neighbor of *last_point* that is not *previous_point*

```
        put neighbor onto curve
        if (neighbor is not labeled BIFURCATION)
            neighbor = WHITE                          ** Remove this pixel from IMAGE
            curve = FOLLOW[curve]
        return curve
    }
    case >1
    {
        ** More than one neighbor, so continue the extraction of curve based on the
            number of possible branches from last_point of curve
        possible_branches = FIND_POSSIBLE_BRANCHES[curve]        ** Section 6.1.2.3
        switch on number of possible_branches                    ** Section 6.1.2.4
        {
            case 0
                ** No possible branches, so curve ends here
                return curve
            case 1
                ** One possible branch, so continue following curve down that branch
                neighbor = first element of possible_branches
                put neighbor onto curve
                if (neighbor is not labeled BIFURCATION)
                    neighbor = WHITE                  ** Remove this pixel from IMAGE
                    curve = FOLLOW[curve]
                return curve
            case >1
                ** Multiple possible branches, so initialize them and end curve here
                    (see section 6.1.2.5)
                INITIALIZE_BRANCHES[last_point, possible_branches]
                return curve
        }
    }
end
```

```
COUNT_NEIGHBORS_FOR_FOLLOWING[ curve ]
last_point = last point on curve
previous_point = point before last point on curve
n_neighbors = 0
for neighbor in the eight-neighbors of last_point
    if (neighbor = BLACK
            and neighbor is not the same point as previous_point
            and (last_point is not labeled SEED or neighbor is not labeled SEED
                    or seed_index of last_point ≠ seed_index of neighbor))
        increment n_neighbors
return n_neighbors
```

6.1.2.3 Finding Possible Branches

The possible branches from a point with two or more neighbors are not always all of the neighbors of that point for three reasons. First, the algorithm does not branch to the previous point on the curve, which may still exist in the thinned fingerprint image. Second, the algorithm does not branch to any pixel that is labeled SEED. Third, the algorithm does not branch diagonally to a neighbor if it can be reached by first branching through a horizontal or vertical neighboring ridge pixel. Thus, the neighbors to which the algorithm can branch are: (a) the horizontal or vertical neighbors of the point that are not labeled SEED, and (b) the diagonal neighbors of the point that are not labeled SEED and that are not neighbors of a point identified in (a). (See figure 38.) The effect of these rules is to prevent unnecessary or unnatural branching (figure 39). The outline of FIND_POSSIBLE_BRANCHES shown below is one possible implementation of the branch finding and counting.

(a) (b)

Figure 38. Branches from point **X**. (a) The algorithm can branch to point **A** and to point **B** if neither is labeled SEED. (b) The algorithm can branch to point **B** only if it is not labeled SEED and if point **A** is labeled SEED (and is therefore not branched to).

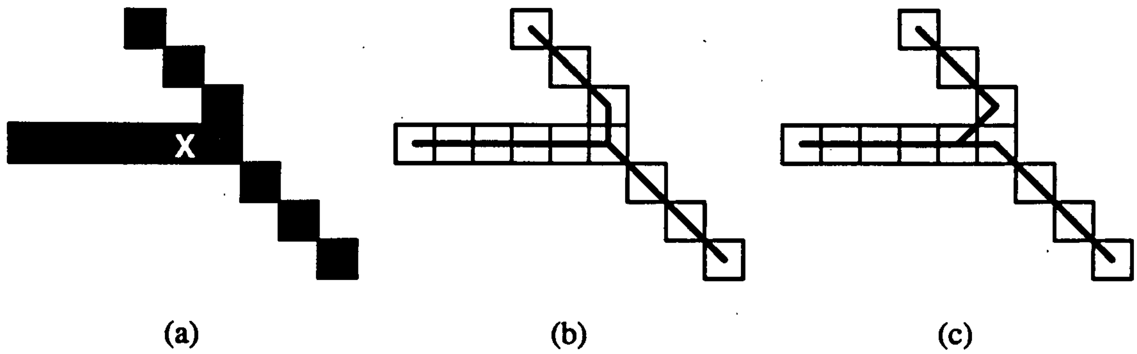


Figure 39. Possible Branch Counting Example. The rules for counting the possible branches from point **X** in the ridge shown in (a) generate the curves shown in (b) instead of those shown in (c).

```
FIND_POSSIBLE_BRANCHES[ curve ]

last_point = last point on curve
previous_point = point before last point on curve
possible_branches = EMPTY
for neighbor in the eight-neighbors of last_point
    if (neighbor = BLACK
            and neighbor is not the same point as previous_point
            and neighbor is not labeled SEED)
        add label POSSIBLE to neighbor
for neighbor in the eight-neighbors of last_point
    if (neighbor is labeled POSSIBLE
            and (neighbor is horizontal from last_point
                or neighbor is vertical from last_point
                or (neighbor is diagonal from last_point
                    and neither eight-neighbor of last_point touching neighbor
                        is labeled POSSIBLE)))
        put neighbor onto possible_branches
for neighbor in the eight-neighbors of last_point
    if (neighbor is labeled POSSIBLE)
        remove label POSSIBLE from neighbor
return possible_branches
```

6.1.2.4 Continued Curve Following Based on Number of Possible Branches

Given a point with two or more neighbors as described in section 6.1.2.2, the process used in following the curve depends on the count of possible branches from that point. If no branches are possible, the curve is complete. If only one branch is possible, the neighbor point corresponding to that branch is added to the curve. If that neighbor point has been labeled BIFURCATION, the curve is complete. Otherwise, the neighbor point is removed from the image and the curve following routine is invoked on the updated curve. Finally, if two or more branches are possible, the current point is a true bifurcation point; it will be labeled BIFURCATION and branches will be initialized from it.

6.1.2.5 Initializing Branches at a True Bifurcation Point

Given a curve with two or more branches possible from its last point (a true bifurcation point), the algorithm initializes or "seeds" new curves from that point and then ends the current curve. All seeds from this point are also labeled with the same seed index, which is unique for each set of seeds. To initialize the new curve seeds, the seed index is first incremented. (The seed index is initialized to 0 before processing a fingerprint.) The following process is then repeated for each possible branch found that is not already labeled BIFURCATION. First, a new curve is initialized with the last point of the original curve. Second, a neighbor point that is a possible branch is added to the curve (but not removed from the image). Third, this neighbor point is labeled SEED and is also labeled with the current seed index. Finally, this initialized seed curve is put onto a list of curves to be processed: the "to-do" list. This process is repeated until all of the possible branches from the last point of the original curve have been processed and the resulting seed curves have been added to the to-do list. The last point on the original curve is labeled BIFURCATION, but is not removed from the fingerprint image. The original curve is then complete.

Note that one possible implementation of seed labeling and indexing is through the use of an auxiliary (seed) array of the same size as the fingerprint array. The locations of the seed array can be initialized to 0 before the fingerprint is processed. If a point is to be labeled SEED, its index can be entered into the corresponding location in the seed array. To check if a point is labeled SEED, then, the algorithm simply accesses the location in the seed array that corresponds to the point. If that location is non-zero, then the point is a seed and the value of its location in the seed array gives its seed index.

```
INITIALIZE_BRANCHES[ last_point, possible_branches ]
increment seed_index
for branch in possible_branches
    ** branch is a neighbor point of last_point
    if (branch is not labeled BIFURCATION)
        curve = EMPTY
        put last_point onto curve
        put branch onto curve
        label branch as SEED
        label branch with seed_index
        put curve onto to_do
label last_point as BIFURCATION
return
```

6.1.2.6 The To-Do List

When a black (ridge) pixel that has two or fewer neighbors is found by the scan across the image, the curve associated with that pixel is extracted from the image by the curve initialization and curve following routines described above. After each such extraction based on a pixel found in the scan, the initialized seed curves in the to-do list must also be followed before the scan continues. Of course, if the to-do list is empty, the scan can continue immediately.,

Branches are placed on the to-do list instead of being followed immediately so that if the image scan finds a pixel in the middle of a curve, it is guaranteed that the two halves of the curve are extracted before any branches from that curve are extracted. If this were not the case, one half of the curve might branch into other curves that in turn branch and that might eventually contain the second half of the original curve. By completing the original curve before considering any branches, the original curve will never be broken.

The second reason for using the to-do list can be demonstrated by considering three curves that meet at a bifurcation point. Assume that one curve that enters the bifurcation has been extracted from the fingerprint image. If the branches from the bifurcation were not placed on the to-do list, the remaining two curves that terminate at the bifurcation point would instead appear to be a single curve going through that point. By placing the branches on the to-do list, the algorithm guarantees that each ridge entering the bifurcation will be represented as a separate curve terminating at the bifurcation point. Because the curves that form the bifurcation share a common endpoint, it is guaranteed that the reconstructed curves (after the encoding/decoding process) will also share this endpoint, thus preserving the bifurcation. If the bifurcation had instead been represented as one curve intersecting the middle of a second curve, the curves might not intersect in the reconstruction since the B-spline process does not guarantee that a reconstructed curve will pass through any of the curve's points other than its endpoints (see section 9).

Given a non-empty to-do list, the initialized seed curves on the list are removed and processed in turn. After removing a seed curve from the list, the last point on the curve is examined. If it is currently labeled BIFURCATION, the curve is complete. If the last point no longer appears in the fingerprint image, the curve is discarded. Otherwise, the point is still in the fingerprint image and the curve should be followed. First, the point is removed from the fingerprint image. Then, the curve is followed as described in section 6.1.2.2. Note that the curve following process may result in new seed curves being put onto the list. After a curve on the list is processed, the next curve on the list is removed and processed as just described. When the to-do list is empty, the scan of the image for black (ridge) pixels continues.

FOLLOW_TO_DO_LIST[ ]

** This function has the side effect of modifying *IMAGE*

```
while to_do is not EMPTY
    curve = next seed curve in to_do list        ** Removes the seed curve from to_do
    last_point = last point on curve
    if (last_point is labeled BIFURCATION)
        put curve onto curve_set
    else
        if (last_point = BLACK)
            last_point = WHITE                   ** Remove this pixel from IMAGE
            curve = FOLLOW[curve]                ** Section 6.1.2.2
            put curve onto curve_set
return
```

6.2 SUMMARY

After curve extraction, all ridges will have been extracted from the thinned fingerprint image and represented as curves. Each curve is represented as an ordered list of points, where each point contains the location of a ridge pixel in the image. Ridges that form a bifurcation will be represented by curves that share a common endpoint (the bifurcation point).

Input

IMAGE	Raw thinned fingerprint image

Output

curve_set	List of fingerprint curves

SECTION 7

RIDGE CLEANING

The ridge cleaning algorithm processes the list of curves generated by the curve extraction algorithm in order to connect curves across small ridge breaks and to remove small offshoot curves, small ridge connections, small ridge segments, and curved ridge endings (see Appendix B for this last process). This has the effect of removing details and artifacts that would require extra data to encode for transmission, and that would contribute little or no relevant information about the fingerprint. Examples of these artifacts and details are illustrated in figure 40. As a final step, any curve sections that cross a region of the fingerprint denoted as a bad block during BHO binarization (see Appendix B) are removed.

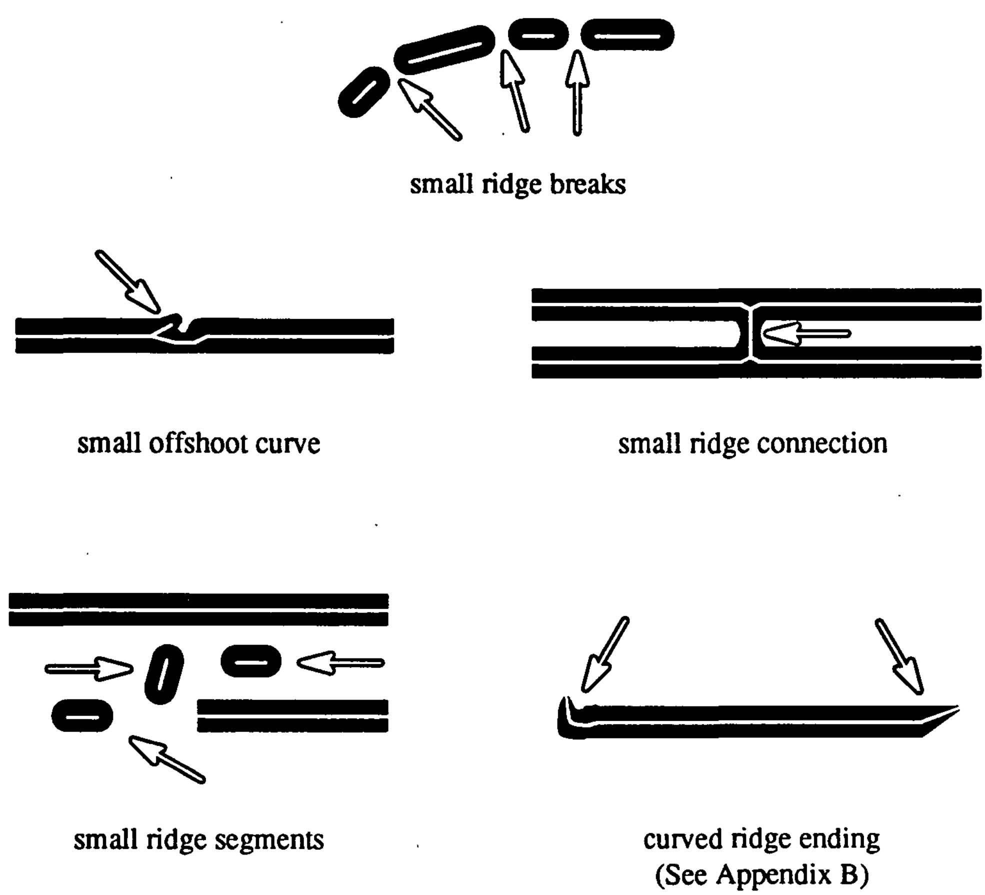


Figure 40. Examples of the Artifacts and Details Modified in Ridge Cleaning

The cleaning process does not modify or remove minutiae from the fingerprint. The small offshoot curves are artifacts that often occur in the ridge thinning process from a pore that is at the edge of a fingerprint ridge, resulting in a small concavity or a small bump in the ridge edge. These are removed because small variations in the contours of the ridge edges are not relevant information in this context. The removal of small offshoot curves is controlled by a selectable parameter indicating the length of the removed curves as a factor of fingerprint average ridge widths. Small offshoot curves are also removed if they are short and their ridges are thin relative to the neighboring ridges.

Curved ridge endings are often caused by a fold or scar in a finger that crosses ridges at a slant. The fold or scar can create a curved appearance at the ridge ending that is retained by thresholding. When these ridges are thinned to a single-pixel width, there may a a small flip or curve at the end of the ridge. This curvature is removed so that the ridge direction at the endpoint more accurately represents the true ridge ending orientation.

Connecting across small ridge breaks saves the overhead of encoding the separate curves while still representing the fingerprint according to the established practices. This process of connecting across small ridge breaks must take place after the small offshoot curves are removed, otherwise two opposing small offshoot curves may be connected, creating a false minutia. The connection across small ridge breaks is controlled by a selectable parameter indicating the size of removed ridge break in terms of the number of overall fingerprint average ridge widths.

Small connections between parallel ridges may come from foreign substances on the finger at the time of printing or other unreliable sources, so these connections are detected and removed. By removing these small ridge connections, the data required to encode the small curve and the overhead associated with the additional curves is saved from having to be transmitted. When these small connections are removed, the ridges on either side, which are represented by two curves each, are joined with their appropriate mates to make single curves.

The small ridge segments are removed last. These small ridge segments primarily reside below the flexion crease of a fingerprint and are not considered to be important, hence they are removed. They are also occasionally found between ridges above the flexion crease. This removal of small ridge segments must take place after the small ridge break connection step because, in some cases, small ridge segments may actually be connected into larger curves.

7.1 ALGORITHM DESCRIPTION

The ridge cleaning process requires three input data items from the previous processes of live-scan fingerprint compression: the curve list representing the fingerprint *curve_list* generated in curve extraction, the chamfered image C generated in ridge thinning, and the ridge direction map *z_blockmap* generated during BHO binarization. The algorithm modifies *curve_list* by removing and joining curves. This reduces the amount of data needed to encode the representation of the fingerprint. The algorithm uses the chamfered image C generated in ridge thinning (section 5) for calculating average ridge widths. Since areas of the chamfered image that correspond to bad blocks identified by *z_blockmap* do not contain accurate ridge width information, all the values of C in bad block areas are set to zero.

Before the actual process of cleaning the curves in *curve_list*, several data items must be initialized. The average ridge width of all ridges in the fingerprint and of neighborhoods in the image must be calculated. These values will be calculated by applying the algorithm described in section 7.1.2 on the ridges in *curve_list* to obtain the average values. The resulting overall average will be referred to as $ridge\_width_{fingerprint}$. The calculation of average ridge width differs from the description of determining the neighborhood average ridge width (section 4.1.3) used by pore filling in that the average ridge width used here is based on the extracted curves in *curve_list*. The value of $ridge\_width_{fingerprint}$ will be used in the cleaning steps to determine important curve size delimiters based on the selectable system parameters $F_{OFFSHOOT\ CURVE}$, F_{RIDGE_BREAK}, and $F_{UNCONNECTED\_CURVE}$ which control the curve connection and removal processes.

A thinned image T is generated by drawing all the points in every curve of *curve_list* into a blank image of the same size as the original fingerprint image. T is used for some condition checking and is updated to match the changes made to *curve_list*.

Initialization continues with the generation of a data item called the *endpoint_map*. The *endpoint_map* serves as a tool to quickly find which curves have an endpoint at any specified location in the fingerprint. The actual implementation of the *endpoint_map* may vary, but the algorithm, by accessing *endpoint_map*, must be able to count the number of endpoints at a location and determine the current locations of the these curves in *curve_list*. This *endpoint_map* must be updated throughout the processing to stay current with any changes made to *curve_list*.

Once the initialization is completed, the algorithm applies each ridge cleaning subprocess in turn, modifying *curve_list* and updating the *endpoint_map* and T as required by the cleaning subprocesses. The first cleaning subprocess to be applied is small offshoot curve removal described in section 7.1.3. After the small offshoot curves have been removed, curved ridge endings are removed by a subprocess described in Appendix B. The next subprocess to be applied to *curve_list* is small ridge break connection described in section 7.1.4. After small ridge break connection is completed, the subprocess small ridge connection removal is applied as described in section 7.1.5. The last two cleaning subprocesses to be applied are small ridge segment removal described in section 7.1.6 and bad block blanking described in Appendix C.

In all the ridge cleaning subprocesses, care must be taken when adding and deleting curves from *curve_list* so that the algorithm's iteration over *curve_list* can continue in a proper fashion. If a curve under consideration is removed, the algorithm must be able to continue onto the next curve on the list in the next step of the iteration. Also, if a new curve is added to *curve_list* during a subprocess, it should be placed in the list in a manner that will allow it to be also considered by the subprocess.

After the ridge cleaning process has been completed, *curve_list* has been modified and will be further processed by the live-scan compression algorithm. The *endpoint_map* and the thinned image ***T*** generated by this process can be deleted at this point because it is not used in further processing. The chamfered image ***C*** generated by ridge thinning can be eliminated at this point, also. A flowchart for the overall ridge cleaning process is illustrated in figure 41.

```
RIDGE_CLEANING[ curve_list, C, z_blockmap ]
    ** This algorithm modifies curve_list and C
    ** The chamfered image C, the thinned image T, endpoint_map, ridge_width_fingerprint,
       and curve_list are globally accessible for the routines called by RIDGE_CLEANING[]
for each block (bi, bj) in z_blockmap              ** Zero bad block chamfer values
    if ( block (bi, bj) is bad )
        for each pixel (i, j) in block (bi, bj)
            C(i, j) = 0
for each curve in curve_list                         ** Initialize endpoint_map
    draw curve into T
    place both endpoints of curve into endpoint_map
ridge_width_fingerprint = average of all ridge widths in the fingerprint
PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS_CURVE[IMAGE]
    ** The five routines below may modify curve_list, endpoint_map and T
SMALL_OFFSHOOT_CURVE_REMOVAL[ curve_list ]
CURVED_RIDGE_ENDING_REMOVAL[ curve_list, z_blockmap ]         ** Appendix B
SMALL_RIDGE_BREAK_CONNECTION[ curve_list ]
SMALL_RIDGE_CONNECTION_REMOVAL[ curve_list ]
SMALL_RIDGE_SEGMENT_REMOVAL[ curve_list ]
BAD_BLOCK_BLANKING[ curve_list, z_blockmap ]                  ** Appendix C
return
```

```
PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS_CURVE[ IMAGE ]
    ** This function is the same as PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS of
       section 4.1.3, except that the ridges used to calculate the widths are taken from
       curve_list instead of from the thinned fingerprint image.
```

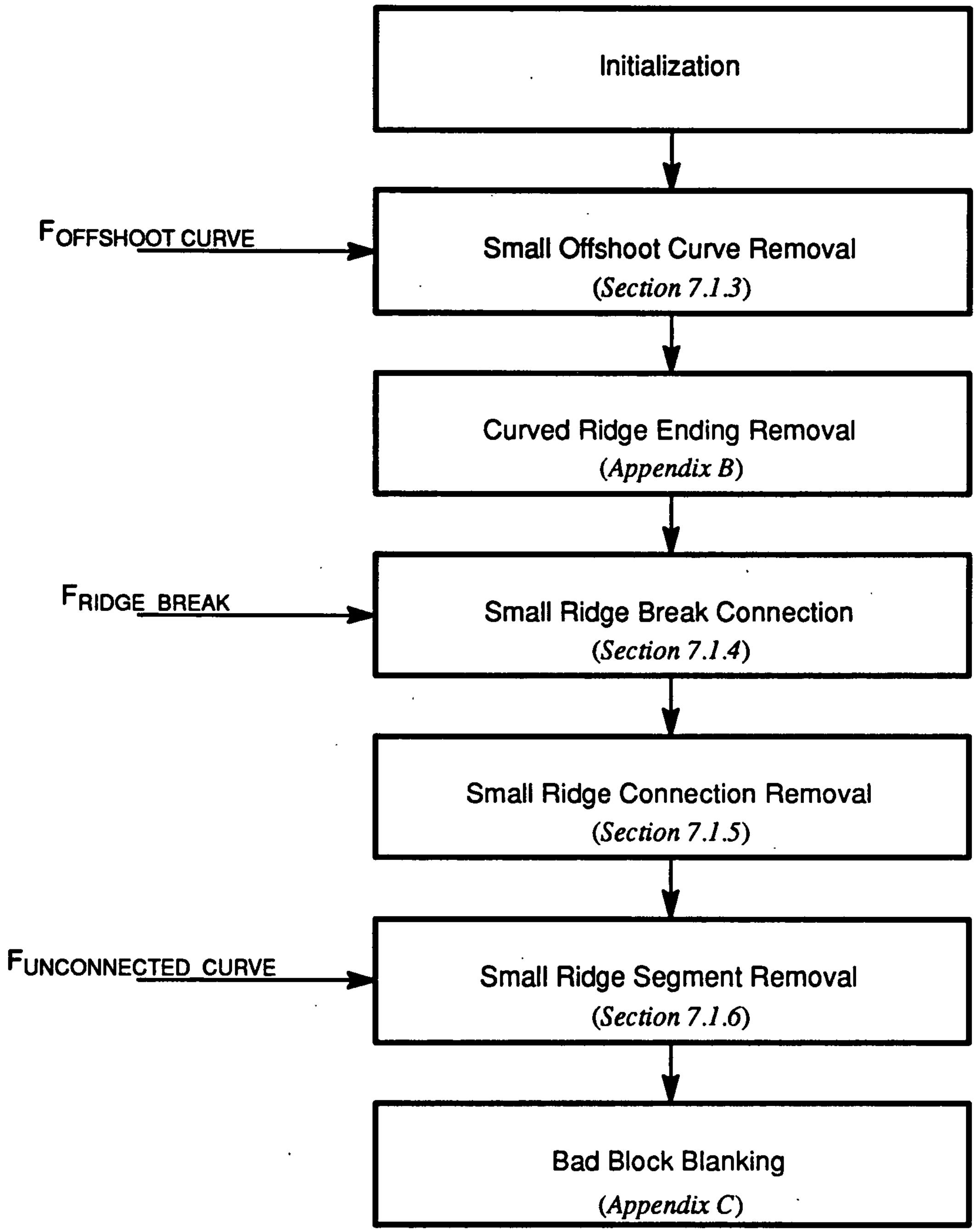


Figure 41. Flowchart of the Ridge Cleaning Process

7.1.1 Definitions

The *thinned_image* provides the ridge cleaning algorithm with a method to efficiently check for collisions when connecting small ridge breaks. A collision occurs when the fill-in curve section connecting across the small ridge break intersects with another curve already in *curve_list*. A *thinned_image* containing the curves in *curve_list* is used for this check. The *thinned_image* must be kept updated on modifications to *curve_list* so that it reflects accurate curve information for this collision check. After the small ridge break removal process is completed, the *thinned_image* can be discarded.

The *endpoint_map* is required to provide the ridge cleaning algorithms with the capability to quickly find curves that have endpoints in common with other curves. (Curve endpoints are defined as the first and last points of a curve.) Without the *endpoint_map*, the algorithms below would need to scan each curve in *curve_list*, comparing a curve's endpoint positions with all the other curves' endpoint positions. The *endpoint_map* is also used to find curves that have endpoints in the neighborhood of a given position. Again, to find these curves is a simple matter of referencing the *endpoint_map* for all the positions within the neighborhood. Without the *endpoint_map*, the algorithms would have to scan the entire list of curves, calculating whether or not each curve had an endpoint within the neighborhood. The endpoint information is used often enough to warrant the memory and processing to generate this map. All that is required for an effective *endpoint_map* is that all endpoints that share the same position are associated, that there is an efficient method of referencing endpoints given a desired position, and that an endpoint's originating curve is associated with the endpoint.

Curve connectivity is determined using the endpoint map described above. The connectivity of a curve can be either unconnected, singly connected, or doubly connected. An unconnected curve does not intersect any other curve. A singly connected curve has exactly one endpoint that intersects with at least one other curve and exactly one endpoint that does not intersect with any other curve. For a doubly connected curve, both endpoints intersect other curves. When curves intersect, their endpoints share the same position. By a property of the extraction algorithm, any curve intersection will involve three or more curves. An intersection of two curves is invalid because they will have been appended together to make one curve. Whether or not an intersection exists at an endpoint can be determined quickly by counting the number of endpoints in *endpoint_map* at the corresponding position of the endpoint. An intersection exists at a position if there is more than one endpoint at that position, but if there is only one endpoint at that position (itself), there is no intersection at that endpoint. Using this quick endpoint intersection test on both endpoints of a curve, the connectivity of the curve can be quickly determined to be either unconnected, singly connected, or doubly connected. Examples of the curve connectivity types are shown in figure 42.

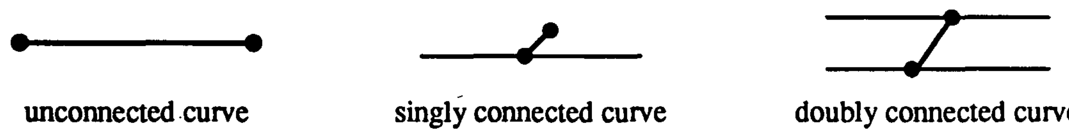


Figure 42. Examples of the Curve Connectivity Types

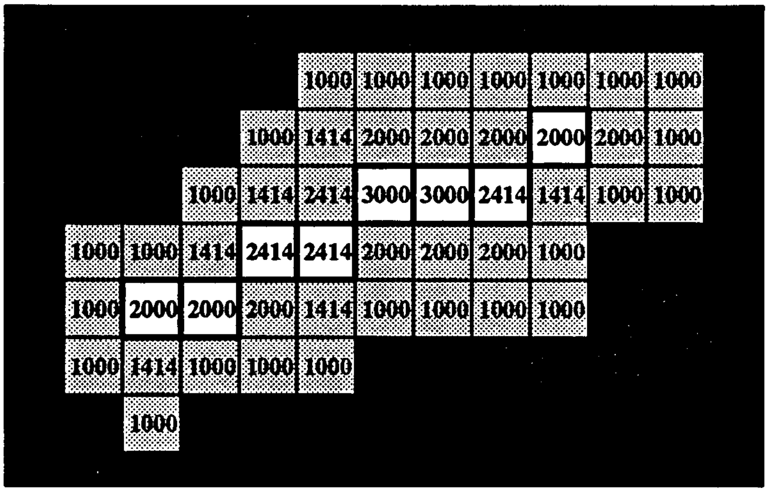
7.1.2 Average Ridge Width

The average ridge width algorithm calculates the average ridge width ($ridge\_width_{ave}$) along a section of a fingerprint curve (*curve*) between the specified starting and ending points P_{start} and P_{end}. The chamfered image C, calculated in the ridge thinning process and modified at the beginning of ridge cleaning, is used because the value of each pixel represents the approximate distance to the nearest edge of its ridge. Only pixels not in bad blocks are considered. To get the ridge width at a particular pixel, the chamfer value at the corresponding location in C is divided by 500.0. The divisor 500.0 is determined to be twice the chamfer value of the pixel normalized by the chamfer scaling factor (1000). This division rescales the chamfer value to represent twice the distance to the nearest ridge edge in pixels. To find $ridge\_width_{ave}$ along a section of *curve*, the algorithm sums the values of the pixels of C at the corresponding positions of the points within the section of *curve* specified by p_{start} and p_{end}. This sum is then divided by the number of pixels in the curve section and further divided by 500.0 for rescaling. The calculation of $ridge\_width_{ave}$ is illustrated in figure 43. If $ridge\_width_{ave}$ of an entire curve is desired, all the chamfer values of the pixels in the curve are summed and then divided by the number of points in the curve and further divided by 500.0. If the average ridge width for the entire fingerprint $ridge\_width_{fingerprint}$ is desired, the chamfer values of all pixels that corresponding to the pixels in all curves of *curve_list* are summed, then divided by the total number of pixels, and further divided by 500.0 for rescaling. The values of $ridge\_width_{ave}$ and $ridge\_width_{fingerprint}$ must be maintained as floating point numbers for the accuracy needed in next steps.

RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[P_{start}, P_{end}, *curve*]

** The chamfer image C is accessed from RIDGE_CLEANING[]

number_of_pixels = 0
sum = 0
for each point (i, j) between P_{start} and P_{end}, inclusive, along *curve*
    if ($C(i,j) \neq 0$)   ** Calculate using only pixels in non-bad blocks
        sum = *sum* + $C(i,j)$
        number_of_pixels = *number_of_pixels* + 1
$ridge\_width_{ave} = sum / (500.0 \times number\_of\_pixels)$
return $ridge\_width_{ave}$



2000
2414
3000
3000
2414
2414
2000
+ 2000

19242

ridge_width$_{ave}$
(19242 / 8) / 500.0 = 4.81 pixels

$$ridge\_width_{ave} = \left(\frac{\sum C_{\text{good ridge pixels}}}{\text{number of good ridge pixels}}\right) / 500.0$$

Figure 43. Calculation of Average Ridge Width, *ridge_width*$_{ave}$

7.1.3 Small Offshoot Curve Removal

Small offshoot curve removal deletes curves from *curve_list* that share a common endpoint on only one end (singly connected) and that are shorter than the smallest, singly connected curve allowed. The algorithm iterates over *curve_list*, considering each curve. If a curve is classified as singly connected, its length is checked. If the curve is either short relative to the fingerprint as a whole (the number of pixels in the curve is less than $F_{\text{OFFSHOOT CURVE}} \times ridge\_width_{fingerprint}$) or both thin and short relative to neighboring curves (both the width of the curve is less than $Z_{\text{WIDTH\_OFFSHOOT}}$ times the local average ridge width and the length of the curve is less than $Z_{\text{LENGTH\_OFFSHOOT}}$ times the local average ridge width), the curve is removed from ***T***, its endpoints are deleted from the *endpoint_map*, and the curve is deleted from *curve_list*. Care must be taken when deleting the curve from *curve_list* so that the iteration can continue in the proper fashion, not neglecting consideration of any curve.

Once all the small offshoot curves are removed, *curve_list* must be further processed to join curves that can be represented as one curve. When a small offshoot curve is deleted, two intersecting curves may be left behind. These two curves can be appended into one larger curve if they are the only curves sharing that endpoint position. To make these required connections, the algorithm iterates over *curve_list*, considering each curve. If a curve has an endpoint whose position is shared by the endpoint of exactly one other curve, all the points of these two curves are joined to generate a new curve.

SMALL_OFFSHOOT_CURVE_REMOVAL[*curve_list*]

```
    ** This algorithm modifies curve_list, endpoint_map, and T
for each curve in curve_list
    z_local_ridge_width = the average of the local average ridge widths at the
                              unconnected endpoint and at the midpoint of curve
    if ((curve is singly connected)
        and ((number of points in curve < F_OFFSHOOT CURVE × ridge_width_fingerprint)
            or ((RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[curve unconnected endpt,
                                                    curve midpoint, curve]
                    < Z_WIDTH_OFFSHOOT × z_local_ridge_width)
                and (number of points in curve
                    < Z_LENGTH_OFFSHOOT × z_local_ridge_width))))
        remove curve from T
        delete endpoints of curve from endpoint_map
        delete curve from curve_list
        set curve a and b to be the curves that shared an endpoint with curve
        JOIN_CURVES[curve a, curve b]
return
```

7.1.3.1 Join Curves

Join_Curves attaches together two curves that have an overlap of endpoints in order to generate a single new curve. Care must be taken that the new curve is traceable between its endpoints without redundant points. This is guaranteed if the points from the first curve are copied into the new curve starting with its non-overlapping endpoint, then the points from the second curve are copied into the new curve starting at its overlap end, skipping the first endpoint in order not to have a repeated point. This new curve is inserted into *curve_list* and the endpoints of this new curve are added to *endpoint_map*. The two curves joined by this combination are deleted from *curve_list* and their endpoints are removed from the *endpoint_map*. Updating of ***T*** is not necessary because the new curve completely overlaps the two appended curves. Care must be taken when modifying *curve_list* so that the iteration can continue in the proper fashion to insure the consideration of every curve, including the newly generated curve.

```
JOIN_CURVES[ curve a, curve b ]
    ** This algorithm modifies curve_list and endpoint_map
    if (curve a ≠ curve b)
    {
        for each point p in curve a from non-connection endpoint to connection endpoint
            append point p to end of curve c
        for each point p in curve b from one past the connection endpoint
                                                    to non-connection endpoint
            append point p to end of curve c

        remove curve a and curve b from curve_list
        append curve c onto curve_list
        update endpoint_map
    }
    return
```

7.1.4 Small Ridge Break Connection

The small ridge break connection algorithm attaches curves that can be considered to be one ridge except for a small break in the ridge. These curves have the property of being approximately colinear and have endpoints that are within an allowable distance. Examples of small ridge breaks are illustrated in figure 44.

To search for these small ridge breaks, the algorithm iterates over *curve_list*, considering each curve. If a curve's number of points is greater than or equal to $\text{RIDGE\_SIZE}_{\text{MIN}}$, its neighboring curves are examined for colinearity. Around each unconnected endpoint of the curve, the algorithm searches for other nearby endpoints. An endpoint is unconnected if, in *endpoint_map*, it does not share its position with any other endpoint. The algorithm searches for nearby endpoints by scanning *endpoint_map* for other unconnected endpoints within a specified neighborhood of the curve's unconnected endpoint. The neighborhood is defined as any pixel within the search radius $radius_{search}$. The value of $radius_{search}$ is set to be the minimum value between the default window size ($\text{RADIUS}_{\text{DEFAULT}}$) and 3.5 × the ridge width at the curve's unconnected endpoint (see section 7.1.2). This calculation of

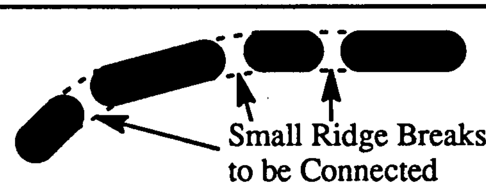


Figure 44. Examples of Small Ridge Breaks to be Connected

$radius_{search}$ allows the algorithm to find other unconnected endpoints of nearby ridges that are within a distance of $F_{RIDGE\_BREAK} \times ridge\_width_{fingerprint}$. An example of the calculation of the search radius is illustrated in figure 45. A list of unconnected endpoints contained in the search area is made that includes their positions and curve identification.

Once this list of unconnected endpoints has been compiled, the algorithm searches for the mutually best connection between curves. This search eliminates processing order dependencies, allowing all candidate connections in a region to be considered at the same time. Each legal ridge break connection between the curve's endpoint and each unconnected endpoint in the search region list is scored by a function (described in section 7.1.4.1) that measures ridge alignment and ridge break size. The algorithm determines the maximum value of all these scores. For each endpoint whose score is equal to the maximum score, a list of the unconnected endpoints in its search region is compiled and scored. If the score for

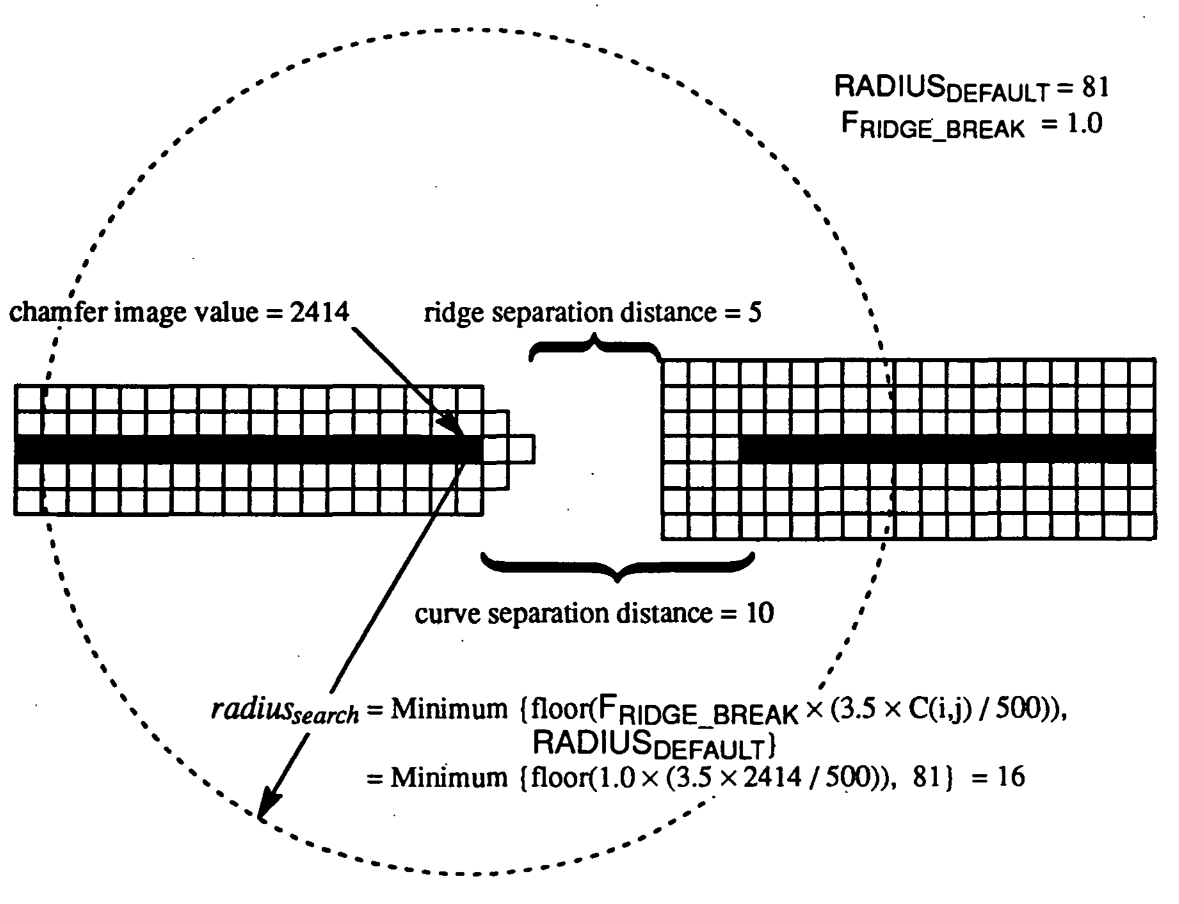


Figure 45. Example of a Search Radius Calculation for the Small Ridge Break Connection Algorithm

the connection to the initiating endpoint is equal to the maximum score of that list, then the connection is considered to be mutually best and the curves of the initiating endpoint and the current endpoint are connected using the curve connection algorithm in section 7.1.4.2.

```
SMALL_RIDGE_BREAK_CONNECTION[ curve_list ]

** This algorithm modifies curve_list, endpoint_map, and T
for each curve in curve_list
    if number of points in curve > RIDGE_SIZE_MIN
        for each unconnected endpoint a in curve
        {
            status = FALSE
            radius_searcha = min(RADIUS_DEFAULT, 3.5 × ridge width at endpoint a)
            candidate_list = all neighboring unconnected endpoints
                                 within radius_searcha of endpoint a
            for each endpoint b in candidate_list
                score of b = CONNECTION_SCORING_FUNCTION[ endpoint a, endpoint b ]
            best_score_a = maximum of all scores of endpoints in candidate_list

            ** Find the mutually best connection by considering all endpoints
               in candidate_list whose score is equal to best_score_a
            for each endpoint b in candidate_list whose score equals best_score_a
            {
                radius_searchb = min(RADIUS_DEFAULT, 3.5 × ridge width at endpoint b)
                check_list = all neighboring unconnected endpoints
                                 within radius_searchb of endpoint b
                for each endpoint c in check_list
                    score of c = CONNECTION_SCORING_FUNCTION[endpoint b,endpoint c]
                best_score_c = maximum of all scores of check_list

                ** Connecting endpoint b and endpoint a is mutually best if the score
                   of endpoint a in the list generated from endpoint b is a maximum
                if score of endpoint a in check_list = best_score_c
                    status = CONNECT_CURVES[ curve of endpoint a, curve of endpoint b ]
                    exit from loop
            }
            if (status)
                exit from loop
        }
return
```

7.1.4.1 The Scoring Function

The scoring function assigns a floating point value that is directly proportional to the "desirability" of the connection being considered between two endpoints. This score is based on the alignment of the two curves near those endpoints and the distance between those endpoints, modified by the average ridge width of the curves near those endpoints. The ends of the curves being considered for connection must first pass several tests. If any of these tests fail, an illegal connection flag is returned to the function that called this scoring function. This will remove the connection from further consideration. Otherwise the floating point score value is returned.

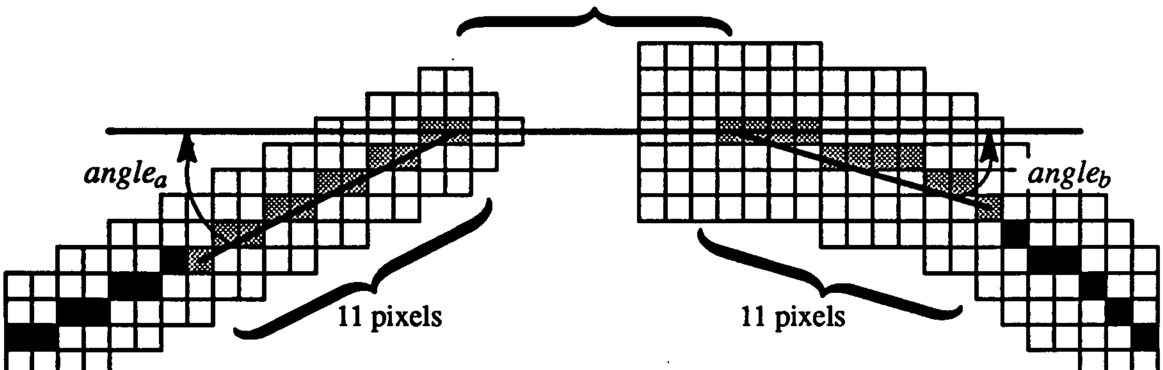
In order to execute the curve end tests and calculate the score, several intermediate values are calculated. First, the Euclidean distance between the endpoints is calculated. The value of floor(distance + 1) is used as the default section size in the further calculations. Second, the end section and end reference point for each curve are determined. This is calculated by finding the curve end's section size ($section\_size_{curve}$), which is defined as the minimum of the default section size and the curve's number of points. The end reference point is the point that is $section\_size_{curve}$ points down from the endpoint. The end section consists of all the points between the endpoint and the end reference point, inclusive. Once both curve ends are defined, the average ridge width for each end section is calculated as described in section 7.1.2. The angle score for each curve is also calculated. The angle score is defined as the cosine of the angle of change at an endpoint caused by traversing from the other endpoint though this endpoint on toward its end reference point. An angle score of 1.0 indicates that the traversal was along a straight line. Before calculating the connection score, the algorithm must determine whether each curve is small enough to be a small ridge segment. This is necessary because if a curve is a small ridge segment, the angle score is prone to error and should be ignored for that curve. If both curves are determined to be small ridge segments, the connection is illegal and the scoring function returns an illegal connection flag. For a curve to be considered a small ridge segment, the curve's number of points must be less than the minimum ridge size and also less than the average ridge width calculated for its end section.

If neither curve is considered to be a small ridge segment, the distance between the endpoints, calculated earlier, is tested for being less than twice the average of the average ridge widths of the two end sections multiplied by F_{RIDGE_BREAK}. If this test fails, the scoring function returns an illegal connection flag. Otherwise, each angle score and the sum of two angle scores are tested for being less than the value of the cosine($A_{COLINEAR}$), where $A_{COLINEAR}$ is the angular limit for colinearity. During development $A_{COLINEAR}$ was set to 45 degrees so that curves within 45 degree of being colinear would be acceptable. A larger value for $A_{COLINEAR}$ will result in a tighter colinearity requirement. If any of the angle scores are less than cosine($A_{COLINEAR}$), the scoring function returns an illegal connection flag. Otherwise, the score is calculated and returned as the product of the two angle scores divided by the distance between the endpoints. An example of this calculation is illustrated in figure 46.

$F_{RIDGE\_BREAK} = 1.0$

$$distance = \sqrt{x^2 + y^2} = \sqrt{10^2 + 10^2} = 10.0$$

(default section size = 11)



$ridge\_width_a = 4.83$ $ridge\_width_b = 7.36$

$angle\_score_a = \cos(angle_a) = 0.910$ $angle\_score_b = \cos(angle_b) = 0.965$

neither curve is smaller than $RIDGE\_SIZE_{MIN}$
and $angle\_score_a \geq \cos(A_{COLINEAR})$
and $angle\_score_b \geq \cos(A_{COLINEAR})$
and $(angle\_score_a + angle\_score_b) \geq \cos(A_{COLINEAR})$
and $distance \leq F_{RIDGE\_BREAK} \times (ridge\_width_a + ridge\_width_b)$
therefore
$connection\_score = (0.910 \times 0.965) / 10.0 = 0.088$

Figure 46. Example of a Connection Scoring Function Calculation

If only one of the curves is considered to be a small ridge segment, its average ridge width and angle score are ignored. Instead, the average ridge width and angle score of the larger curve is used in place of these values for the small ridge segment. The distance between endpoints is tested for being less than twice the average ridge width of the larger curve's end section. If this test fails, the scoring function returns an illegal connection flag. Otherwise, the angle score of the larger curve is tested for being less than the cosine($A_{SEGMENT}$). During development $A_{SEGMENT}$ was set to 60 degrees, which is slightly less restrictive than $A_{COLINEAR}$ used above in checking the colinearity of two larger curves. If the angle score is less than cosine($A_{SEGMENT}$), the scoring function returns an illegal connection flag. Otherwise, the score is calculated and returned as the square of the larger curve's angle score divided by the distance between the endpoints.

```
CONNECTION_SCORING_FUNCTION[ endpoint a , endpoint b ]

curve_a = the curve that contains endpoint a
curve_b = the curve that contains endpoint b
distance = Euclidean distance between endpoint a and endpoint b
section_size = floor(distance + 1.0)
section_size_a = minimum(section_size, the number of points in curve_a)
ref_a = the point that is section_size_a points down curve_a from endpoint a
ridge_width_a = RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[ endpoint a, ref_a, curve_a]
angle_score_a = cosine(angle of change traversed from ref_a
                        through endpoint a to endpoint b)
section_size_b = minimum(section_size, the number of points in curve_b)
ref_b = the point that is section_size_b points down curve_b from endpoint b
ridge_width_b = RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[ endpoint b, ref_b, curve_b ]
angle_score_b = cosine(angle of change traversed from ref_b
                        through endpoint b to endpoint a)

if (number of points in curve_a < minimum(RIDGE_SIZE_MIN, ridge_width_a))
    curve_a is a small ridge segment
if (number of points in curve_b < minimum(RIDGE_SIZE_MIN, ridge_width_b))
    curve_b is a small ridge segment

if ((curve_a is a small ridge segment) and (curve_b is a small ridge segment))
    return ILLEGAL_CONNECTION
if ((curve_a is not a small ridge segment) and (curve_b is not a small ridge segment))
    if ((angle_score_a < cos(A_COLINEAR)) or (angle_score_b < cos(A_COLINEAR))
        or ((angle_score_a + angle_score_b) < cos(A_COLINEAR))
        or (distance > F_RIDGE_BREAK × (ridge_width_a + ridge_width_b)))
        return ILLEGAL_CONNECTION
    connection_score = (angle_score_a × angle_score_b) / distance
if ((curve_a is not a small ridge segment) and (curve_b is a small ridge segment))
    if ((angle_score_a < cos(A_SEGMENT))
        or (distance > F_RIDGE_BREAK × (ridge_width_a + ridge_width_a)))
        return ILLEGAL_CONNECTION
    connection_score = angle_score_a^2 / distance
```

if (($curve_a$ is a small ridge segment) **and** ($curve_b$ is not a small ridge segment))
    if (($angle\_score_b < \cos(A_{SEGMENT})$)
        or ($distance > F_{RIDGE\_BREAK} \times (ridge\_width_b + ridge\_width_b)$))
        return ILLEGAL_CONNECTION
    $connection\_score = angle\_score_b^2 \,/\, distance$

return *connection_score*

7.1.4.2 Curve Connection

Curves are connected by generating a single new curve from the two curves to be connected and a fill-in curve section between their endpoints. First, the fill-in section to connect the curves between the endpoints is calculated. This is done first because if the fill-in curve section overlaps any other curve point in the thinned image, there is a potential crossing of ridges. If this happens, the connection should not be made as it might add false minutiae. This situation can be detected when generating the fill-in section. The fill-in section is generated as a sequence of points calculated as a straight line starting at both endpoints and meeting in the middle. Care must be taken that the resulting connectivity of the complete curve follows the connectivity properties of the curve extraction algorithm (see section 6).

Once the fill-in section is successfully calculated, the new curve is created with its size equal to the sum of the number of points in the two curves and the number of points generated by the fill-in section. The points of the first curve are copied into the new curve starting with the end that will not be connected and ending with the connecting endpoint. The fill-in section is then copied into the new curve maintaining the proper contiguous connections. Lastly, the points of the second curve are copied into the new curve, again maintaining proper connections so the final product is a continuous curve with each point eight connected to its neighbors. The new curve is added to *curve_list* in a manner that insures continued proper iteration over the list. The two curves that were connected are deleted from *curve_list*. The *endpoint_map* is updated by deleting the endpoints from the connected curves and adding the endpoints of the newly generated curve. The thinned image ***T*** is updated by drawing in the fill-in section, thereby connecting the two curves in the thinned image. When updating these items for the two connecting curves, care must be taken to determine whether these two curves are actually just one curve; removing the same curve twice from *curve_list* and *endpoint_map* must be prevented.

```
CONNECT_CURVES[ curve a, curve b ]
    ** This algorithm modifies curve_list, endpoint_map, and T
        generate the fill-in section between the connection endpoints of curves a and b

    ** If the fill-in section overlaps another curve, this is an illegal connection
    if (fill-in section intersects any curve in thinned image, T)
        return FALSE

    for each point p in curve a from non-connection endpoint to connection endpoint
        append point p to end of curve c
    for each point p in the fill-in section
        append point p to end of curve c
    for each point p in curve b from connection endpoint to non-connection endpoint
        append point p to end of curve c

    remove curve a and curve b from curve_list
    append curve c onto curve_list
    update T and endpoint_map
    return TRUE
```

7.1.5 Small Ridge Connection Removal

The small ridge connection removal algorithm deletes small doubly connected curves from *curve_list* that bridge across two roughly parallel curves (see figure 47). The algorithm iterates over each curve in *curve_list* to test for and remove such bridge curves. If a curve is classified as doubly connected and is shorter than $L_{DOUBLY\_CONNECTED}$ points, the curve is considered further, otherwise the iteration continues.

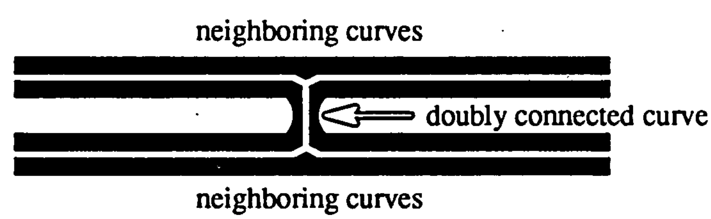


Figure 47. Example of a Small Ridge Connection

Given a curve for further consideration, the algorithm tests the number of endpoints in the *endpoint_map* for each of the curve's endpoints. If tests determine that exactly three endpoints are present at each end, this curve is considered further, otherwise the iteration continues. This check is done because, if the curve's intersections are more complicated than three-way overlaps, this curve may be in a very complicated section of the fingerprint where appropriate removal decisions are not possible. Many of these complicated areas reside in bad blocks and will be removed during bad block blanking (see Appendix C).

To continue testing of the doubly connected curve, the algorithm determines the reference points and end sections for the four neighboring curves connected by overlapping endpoints to the doubly connected curve (see figure 48). If $l_{doubly\_connected}$ represents the number of points in the doubly connected curve, the reference point of a neighboring curve is defined to be $l_{doubly\_connected}$ points down the curve from the curve's overlapping endpoint. The neighboring curve section between the endpoint and the reference point is referred to as the end section and contains ($l_{doubly\_connected}$ + 1) points, including the endpoint and the reference point. If a curve contains fewer than max($l_{doubly\_connected}$/2, 3) points, which does not allow reasonable accuracy in curve direction, the consideration of this doubly connected curve is aborted and the iteration over *curve_list* continues. If a neighboring curve has fewer than $l_{doubly\_connected}$ points, the entire curve is used as the end section. The four end sections and their associated reference points will be used to test the relative thickness of the doubly connected curve, the connection angle, and the parallelism of the two ridges connected by the doubly connected curve to decide whether the curve should be deleted.

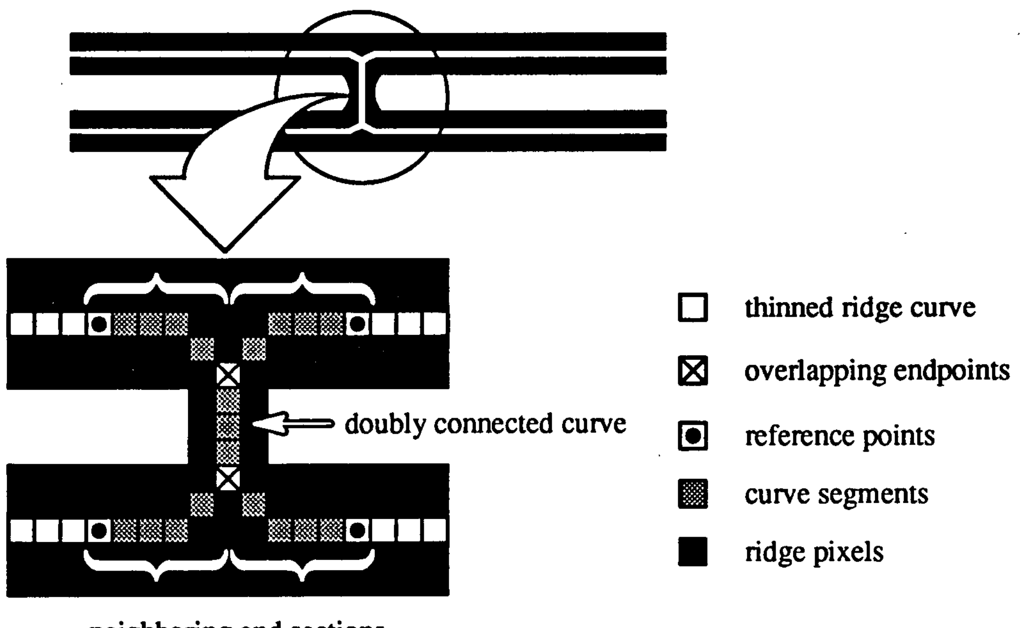


Figure 48. Definition of the Four Neighboring End Sections of a Doubly Connected Curve

If all four neighboring end sections and reference points are successfully determined, the algorithm tests to see that the doubly connected curve is sufficiently thinner than the neighboring ridges. This is accomplished by calculating the average ridge widths, as described in section 7.1.2, for the doubly connected curve and the four neighboring end sections. If the average ridge width of the doubly connected curve is less than or equal to $W_{DOUBLY\_CONNECTED} \times$ the average of the four neighboring end sections' average ridge widths, and the number of points of the doubly connected curve is less than $F_{DOUBLY\_CONNECTED} \times$ the average of the average ridge widths of the four neighboring end sections, the algorithm continues its consideration of this doubly connected curve. Otherwise, the algorithm no longer considers this curve and instead continues to iterate over *curve_list.*

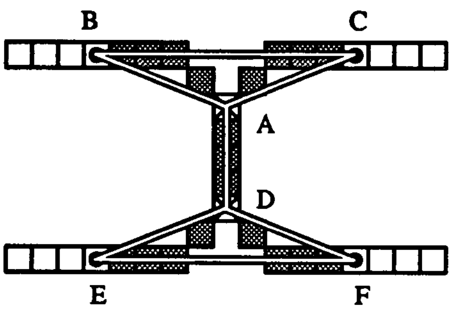
Next, the algorithm tests that the neighboring ridges are roughly parallel. This is accomplished by testing the angle between the two neighboring curves that pass through the reference points associated with each overlapping endpoint. If the angle is less than the angular limit for parallelism ($A_{PARALLEL}$), the neighboring curves are considered to be parallel and the algorithm continues consideration of the doubly connected curve. Otherwise, the algorithm no longer considers this curve and instead continues to iterate over *curve_list.*

Finally, the algorithm tests that the angles of attachment at the overlapping endpoints roughly form an "H". To check the colinearity of the two neighboring curves at an endpoint, the angle between the reference ends associated with the endpoint (with the vertex of the angle located at the endpoint) is tested to determine if it is greater than the angular limit for straightness ($A_{STRAIGHT}$). If the angles at both endpoints meet the straightness criterion, the algorithm continues to consider this doubly connected curve, otherwise the algorithm aborts consideration of this curve and continues the iteration over *curve_list.* Next, the algorithm tests the angle of attachment of the doubly connected curve to each of the four end sections. If the angle between the doubly connected curve and each end section is within A_{ATTACH} degrees of being perpendicular, this doubly connected curve has passed all the tests for being a small ridge connection and can be removed. The criteria for removing a small ridge connection are summarized in figure 49. The limits for sizes and angles indicated in the parameter summary below were the values used during development and may be selectable.

If a doubly connected curve passes all the tests, the curve is removed from ***T***, its endpoints are deleted from the *endpoint_map*, and the curve is deleted from *curve_list.* After the doubly connected curve has been deleted, each pair of neighboring curves having overlapping endpoints can be represented as a single curve. These pairs of curves must be combined in the same manner described in section 7.1.3.1., resulting in two curves from the original four.

Neighbor curve parallelism criteria

$$\frac{\vec{BC} \cdot \vec{EF}}{\|\vec{BC}\| \, \|\vec{EF}\|} > \text{cosine}(A_{PARALLEL})$$



Endpoint overlap criteria

Three overlapping endpoints
on both ends of curve

Doubly connected ridge size criteria

doubly connected curve's average ridge width
is less than or equal to
$W_{DOUBLY\_CONNECTED} \times$ average ridge width of the end sections
AND
number of points in the doubly connected curve
is less than
$F_{DOUBLY\_CONNECTED} \times$ average ridge width of the end sections
AND
number of points of each neighboring end section
is greater than
MAXIMUM(doubly connected curve number of points/2, 3)

Angle of attachment criteria

$$\frac{\vec{AB} \cdot \vec{AC}}{\|\vec{AB}\| \, \|\vec{AC}\|} < \text{cosine}(A_{STAIGHT}) \qquad \left| \frac{\vec{AD} \cdot \vec{AB}}{\|\vec{AD}\| \, \|\vec{AB}\|} \right| < \text{cosine}(A_{ATTACH}) \qquad \left| \frac{\vec{AD} \cdot \vec{AC}}{\|\vec{AD}\| \, \|\vec{AC}\|} \right| < \text{cosine}(A_{ATTACH})$$

$$\frac{\vec{DE} \cdot \vec{DF}}{\|\vec{DE}\| \, \|\vec{DF}\|} < \text{cosine}(A_{STAIGHT}) \qquad \left| \frac{\vec{DA} \cdot \vec{DE}}{\|\vec{DA}\| \, \|\vec{DE}\|} \right| < \text{cosine}(A_{ATTACH}) \qquad \left| \frac{\vec{DA} \cdot \vec{DF}}{\|\vec{DA}\| \, \|\vec{DF}\|} \right| < \text{cosine}(A_{ATTACH})$$

$\vec{AB}$ indicates a vector from point A to point B
· indicates the vector dot product

$\| \;\; \|$ is the Euclidean norm
$| \;\; |$ is the absolute value

Figure 49. Criteria for Removing a Small Ridge Connection

SMALL_RIDGE_CONNECTION_REMOVAL[*curve_list*]

```
    ** This algorithm modifies curve_list, endpoint_map, and T
    for each curve in curve_list
        if ((curve is double connected with exactly three common endpoints at each end)
                and ( length of curve < L_DOUBLY_CONNECTED))
            set endpoint_0 and endpoint_1 to be the endpoints of curve
            set curves a and b to be curves that share endpoints with curve at endpoint_0
            set endpoint_a and endpoint_b to be the overlapping endpoints of these curves
            set curves c and d to be the other two curves that share endpoints with curve
                    at endpoint_1
            set endpoint_c and endpoint_d to be the overlapping endpoints of these curves
            reference_length = max(length of curve, 3)
            if all of the lengths of curves a, b, c, or d > reference_length
                ref_a = the point on curve a that is reference_length points from endpoint_a
                ref_b = the point on curve b that is reference_length points from endpoint_b
                ref_c = the point on curve c that is reference_length points from endpoint_c
                ref_d = the point on curve d that is reference_length points from endpoint_d
                w = ( RIDGE_SECTION_AVERAGE_RIDGE_WIDTH(endpoint_a, ref_a, curve a)
                      + RIDGE_SECTION_AVERAGE_RIDGE_WIDTH(endpoint_b, ref_b, curve b)
                      + RIDGE_SECTION_AVERAGE_RIDGE_WIDTH(endpoint_c, ref_c, curve c)
                      + RIDGE_SECTION_AVERAGE_RIDGE_WIDTH(endpoint_d, ref_d, curve d))
                      / 4.0
                if ((RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[endpoint_0, endpoint_1, curve]
                            < W_DOUBLY_CONNECTED)
                        and (length of curve < F_DOUBLY_CONNECTED × w))
                    ** Check angle of attachment criteria
                    if ((DOT_PRODUCT(ref_a, endpoint_0, ref_b) < cos(A_STRAIGHT))
                        and (DOT_PRODUCT(ref_c, endpoint_1, ref_d) < cos(A_STRAIGHT))
                        and (|DOT_PRODUCT(ref_a, endpoint_0, endpoint_1)| < cos(A_ATTACH))
                        and (|DOT_PRODUCT(ref_b, endpoint_0, endpoint_1)| < cos(A_ATTACH))
                        and (|DOT_PRODUCT(ref_c, endpoint_1, endpoint_0)| < cos(A_ATTACH))
                        and (|DOT_PRODUCT(ref_d, endpoint_1, endpoint_0)| < cos(A_ATTACH)))
                        ** Check neighbor curve parallelism criteria
                        if |DOT_PRODUCT(ref_a, ref_b, ref_d – (ref_c – ref_b))| > cos(A_PARALLEL)
                            remove curve from T
                            delete endpoints of curve from endpoint_map
                            delete curve from curve_list
    return
```

```
DOT_PRODUCT[ points a, b, c ]
  return the normalized dot product of the vectors from b to a and from b to c
```

7.1.6 Small Ridge Segment Removal

The small ridge segment removal algorithm deletes curves from *curve_list* that do not share endpoint positions with any other curve (hence they are unconnected) and that have fewer numbers of points than the minimum allowed for an unconnected curve. The algorithm iterates over *curve_list* considering each curve. The curve connectivity test described above is applied to each curve. If a curve is classified as unconnected, its length is checked. If the curve is either short relative to the fingerprint as a whole (the number of pixels in the curve is less than $F_{UNCONNECTED\_CURVE} \times ridge\_width_{fingerprint}$) or both thin and short relative to neighboring curves (both the width of the curve is less than $Z_{WIDTH\_UNCONNECTED}$ times the local average ridge width and the length of the curve is less than $Z_{LENGTH\_UNCONNECTED}$ times the local average ridge width), the curve is removed from ***T***, its endpoints are deleted from the *endpoint_map*, and the curve is deleted from *curve_list*.

```
SMALL_RIDGE_SEGMENT_REMOVAL[ curve_list ]
  ** This algorithm modifies curve_list, endpoint_map, and T
  for each curve in curve_list
      z_local_ridge_width = the average of the local average ridge widths at the
                                     endpoints of curve
      if ((curve is unconnected)
          and ((number of pts in curve < F_UNCONNECTED_CURVE × ridge_width_fingerprint)
              or ((RIDGE_SECTION_AVERAGE_RIDGE_WIDTH[curve first endpt,
                                                      curve last endpt, curve]
                        < Z_WIDTH_UNCONNECTED * z_local_ridge_width)
                    and (number of points in curve
                        < Z_LENGTH_UNCONNECTED * z_local_ridge_width))))
          remove curve from T
          delete endpoints of curve from endpoint_map
          delete curve from curve_list
  return
```

7.2 SUMMARY

Parameters

A_{ATTACH} = 30 degrees	Angular limit for perpendicular attachment
$A_{COLINEAR}$ = 45 degrees	Angular limit for colinearity
$A_{PARALLEL}$ = 45 degrees	Angular limit for parallelism of neighboring ridges
$A_{SEGMENT}$ = 60 degrees	Angular limit for colinearity with a small segment
$A_{STRAIGHT}$ = 90 degrees	Angular limit for straightness
$F_{DOUBLY\_CONNECTED}$ = 2.25	Maximum length of a doubly connected curve in terms of the average of its neighboring end sections' average ridge widths
$F_{OFFSHOOT\ CURVE}$ = 2.0	Length of the smallest allowable singly connected curve in terms of $ridge\_width_{fingerprint}$
F_{RIDGE_BREAK} = 1.0	Maximum length of a possibly connectable ridge break in terms of $ridge\_width_{fingerprint}$
$F_{UNCONNECTED\_CURVE}$ = 5.0	Length of the smallest allowable unconnected curve in terms of $ridge\_width_{fingerprint}$
$L_{DOUBLY\_CONNECTED}$ = 20	Maximum length of a doubly connected curve to be considered for removal
$RADIUS_{DEFAULT}$ = 81	Default search radius for the small ridge break connection algorithm
$RIDGE\_SIZE_{MIN}$ = 5	Minimum length of a curve allowed to be used in calculating colinearity
$W_{DOUBLY\_CONNECTED}$ = 0.95	Maximum average ridge width of the doubly connected curve in terms of the average of its neighboring end sections' average ridge widths
$Z_{LENGTH\_OFFSHOOT}$ = 5.0	Length of the smallest allowable singly connected curve in terms of the local average ridge width
$Z_{LENGTH\_UNCONNECTED}$ = 10.0	Length of the smallest allowable unconnected curve in terms of the local average ridge width
$Z_{WIDTH\_OFFSHOOT}$ = 0.65	Width of the smallest allowable singly connected curve in terms of the local average ridge width
$Z_{WIDTH\_UNCONNECTED}$ = 0.65	Width of the smallest allowable unconnected curve in terms of the local average ridge width

Input

curve_list	The list of curves for the live-scan fingerprint
C	Chamfered image calculated as part of ridge thinning (section 5)
z_blockmap	Ridge direction data structure

Output

modified *curve_list*

Calculated values

T	Thinned image regenerated from *curve_list* and update as the *curve_list* is modified
endpoint_map	See definition in section 7.1.1
$ridge\_width_{fingerprint}$	The average ridge width for the entire fingerprint

SECTION 8

RIDGE SMOOTHING

Ridge smoothing processes the cleaned, extracted curves of a fingerprint's ridges to produce smoother versions of those same curves. This smoothing of the fingerprint ridges removes topologically insignificant deviations in the curve. A smoother curve will ultimately require fewer spline points to represent it; hence it will compress more efficiently. An illustration of this situation is shown in figure 50. Notice that at each small bump in the curve there are several spline points wasted on representing more detailed information about that curve than is desired. By smoothing the curve we eliminate those extra points, reducing the information to be encoded about that curve. However, it should be noted that the smoothing process preserves the general shape and the exact locations of the endpoints of each curve, therefore preserving the precise location of minutiae points.

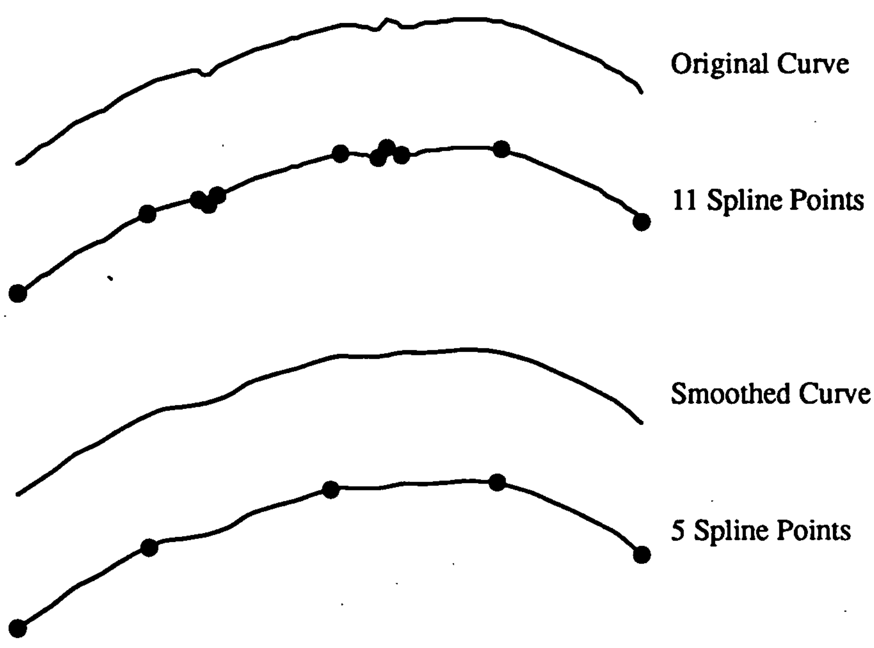


Figure 50. Illustration of the Difference in the Number of the Spline Points on a Curve and Its Smoothed Counterpart

Ridge smoothing is accomplished by a window filter that averages pixel coordinates along each curve in the fingerprint curve list. A window filter is a filter that applies a function on each subgroup of adjacent pixels as it traverses over the entire group of pixels.

Each curve consists of a list of pixel coordinates. In this algorithm, the window filter averages the pixel coordinates of a small group of pixels in a curve as it traverses the entire list of pixels in a curve. This has the effect of low-pass filtering the shape of each curve, making the curve smoother. The amount of smoothing effect is controlled by the size of the window used. The window size used during development was 15, but this value, like all other parameters, is selectable. This parameter is referred to as the target window size in the algorithm description below.

8.1 ALGORITHM DESCRIPTION

It is assumed that each curve is represented as an ordered list of pixels and that the positional coordinates are available for each pixel in the curve. It is also assumed that the first and last points in the ordered list of pixels are the endpoints for the curve. The properties of the curve extraction algorithm described earlier in section 6 guarantee these assumptions. In order to calculate a smoothed curve from an original curve, it is required that a new ordered list of pixels be generated to represent the new smoothed curve. The original pixels of a curve must be available throughout the smoothing process on that curve. Once a new ordered list for a curve is completely generated, the original ordered list for that curve may be discarded. By a property of the smoothing algorithm, the number of pixels in the new ordered list is guaranteed to be less than or equal to the sum of the number of pixels in the original ordered list and the target window size W. In practice, however, due to overlapping pixel removal, the number of new pixels is less than or equal to the original number of pixels.

The smoothing algorithm is applied to each ridge in the fingerprint independently. The three components to the smoothing algorithm are window positioning, candidate pixel calculation, and new pixel addition. Window positioning is the most complicated and the most important component in the smoothing algorithm. It controls which pixel positions are averaged together to create a candidate pixel position, and it also ensures the exact preservation of the curve's endpoints. Candidate pixel calculation generates a new candidate pixel position that is considered by the new pixel addition step for inclusion into the new smoothed curve.

Window positioning moves a window over the ordered list of pixels of a curve starting at one endpoint and stopping at the other endpoint while maintaining the front and back boundaries of the window and the current window size, $w_{current}$. The front boundary is defined to be the boundary over which new pixels are added to the window. The back boundary is defined to be the boundary over which pixels leave the window. The value of $w_{current}$ indicates the number of pixels within the window at a particular iteration. At the first iteration on a curve, $w_{current}$ is initialize to be 1, and the front and back boundaries are set to point at the starting endpoint. For the subsequent iterations until $w_{current}$ equals W or the front boundary reaches the stopping endpoint, $w_{current}$ is incremented by one and the front boundary is moved one pixel down the ordered list of pixels. If $w_{current}$ reaches W, the subsequent iterations move the window by moving both the front and back boundaries one pixel down the ordered list of pixels. When the front boundary reaches the stopping endpoint, the subsequent iterations move and shrink the window by decrementing $w_{current}$ by one and moving the back boundary one pixel down the ordered list of pixels. Iteration stops when $w_{current}$ is one and the front and back boundaries are at the stopping endpoint.

At each iteration of the window positioning the candidate pixel position is calculated as the average position of the pixels within the window. This is accomplished by averaging the row coordinate values of the pixels within the window and averaging the column coordinate values of the pixels within the window. The process of averaging these coordinates can be accelerated by keeping a running sum of the row and column coordinate values currently within the window. When the window is moved, the row and column coordinate values of the pixel leaving the window is subtracted from the respective sums, and the row and column coordinate values of the new pixel entering the window is added to the respective sums. Then the average row and column positions can be obtained by dividing these sums by $w_{current}$. This method reduces the algorithm complexity from an order n^2 to an order n. These average values, which are real numbers, are rounded to integer coordinates by selecting the nearest integer, (i.e., floor(average_value + 0.5)).

Before the new candidate pixel can be appended to the new ordered list of pixels, it must be tested to ensure it is not identical to the previously added pixel on the new list. This test is accomplished by checking if the new pixel's coordinate is the same as the coordinate of the last pixel currently in the new ordered list. If the coordinates do not match, the candidate pixel is appended to the end of the new ordered list. Otherwise, the new pixel is redundant and is not added to the list. Note that in the first iteration of the window positioning, the candidate pixel is automatically added because the new ordered list of pixels is empty. This process of curve smoothing is illustrated on an example curve in figure 51.

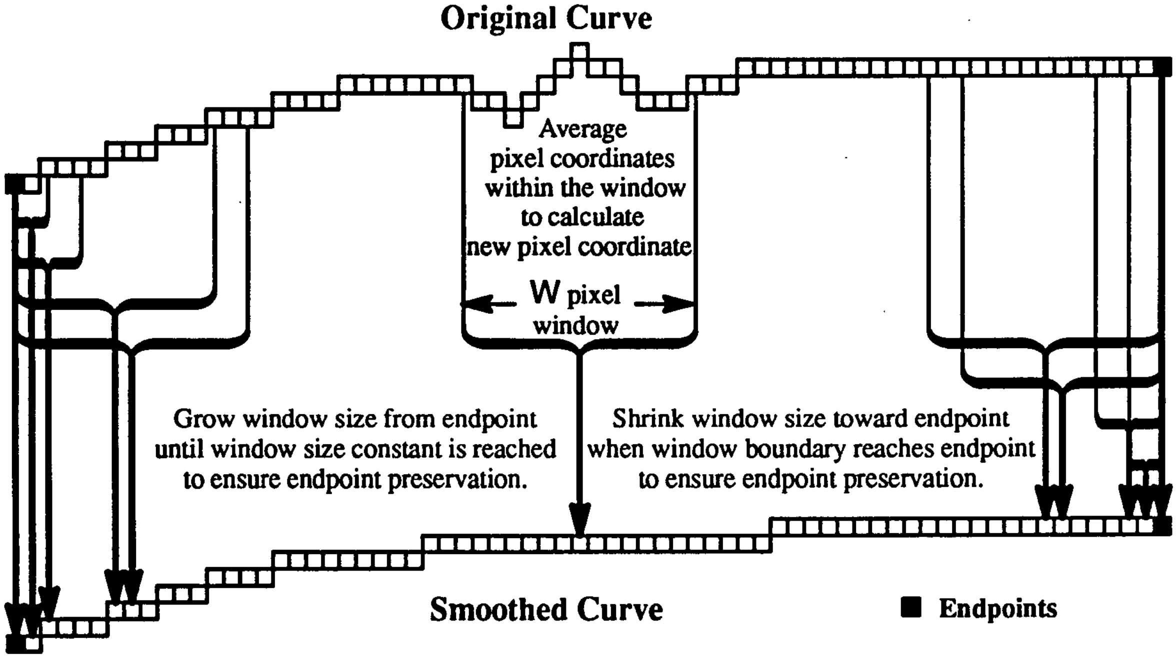


Figure 51. Illustration of the Curve Smoothing Algorithm

8.2 SUMMARY

Parameters

W	The window size constant for the smoothing window

Input

curve_list	The list of fingerprint curves from the ridge cleaning process

Output

smooth_curve_list	The list of fingerprint curves after having been smoothed

Calculated Values

$w_{current}$	The current window size
s_{row}	The running sum of the row coordinate values within the current window
s_{column}	The running sum of the column coordinate values within the current window
μ_{row}	The mean value of the row coordinate values within the current window
μ_{column}	The mean value of the column coordinate values within the current window

RIDGE_SMOOTHING[*curve_list*]

```
** This algorithm modifies curve_list
for each curve in the fingerprint curve_list
{
    generate a smooth curve on the smooth_curve_list to contain the processed curve

    s_row = 0.0
    s_column = 0.0
    w_current = 0
    set pixel b to first pixel in curve
    n = minimum [ W , number of pixels in curve ]

** Expand window while moving it until w_current reaches n
    for each pixel a in curve from first pixel to nth pixel
    {
        s_row = s_row + row coordinate of pixel a
        s_column = s_column + column coordinate of pixel a
        w_current = w_current + 1
        μ_row = floor(s_row/w_current + 0.5)
        μ_column = floor(s_column/w_current + 0.5)

        if (the pixel (μ_row, μ_column) ≠ the last pixel in smooth_curve)
            add pixel (μ_row, μ_column) to the end of smooth_curve
    }

** Move window with w_current set to n until reaching last pixel in curve
    while pixel a is not last pixel in curve
    {
        s_row = s_row + row coordinate of pixel a – row coordinate of pixel b
        s_column = s_column + column coordinate of pixel a – column coordinate of pixel b
        μ_row = floor(s_row/w_current + 0.5)
        μ_column = floor(s_column/w_current + 0.5)

        if (the pixel (μ_row, μ_column) ≠ the last pixel in smooth_curve)
            add pixel (μ_row, μ_column) to the end of smooth_curve
        set pixel b to next pixel in curve
    }
```

** Shrink window while moving it until window only contains the last pixel

while pixel b is not the last pixel in *curve*

{

$s_{row} = s_{row}$ – row coordinate of pixel b

$s_{column} = s_{column}$ – column coordinate of pixel b

$w_{current} = w_{current} - 1$

$\mu_{row} = \text{floor}(s_{row}/w_{current} + 0.5)$

$\mu_{column} = \text{floor}(s_{column}/w_{current} + 0.5)$

if (the pixel $(\mu_{row}, \mu_{column}) \neq$ the last pixel in *smooth_curve*)

add pixel $(\mu_{row}, \mu_{column})$ to the end of *smooth_curve*

set pixel b to next pixel in *curve*

}

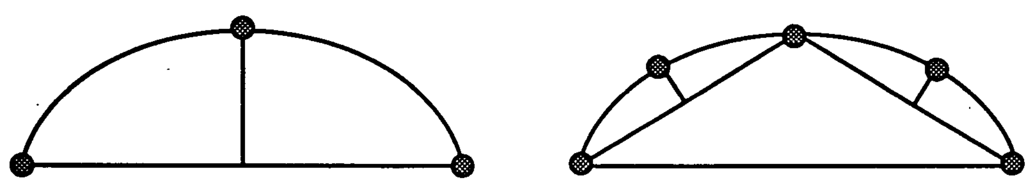
}

end

SECTION 9

CHORD SPLITTING

Chord splitting selects the fingerprint ridge points that will be used by the B-spline algorithm to reconstruct the ridge during decompression. The input to the process is an array containing an ordered set of all of the points representing a ridge. The output of the process is an ordered subset of the original ridge points. The algorithm is iterative, selecting the subset of points based upon the perpendicular distance to a line (chord) connecting the current ridge segment endpoints. This perpendicular distance is the error, or residue, for the current chord segment. A greater number of selection points result from a smaller allowable error (see figure 52).



Larger Allowable Error

Smaller Allowable Error

Figure 52. Effects of Allowable Error or Residue

9.1 ALGORITHM DESCRIPTION

The variables and parameters that the chord splitting algorithm uses are described below.

$x(i)$ The x coordinate information for the ith point on a curve segment

$y(i)$ The y coordinate information for the ith point on a curve segment

$d(i)_{jk}$ The perpendicular distance from point $(x(i), y(i))$ to the line segment with endpoints described by $(x(j), y(j))$ and $(x(k), y(k))$.

ALLOWABLE_RESIDUE The smallest acceptable perpendicular distance between curve segment and the chord segment

The chord splitting process involves several steps (figure 53) described at length in the following paragraphs. Two arrays, x and y, are passed to the chord splitting function. Array

x contains the x coordinate information for the given line segment; similarly, array y contains y coordinate information. In the input arrays, the first endpoint is referenced by j, the second endpoint is referenced by k. To calculate the residue distance, $d(i)_{jk}$, an array containing intermediate values describing the endpoint ridge segment is maintained [6]. The values within the array are

$$a(0) = y(j) - y(k)$$
$$a(1) = x(k) - x(j)$$
$$a(2) = (y(k) \times x(j)) - (y(j) \times x(k)).$$

The algorithm begins by calculating the perpendicular distance from each point on the input segment to the line segment connecting the ridge endpoints. For each point within the input ridge (subscript i), the distance is calculated using the following formula:

$$d(i)_{jk} = (a(0) \times x(i)) + (a(1) \times y(i)) + a(2).$$

The largest distance is found and the index is stored in m. If the largest distance is greater than ALLOWABLE_RESIDUE, it is acceptable and the point indexed by m is added to a linked list that stores valid spline points.

If the largest distance was above the threshold, the process is repeated for a new segment defined by the first endpoint and the point indexed by m. The algorithm continues to find the largest acceptable point. With each successful iteration a smaller line segment is defined. When an acceptable distance is not found, the algorithm moves to areas not yet investigated. The new areas are investigated by defining a new line segment with the last valid point found and the unused endpoint of the previous segment, then repeating the process described above.

In the special case of a loop, which consists of a single endpoint, the ridge endpoints are defined differently. One endpoint is defined as the actual ridge endpoint, while the other endpoint is defined as the point along the ridge that is furthest away from the ridge endpoint. The selection of these endpoints effectively divides the loop into two segments. Each of the two segments is then processed using the normal chord splitting process.

Figure 54 shows the sequence of iterations in processing a simple arc. In this figure, a line segment is joined from the original ridge endpoints labeled A and B. The largest perpendicular distance is found to be located at C. The second iteration creates a new line segment AC. The largest perpendicular distance between the ridge points A and C is found to be D. The process continues, using AD as a line segment. In this case, an acceptable distance is not found. Next, the algorithm uses the line segment joining DC, where an acceptable distance is not found. Point E is the largest distance from segment CB. A check is then made for segments CE and EB, with no valid points being found. The final points retained by the algorithm are A, D, C, E, B, in that order.

The chord splitting process is performed by a recursive function. The recursive function maintains the ordering of the selected points. This is particularly important, since the B-spline program that will use the points selected by the chord splitting algorithm expects the points to be ordered.

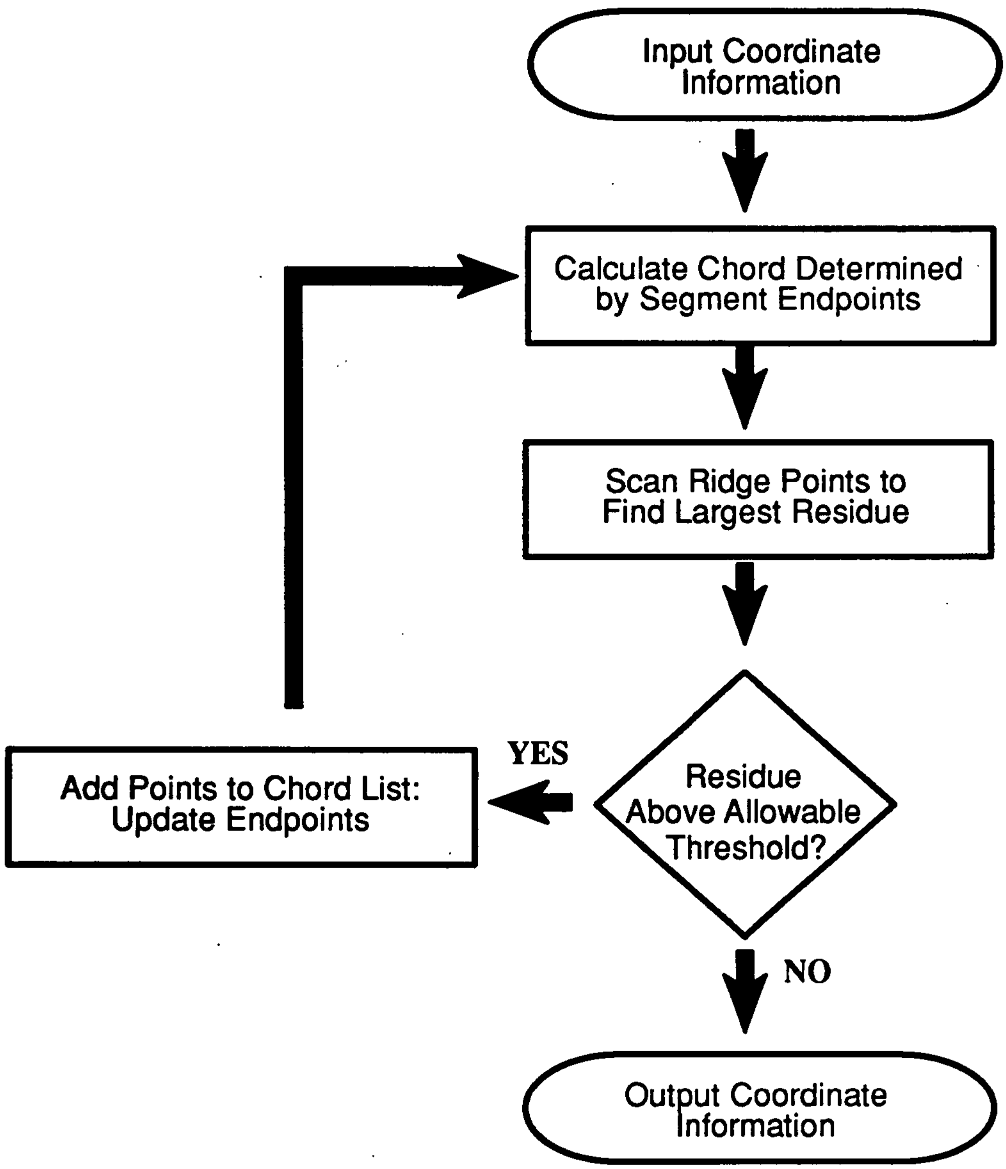


Figure 53. Flowchart of Operations

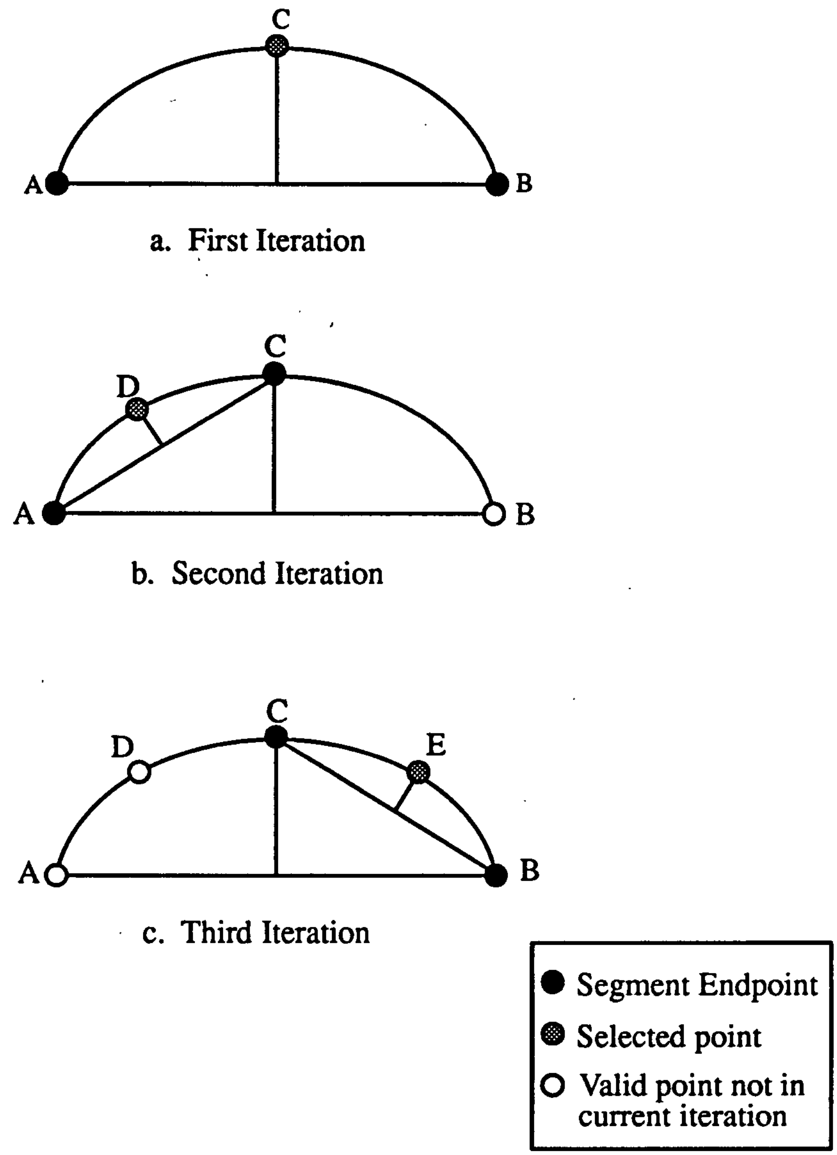


Figure 54. Sequence of Iterations

9.2 SUMMARY

This section provides parameters, input variables, output variables, and pseudocode for the chord splitting algorithm.

Parameters

ALLOWABLE_RESIDUE	Smallest acceptable perpendicular distance between the curve segment and the chord segment

Input

curve_list	List of curves that represent the fingerprint

Output

modified *curve_list*	List of curves which now includes chord points for each curve

```
** Algorithm to select the fingerprint ridge points to be used by the reconstruction algorithm
CALCULATE_CHORD_POINTS[curve_list]
    ** temp_chord_points and the arrays x and y are globally accessible by the routines
       called by this process.
    for each curve in curve_list
        ** x is an array which holds x coordinate information for curve
        ** y is an array which holds y coordinate information for curve
        if (the number of points in curve > 1)
        {
            j = first index of the coordinate arrays for curve
            k = last index of the coordinate arrays for curve

            initialize temp_chord_points with the first and last point of curve

            LINE_FITTING[ j, k ]  ** Refers to x, y, and temp_chord_points;
                                     and modifies temp_chord_points

            copy the chord points in temp_chord_points into the chord points for curve
        }
        else
            copy the one point of curve into the chord points for curve

    return
```

LINE_FITTING[j, k]

```
    ** The values of temp_chord_points and the arrays x and y are globally accessible and
       modifiable by this routine.

first_endpoint =(x(j), y(j))
second_endpoint =(x(k), y(k))

if (first_endpoint = second_endpoint)
{
    ** Special case if the curve is a loop
    m = index between j and k where (x(m), y(m)) has
            the largest distance from first_endpoint
    residue = perpendicular distance from (x(m), y(m)) to first_endpoint
}
else
{
    chord = the line passing through first_endpoint and second_endpoint
    residue = 0
    previous_residue = 0
    same_residue = 0
    for i from j to k
    {
        point = (x(i), y(i))
        p_distance = perpendicular distance from the point on the
                        curve to the chord connecting the endpoints

        ** If there are consecutive points with the same perpendicular distance,
           find the middle one

        if (p_distance ≥ residue)
        {
            residue = p_distance
            if (residue = previous_residue)
            {
                increment same_residue
            }
            else
            {
```

```
                same_residue = 0
            }
            m = i
        }
        previous_residue = p_distance
    }
    ** Find the midpoint if there are consecutive points with the
       same perpendicular distance
    if (same_residue ≠ 0)
        m = m – (same_residue ÷ 2)
}
if (residue > ALLOWABLE_RESIDUE)
{
    insert the point (x(m), y(m)) between the points in temp_chord_points that
                        correspond to first_endpoint and second_endpoint

    ** Recursive processing to continually divide segment into smaller segments

    ** Line fitting for left hand side
    LINE_FITTING[j, m]

    ** Line fitting for right hand side
    LINE_FITTING[m, k]
}
end
```

SECTION 10

CURVE SORTING

After the chord splitting process has been completed, there is no particular order to the resulting list of curves. Absolute coordinates could be used to encode the positions of the curve endpoints from this list, but this would not, in general, be very efficient. It was found that fewer bits are needed to encode the curve endpoint positions if relative offsets *between* curves are used. Thus, to further improve encoding efficiency, the list of curves are reordered to minimize the relative offsets between consecutive curves. Therefore, it is desirable to sort the curve list by closest relative offsets, taking advantage of curves that are grouped closely together to maximize encoding efficiency. The sorting process described below generates a new sorted list. The first curve in this list is represented using an absolute coordinate and the remaining curves are represented with relative offsets. Figure 55 shows an example of the results of the sorting process applied to a list of four curves.

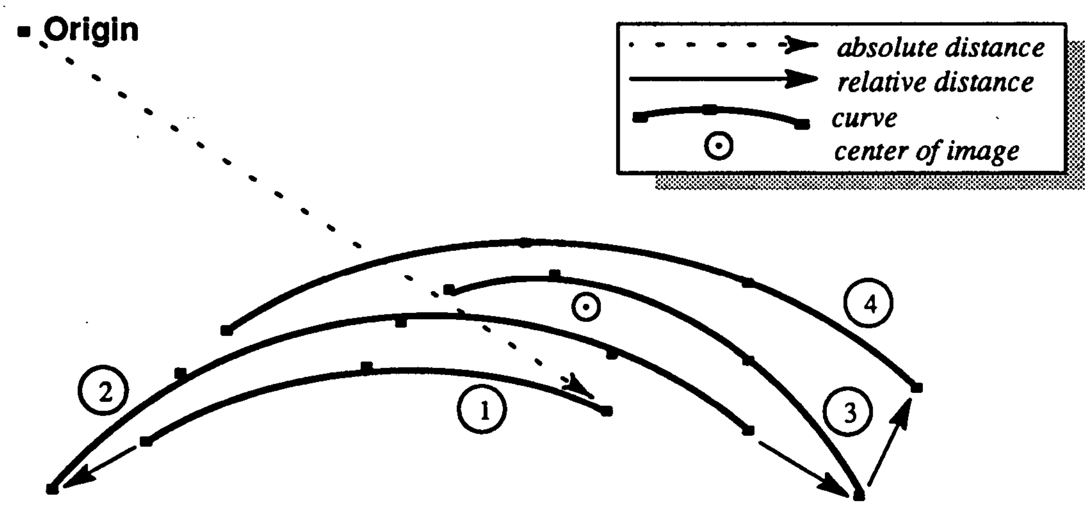


Figure 55. Results of the Sorting Process

The algorithm developed to sort the curve list does not calculate the optimal curve order, because this would be far too computationally intensive. Therefore, the algorithm described in this section is a heuristic that is far less computationally complex than optimal ordering, but still provides an efficient ordering of the fingerprint curves.

10.1 ALGORITHM DESCRIPTION

The sorting algorithm is a two stage algorithm that receives an unordered list of curves and generates an ordered list of curves. The first stage sorts the curves by repeatedly transferring the curve from the original unordered list (*unsorted_list*) that is closest to the last curve of the ordered list being generated (*sorted list*) onto the end of *sorted list*. This first stage (selective processing) usually places the entire original unsorted list of curves onto the sorted list of curves. Only when the first stage fails to place all the curves onto *sorted_list* (under conditions described in section 10.1.1.3) is the second stage reached. This stage (cyclic processing) takes any remaining unsorted curves in the original list and inserts them into the sorted list.

In both stages of sorting, a curve is selected or inserted according to a "best fit" criterion which is based on inter-curve offsets. When searching for a closest curve, the best fit criterion must be applied to every inter-curve offset (*jump*) between each endpoint of the curve being considered and each endpoint of every remaining unsorted curve to determine which of the unsorted curves minimizes *jump*. This requires that a total of four inter-curve offsets must be compared for every curve that is a candidate. Therefore, to uniquely identify *jump* for a pair of curves, the endpoints of the two curves that define this inter-curve offset must be recorded. (See section 10.1.1.1 for further details regarding the handling of these situations.)

CURVE_SORTING[*unsorted_list*]

```
sorted_list = SELECTIVE_PROCESSING[ unsorted_list ]
if (unsorted_list is not empty)
    sorted_list = CYCLIC_PROCESSING[ unsorted_list, sorted_list ]
return sorted_list
```

10.1.1 First Stage: Selective Processing

Selective processing sorts the original list of curves so as to minimize inter-curve offsets. Prior to selective processing, the curve that has either endpoint closest to the center of the image is found and is designated as the first curve in the sorted list. After the first curve is found, the remainder of the processing repeatedly selects the unsorted curve that is closest to the last curve in the sorted list and appends it to the list (see section 10.1.1.1). The flowchart in figure 56 is an overview of the selective processing stage.

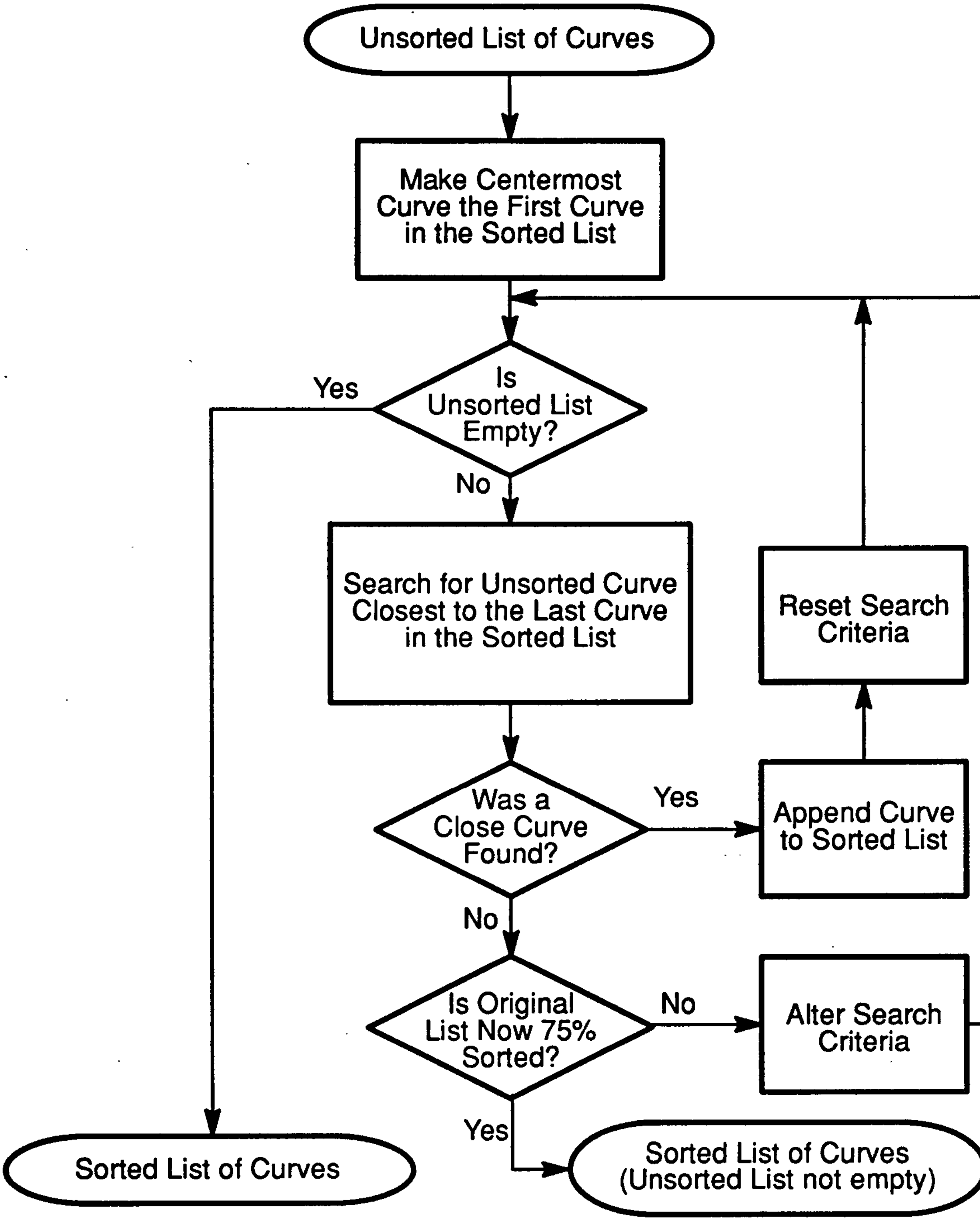


Figure 56. Flowchart of Selective Processing

```
SELECTIVE_PROCESSING[unsorted_list]

penalty_size = P_INIT                                  ** Initialized for global use

** First, find the curve closest to the center of the image and make it the first
   curve in the sorted list
curve = curve in unsorted_list that is closest to the center of the image
put curve into sorted_list

status = CONTINUE_FIRST_STAGE
while (status = CONTINUE_FIRST_STAGE)
{
    last_curve = last curve in sorted_list             ** Curve to be jumped from
    max_offset = D_SELECT                              ** Initialize the filter value

    ** The following three variables are initialized for global use. These values are
       modified in SEARCH_FOR_THE_BEST_FIT_CURVE[] and used in
    RESULTS_CHECKING[] to indicate the closest curve to last_curve
    closest_curve = NULL
    endpoint_flag = NULL
    reverse_flag = NULL

    ** Find next curve from unsorted_list, repeating the search
       with larger limits if necessary
    status = REPEAT_FIRST_STAGE_SEARCH
    while (status = REPEAT_FIRST_STAGE_SEARCH)
    {
        ** Look for a curve that is close to the last curve in the sorted list
        SEARCH_FOR_THE_BEST-FIT_CURVE[ max_offset ]

        ** Check to see if a close curve was found and perform the appropriate actions
        status = RESULTS_CHECKING[]
        if (status = REPEAT_FIRST_STAGE_SEARCH)
            max_offset = 2 × max_offset                ** The search is repeated using
                                                          twice the filter value (max_offset)
    }
}
return sorted_list
```

10.1.1.1 Search for the Best-Fit Curve

The search process examines the unsorted list of curves to find a close curve to jump to from the last curve in the sorted list. This routine computes the distance from the last curve in the sorted list to each curve in the unsorted list. Each distance comprises two values: an x offset and a y offset. Each offset is a component of the jump vector and is the magnitude of the coordinate difference from an endpoint of one curve to the endpoint of another curve. For example, the distance between endpoints (180, 200) and (140, 235) is $\langle 40, 35 \rangle$.

When computing the jump vector from the last curve in the sorted list to a curve in the unsorted list, there are four distance (jumping) scenarios to consider: the first point of the last curve in the sorted list to the first point of the current curve; the last point of the last curve in the sorted list to the first point of the current curve; the first point of the last curve in the sorted list to the last point of the current curve; and, finally, the last point of the last curve in the sorted list to the last point of the current curve. Figure 57 depicts the four distance scenarios, where each arrow represents a different endpoint offset between the two curves.

In addition to the jump distance, the reference endpoint (first or last) of the last curve as well as the closest endpoint (first or last) of the closest curve to the last curve must be noted. The reference endpoint of a curve is the endpoint from which to jump to the next curve. In situations where the closest endpoint of the closest curve is in fact its last endpoint, the list of points representing this curve are reversed. The reference endpoint information is retained using a flag, because it is required by the encoding and decoding processes. For example, this flag (the *reference_end_flag*) would be set to LAST_ENDPOINT when jumping from its last endpoint, and to FIRST_ENDPOINT when jumping from its first endpoint. (The values FIRST_ENDPOINT and LAST_ENDPOINT are used for the remainder of the document and reflect the usage in the previous example.)

Given the last curve in the sorted list, the values of the best jump vector, $\langle best\_jump_x, best\_jump_y \rangle$, represent the jump to the closest curve in the unsorted list and are determined by comparing the jumping scenarios of this last curve to every curve left in the unsorted list. Prior to the search over the unsorted list, both offset values representing the best jump vector are initialized to the value MAX_{OFFSET}. MAX_{OFFSET} is defined as the larger of the width and height of the image, plus one.

To avoid unnecessary computation, a filter test is applied before the distance comparison for each jumping scenario. The components of the inter-curve offset ($current\_jump_x$ and $current\_jump_y$) for a jumping scenario are compared to *max_offset*, the filter value. If both $current\_jump_x$ and $current\_jump_y$ are less than *max_offset*, the distance comparison (section 10.1.1.2) is applied for this jumping scenario.

Empirical analysis during development has shown that 128 is the best initial value of *max_offset* when dealing with an image of 450 pixels (horizontal) by 600 pixels (vertical).

For the remainder of the document, the value of D_{SELECT} (initial *max_offset* during the selection sort) is 128.

The best jump vector values are then used by the distance comparison process to keep track of the offset to the closest curve found so far in the current search. Therefore, if the search finds a valid closest curve, $best\_jump_x$ and $best\_jump_y$ will reflect the offsets to this curve.

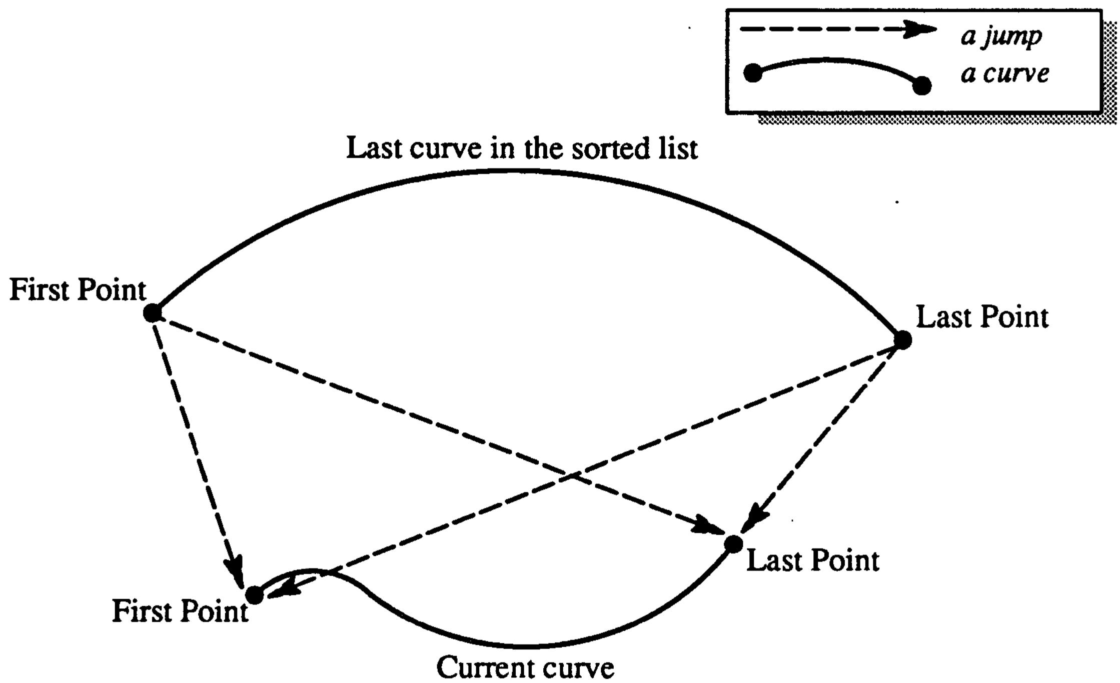


Figure 57. The Four Possible Jumping Scenarios

SEARCH_FOR_THE_BEST-FIT_CURVE[*max_offset*]

```
best_jump_x = MAX_OFFSET
best_jump_y = MAX_OFFSET

for each curve in unsorted_list
{
    ** Distance comparison is called four times, once for each
       of the four jumping scenarios

    ** Evaluate the first-to-first scenario
    current_jump_x = abs[curve_first_endpoint_x - last_curve_first_endpoint_x]
    current_jump_y = abs[curve_first_endpoint_y - last_curve_first_endpoint_y]
    if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
        if (DISTANCE_COMPARISON[current_jump, best_jump] = TRUE)
            endpoint_flag = FIRST_ENDPOINT
            reverse_flag = FALSE
            closest_curve = curve
            best_jump = current_jump

    ** Check to see if the first-to-last scenario is better
    current_jump_x = abs[curve_last_endpoint_x - last_curve_first_endpoint_x]
    current_jump_y = abs[curve_last_endpoint_y - last_curve_first_endpoint_y]
    if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
        if (DISTANCE_COMPARISON[current_jump, best_jump] = TRUE)
            endpoint_flag = FIRST_ENDPOINT
            reverse_flag = TRUE
            closest_curve = curve
            best_jump = current_jump

    ** Check if the last-to-first scenario is better
    current_jump_x = abs[curve_first_endpoint_x - last_curve_last_endpoint_x]
    current_jump_y = abs[curve_first_endpoint_y - last_curve_last_endpoint_y]
    if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
        if (DISTANCE_COMPARISON[current_jump, best_jump] = TRUE)
            endpoint_flag = LAST_ENDPOINT
            reverse_flag = FALSE
            closest_curve = curve
            best_jump = current_jump
```

```
    ** Check if the last-to-last scenario is better
    current_jump_x = abs[curve_last_endpoint_x – last_curve_last_endpoint_x]
    current_jump_y = abs[curve_last_endpoint_y – last_curve_last_endpoint_y]
    if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
        if (DISTANCE_COMPARISON[current_jump, best_jump] = TRUE)
            endpoint_flag = LAST_ENDPOINT
            reverse_flag = TRUE
            closest_curve = curve
            best_jump = current_jump
}

** The global values of closest_curve, best_jump, endpoint_flag and reverse_flag
   indicate the best fit curve found and its jump information
return
```

10.1.1.2 Distance Comparison

This section describes the distance comparison used to determine if a jump is better than the best jump, which consists of the two values $best\_jump_x$ and $best\_jump_y$. The variables $current\_jump_x$ and $current\_jump_y$ contains the pair of endpoint offsets (jump vector) from the last curve in the sorted list to the curve currently being processed.

Distance comparison uses several auxiliary functions and values. **MAX_BITS** returns the larger of the number of bits necessary to represent the *x* or *y* offset in a jump vector. **SUM_BITS** returns the aggregate number of bits necessary to represent both the *x* and *y* offsets. **SUM_DISTANCES** returns the sum of the *x* and *y* offsets. These functions are used with the current jump vector and best jump vector to obtain the values *max_current_bits*, *max_best_bits*, *sum_current_bits*, *sum_best_bits*, *sum_current_distance*, and *sum_best_distance*. The calculation of these values is explained in the pseudocode at the end of this subsection.

To prefer jumps whose offset components are roughly equivalent in magnitude, the algorithm first compares *max_current_bits* and *max_best_bits* to *penalty_size* (the penalty test). If both *max_current_bits* and *max_best_bits* are less than or equal to *penalty_size*, the current jump will be considered better than the best jump if *sum_current_distance* is less than *sum_best_distance* and if *sum_current_bits* is less than or equal to *sum_best_bits*.

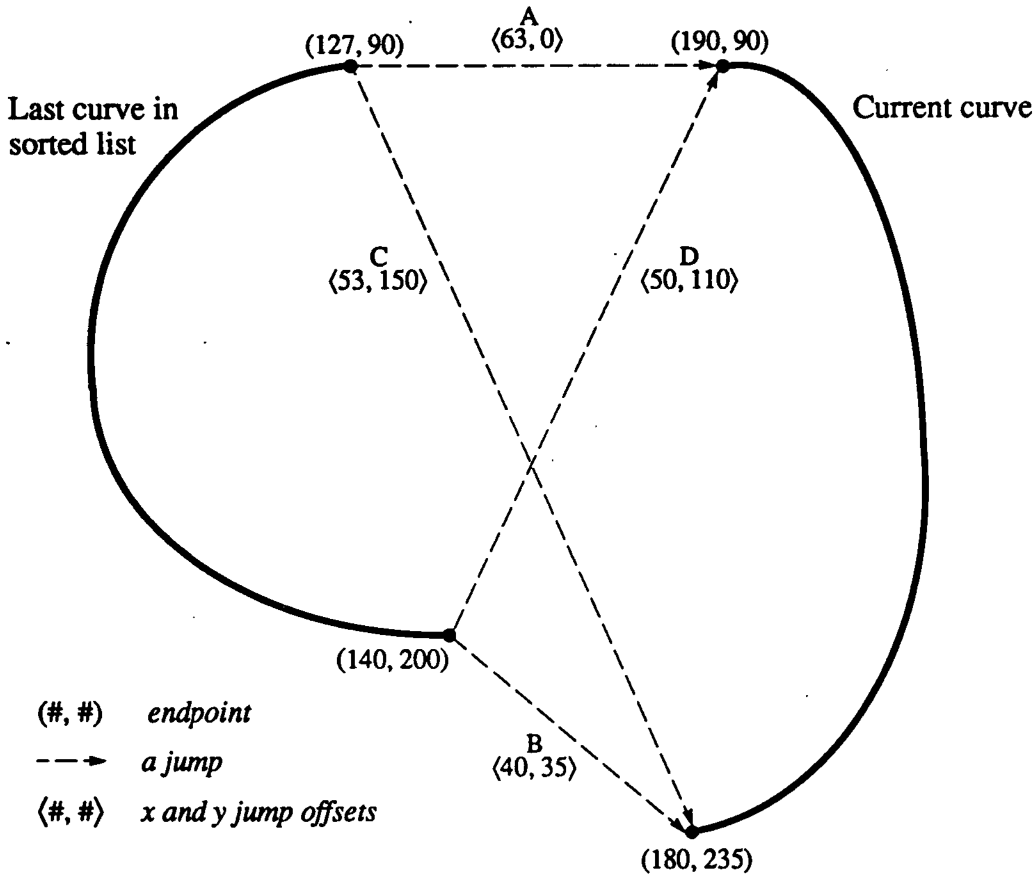
Otherwise, if either (or both) *max_current_bits* or *max_best_bits* is greater than *penalty_size*, then the current jump will be considered better than the best jump if either of the following two conditions are true:

1. *max_current_bits* is less than *max_best_bits*.

2. *max_current_bits* is equal to *max_best_bits*, and *sum_current_bits* is less than the *sum_best_bits*.

Figure 58 shows an example of comparing the four jumping scenarios between the last curve in the sorted list and the current candidate curve from the unsorted list. By passing both the filter test and the penalty test, the current curve becomes the best curve and *best_jump* is set to *current_jump*. Searching continues until every unsorted curve is examined and the *best_jump* over the entire unsorted list is found.

Empirical analysis during development has shown that 6 is the best initial value of *penalty_size* when dealing with an image of 450 pixels (horizontal) by 600 pixels (vertical). For the remainder of the document, the value for P_{INIT} (initial *penalty_size*) is 6.



jumps	offsets	sum bits	max bits	sum distances
A	⟨63, 0⟩	6 + 1 = 7	6	63 + 0 = 63
B	⟨40, 35⟩	6 + 6 = 12	6	40 + 35 = 75
C	⟨53, 150⟩	6 + 8 = 14	8	53 + 150 = 203
D	⟨50, 110⟩	6 + 7 = 13	7	50 + 110 = 160

Given that the *penalty_size* is 6, only jumps A and B satisfy the penalty test. Because *max bits* for jumps A and B are equal, the *sum bits* must be compared. Based on this comparison, jump A is selected as the best jump.

Figure 58. Example of Comparing the Four Jumping Scenarios Between the Last Curve in the Sorted List and the Current Candidate Curve

```
DISTANCE_COMPARISON[current_jump, best_jump]

    ** Now perform the penalty test
    max_current_bits = MAX_BITS[current_jump]
    max_best_bits = MAX_BITS[best_jump]
    ** max_best_bits is global
    sum_current_bits = SUM_BITS[current_jump]
    sum_best_bits = SUM_BITS[best_jump]
    sum_current_distance = SUM_DISTANCE[current_jump]
    sum_best_distance = SUM_DISTANCE[best_jump]
    if (max_current_bits ≤ penalty_size and max_best_bits ≤ penalty_size) ** Penalty Test
    {
        ** Offset components are roughly equivalent in magnitude
        if (sum_current_distance < sum_best_distance
                and sum_current_bits ≤ sum_best_bits)
            return TRUE
    }
    else
    {
        ** Offset components are not roughly equivalent in magnitude
        if (max_current_bits < max_best_bits)
            return TRUE
        else
        {
            if (max_current_bits = max_best_bits
                    and sum_current_bits < sum_best_bits)
                return TRUE
        }
    }
    return FALSE
```

The function MAX_BITS[ ] returns the largest number of bits necessary to represent the magnitude of the *x* or *y* offset.

```
MAX_BITS[jump]

    return max[NUM_BITS[jump_x], NUM_BITS[jump_y]]
```

The function SUM_BITS[ ] returns the sum of the number of bits necessary to represent the magnitudes from a pair of x and y offsets.

SUM_BITS[*jump*]

return NUM_BITS[$jump_x$] + NUM_BITS[$jump_y$]

The function SUM_DISTANCE[ ] returns the sum of the magnitudes from a pair of x and y offsets.

SUM_DISTANCE[*jump*]

return $jump_x + jump_y$

The function NUM_BITS[n] returns the smallest number of binary bits needed to represent the absolute value of the integer value n.

NUM_BITS[n]

return floor($\log_2(n)$ + 1)

10.1.1.3 Results Checking

Results checking determines if a sufficiently close curve has been found. If so, the closest curve is added to the sorted list. This curve to be appended may need to have the order of its points reversed. The assignment of the appropriate endpoint reference value to *reference_end_flag* must take such a reversal into account. This is necessary because it has been defined that a curve in the sorted list always jumps to the *first* point of the next curve in the sorted list; therefore, if the last curve jumps to the last point of the closest curve, the closest curve must be reversed prior to appending it to the sorted list. Also, *reference_end_flag* of the last curve in the sorted list must be saved to indicate which endpoint of the last curve was used to jump to the first endpoint of the closest curve. Before continuing, the penalty value is set to be the larger of itself or *max_best_bits*, because future jumps should be allowed to use as many bits as do existing jumps in the sorted list.

If a curve is not found during this search that is close enough to satisfy the distance comparison, there are two options available: the filter value is doubled and the selection sort begins again, or cyclic processing begins. This decision is based on the length of the unsorted list. If the length of the unsorted list is at or below 25 percent of its original length, then cyclic processing begins; otherwise, the filter value is doubled and the selection sort begins again. The capacity value, set at 25 percent during testing, is defined by the parameter $C_{\%}$. If the first option is chosen and a closest curve is found after further passes through the unsorted list, then the value *max_offset* must be reset to its initial value of D_{SELECT} before continuing the selective process.

Given this closest curve, results checking will decide which of the following operations to perform: to append the closest curve to the end of the sorted list; to indicate that the search should be repeated with a larger filter value; or to indicate that first stage processing has completed and second stage processing should begin.

RESULTS_CHECKING[]

```
** At least one of the best_jump offsets will no longer be set to its initialization value if
   a close curve has been found
if ((best_jump_x ≠ MAX_OFFSET) and (best_jump_y ≠ MAX_OFFSET))
{
** A close curve has been found, so append closest_curve to the sorted list
    last_curve_reference_end_flag = endpoint_flag
    if (reverse_flag = LAST_ENDPOINT)
        reverse the order of the spline points of closest_curve

    append closest_curve to sorted_list
    ** closest_curve is the new last curve in sorted_list

    penalty_size = max[max_best_bits, penalty_size]

    if (unsorted_list is empty)
        return FIRST_STAGE_FINISHED      ** First stage has successfully sorted all curves
    else
        return CONTINUE_FIRST_STAGE      ** Find next curve using the first stage sorting
}
** If a close curve was not found, determine if entry into the cyclic
   stage is necessary by checking if the number of curves in the unsorted list
   is down to C%, or less, of its original size
else if (percentage of curves in unsorted_list ≤ C% of curves in original unsorted_list)
    return FIRST_STAGE_FINISHED          ** Proceed onto the second stage sort
else     ** Cyclic processing is not permissible try once again to find a close curve
    return REPEAT_FIRST_STAGE
```

10.1.2 Second Stage: Cyclic Processing

If the cyclic stage in the sorting process is reached, the unsorted list is processed until it is completely empty. This process iterates through the unsorted list and attempts to find a place to insert each curve on that list between two curves in the sorted list. Like the first stage, the cyclic stage has a search subprocess, a distance subprocess, and a results checking subprocess. However, there exist some differences between the two processes. The first is that during cyclic processing a filter value is associated with each unsorted curve. Second, in the cyclic stage, if the search does not yield an insertion location in the sorted list, the curve is put onto the end of the unsorted list and its associated filter value is doubled.

Upon entering the cyclic stage, the filter value for every unsorted curve is initialized to D_{CYCLIC}. The value of D_{CYCLIC} used during testing was empirically determined to be 64. For the remainder of this document the parameter representing this value is referred to as D_{CYCLIC}. Each step in cyclic processing consists of three subprocesses: searching for an insertion location, linkage comparison, and results checking. These steps are performed until every remaining curve in the unsorted list has been placed in the sorted list. Once the unsorted list is empty, the sorted list is the same size as the original unsorted list, since all of the curves have been transferred to it. The flowchart in figure 59 is an overview of the cyclic processing stage.

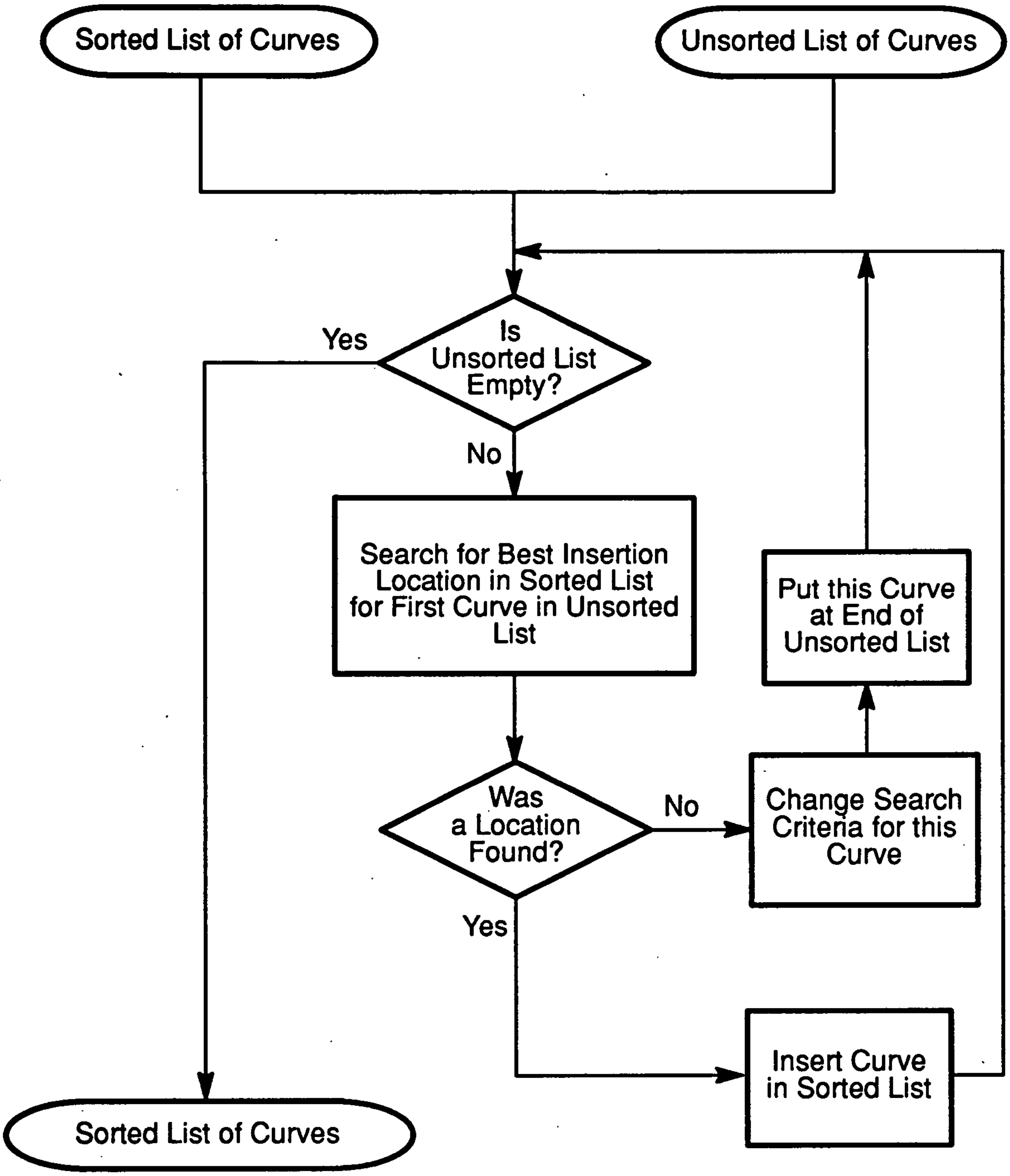


Figure 59. Flowchart of Cyclic Processing

```
CYCLIC_PROCESSING[ unsorted_list, sorted_list ]
    ** Initialize the filter value of every curve remaining in unsorted_list
    for each curve in unsorted_list
        curve_filter_value = D_CYCLIC

    while (unsorted_list is not empty)
        ** Initialize the following seven variables for global use
        saved_endpoint_flag_one = NULL
        saved_endpoint_flag_two = NULL
        saved_reverse_flag_one = NULL
        saved_reverse_flag_two = NULL
        first_curve = first curve in unsorted_list
        best_insertion_linkage_x_offset_to = first_curve_filter_value
        best_insertion_linkage_y_offset_to = first_curve_filter_value
        best_insertion_linkage_x_offset_from = first_curve_filter_value
        best_insertion_linkage_y_offset_from = first_curve_filter_value

        ** Try to find an insertion location for first_curve in the sorted list
        best_insertion_location = SEARCH_FOR_THE_BEST_INSERTION_LOCATION[]

        ** Test whether an insertion location was found for first_curve in sorted_list
        RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE[best_insertion_location]

    return sorted_list
```

10.1.2.1 Search for the Best Insertion Location

The search routine in cyclic processing operates similarly to the search routine in the first stage; however, instead of searching the unsorted list for an appropriate curve to append, the sorted list is searched for two curves (labeled i and $i+1$) that comprise an insertion location for an unsorted curve. When searching for an insertion location for an unsorted curve, the filter value *max_offset*, which is used for any distance comparisons, is set to the filter value associated with that curve. The two curves i and $i+1$ must be adjacent in the sorted list and a series of jumps must be found to go from *curve i* to the first curve in the unsorted list, then from this unsorted curve to *curve* $i+1$. Figure 60 shows the "before" and "after" appearance of an insertion location. Unlike the first stage, which has a single best jump, two different sets of best jump values must be determined. One set describes the jump from *curve i* to the candidate insertion curve, and the other set describes the jump from the candidate curve to

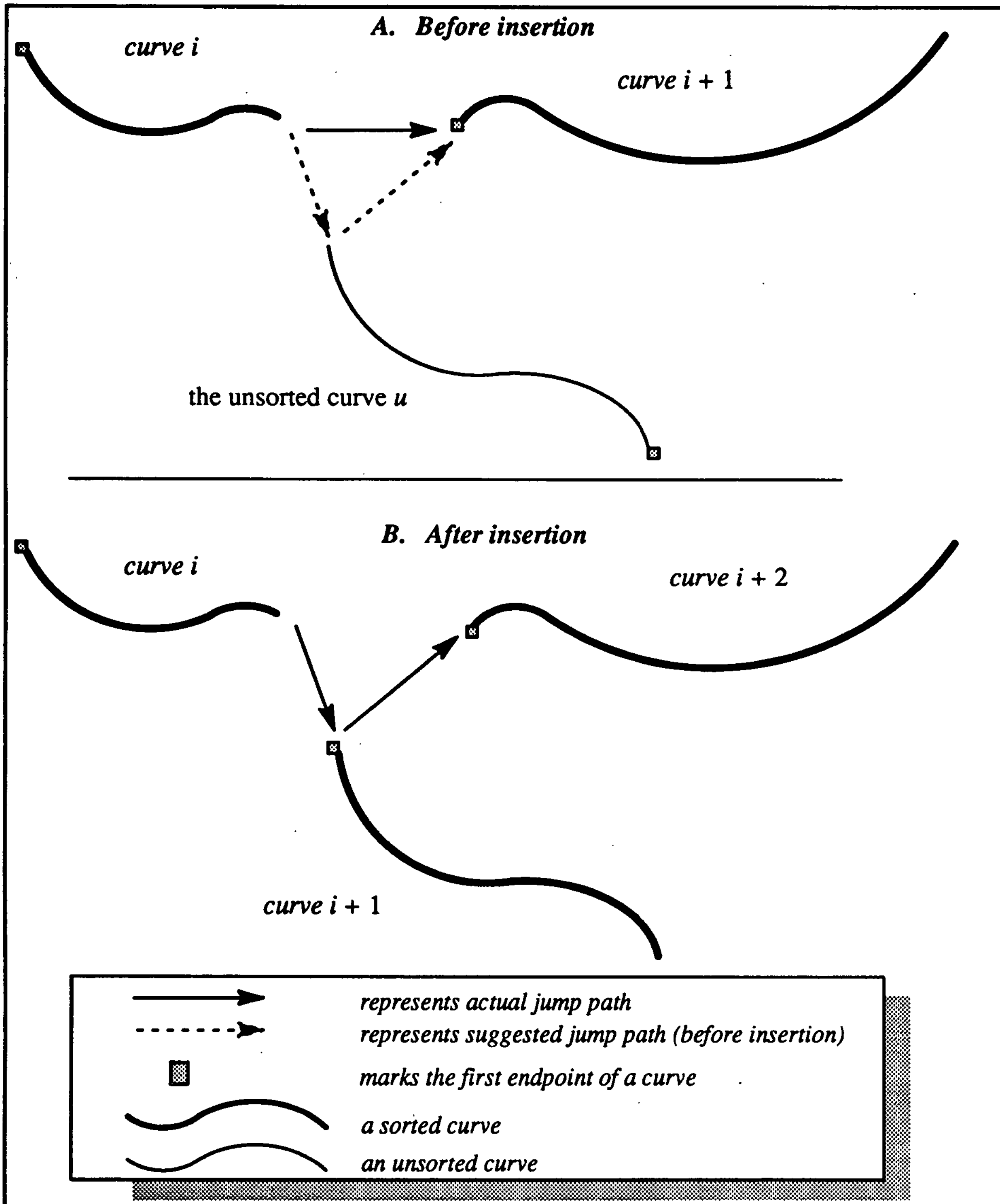


Figure 60. Insertion of an Unsorted Curve into the List of Sorted Curves

curve $i+1$. These two best jumps are the best jumping scenario that leads *to* the unsorted curve from *curve i*, and the best jumping scenario that leads *from* the unsorted curve to *curve* $i+1$. The best jumping scenarios are determined by applying the distance comparison criteria (from the first stage) over the four possible jumps between *curve i* and the unsorted curve, and also over the four possible jumps between the unsorted curve and *curve* $i+1$. The two best jumps together are called the *insertion linkage* for the insertion location between *curve i* and *curve* $i+1$. If two sets of *best_jumps* are found that pass the distance comparison criteria and result from the jumps of two adjacent sorted list curves and the unsorted curve, then a complete insertion linkage has been found. Due to the relative distance between the unsorted curve and curves i and $i+1$, however, it is possible that two *best_jumps* may not be found because one or more of the curves might not pass the filter test (section 10.1.1.2). If two feasible jumps are not found for an insertion location then the insertion location is abandoned as a possibility for the current unsorted curve. For each insertion location that yields a complete insertion linkage, a comparison against the *best_insertion_linkage* is performed to determine which insertion linkage represents the best insertion location. This comparison is described in the next section.

SEARCH_FOR_THE_BEST_INSERTION_LOCATION[]

```
max_offset = first_curve_{filter_value}
best_insertion_location = NULL          ** Initialize to indicate no location found

for each curve from the first curve in sorted_list
                    to the curve before the last curve in sorted_list
    curve_plus_one = the curve following curve in sorted_list

    ** Initialize variables
    best_jump_to_unsorted_curve_x = first_curve_{filter_value}
    best_jump_to_unsorted_curve_y = first_curve_{filter_value}
    best_jump_from_unsorted_curve_x = first_curve_{filter_value}
    best_jump_from_unsorted_curve_y = first_curve_{filter_value}
```

```
** Calculate and determine the best endpoint offset of the four endpoint offset
   pairs from curve to first_curve

** Check to see if the first-to-first scenario (from curve i to
   the candidate unsorted curve) is acceptable
current_jump_x = abs[first_curve_first_endpoint_x - curve_first_endpoint_x]
current_jump_y = abs[first_curve_first_endpoint_y - curve_first_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump, best_jump_to_unsorted_curve]
            = TRUE)
        endpoint_flag_one = FIRST_ENDPOINT
        reverse_flag_one = FALSE
        best_jump_to_unsorted_curve = current_jump

** Check to see if the first-to-last scenario (from curve i to
   the candidate unsorted curve) is better
current_jump_x = abs[first_curve_last_endpoint_x - curve_first_endpoint_x]
current_jump_y = abs[first_curve_last_endpoint_y - curve_first_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump, best_jump_to_unsorted_curve]
            = TRUE)
        endpoint_flag_one = FIRST_ENDPOINT
        reverse_flag_one = TRUE
        best_jump_to_unsorted_curve = current_jump

** Check to see if the last-to-first scenario (from curve i to
   the candidate unsorted curve) is better
current_jump_x = abs[first_curve_first_endpoint_x - curve_last_endpoint_x]
current_jump_y = abs[first_curve_first_endpoint_y - curve_last_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump, best_jump_to_unsorted_curve]
            = TRUE)
        endpoint_flag_one = LAST_ENDPOINT
        reverse_flag_one = FALSE
        best_jump_to_unsorted_curve = current_jump
```

```
** Check to see if the last-to-last scenario (from curve i to
   the candidate unsorted curve) is better
current_jump_x = abs[first_curve_last_endpoint_x − curve_last_endpoint_x]
current_jump_y = abs[first_curve_last_endpoint_y − curve_last_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump, best_jump_to_unsorted_curve]
            = TRUE)
    {
        endpoint_flag_one = LAST_ENDPOINT
        reverse_flag_one = TRUE
        best_jump_to_unsorted_curve = current_jump
    }

** Now determine the best endpoint offset of the four endpoint offset pairs
   from first_curve to curve_plus_one

** Check to see if the first-to-first scenario (from candidate unsorted curve
   to sorted curve i+1) is better
current_jump_x = abs[curve_plus_one_first_endpoint_x − first_curve_first_endpoint_x]
current_jump_y = abs[curve_plus_one_first_endpoint_y − first_curve_first_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump,best_jump_from_unsorted_curve]
            = TRUE)
        endpoint_flag_two = FIRST_ENDPOINT
        reverse_flag_two = FALSE
        best_jump_from_unsorted_curve = current_jump

** Check to see if the first-to-last scenario (from candidate unsorted curve
   to sorted curve i+1) is better
current_jump_x = abs[curve_plus_one_last_endpoint_x − first_curve_first_endpoint_x]
current_jump_y = abs[curve_plus_one_last_endpoint_y − first_curve_first_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump,best_jump_from_unsorted_curve]
            = TRUE)
        endpoint_flag_two = FIRST_ENDPOINT
        reverse_flag_two = TRUE
        best_jump_from_unsorted_curve = current_jump
```

```
** Check to see if the last-to-first scenario (from candidate unsorted curve
   to sorted curve i+1) is better
current_jump_x = abs[curve_plus_one_first_endpoint_x − first_curve_last_endpoint_x]
current_jump_y = abs[curve_plus_one_first_endpoint_y − first_curve_last_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump,best_jump_from_unsorted_curve]
            = TRUE)
        endpoint_flag_two = LAST_ENDPOINT
        reverse_flag_two = FALSE
        best_jump_from_unsorted_curve = current_jump

** Check to see if the last-to-last scenario (from candidate unsorted curve
   to sorted curve i+1) is better
current_jump_x = abs[curve_plus_one_last_endpoint_x − first_curve_last_endpoint_x]
current_jump_y = abs[curve_plus_one_last_endpoint_y − first_curve_last_endpoint_y]
if (current_jump_x < max_offset and current_jump_y < max_offset)   ** The filter test
    if (DISTANCE_COMPARISON[current_jump,best_jump_from_unsorted_curve]
            = TRUE)
        endpoint_flag_two = LAST_ENDPOINT
        reverse_flag_two = TRUE
        best_jump_from_unsorted_curve = current_jump

** If the best_jump values are all uninitialized, then this is a feasible insertion
   location and therefore must be tested to see if it is better than the best
   insertion location found so far
if ((best_jump_to_unsorted_curve_x ≠ first_curve_filter_value)
        and (best_jump_to_unsorted_curve_y ≠ first_curve_filter_value)
        and (best_jump_from_unsorted_curve_x ≠ first_curve_filter_value)
        and (best_jump_from_unsorted_curve_y ≠ first_curve_filter_value))
    current_insertion_linkage_to_jump = best_jump_to_unsorted_curve
    current_insertion_linkage_from_jump = best_jump_from_unsorted_curve

    if (LINKAGE_COMPARISON(current_insertion_linkage, best_insertion_linkage)
            = TRUE)
    ** If the current insertion linkage is the best , set flags
    {
        saved_endpoint_flag_one = endpoint_flag_one
        saved_endpoint_flag_two = endpoint_flag_two
        saved_reverse_flag_one = reverse_flag_one
```

saved_reverse_flag_two = *reverse_flag_two*
best_insertion_linkage = *current_insertion_linkage*
best_insertion_location = *curve*
}

return *best_insertion_location*

10.1.2.2 Linkage Comparison

This section describes the conditions for determining the best insertion location among all feasible candidate locations in the sorted curve list. The *best_insertion_linkage* is the linkage that best satisfies the following two tests: (1) the linkage has the smallest value for the maximum number of bits of the four component offsets in the insertion linkage, and (2) the linkage has the most component offsets that are less than or equal to S, where S is a parameter that was set to 15 during testing and has been empirically determined to give the best encoding results.

LINKAGE_COMPARISON[*current_insertion_linkage*, *best_insertion_linkage*]

** Set *current_numbits* to the number of bits of the largest offset magnitude in *current_insertion_linkage*

current_numbits = NUM_BITS[max[$current\_insertion\_linkage_{x\_offset\_to}$,
$current\_insertion\_linkage_{y\_offset\_to}$,
$current\_insertion\_linkage_{x\_offset\_from}$,
$current\_insertion\_linkage_{y\_offset\_from}$]]

** Set *current_quantity_smalls* to the sum of the offsets from *current_insertion_linkage* that are ≤ S. This is determined by the function IS_SMALL[].

current_quantity_smalls = IS_SMALL[$current\_insertion\_linkage_{x\_offset\_to}$]
+ IS_SMALL[$current\_insertion\_linkage_{y\_offset\_to}$]
+ IS_SMALL[$current\_insertion\_linkage_{x\_offset\_from}$]
+ IS_SMALL[$current\_insertion\_linkage_{y\_offset\_from}$]

** Set *best_numbits* to the number of bits of the largest offset magnitude in *best_insertion_linkage*

best_numbits = NUM_BITS[max[$best\_insertion\_linkage_{x\_offset\_to}$,
$best\_insertion\_linkage_{y\_offset\_to}$,
$best\_insertion\_linkage_{x\_offset\_from}$,
$best\_insertion\_linkage_{y\_offset\_from}$]]

** Set *best_quantity_smalls* to the number of offsets from *best_insertion_linkage* that are ≤ S, which is calculated using the function **Is_Small[]**.

best_quantity_smalls = **Is_Small**[$best\_insertion\_linkage_{x\_offset\_to}$]
+ **Is_Small**[$best\_insertion\_linkage_{y\_offset\_to}$]
+ **Is_Small**[$best\_insertion\_linkage_{x\_offset\_from}$]
+ **Is_Small**[$best\_insertion\_linkage_{y\_offset\_from}$]

** If the current linkage uses as many or fewer bits to represent offsets as the best linkage and also has more small words, then it is better than the best linkage

if (((*current_numbits* ≤ *best_numbits*)
  and (*current_quantity_smalls* > *best_quantity_smalls*))
  return TRUE

** If the current linkage uses fewer bits to represent offsets than the best linkage and has as many or more small words than the best linkage, then it is better than the best linkage

else if ((*current_numbits* < *best_numbits*)
  and (*current_quantity_smalls* ≥ *best_quantity_smalls*)))
  return TRUE

** Otherwise, the current linkage uses as many or more bits and also has as many or fewer small words and the best linkage remains unchanged

else
  return FALSE

The function **Is_Small**[] determines if *value* ≤ S, and returns one if this condition is true, otherwise it returns zero.

Is_Small[*value*]

if (*value* ≤ S)
  return 1
else
  return 0

10.1.2.3 Results Checking and Insertion of Unsorted Curve

If an insertion location for a curve in the unsorted list is found, operations are performed to insert that curve into the sorted list, reversing the order of its points if necessary, and setting or resetting *reference_end_flag* for the appropriate curves. In figure 60, it is apparent that the unsorted curve *u* would need to be reversed because the last endpoint of *u* is closer to curve *i* than to the first endpoint. Reversing the unsorted curve *u* directly affects the reference end flag for *u*. In part A of the figure, the reference end flag is initially set to represent jumping from the last endpoint. However, once *u* is reversed, the reference end flag needs to be changed to represent jumping from the first endpoint (see part B of figure 60). After the curve has been inserted, if the number of bits of any value in the *best_insertion_linkage* is larger than the penalty value, *penalty_size* is set to this maximum value.

If an insertion location is found for an unsorted curve, the curve is inserted there. If an insertion location is not found, however, this unsorted curve is placed at the end of the unsorted list and the filter value associated with this curve is doubled. In either case, the unsorted list will now have a new first curve (unless the curve being processed is the only remaining unsorted curve).

RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE[]

    ** *Best_insertion_location* will be set to a value other than NULL
       when a good insertion location has been found.

if (*best_insertion_location* ≠ NULL)
{
    before_curve = *best_insertion_location*
    after_curve = the curve after *best_insertion_location*

    insert *first_curve* between the curves *before_curve* and *after_curve*
    ** The second curve in *unsorted_list* is now the first curve

    $before\_curve_{reference\_end\_flag}$ = *saved_endpoint_flag_one*
    $first\_curve_{reference\_end\_flag}$ = *saved_endpoint_flag_two*
    if (*saved_reverse_flag_one* = TRUE)
        toggle $first\_curve_{reference\_end\_flag}$
        reverse the order of the spline points of *first_curve*
    if (*saved_reverse_flag_two* = TRUE)
        toggle $after\_curve_{reference\_end\_flag}$

```
        reverse the order of the spline points of curve after_curve
    to_endpoint_offset = best_insertion_linkage_to_endpoint_offset
    from_endpoint_offset = best_insertion_linkage_from_endpoint_offset
    component_max = NUM_BITS[max[to_endpoint_offset_x,
                                 to_endpoint_offset_y,
                                 from_endpoint_offset_x,
                                 from_endpoint_offset_y]]
    penalty_size = max[penalty_size, component_max]
}
else
{
** An insertion location was not found, therefore move this curve
   to the end of the unsorted list and double its filter value

    first_curve_filter_value = first_curve_filter_value * 2
    move first_curve to the end of unsorted_list
    ** The value of max_offset will be larger the next time first_curve is processed
}
** The curve that was originally second on the unsorted list is now first
return
```

10.2 SUMMARY

The parameter values used during development and testing of the sorting algorithm, as well as the constants, input variables, and output variables, are listed below.

Parameters

$P_{INIT} = 6$	Initial value for the penalty variable
$D_{SELECT} = 128$	Initial value assigned to the filter variable upon entering the selective processing stage
$D_{CYCLIC} = 64$	Initial filter value assigned to each curve upon entering The cyclic processing stage
$S = 15$	Limit used to test whether one insertion linkage has more small offsets than another insertion linkage
$C_{\%} = 25\%$	Maximum percentage of curves that can exist in the *unsorted_list* before the cyclic processing stage will begin if SEARCH_FOR_THE_BEST-FIT_CURVE fails
$MAX_{OFFSET} = 601$	The larger of the width and height of the image, plus one

Constants

FIRST_ENDPOINT	Flag assigned to the *reference_end_flag* of a curve when the first endpoint is used as the reference endpoint for the jump to the curve following this curve in *sorted_list*
LAST_ENDPOINT	Flag assigned to the *reference_end_flag* of a curve when the last endpoint is used as the reference endpoint for the jump to the curve following this curve in *sorted_list*

Input

unsorted_list	List of curves from the chord splitting process

Output

sorted_list	List of curves sorted by inter-curve offsets

SECTION 11

ENCODING

The encoding step follows the sorting process and has two purposes: (1) to prepare the fingerprint data for transmission, and (2) to compress the fingerprint information even further by representing it in an efficient bit-stream format. Once the data has been encoded and transmitted, the decoding step (described in section 12) reverses the process to extract and reformat the information into a more usable form. This decoded data can then be interpreted correctly by the spline reconstruction process to regenerate the image.

As shown in figure 61, the encoded data stream consists of two types of information: the fingerprint header, and the curve or ridge information. The fingerprint header consists of general data about the encoded fingerprint, which will be used by the decoding process. There is only one header record in the data stream for each fingerprint. The second type of information is the ridge data, including one ridge record for each of the ridges in the fingerprint. The ridge data consists of jump information from the endpoint of the last ridge encoded, a header of general ridge information, and the relative distances (delta offsets) between points of the ridge. Each of these will be discussed in more detail in the following sections. In summary, if the fingerprint being encoded has n ridges, the encoded data stream will contain one header record and n ridge records.

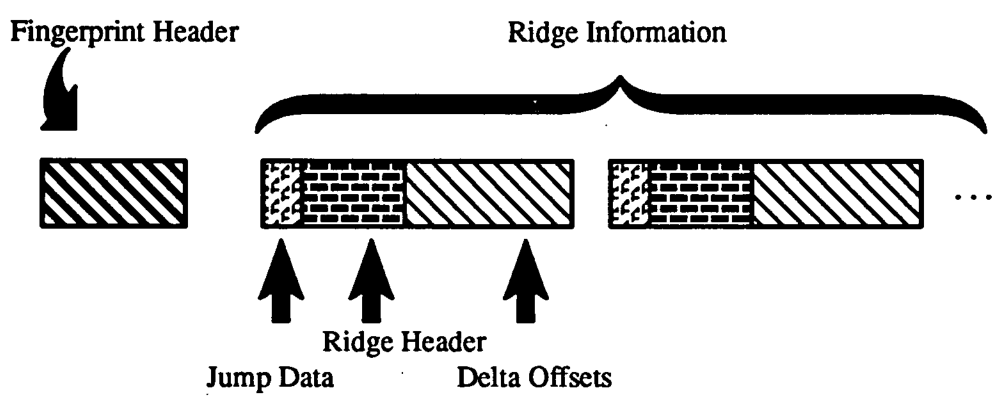


Figure 61. Encoded Fingerprint Components

Many different techniques are used to encode the data efficiently, including relative values (differential encoding), Huffman encoding, duplication elimination, a process referred to as *short/long* word encoding, and bit packing. Each of these techniques is used to reduce the number of bits required to represent data within the fingerprint. A savings of just a few bits per curve (or per point within a curve) can amount to a savings of many bits for the entire fingerprint.

11.1 EXPLANATION OF TERMS

In this section, several terms and concepts will be described that are used frequently in the subsequent sections. The first two terms, *jump values* and *delta offsets*, describe relative coordinate distances between points in separate ridges and within a ridge, respectively. The third term, *reference end*, describes the end of the ridge from which a jump is made to reach the next curve. The *monotonicity type* describes the sign fluctuation pattern for the x and y relative coordinates (delta offsets) along a ridge.

11.1.1 Delta Offsets

Figures 62 and 63 illustrate two methods that can be used to describe relative distance values used in encoding. Both relative distances are determined by computing the differences between the respective x and y values of two adjacent, or consecutive, points. The first term, delta offset, is used to describe relative distances between points along a ridge. For example, if the absolute coordinates for the first and second points in a fingerprint curve are (10,14) and (15,19), the second point can be represented relative to the first as (dx,dy) or (+5,+5) (i.e., $dx = 15 - 10$, and $dy = 19 - 14$).

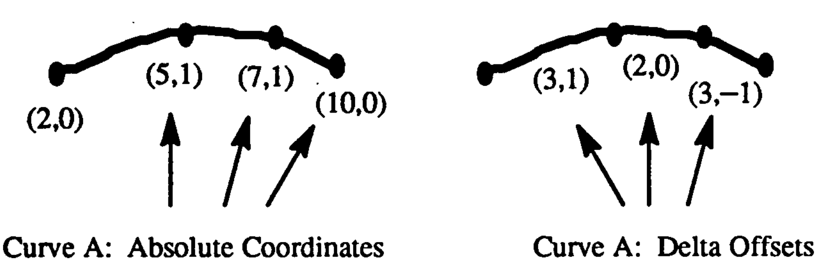


Figure 62. Absolute Coordinates and Delta Offsets Within a Curve

11.1.2 Jump Values and Reference End

Jump values (see figure 63) describe the relative distances between an endpoint of one ridge and the first endpoint of the next consecutive ridge as it is listed in the data stream. This does not necessarily mean that the jump is from the last point of one curve to the first of the next, since this may not create the shortest jump distance. The sorting process determines the best way to make the jump from one ridge to the next, and the reference end (see figure 64) is used to describe which end of the first ridge is jumped from to get to the next.

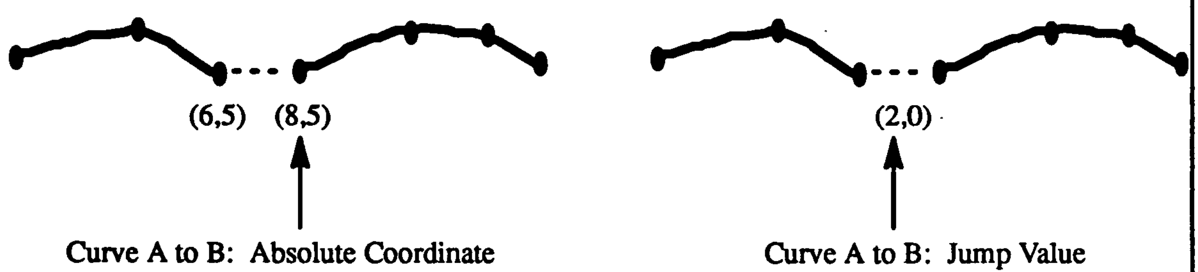


Figure 63. Absolute Coordinates and Jump Values Between Curves

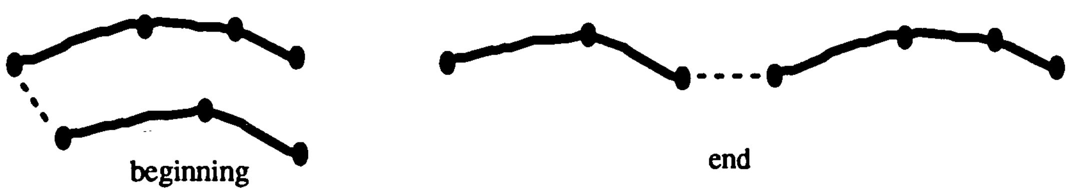


Figure 64. Reference End Values

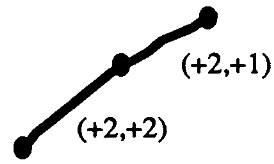
11.1.3 Monotonicity Type

Monotonicity type refers to the sign fluctuations determined for the delta offsets of a particular ridge. The four types are *monotonic both, monotonic delta x, monotonic delta y*, and *non-monotonic* (see figure 65). Recall that the delta offset values are relative distance values calculated between adjacent points along a ridge. These offsets in x and y must contain a sign flag in order to determine if there is a relative increase or decrease in the value from the last point. For example, without sign information, a (5,5) delta offset value could be interpreted as either (+5,+5), (+5,–5), (–5,+5), or (–5,–5). If a ridge can be characterized as having constant positive or negative sign values in the x and/or y coordinate, a bit savings can be achieved by encoding the pattern and sign once, and not explicitly for every value.

The sign fluctuations are determined independently for the x delta offsets and the y delta offsets along a ridge. Monotonic both refers to the case where all of the x values have the

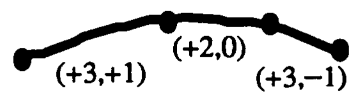
Monotonic Both

- delta x values have same sign
- delta y values have same sign



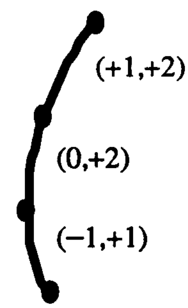
Monotonic Delta x

- delta x values have same sign
- delta y signs oscillate between positive and negative



Monotonic Delta y

- delta x signs oscillate between positive and negative
- delta y values have same sign



Non-Monotonic

- delta x signs oscillate between positive and negative
- delta y signs oscillate between positive and negative

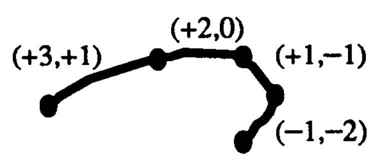


Figure 65. Sign Monotonicity Type

same sign and all of the y values have the same sign. Monotonic delta x and monotonic delta y refer to consistent signs along either the x or y values, as appropriate. Finally, non-monotonic describes those cases where both the x and y sign values fluctuate.

11.2 DESCRIPTION OF ENCODING TECHNIQUES

The following sections briefly describe several of the techniques used in encoding the flat live-scan searchprint information.

11.2.1 Relative Values

The first encoding technique, relative values, allows numbers to be specified in terms of a reference, which is provided in the fingerprint header information. Three areas where relative values are used include coordinate distances between curves (jump values), coordinate distances within curves (delta offsets), and the number of deltas per curve. Encoding this information in relative terms can provide a significant savings in the number of bits required to represent the word size(s) necessary for these values.

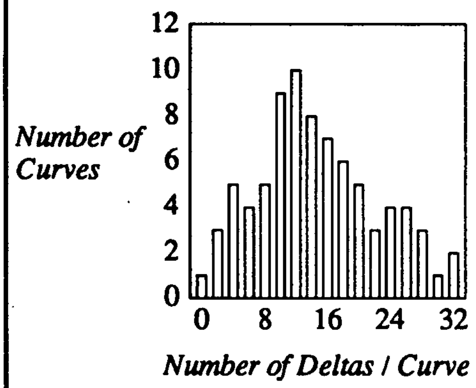
For relative distances, the reference value is the first ridge point of the fingerprint; this is the only absolute coordinate given in the data stream. The rest of the coordinates are determined by computing the differences between the respective x and y values of two adjacent points or coordinates. Using relative distances can provide a substantial reduction in the word size necessary to represent the position of a point. For example, since a flat live-scan searchprint file size is 450 pixels by 600 pixels for this study, an absolute coordinate may require as many as nine bits to represent an x value and ten bits to represent a y value. If relative coordinates are used, many fewer bits may be required for both the x and y values. Given that several hundred spline points have to be represented for a typical image, this can amount to a substantial savings.

Relative values are also used to represent the number of deltas per curve. The number of deltas per curve is important for later stages when the curves will be regenerated; however, this value can never be zero, since one point curves are not allowed. The minimum number of deltas per curve is calculated independently for each fingerprint and will generally be one, although the algorithm allows higher values. The minimum number is recorded in the header information for the fingerprint and all curves are specified relative to this value. That is, for each curve, the number of deltas is calculated as the actual number of deltas for that curve, minus the minimum number of deltas for all curves in the fingerprint. For example, given that the minimum number of deltas per curve for a particular fingerprint is one, a curve having 16 deltas will actually be encoded as having 15 deltas (i.e., 16 - 1 = 15).

An example of the bit savings achieved by relative values applied to the number of deltas per curve is given in figure 66. In this example, the original minimum number of deltas for all of the ridges in a fingerprint is one and the maximum is 32. Normally, six bits would be required to represent the maximum number. If relative values are used instead, the new minimum would be zero and the new maximum would be 31. Since 31 only requires five bits to represent, there would be a savings of one bit for the word size required.

11.2.2 Huffman Codes

With Huffman encoding, bit savings are achieved based on the frequency of occurrences of certain values (symbols), since this type of encoding assigns the most frequently used symbols to the shortest codes [7,8]. Each Huffman code is unique in that no complete Huffman code word comprises the initial sequence of bits in another Huffman code word. An example series of Huffman codes for four symbols is: 0, 10, 110, 111. Notice that the



Number of deltas per curve:
Original minimum = 1
Original maximum = 32 (6 bits)
For every curve, subtract 1 from number of deltas:
New minimum = 0
New maximum = 31 (5 bits)

➡ *Savings of 1 bit to represent the number of deltas per curve*

Figure 66. Number of Deltas per Curve Example

"0" code cannot be misinterpreted as any other code, since no other code starts with 0. Similarly, in the case of the "10" code, no other code starts with 10, and so on.

Huffman codes are used to encode the sign monotonicity type of fingerprint ridges. Assigning a monotonicity type allows the encoder to make certain assumptions about the signs of the delta offsets within a ridge. The encoder can then take advantage of redundancy by using another technique called *duplication elimination*, which will be discussed further in section 11.2.3. For monotonicity type, suppose that the distribution of ridges of each type is ordered by frequency and given in table 1. Using the Huffman codes given, the number of bits required to represent this information is: $(1 \times 60) + (2 \times 20) + (3 \times 15) + (3 \times 5) = 160$ bits. Using a straight (natural) two-bit code to represent the four symbols (i.e., 00, 01, 10, 11) would require: $2 \times 100 = 200$ bits. In this simple example, the savings from using Huffman codes is 40 bits over a straight two-bit code.

Table 1. Monotonicity Types and Huffman Codes

Monotonicity Type	# Curves	Huffman Code	# Bits
Non-Monotonic	60	0	1
Monotonic Delta x	20	10	2
Monotonic Delta y	15	110	3
Monotonic Both	5	111	3

11.2.3 Duplication Elimination

If sign monotonicity exists in the ridges, it is redundant (and costly) to assign a sign bit for every offset value in the data stream. So, monotonicity types are determined for each curve in order to identify patterns. Once the monotonicity type has been determined, the encoder can specify the sign once in the ridge header and avoid designating a sign for every offset value. For offset values with fluctuating signs along a curve, the sign bits are supplied with every offset value.

11.2.4 Short Word/Long Word

In the cases of the number of deltas per curve, delta offsets, and jump values, it may be advantageous to use more than one word size in representing the values. However, multiple word sizes incur some overhead, since varying the word size requires a flag to indicate which word size is being used. Therefore, it is necessary to perform a trade-off analysis to determine the most efficient representation.

The encoding algorithm allows a maximum of two word sizes for the number of deltas per curve and delta offsets (x and y), and three word sizes for jump values (x and y). As discussed in the following sections, two word sizes are allocated a two-bit flag in the fingerprint header (this can be implemented as a one-bit flag), and three word sizes for jump values require a one or two-bit flag, since Huffman codes are used. Note that within the delta offset and jump value categories, the word sizes for the x and y values are computed separately. The results of the word size trade-off analysis may indicate that the best representation is for the x values to have a different number of word sizes than the y values.

Since the best word size or sizes to use depends upon the distribution of specific values in a particular fingerprint, the calculations should be performed independently for each fingerprint. The easiest way to perform multiple word size analyses is first to calculate a frequency distribution by putting the values into bins indexed by the number of bits required to represent the values. Examples of these frequency distributions by number of bits are shown in figures 67, 68, and 69.

The calculation of the number of bits required for just one word size should be performed for any case where multiple word sizes are allowed. This provides a value for comparison that, for some distributions, may be the most efficient representation. Since the use of just one word size does not require flag bits, the calculation of the number of bits required is very straightforward, and is shown in the following equation:

$$B = L \sum_{i=1}^{L} f(i) \qquad (1)$$

where B denotes the total number of bits, L is the word size (number of bits) required for the largest value calculated, and $f(i)$ is the number of values to be encoded that can be represented with i bits.

11.2.4.1 Delta Offset and Number of Deltas per Curve Calculations

For the number of deltas per curve and delta offsets, equation 1 is used to calculate the number of bits required when only one word size is used. In addition, the minimum number of bits required for two word sizes must be determined:

$$B = (S + n_s) \sum_{i=1}^{S} f(i) + (L + n_l) \sum_{i=S+1}^{L} f(i) \qquad (2)$$

where B denotes the total number of bits, S is the short word size (in bits), L is the word size (in bits) required for the largest value calculated, n_s and n_l are the number of bits required for a short flag and a long flag respectively, and $f(i)$ is the number of values that can be represented with i bits. When only two word sizes are used, the flag sizes n_s and n_l are equal to one.

This value is calculated for every possible short word size, with the long word size remaining fixed, since the long word size must always represent the largest value. The first term gives the number of bits required for values representable by the short word size, the second term is the number of bits for the rest of the values, and the third term expresses the number of bits required for flag bits. The short word size that gives the minimum number of bits is determined and compared to the number of bits required if only a single word size is used. The best approach is then chosen.

Figure 67 shows an example of calculating the two best word sizes for the number of deltas per curve. The resulting 1221 bits calculated for two word sizes (including flag bits) is a much better choice than the 1795 bits required for a fixed word size.

Figure 68 provides an extensive example of calculating the best word sizes for the x and y values of the given delta offsets. Note that the best two word sizes for x require 5893 bits (including flag bits), and a fixed word size requires 7007 bits. For the y component of the delta offset, 5548 bits (including flag bits) are required by the best two word sizes, compared to 7007 bits for a fixed word size. The better choice is two word sizes for both the x and y components.

11.2.4.2 Jump Value Calculations

Due to the distribution of jump values, a maximum of three word sizes is allowed. Since ridge bifurcations are actually split into three curves by the curve extraction routine and these curves have common endpoints, there are often many zero jump values. Another alternative

Number of Deltas per Curve Distribution:

# bits	# deltas	#curves
1	0 – 1	222
2	2 – 3	89
3	4 – 7	35
4	8 – 15	12
5	16 – 31	1

Short/Long Word Size Calculations:

short word	long word	total # bits
1	5	907
2	5	862
3	5	1103
4	5	1437

Fixed Word Size: 5 bits × 359 curves = 1795 bits ⇒ no flag bits needed
Two Word Sizes: 1221 bits ⇒ includes 359 flag bits

Figure 67. Short/Long Word Sizes for Number of Deltas per Curve

Fixed Word Size:

Delta x:
7 bits × 1001 pts = 7007 bits
Delta y:
7 bits × 1001 pts = 7007 bits
⇒ no flag bits needed

Delta Offset Distribution:

# bits	Value	# of delta x	# of delta y
1	0 – 1	99	155
2	2 – 3	142	195
3	4 – 7	217	265
4	8 – 15	247	197
5	16 – 31	224	136
6	32 – 63	70	49
7	64 – 127	2	4

Delta Short/Long Word Size Calculations:

short word	long word	delta x bits	delta y bits
1	7	6413	6077
2	7	5802	5257
3	7	5175	4547
4	7	4892	4571
5	7	5149	5111
6	7	6008	6010

Two Word Sizes:

Delta x: 5893 bits
Delta y: 5548 bits
⇒ includes 1001 flag bits

Figure 68. Short/Long Word Sizes for Delta Offsets

for calculating word sizes for jump values is to use zero as one word size, and calculate the short and long word sizes from the distribution of jumps greater than zero. Again, comparisons must be made to determine whether one, two, or three word sizes is best.

For one fixed word size, the calculation is the same as for equation 1; for the short and long word values, the calculations are described by equation 2. The three word size calculation is the same as described in equation 2 with the zero jump distances removed from the distribution list. Short and long word sizes are calculated on all jumps other than zero, since zero has its own word. Fixed Huffman codewords (i.e., 0, 10, 11) are used to represent the three word size flags, where one bit is used for jump values of zero, and two bits are used for the other two word sizes. The Huffman code "0" for jump distances of zero is actually very efficient, since it requires no additional information (i.e., no sign or magnitude). Once all of the calculations are done, the best choice is made from one, two, and three word sizes.

An example of jump values and the three types of word size calculations is given in figure 69. In this example, a fixed word size for the x component of the jump value would require 2506 bits, two word sizes would require a total of 1298 bits (including flag bits), and three word sizes need only 1198 bits (including flag bits). For the y component, a fixed word size would require 2506 bits, two word sizes would require a total of 1310 bits (including flag bits), and three word sizes need only 1238 bits (including flag bits). Three word sizes is the best choice for both the x and y components in this case.

11.2.5 Bit Packing

Bit packing refers to the creation of a bit stream with the bit patterns generated by the encoding techniques described in the previous sections. The bit stream contains variable length bit patterns concatenated, from which the decoding routines can reconstruct the original information.

11.3 BIT STREAM COMPONENTS

As shown in figure 61, the bit stream consists of two major components: the fingerprint header, and the ridge information. These two components are described in more detail in the following sections.

11.3.1 The Fingerprint Header

The fingerprint header is composed of image size parameters and information necessary to interpret the ridge data. Word sizes determined for delta offsets, jump values, and the number of deltas per curve are found here, as well as the minimum number of deltas per curve for the fingerprint and the Huffman codes for interpreting monotonicity type. For the word sizes, the first value gives the number of word sizes expected to follow for that type of

Fixed Word Size:

Jump x: 7 bits × 358 pts = 2506 bits

Jump y: 7 bits × 358 pts = 2506 bits

⇒ no flag bits needed

Two Word Sizes:

Jump x: 1298 bits

Jump y: 1310 bits

⇒ includes 358 flag bits

Jump Offset Distribution:

# bits	Value	# of jump x	# of jump y
0	0	206	203
1	1	55	56
2	2 – 3	20	7
3	4 – 7	19	20
4	8 – 15	30	37
5	16 – 31	15	22
6	32 – 63	12	11
7	64 – 127	1	2

Jump Short/Long Word Size Calculations:

short word	long word	jump x bits	jump y bits
1	7	734	749
2	7	689	770
3	7	688	753
4	7	**692**	725
5	7	786	**801**
6	7	913	932

Three Word Sizes:

Jump x: 1198 bits

⇒ includes 510 flag bits

Jump y: 1238 bits

⇒ includes 513 flag bits

Figure 69. Short/Long Word Sizes for Jump Values

information, and then the actual word sizes. In the case of delta offsets and jump values, word sizes are provided for both x and y components.

One Huffman code is assigned to each of the four monotonicity types depending upon the frequency of occurrence (see section 11.2.2). The four fixed Huffman codewords in the fingerprint header are listed in order in table 2. Also shown in this figure are two example assignments of monotonicity types to the Huffman codewords. These assignments are based on the frequency of sign types within two hypothetical fingerprints. The monotonicity types are defined using a two-bit code. One example definition is given in table 3.

Table 4 describes the fingerprint header information in detail with the number of bits expected for each field. The number of bits in the header can range from a minimum of 51 bits to a maximum of 79 bits.

11.3.2 The Ridge Information

The information for a given ridge consists of three major data segments and is encoded based upon the parameters given in the fingerprint header. The three segments are jump

Table 2. Example of Monotonicity Type Assignments to Huffman Codewords

Fixed Huffman Code	Assignment 1	Assignment 2
Code 0	01	11
Code 10	11	01
Code 110	00	10
Code 111	10	00

Table 3. Monotonicity Type Codes

Monotonic Both	00
Monotonic Delta x	01
Monotonic Delta y	10
Non-monotonic	11

values, ridge header information, and delta offsets. This information structure is the same for all encoded ridges.

The jump values provide relative distance data from the reference end of the previous ridge (except for the first ridge where an absolute coordinate is used). The ridge header provides specific information required for that particular ridge, such as the number of delta offsets, the reference end, and the monotonicity type. Following the jump values and the ridge header are the delta offset values, a set of x and y values for each offset along the ridge. If the ridge is defined by n points, then the number of delta offsets is $n-1$. Zero/short/long word flags and sign flags are provided as appropriate. Table 5 gives detailed information about the fields and number of bits found in the ridge information for each curve.

The first curve in the fingerprint is encoded slightly differently from the other curves. Absolute coordinates are specified for the jump to this curve to provide context for every other point. In addition, no sign bits are used, since the absolute coordinates are always positive. For flat live-scan searchprints with a width of 450 pixels and a length of 600 pixels, nine bits are used to represent the x value and 10 bits are used to represent the y value. Delta offsets are then used for every other point in the first curve and jump values are used to reach all remaining curves.

Table 4. Fingerprint Header

Field	Number of Bits
Image Width	16
Image Height	16
Number of Ridges	11
Number of Word Sizes for Delta Offset *x* (maximum of 2)	2
Delta *x* Word Size 1	4
Delta *x* Word Size 2 (optional)	4
Number of Word Sizes for Delta Offset *y* (maximum of 2)	2
Delta *y* Word Size 1	4
Delta *y* Word Size 2 (optional)	4
Number of Word Sizes for Jump Value *x* (maximum of 3)	2
Jump *x* Word Size 1	4
Jump *x* Word Size 2 (optional)	4
Jump *x* Word Size 3 (optional)	4
Number of Word Sizes for Jump Value *y* (maximum of 3)	2
Jump *y* Word Size 1	4
Jump *y* Word Size 2 (optional)	4
Jump *y* Word Size 3 (optional)	4
Number of Word Sizes for Number of Deltas/Curve (maximum of 2)	2
Number of Deltas Word Size 1	4
Number of Deltas Word Size 2 (optional)	4
Minimum Number of Deltas	2
Coordinate Sign Huffman Codes	
Code 0	2
Code 10	2
Code 110	2
Code 111	2
	Minimum 83 bits Maximum 111 bits

Table 5. Ridge Information

Field	Number of Bits
Jump Values	
Jump *x* Zero/Short/Long Word Flag	1–2
Jump *x* Value	0, *S*, or *L*
Jump *x* Sign	0–1
Jump *y* Zero/Short/Long Word Flag	1–2
Jump *y* Value	0, *S*, or *L*
Jump *y* Sign	0–1
Ridge Header Information	
Number of Deltas Short/Long Word Flag	1
Number of Deltas Value	*S* or *L*
Reference End	1
Monotonicity Sign Type	1–3
Sign (if Monotonic)	0–1
Sign (if Monotonic Both)	0–1
Delta Offsets	
Delta *x* Short/Long Word Flag	1
Delta *x* Value	*S* or *L*
Delta *x* Sign	0–1
Delta *y* Short/Long Word Flag	1
Delta *y* Value	*S* or *L*
Delta *y* Sign	0–1
. . .	

Note: In this table, *S* represents the number of bits required for the short word size, and *L* represents the number of bits required for the long word size.

11.4 ALGORITHM DESCRIPTION AND SUMMARY

Figure 70 shows the flowchart for the encoding process. The encoding process actually consists of three stages: calculating relative distances, determining properties of the fingerprint, and, finally, encoding the data. Pseudocode is provided in the following sections for each of these steps.

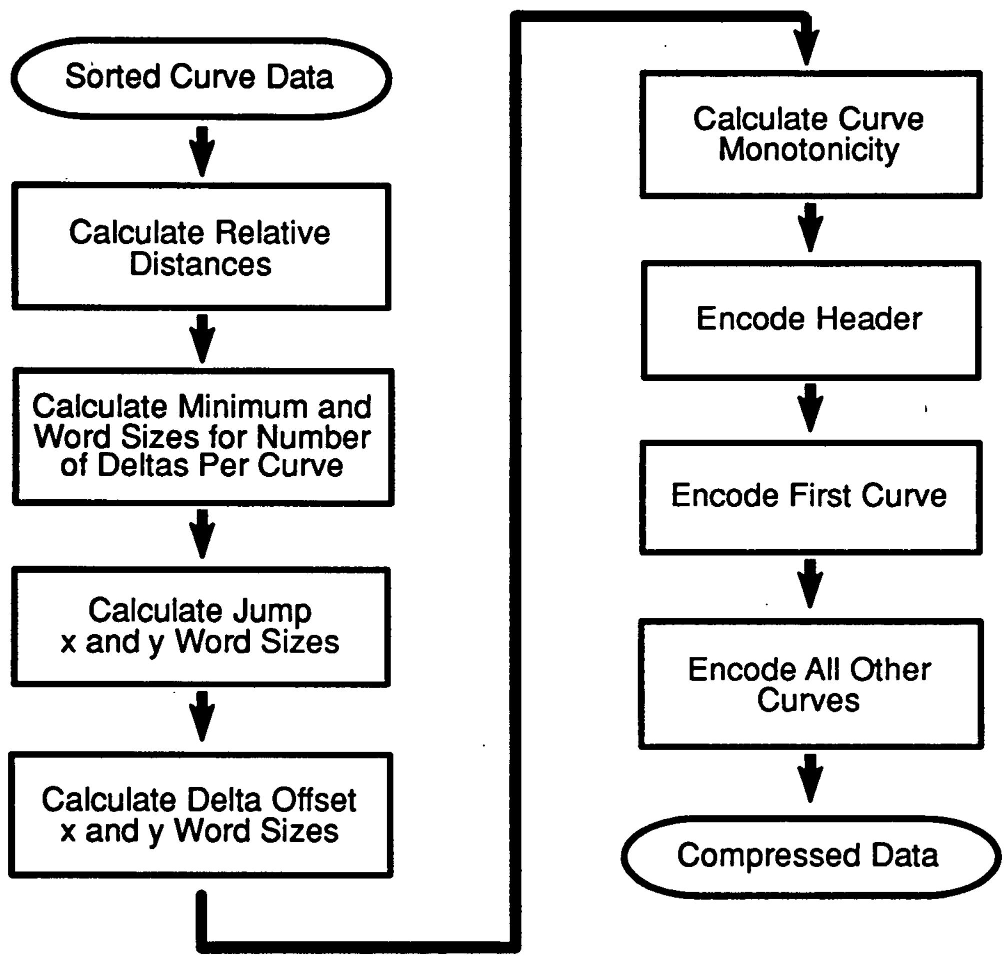


Figure 70. Encoding Flowchart

Parameters

$BITS_{IMAGE\_SIZE} = 16$	The number of bits used to represent the image size in pixels horizontally and vertically
$BITS_{NUMBER\_OF\_WORD\_SIZES} = 2$	The number of bits used to represent the number of word sizes in a word_size coding scheme
$BITS_{WORD\_SIZE} = 4$	The number of bits used to represent a word size in a word_size coding scheme
$BITS_{HUFFMAN\_INDEX} = 2$	The number of bits used to represent the sign monotonicity type index that is assigned to a particular Huffman symbol
$BITS_{NUMBER\_OF\_CURVES} = 11$	The number of bits used to represent the number of curves in the fingerprint curve list
$BITS_{X\_COORDINATE} = 9$	The number of bits used to represent an absolute x-coordinate in the live-scan fingerprint image (based on the width of the image)
$BITS_{Y\_COORDINATE} = 10$	The number of bits used to represent an absolute y-coordinate in the live-scan fingerprint image (based on the height of the image)
$BITS_{MINIMUM\_NUMBER\_OF\_DELTA} = 2$	The number of bits used to represent the minimum number of deltas of any curve of the curve list
$SIZES_{DELTAS} = 2$	Maximum number of word sizes allowed for encoding the deltas of curves
$SIZES_{JUMPS} = 3$	Maximum number of word sizes allowed for encoding the jumps between curves
$SIZES_{NUM\_DELTAS} = 2$	Maximum number of word sizes allowed for encoding the number of deltas in curves

Input

curve_list	The final list of curves from the live-scan fingerprint which are to be encoded into a data stream

Output

An encoded data stream representing a live-scan fingerprint

Calculated Values

$delta_{minimum\_per\_curve}$	Minimum number of deltas of any curve of the curve list (not to exceed that which can be represented by $BITS_{MINIMUM\_NUMBER\_OF\_DELTA}$)

```
ENCODE_FINGERPRINT[curve_list]

    ** The value of delta_minimum_per_curve is globally available to all the functions below.
    CALCULATE_RELATIVE_DISTANCES[curve_list]                    ** See section 11.4.1
    DETERMINE_FINGERPRINT_DATA_PROPERTIES[curve_list]           ** See section 11.4.2
    ENCODE_CURVE_LIST[curve_list]                               ** See section 11.4.3
    return
```

11.4.1 Calculating Relative Distances

To prepare the data for further processing the relative distance values are calculated between points passed from the sorting routine (see section 11.2.1). This includes both the delta offset values for points within a ridge and the jump values from endpoint to endpoint.

```
CALCULATE_RELATIVE_DISTANCES[curve_list]

    ** Calculate the delta offsets for all the curves in curve_list
    ** The jumps and deltas calculated here are stored in association with their curves so
       that they are available for further processing
    for each curve in curve_list
        DETERMINE_CURVE_DELTA_OFFSETS[curve]

    ** Calculate jump offsets for all the curves in curve_list
    for each curve b from second curve to last curve in curve_list
        curve a = the previous curve in curve_list before curve b
        jump from curve a to curve b = DETERMINE_CURVE_JUMP_OFFSETS[curve a,curve b]
    return
```

```
DETERMINE_CURVE_DELTA_OFFSETS[curve]                            ** See section 11.1.1

    for each point b in curve from second point to last point
        point a = the previous point in curve before point b
        delta_x from point a to point b = b_x − a_x
        delta_y from point a to point b = b_y − a_y
    return
```

DETERMINE_JUMP_OFFSET[curve *a*, curve *b*] ** See section 11.1.2

if (reference_end of curve *a* = FIRST_ENDPOINT)
    ref_pt = first point in curve *a*
else if (reference_end of curve *a* = LAST_ENDPOINT)
    ref_pt = last point in curve *a*
first_pt = first point in curve *b*
$jump_x$ from curve *a* to curve *b* = $first\_pt_x - ref\_pt_x$
$jump_y$ from curve *a* to curve *b* = $first\_pt_y - ref\_pt_y$
return *jump* from curve *a* to curve *b*

11.4.2 Determining Fingerprint Data Properties

This stage of processing determines the various values needed for encoding the data. Word sizes are calculated for five types of fingerprint information: the number of deltas per curve, jump values (x and y), and delta offsets (x and y) (see the description of each of these word size calculations in section 11.2.4). In addition, the monotonicity codes are generated based upon the sign fluctuation patterns in the fingerprint ridges (see section 11.2.2), and the minimum number of deltas per curve is found.

DETERMINE_FINGERPRINT_DATA_PROPERTIES[]

** Generate the word sizes for encoding the number of delta offsets in each curve
$delta_{minimum\_per\_curve}$ = minimum number of deltas per curve for all curves in *curve_list* not exceeding that which can be written in $\mathrm{BITS}_{\mathrm{MINIMUM\_NUMBER\_OF\_DELTA}}$

histogram = GENERATE_HISTOGRAM[number of deltas of each curve − $delta_{minimum\_per\_curve}$]
$word\_sizes_{num\_deltas}$ = DETERMINE_WORD_SIZES[*histogram*, $\mathrm{SIZES}_{\mathrm{NUM\_DELTA}}$]

** Generate the word sizes for encoding the $jump_x$
histogram = GENERATE_HISTOGRAM[all the $jump_x$]
$word\_sizes_{jumpx}$ = DETERMINE_WORD_SIZES[*histogram*, $\mathrm{SIZES}_{\mathrm{JUMP}}$]

** Generate the word sizes for encoding the $jump_y$
histogram = GENERATE_HISTOGRAM[all the $jump_y$]
$word\_sizes_{jumpy}$ = DETERMINE_WORD_SIZES[*histogram*, $\mathrm{SIZES}_{\mathrm{JUMP}}$]

** Generate the word sizes for encoding the $delta_x$
histogram = GENERATE_HISTOGRAM[all the $delta_x$]
$word\_sizes_{deltax}$ = DETERMINE_WORD_SIZES[*histogram*, $\mathrm{SIZES}_{\mathrm{DELTA}}$]

** Generate the word sizes for encoding the $delta_y$
histogram = GENERATE_HISTOGRAM[all the $delta_y$]
$word\_sizes_{deltay}$ = DETERMINE_WORD_SIZES[*histogram*, $\mathrm{SIZES}_{\mathrm{DELTA}}$]

** Assign Huffman symbol to *curve_sign_monotonicity* (See section 11.2.2)
for each *curve* **in** *curve_list*
    sign monotonicity of *curve* = DETERMINE_CURVE_SIGN_MONOTONICITY[*curve*]
    count the number of curves of each sign monotonicity type
assign the Huffman symbol 0 to the most common *curve_sign_monotonicity*
assign the Huffman symbol 10 to the next most common *curve_sign_monotonicity*
assign the Huffman symbol 110 and 111 to the remaining two *curve_sign_monotonicity*
return

GENERATE_HISTOGRAM[list of *magnitudes*]

```
** Use log2(0) = −1 to separate zero-valued elements from elements with magnitude 1
initialize all histogram bins to 0
for each magnitude in the list
    increment by one the bin of the histogram representing floor[log2(magnitude) + 1]
return histogram
```

DETERMINE_WORD_SIZES[*histogram*, *maximum_number_of_word_sizes*]

```
** See section 11.2.4
length_LW = the largest number of bits needed to represent any element of histogram
total = total number of elements in histogram
bits_min = length_LW × total
number_of_word_sizes = 1
if (maximum_number_of_word_sizes ≥ 2)
    total_SW = 0
    for SW from 0 to length_LW −1
        total_SW = total_SW + number of elements in histogram bin SW
        bits = total_SW × SW + (total − total_SW) × length_LW + total
        if (bits < bits_min)
            length_SW = SW
            number_of_word_sizes = 2
            bits_min = bits
if (maximum_number_of_word_sizes = 3)
    total_zero = number of elements in histogram equal to 0
    total_SW = 0
    for SW from 1 to length_LW −1
        total_SW = total_SW + number of elements in histogram bin SW
        bits = total_SW × (SW+2) + (total − total_SW − total_zero) × (length_LW +2) + total_zero
        if (bits < bits_min)
            length_zero = 0
            length_SW = SW
            number_of_word_sizes = 3
            bits_min = bits
return word sizes
```

DETERMINE_CURVE_SIGN_MONOTONICITY[*curve*] ** See section 11.1.3

```
if (the number of deltas in curve > 0)
    curve_sign_x = SIGN[first delta_x in curve]
    monotonic_x = TRUE
    for each delta_x from the second delta_x to the last delta_x in curve
        if (curve_sign_x = ZERO)
            curve_sign_x = SIGN[delta_x]                ** SIGN[] is defined below
        if ((SIGN[delta_x] ≠ curve_sign_x) and (SIGN[delta_x] ≠ ZERO))
            monotonic_x = FALSE
            break from loop

    curve_sign_y = SIGN[first delta_y in curve]
    monotonic_y = TRUE;
    for each delta_y from the second delta_y to the last delta_y in curve
        if (curve_sign_y = ZERO)
            curve_sign_y = SIGN[delta_y]
        if ((SIGN[delta_y] ≠ curve_sign_y) and (SIGN[delta_y] ≠ ZERO))
            monotonic_y = FALSE
            break from loop

    ** If a curve_sign is ZERO, force it to POSITIVE for encoding purposes
    if (curve_sign_x = ZERO)
        curve_sign_x = POSITIVE
    if (curve_sign_y = ZERO)
        curve_sign_y = POSITIVE

    if (monotonic_x and monotonic_y)
        curve_sign_monotonicity = MONOTONIC_BOTH
    else if (monotonic_x)
        curve_sign_monotonicity = MONOTONIC_DX
    else if (monotonic_y)
        curve_sign_monotonicity = MONOTONIC_DY
    else
        curve_sign_monotonicity = NON_MONOTONIC

    return curve_sign_monotonicity
end
```

```
SIGN[value]
if (value > 0)
    return POSITIVE
else if (value < 0)
    return NEGATIVE
else
    return ZERO
end
```

11.4.3 Encoding

Once all the auxiliary information has been calculated, the actual encoding of the bit stream can begin. First, the header is encoded with the information shown in table 4. Then, the first curve is encoded with absolute coordinates being given for the first point of this curve. The rest of the first curve (the ridge header and delta offset information) is the same as shown in table 5. Finally, all other curves are encoded as shown in table 5.

```
ENCODE_CURVE_LIST[curve_list]                                    ** See section 11.3
ENCODE_HEADER[ ]

OUTPUT_STREAM[number of curves in curve_list, BITS_NUMBER_OF_CURVES]

** Encode first curve of curve_list
OUTPUT_STREAM[x-coordinate of first point in first curve, BITS_X_COORDINATE]
OUTPUT_STREAM[y-coordinate of first point in first curve, BITS_Y_COORDINATE]
ENCODE_CURVE_DELTAS[first curve of curve_list]

** Encode the rest of the curves in curve_list
for each curve from second curve to last curve in curve_list
    ENCODE_JUMP[curve previous to curve in curve_list, curve]
    ENCODE_CURVE_DELTAS[curve]
return
```

ENCODE_HEADER[ ] ** See section 11.3.1

** Write image size to stream
OUTPUT_STREAM[I_{width}, $\text{BITS}_{\text{IMAGE\_SIZE}}$]
OUTPUT_STREAM[I_{height}, $\text{BITS}_{\text{IMAGE\_SIZE}}$]

** Write code strategies to stream (header)
ENCODE_WORD_SIZES[$word\_sizes_{deltax}$]
ENCODE_WORD_SIZES[$word\_sizes_{deltay}$]
ENCODE_WORD_SIZES[$word\_sizes_{jumpx}$]
ENCODE_WORD_SIZES[$word\_sizes_{jumpy}$]
ENCODE_WORD_SIZES[$word\_sizes_{num\_deltas}$]
OUTPUT_STREAM[$delta_{minimum\_per\_curve}$, $\text{BITS}_{\text{MINIMUM\_NUMBER\_OF\_DELTA}}$]
OUTPUT_STREAM[sign monotonicity type for symbol 0, $\text{BITS}_{\text{HUFFMAN\_INDEX}}$]
OUTPUT_STREAM[sign monotonicity type for symbol 10, $\text{BITS}_{\text{HUFFMAN\_INDEX}}$]
OUTPUT_STREAM[sign monotonicity type for symbol 110, $\text{BITS}_{\text{HUFFMAN\_INDEX}}$]
OUTPUT_STREAM[sign monotonicity type for symbol 111, $\text{BITS}_{\text{HUFFMAN\_INDEX}}$]
return

ENCODE_WORD_SIZES[*word_sizes*]

** *word_sizes* is a list of word sizes used in encoding particular types of data (e.g., $word\_sizes_{num\_deltas}$ indicates the sizes of words in bits used in encoding the number of deltas in a curve)
OUTPUT_STREAM[number of word sizes in *word_sizes*, $\text{BITS}_{\text{NUMBER\_OF\_WORD\_SIZES}}$]
for each *word_size* **in** *word_sizes*
  OUTPUT_STREAM[*word_size*, $\text{BITS}_{\text{WORD\_SIZE}}$]
return

OUTPUT_STREAM[*value*, *n*] ** See section 11.2.5

** Note: If *n* is missing on invocation of **OUTPUT_STREAM**[], the number of bits required to append *value* will be obvious from the definition of *value* (e.g., a Huffman symbol)
append *value* in *n* bits onto end of the encoded data stream
return

ENCODE_USING_WORD_SIZES[*magnitude*, *word_sizes*]

```
for word_size from smallest to largest
    if (ceil[log2(magnitude)] ≤ word_size)
        OUTPUT_STREAM[Huffman symbol for word_size]
        if (word_size ≠ 0)
            OUTPUT_STREAM[magnitude, word_size]
    break from loop
return
```

ENCODE_JUMP[curve *a*, curve *b*]

```
ENCODE_JUMP_REFERENCE_END[curve a]
ENCODE_USING_WORD_SIZES[jump_x from curve a to curve b, word_sizes_jumpx]
ENCODE_SIGN[SIGN[jump_x from curve a to curve b]]
ENCODE_USING_WORD_SIZES[jump_y from curve a to curve b, word_sizes_jumpy]
ENCODE_SIGN[SIGN[jump_y from curve a to curve b]]
return
```

ENCODE_JUMP_REFERENCE_END[*curve*]

```
if (reference end of curve = FIRST_ENDPOINT)
    OUTPUT_STREAM[0, 1]
else if (reference end of curve = LAST_ENDPOINT)
    OUTPUT_STREAM[1, 1]
return
```

ENCODE_SIGN[*sign*]

```
    ** Note: If sign is ZERO, nothing is appended to the output_stream
if (sign = NEGATIVE)
    OUTPUT_STREAM[1, 1]
else if (sign = POSITIVE)
    OUTPUT_STREAM[0, 1]
return
```

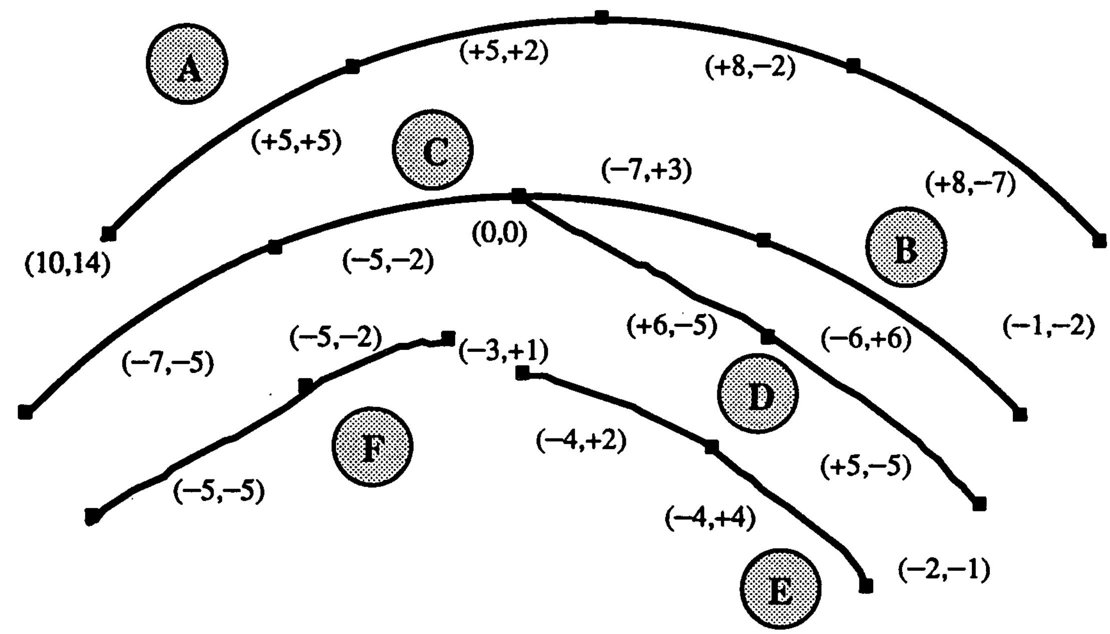
ENCODE_CURVE_DELTAS[*curve*]

```
delta_count = number of deltas in curve – delta_minimum_per_curve
ENCODE_USING_WORD_SIZES[delta_count, word_sizes_num_deltas]

if (sign monotonicity of curve = MONOTONIC_BOTH)
    OUTPUT_STREAM[Huffman symbol for MONOTONIC_BOTH]
    ENCODE_SIGN[signx of curve]
    ENCODE_SIGN[signy of curve]
    for each delta in curve
        ENCODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        ENCODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
else if (sign monotonicity of curve = MONOTONIC_DX)
    OUTPUT_STREAM[Huffman symbol for MONOTONIC_DX]
    ENCODE_SIGN[signx of curve]
    for each delta in curve
        ENCODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        ENCODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        ENCODE_SIGN[SIGN[delta_y]]
else if (sign monotonicity of curve = MONOTONIC_DY)
    OUTPUT_STREAM[Huffman symbol for MONOTONIC_DY]
    ENCODE_SIGN[signy of curve]
    for each delta in curve
        ENCODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        ENCODE_SIGN[SIGN[delta_x]]
        ENCODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
else if (sign monotonicity of curve = NON_MONOTONIC)
    OUTPUT_STREAM[Huffman symbol for NON_MONOTONIC]
    for each delta in curve
        ENCODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        ENCODE_SIGN[SIGN[delta_x]]
        ENCODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        ENCODE_SIGN[SIGN[delta_y]]
return
```

11.5 FINGERPRINT EXAMPLE

This section contains a very simple example to illustrate the encoding process. The example contains only six fingerprint ridges (see figure 71). It has been constructed to illustrate delta offsets, jump values, reference ends, word sizes, monotonicity types, and encoding to create the bit stream. Word size calculations are not explicitly shown, but can be easily derived. Table 6 shows the fingerprint header information for this example, and table 7 shows the encoded ridge information. Both tables show the information first in decimal and then in binary bit-stream form.



Number of Deltas:
Curve A = 4; Curves B,C,D,E,F = 2;
Minimum Number of Deltas/Curve = 2
Number of Deltas Word Sizes = 1 and 2 bits
Delta x Offsets:
Delta x Short Word Size = 3 bits
Delta x Long Word Size = 4 bits
Delta y Offsets:
Delta y Short Word Size = 2 bit
Delta y Long Word Size = 3 bits

Jump x Values:
Jump x Short Word Size = 1 bit
Jump x Long Word Size = 2 bits
Jump y Values:
Jump y Short Word Size = 1 bits
Jump y Long Word Size = 2 bits
Monotonicity:
Monotonic Both = 4 curves
Monotonic Delta x = 2 curves

Figure 71. Encoding Example Ridges

Table 6. Encoded Fingerprint Header

Decimal Values

Image Width 450
Image Height 600

Delta Offsets						Jump Values									Deltas/Curve				Huffman Codes			
# rdgs	# wd	xwd sz	xwd sz	# wd	ywd sz	ywd sz	# wd	xwd sz	xwd sz	xwd sz	# sz	ywd wd	ywd sz	ywd sz	# wd	wd sz	wd sz	min del	code 0	code 10	code 110	code 111
6	2	3	2	2	4	3	2	1	1		2	2	2		2	1	2	2	0	1	2	3

Binary Values

0000000111000010 0000001001011000
00000000110 10 0011 0100 10 0010 0011 10 0001 0010 10 0001 0010 10 0001 0010 10 00 01 10 11

Table 7. Encoded Ridge Information

Decimal Values

	Jump Values						Ridge Header						Delta Offsets					
	wd s/l	val	sign	wd s/l	val	sign	wd s/l	# of del	ref end	sign type	sign	sign	wd s/l	val	sign	wd s/l	val	sign
A		10			14		1	2	1	10	0		0	5		1	5	0
													0	5		0	2	0
													1	8		0	2	1
													1	8		1	7	1
B	0	1	1	1	2	1	0	0	1	0	1	0	0	6		1	6	
													0	7		0	3	
C	0	0		0	0		0	0	0	0	1	1	0	5		0	2	
													0	7		1	5	
D	0	0		0	0		0	0	1	0	0	1	0	6		1	5	
													0	5		1	5	
E	1	2	1	0	1	1	0	0	1	0	1	0	0	4		1	4	
													0	4		0	2	
F	1	3	1	0	1	0	0	0		0	1	1	0	5		0	2	
													0	5		1	5	

Binary Values

000001010 0000001100 1 10 1 10 0 0 101 1 101 0 0 101 0 10 0 1 1000 0 10 1 1 1000 1 111 1
0 1 1 1 10 1 0 0 1 0 1 0 0 110 1 110 0 111 0 11
0 0 0 0 0 0 0 0 1 1 0 101 0 10 0 111 1 101
0 0 0 0 0 0 1 0 0 1 0 110 1 101 0 101 1 101
1 10 1 0 1 1 0 0 1 0 1 0 0 110 1 110 0 110 0 10
1 11 1 0 1 0 0 0 0 1 1 0 101 0 10 0 101 1 101

SECTION 12

DECODING

Decoding the bit stream after transmission is a strictly mechanical process. Since only transmission of the data occurs between encoding and decoding, the form of the data to be decoded is the same as shown in figure 61, and tables 4 and 5, of section 11. The fingerprint header information is parsed and interpreted first, providing the information needed to decode the subsequent ridge information. After the decoding process interprets and expands all of the fingerprint header and ridge information, it is passed to the final processing stage, the ridge reconstruction algorithm.

The decoding algorithm is position-based and flag-based. That is, each category of information is interpreted either by its position in the bit stream, or by a flag preceding it, which tells the decoder how to interpret the subsequent information. It is basically the reverse of encoding, but much simpler, since no analyses of the data are performed. The types of flags used are short/long word flags, reference end flags, monotonicity flags, and sign flags.

12.1 ALGORITHM DESCRIPTION

The decoding process consists of three steps (see figure 72). First, the fingerprint header information is parsed and decoded. Information extracted from the fingerprint header is then used to assist in the second step, parsing and decoding the ridge information. Finally, after all of the binary bit stream has been parsed and interpreted, relative values, such as delta offsets and jump values, are converted to absolute coordinates.

Parsing and interpreting the fingerprint header is very straightforward. The values are interpreted one by one, as shown in table 4 of section 11. Notice that the size of the header varies with the number of word sizes calculated for delta offsets, jump values, and number of deltas per curve. This, in turn, depends upon the distribution of these numbers in each fingerprint. Table 3 of section 11 shows the monotonicity type codes used for the two-bit allocation in assigning Huffman codes.

As with the encoder, the decoder expects the first point of the first curve of the ridge information to be represented with absolute coordinates and all following points to be represented in relative coordinates. This first absolute coordinate provides the context for converting all subsequent points to absolute coordinates. For flat live-scan fingerprints used in generating and testing these algorithms, the image width is 450 pixels and the image height is 600 pixels, requiring nine and 10 bits, respectively, for the absolute coordinate. Note that in this instance no short/long word flag is needed.

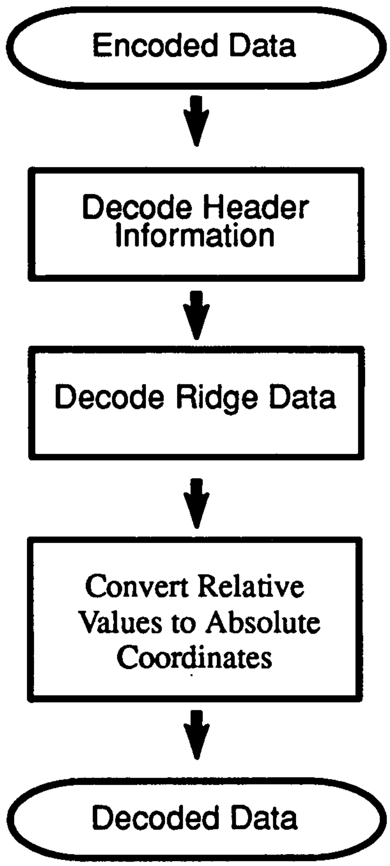


Figure 72. Decoding Processing Steps

Following the first curve, all remaining curves are expected to be in the form shown in table 5 of section 11. These values are also parsed and interpreted one by one. The interpretation of one value may preclude the need for another value. For instance, a jump value of zero eliminates the need for a sign flag, and a monotonicity sign type of monotonic delta x (or y), or monotonic both, eliminates the need for some or all coordinate sign flags. Note that one bit is used to represent the word size for delta offsets and number of deltas per curve, where zero indicates a short word size and one indicates a long word size. Zero or two bits are required for Huffman encoded jump value word sizes, since three word sizes are allowed: "0" indicates the zero word size, "10" the short word size, and "11" the long word size.

12.2 SUMMARY

This section provides parameters, input variables, output variables, and pseudocode for the decoding algorithm.

Parameters

$BITS_{IMAGE\_SIZE} = 16$	The number of bits used to represent the image size in pixels horizontally and vertically
$BITS_{NUMBER\_OF\_WORD\_SIZES} = 2$	The number of bits used to represent the number of word sizes in a word_size coding scheme
$BITS_{WORD\_SIZE} = 4$	The number of bits used to represent a word size in a word_size coding scheme
$BITS_{HUFFMAN\_INDEX} = 2$	The number of bits used to represent the sign monotonicity type index which is assigned to a particular Huffman symbol
$BITS_{NUMBER\_OF\_CURVES} = 11$	The number of bits used to represent the number of curves in the fingerprint curve list
$BITS_{X\_COORDINATE} = 9$	The number of bits used to represent an absolute x coordinate in the live-scan fingerprint image (based on the width of the image)
$BITS_{Y\_COORDINATE} = 10$	The number of bits used to represent an absolute y coordinate in the live-scan fingerprint image (based on the height of the image)
$BITS_{MINIMUM\_NUMBER\_OF\_DELTA} = 2$	The number of bits used to represent the minimum number of deltas of any curve of the curve list

Input

An encoded data stream representing a live-scan fingerprint

Output

curve_list	The reconstructed list of curves from the live-scan fingerprint which had been encoded into a data stream

Calculated Values

$delta_{minimum\_per\_curve}$	Minimum number of deltas of any curve of the curve list (not to exceed that which can be represented by $BITS_{MINIMUM\_NUMBER\_OF\_DELTA}$)

```
DECODE_CURVE_LIST[encoded fingerprint data stream]
DECODE_HEADER[ ]
INPUT_STREAM[number of curves in curve_list, BITS_NUMBER_OF_CURVES]
    ** Decode first curve of curve_list
INPUT_STREAM[x-coordinate of first point in first curve, BITS_X_COORDINATE]
INPUT_STREAM[y-coordinate of first point in first curve, BITS_Y_COORDINATE]
DECODE_CURVE_DELTAS[first curve of curve_list]
    ** Decode the rest of the curves in curve_list
for each curve from second curve to last curve in curve_list
    DECODE_JUMP[curve previous to curve in curve_list, curve]
    DECODE_CURVE_DELTAS[curve]
    ** Reconstruct the absolute coordinates for the points in each curve of curve_list
APPLY_CURVE_DELTA_OFFSETS[first curve of curve_list]
for each curve b from second curve to last curve in curve_list
    curve a = the previous curve in curve_list before curve b
    APPLY_JUMP_OFFSETS[curve a, curve b]
    APPLY_CURVE_DELTA_OFFSETS[curve b]
return curve_list
```

```
DECODE_HEADER[ ]
    ** Read image dimensions from stream
INPUT_STREAM[I_width, BITS_IMAGE_SIZE]
INPUT_STREAM[I_height, BITS_IMAGE_SIZE]
    ** Read coding strategies from stream (header)
DECODE_WORD_SIZES[word_sizes_deltax]
DECODE_WORD_SIZES[word_sizes_deltay]
DECODE_WORD_SIZES[word_sizes_jumpx]
DECODE_WORD_SIZES[word_sizes_jumpy]
DECODE_WORD_SIZES[word_sizes_num_deltas]
INPUT_STREAM[delta_minimum_per_curve, BITS_MINIMUM_NUMBER_OF_DELTA]
INPUT_STREAM[sign monotonicity type for symbol 0, BITS_HUFFMAN_INDEX]
INPUT_STREAM[sign monotonicity type for symbol 10, BITS_HUFFMAN_INDEX]
INPUT_STREAM[sign monotonicity type for symbol 110, BITS_HUFFMAN_INDEX]
INPUT_STREAM[sign monotonicity type for symbol 111, BITS_HUFFMAN_INDEX]
return
```

DECODE_WORD_SIZES[*word_sizes*]

** *word_sizes* is a list of word sizes used in encoding particular types of data (e.g., $word\_sizes_{num\_deltas}$ indicates the sizes of words in bits used in encoding the number of deltas in a curve)

INPUT_STREAM[number of word sizes in *word_sizes*, $BITS_{NUMBER\_OF\_WORD\_SIZES}$]
for each *word_size* **in** *word_sizes*
    INPUT_STREAM[*word_size*, $BITS_{WORD\_SIZE}$]
return

INPUT_STREAM[*value*, *n*]

** Note: If *n* is missing on invocation of INPUT_STREAM[ ], the number of bits required to read *value* will be obvious from the definition of *value* (e.g., a Huffman symbol)

read *value* in *n* bits from the encoded data stream
return

DECODE_JUMP[curve *a*, curve *b*]

DECODE_JUMP_REFERENCE_END[curve *a*]
DECODE_USING_WORD_SIZES[$jump_x$ from curve *a* to curve *b*, $word\_sizes_{jumpx}$]
DECODE_SIGN_FOR_VALUE[$jump_x$ from curve *a* to curve *b*]
DECODE_USING_WORD_SIZES[$jump_y$ from curve *a* to curve *b*, $word\_sizes_{jumpy}$]
DECODE_SIGN_FOR_VALUE[$jump_y$ from curve *a* to curve *b*]
return

DECODE_JUMP_REFERENCE_END[*curve*]

INPUT_STREAM[*flag*, 1]
if (*flag* = 0)
    reference end of *curve* = FIRST_ENDPOINT
else
    reference end of *curve* = LAST_ENDPOINT
return

```
DECODE_USING_WORD_SIZES[magnitude, word_sizes]
    INPUT_STREAM[Huffman symbol for word_size]
    if (word_size ≠ 0)
        INPUT_STREAM[magnitude, word_size]
    return
```

```
DECODE_SIGN_FOR_VALUE[value]
    ** Note: If sign was ZERO, nothing was appended to the stream
    if (value ≠ 0)
        INPUT_STREAM[flag, 1]
        if (flag = 1)
            negate value
    return
```

```
DECODE_CURVE_DELTAS[curve]
DECODE_USING_WORD_SIZES[delta_count, word_sizes_num_deltas]
number of deltas in curve = delta_count + delta_minimum_per_curve

INPUT_STREAM[Huffman symbol for sign monotonicity of curve]
if (sign monotonicity of curve = MONOTONIC_BOTH)
    DECODE_SIGN[sign_x of curve]
    DECODE_SIGN[sign_y of curve]
    for each delta in curve
        DECODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        APPLY_SIGN_TO_VALUE[sign_x of curve, delta_x]
        DECODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        APPLY_SIGN_TO_VALUE[sign_y of curve, delta_y]
else if (sign monotonicity of curve = MONOTONIC_DX)
    DECODE_SIGN[sign_x of curve]
    for each delta in curve
        DECODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        APPLY_SIGN_TO_VALUE[sign_x of curve, delta_x]
        DECODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        DECODE_SIGN_FOR_VALUE[delta_y]
else if (sign monotonicity of curve = MONOTONIC_DY)
    DECODE_SIGN[sign_y of curve]
    for each delta in curve
        DECODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        DECODE_SIGN_FOR_VALUE[delta_x]
        DECODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        APPLY_SIGN_TO_VALUE[sign_y of curve, delta_y]
else if (sign monotonicity of curve = NON_MONOTONIC)
    for each delta in curve
        DECODE_USING_WORD_SIZES[delta_x, word_sizes_deltax]
        DECODE_SIGN_FOR_VALUE[delta_x]
        DECODE_USING_WORD_SIZES[delta_y, word_sizes_deltay]
        DECODE_SIGN_FOR_VALUE[delta_y]
return
```

DECODE_SIGN[*sign*]

```
** Note: If sign was ZERO, nothing was appended to the stream
INPUT_STREAM[flag, 1]
if (flag = 1)
    sign = NEGATIVE
else
    sign = POSITIVE
return
```

APPLY_SIGN_TO_VALUE[*sign*, *value*]

```
if (sign = NEGATIVE)
        negate value
return
```

APPLY_JUMP_OFFSET[curve *a*, curve *b*]

```
if (reference end of curve a = FIRST_ENDPOINT)
    ref_pt = first point in curve a
else if (reference end of curve a = LAST_ENDPOINT)
    ref_pt = last point in curve a
first_pt = first point in curve b
first_pt_x = ref_pt_x + jump_x from curve a to curve b
first_pt_y = ref_pt_y + jump_y from curve a to curve b
return
```

APPLY_CURVE_DELTA_OFFSETS[*curve*]

```
for each point b from second point to last point in curve
    point a = the previous point in curve before point b
    b_x = a_x + delta_x from point a to point b
    b_y = a_y + delta_y from point a to point b
return
```

12.3 EXAMPLE

In order to illustrate the decoding process, the same example from the encoding section will be used. Figures 6 and 7 of section 11 show the fingerprint header and the ridge information bit streams. Figure 73 illustrates parsing and interpreting just the fingerprint header information in the bit stream. Figure 74 illustrates parsing and decoding the ridge information data for the first curve. Note that absolute coordinate positions are given instead of jump values for this ridge. Figure 75 shows parsing and decoding of the second ridge. Although it is not shown, the process would proceed similarly for the remaining four ridges in this example. (Note: The binary bit stream values in these figures are shown grouped to show the parsed structure.)

Bit Stream:

0000000111000010 0000001001011000
000000110 10 0011 0100 10 0010 0011 10 0001 0010 10 0001 0010 10 0001 0010 11 00 01 10 11

Parsing:

Bit Stream	# Parse Bits	Value	Interpretation
0000000111000010	16	450	Image is 450 pixels wide
0000001001011000	16	600	Image is 600 pixels high
00000000110	11	6	6 ridges
10	2	2	delta offset: 2 word sizes
0011	4	3	delta offset short *x* word size
0100	4	4	delta offset long *x* word size
10	2	2	delta offset: 2 word sizes
0010	4	2	delta offset short *y* word size
0011	4	3	delta offset long *y* word size
10	2	2	jump value: 2 word sizes
0001	4	1	jump value short *x* word size
0010	4	2	jump value long *x* word size
10	2	2	jump value: 2 word sizes
0001	4	1	jump value short *y* word size
0010	4	2	jump value long *y* word size
10	1	2	num. deltas per curve: 2 word sizes
0001	4	1	num. deltas per curve short word size
0010	4	2	num. deltas per curve long word size
11	2	3	minimum number of deltas per curve
00	2	0	monotonic both
01	2	1	monotonic delta *x*
10	2	2	monotonic delta *y*
11	2	3	non-monotonic

Figure 73. Fingerprint Header Parsing

Bit Stream:

000001010 0000001100 1 10 1 10 0 0 101 1 101 0 0 101 0 10 0 1 1000 0 10 1 1 1000 1 111 1

Parsing:

Bit Stream	# Parse Bits	Value	Interpretation
000001010	9	10	x coordinate of first point
0000001100	10	14	y coordinate of first point
1	1	1	num. deltas per curve long word
10	2	2	4 deltas (2 + min. num. deltas per curve)
1	1	1	reference end: LAST_ENDPOINT
10	2	2	monotonic delta x
0	1	0	positive monotonic sign for delta x
0	1	0	delta offset short x word
101	3	5	delta offset x value
1	1	1	delta offset long y word
101	3	5	delta offset y value
0	1	0	positive delta offset y sign
0	1	0	delta offset short x word
101	3	5	delta offset x value
0	1	0	delta offset short y word
10	2	2	delta offset y value
0	1	0	positive delta offset y sign
1	1	1	delta offset long x word
1000	4	8	delta offset x value
0	1	0	delta offset short y word
10	2	2	delta offset y value
1	1	1	negative delta offset y sign
1	1	1	delta offset long x word
1000	4	8	delta offset x value
1	1	1	delta offset long y word
111	3	7	delta offset y value
1	1	1	negative delta offset y sign

Reconstruction:

Decoded Relative Coordinates: (10,14), (+5,+5), (+5,+2), (+8,−2), (+8,−7)

Decoded Absolute Coordinates: (10,14), (15,19), (20,21), (28,19), (36,12)

Figure 74. Ridge Information Decoding: First Curve

Bit Stream:

0 1 1 1 10 1 0 0 1 0 1 0 0 110 1 110 0 111 0 11

Parsing:

Bit Stream	# Parse Bits	Value	Interpretation
0	1	0	jump value short x word
1	1	1	jump value x value
1	1	1	negative sign
1	1	1	jump value long y word
10	2	2	jump value y value
1	1	1	negative sign
0	1	0	num. deltas per curve short word
0	1	0	2 deltas (0 + min. num. deltas per curve)
1	1	1	reference end: LAST_ENDPOINT
0	1	0	monotonic both
1	1	1	negative sign for all delta x
0	1	0	positive sign for all delta y
0	1	0	delta offset short x word
110	3	6	delta offset x value
1	1	1	delta offset long y word
110	3	6	delta offset y value
0	1	0	delta offset short x word
111	3	7	delta offset x value
0	1	0	delta offset short y word
11	2	3	delta offset y value

Reconstruction:

Decoded Relative Coordinates: (−1,−2), (−6,+6), (−7,+3)

Decoded Absolute Coordinates: (35,10), (29,16), (22,19)

Figure 75. Ridge Information Decoding: Second Curve

SECTION 13

RIDGE RECONSTRUCTION

After decoding, ridge reconstruction regenerates the single pixel width representation of the fingerprint ridges using a B-spline algorithm. The input to the B-spline algorithm is a set of ordered control points previously determined by the chord splitting stage of processing. For each set of ordered points, the output of the B-spline algorithm is a reconstructed fingerprint ridge. Multiple sets of ordered points describe all of the ridges within one fingerprint image. Figure 76 illustrates the curve that would be generated by the B-spline process given an input set of five points. B-splines differ from other spline curves in that the resulting curve does not necessarily pass directly through the set of input points, which results in a more uniform and smooth curve [9].

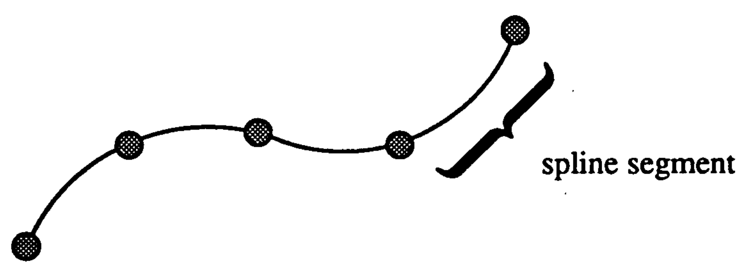


Figure 76. B-spline Curve Representation

13.1 ALGORITHM DESCRIPTION

Figure 77 describes the processing steps of the B-spline algorithm. The algorithm that computes the B-splines uses a set of four consecutive control points to calculate each spline segment [10]. Due to the nature of the process, it is also necessary to duplicate the original coordinate segment endpoints four times to ensure that the spline curve is drawn to the endpoints.

A B-spline is determined using two input arrays, x and y. The B-spline algorithm creates curve segments between successive points P_i and P_{i+1} for each curve to be constructed. It is not necessary to calculate a B-spline for ridges containing one or two points. A ridge containing one point will be represented as a single pixel, and a ridge containing two points will be represented by a line. The curve segment between points P_i and P_{i+1} is constructed by calculating $x(t)$ and $y(t)$ as t increases from zero to one:

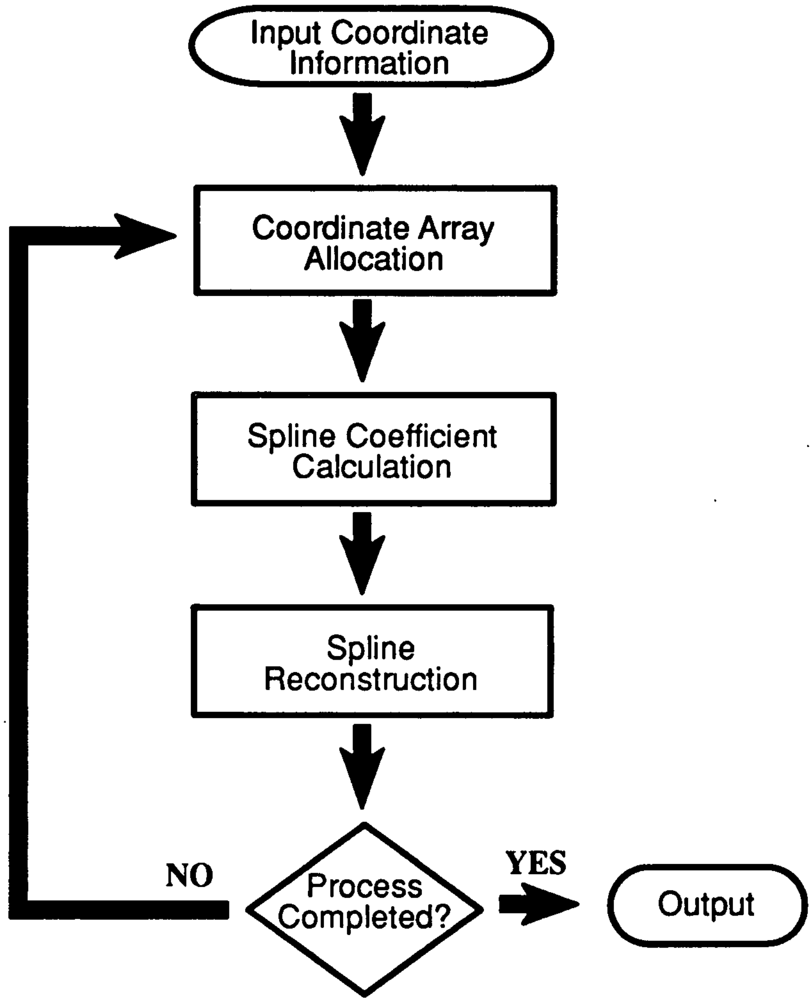


Figure 77. Flow Chart of Operations

$$x(t) = \mathbf{A}[x, i] + t(\mathbf{B}[x, i] + t(\mathbf{C}[x, i] + t\mathbf{D}[x, i]))$$
$$y(t) = \mathbf{A}[y, i] + t(\mathbf{B}[y, i] + t(\mathbf{C}[y, i] + t\mathbf{D}[y, i]))$$

where **A**, **B**, **C**, and **D** are functions defined as:

$$\mathbf{A}[x, i] = (x(i-1) + 4x(i) + x(i+1)) \div 6$$
$$\mathbf{B}[x, i] = (-x(i-1) + x(i+1)) \div 2$$
$$\mathbf{C}[x, i] = (x(i-1) - 2x(i) + x(i+1)) \div 2$$
$$\mathbf{D}[x, i] = (-x(i-1) + 3x(i) - 3x(i+1) + x(i+2)) \div 6$$

The functions are similarly defined for A[y, i], B[y, i], C[y, i], D[y, i].

The coefficients are computed for each point in the x and y input arrays. After the coefficients are computed for an individual point, a loop increasing from zero to N is executed. Within this loop, $x(t)$ and $y(t)$ are calculated. Starting with the second input point, a line segment is drawn for each successive $x(t)$ and $y(t)$. The process ends when the input arrays are exhausted.

13.2 SUMMARY

This section provides parameters, input variables, output variables, and pseudocode for the B-spline algorithm.

Parameters

N = 30	Number of iterations

Input Variables

curve_list	List containing the coordinate control points for a group of curves with the endpoints added four times to each curve

Output Variables

spline_x, spline_y	Arrays that hold spline coordinates

** Algorithm used to construct a smooth curve using a given set of ordered coordinates

```
B-SPLINE[curve_list]
while curve in curve_list
{
    ** x and y are arrays that hold coordinate information
    x = ordered set of x coordinates
    y = ordered set of y coordinates

    ** Spline coefficient calculations
    for i from 1 to number_of_points_in_curve
            for j from 0 to N
                    t = j ÷ N
                    x_coordinate = A[y, i] + t(B[y, i] + t(C[y, i] + tD[y, i]))
                    y_coordinate = A[y, i] + t(B[y, i] + t(C[y, i] + tD[y, i]))
                    spline_x = x_coordinate
                    spline_y = y_coordinate
  ** At each iteration, the calculated x and y coordinate positions may be used to
     reconstruct an image array, using the spline_y array for row values and the
     spline_x array for column values.
}
return
```

** The following functions are also used to calculate A[y, i], B[y, i], C[y, i], D[y, i]

```
A[x, i]
return (−x(i−1) + 3x(i) − 3x(i+1) + x(i+2)) ÷ 6
```

```
B[x, i]
return (x(i−1) − 2x(i) + x(i+1)) ÷ 2
```

```
C[x, i]
return (−x(i−1) + x(i+1)) ÷ 2
```

```
D[x, i]
return (x(i−1) + 4x(i) + x(i+1)) ÷ 6
```

LIST OF REFERENCES

1. *National Crime Information Center (NCIC) 2000 Request for Proposal (RFP). RFP 5060. Attachment 1. National Crime Information Center (NCIC) 2000 System Requirements*, Federal Bureau of Investigation, Washington, D.C.

2. *An Analysis of Standards in Fingerprint Identification*, June 1972, FBI Law Enforcement Bulletin, Federal Bureau of Investigation, U.S. Department of Justice, Washington, D.C.

3. Lepley, M. A., April 1994, *NCIC 2000 Image Compression Algorithms, Volume II: Mugshot Compression*, MTR–94B0000021V2, The MITRE Corporation, Bedford, MA.

4. Horn, B. K. P., 1986, *Robot Vision*, Cambridge, MA: MIT Press, pp. 49–53.

5. Barrow, H. G., J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf, August 1977, "Parametric Correspondence and Chamfer Matching: Two New Techniques for Image Matching," *Proceedings of International Joint Conference on Artificial Intelligence*, Vol. 2, pp. 659–663, Cambridge, MA.

6. Pavlidis, Theo, 1982, *Algorithms for Graphics and Image Processing*, Rockville, MD: Computer Science Press, pp. 283–287.

7. Hamming, R. W., 1986, *Coding and Information Theory*, Second Edition. New Jersey: Prentice-Hall.

8. Gonzalez, R. C. and P. Wintz, 1977, *Digital Image Processing*, Reading, MA: Addison-Wesley Publishing Co.

9. Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, 1990, *Computer Graphics: Principles and Practice*, Second Edition, Reading, MA: Addison-Wesley Publishing Co., p. 491.

10. Ammeraal, Leendert, 1986, *Programming Principles in Computer Graphics*, New York: John Wiley and Sons, pp. 28–33.

11. *Home Office Algorithm Package*, Volumes 1 and 2, Issue 1, April 1992, Department of the Government of the United Kingdom, London, England.

APPENDIX A

MODIFIED BHO BINARIZATION

The algorithm used for thresholding the fingerprint image is based upon the Home Office Automatic Fingerprint Recognition System (HOAFRS) Encoder. This algorithm generates a smoothed ridge direction map, thresholds each 24×24 block based upon the primary indicated direction, and then extracts minutiae points. We used the portion of the Encoder that generates the thresholded image, hereafter referred to as British Home Office (BHO) binarization, with a few minor modifications to accommodate variable image sizes. The BHO algorithm is described in detail in the Home Office Algorithm Package Volume 1: Description of Encoders and Matchers [11]. This section describes how the source code provided by the Home Office was modified for use with the Flat Live-Scan Searchprint Compression algorithm.

A.1 SOURCE CODE ALTERATIONS

The modifications described in this section fall into two categories: those that are implementation dependent and those that are required modifications to the HOAFR algorithm. Implementation dependent details, such as the FORTRAN-to-C conversion and methods used for faster execution, are mentioned here only as a guide. Required changes are necessary to process variable image sizes, to obtain acceptable encoded fingerprints, and to produce the required output files. The required changes must be implemented and will be prefixed by a "*" in the following subsections.

A.1.1 FORTRAN-to-C Conversion

The five source files of the Encoder that are used in BHO binarization are "encoder.f", "initrg.f", "insspr.f", "main12.f", and "main3.f". Since the original source code was in FORTRAN, "f2c", a public domain FORTRAN-to-C conversion program, was used to create a C version of the code. This conversion program, written by David Gay, Stu Feldman, Mark Maimone, and Norm Schryer, is available via electronic transfer from research.att.com. This C version was then modified to make it possible to compile without the include file ("f2c.h") and the FORTRAN libraries required by "f2c". The changes needed were:

- References to `logical` were changed to `int`.
- References to `integer` were changed to `int`.
- References to `real` were changed to `float`.
- Unnecessary static declarations were removed.
- Global data structures were moved to an include file.
- All the I/O was changed since the output of f2c for I/O code is indecipherable.

– Replaced the functions `min`, `max`, and `dmax` by macros.
– Added the `rmod` function source to the code.
– Removed #include "f2c.h" from each file.

A.1.2 Variable Image Size Accommodation

The stand-alone C version was modified to generate only the thresholded image and was then modified to handle rectangular images of any size. This change required careful attention to detail to determine the meaning of many of the hardwired constants in the code. (Although the capability is not being used at this time, changes were also made to allow smaller block sizes, if desired.) In summary, the changes at this stage were:

*– Output a thresholded image with ridges being black and valleys being white.
*– Variables were created to contain the following information about the image: height, width, block size, the number of blocks contained in the horizontal and vertical direction, and the positions of blocks. The variables are initialized as the image is read.
*– Constants within the code were replaced by the appropriate variables or combinations of variables. For example, m×11520 + 1920 became m×(blocksize×iwidth) + (4×iwidth).
*– Arrays whose sizes vary according to the input image size were allocated dynamically and freed when they were no longer needed.
*– The ridge direction consistency checking function `consis` was modified to allow the spiraling portion of the algorithm to reach all parts of a rectangular image.

Please note that changes made up to this point have no effect on the behavior of the algorithm on a 512×512 or a 480×480 image.

A.1.3 Change to BHO Algorithmic Behavior

As a result of testing, we determined that some undesirable effects were produced by the original algorithm. Therefore, the following changes were made to the BHO binarization algorithm.

*– Removal of `cnnect` call and function.

*– An absolute upper threshold, *zz_top*, is generated on a per image basis, for use in non-blanked blocks. This allows pixels with a gray level above this threshold to be automatically marked as BACKGROUND pixels. This change was required in the functions `ifilt7` and `binblk` which do the directional and non-directional thresholding.

Before calculating the threshold *zz_top*, a determination is made whether the image gray values have saturated, i.e., whether there are an inordinate number of white pixels because of the particular brightness or contrast level settings in effect during image capture. Assuming that 255 corresponds to the maximum gray value, i.e., white, the image is deemed to be saturated if the number of pixels with gray value 255 is greater than $Z_{SATURATION\_RATIO}$ times the number of pixels with gray value 254.

If the image is saturated, thresholding is not effective and *zz_top* is set to 255. Otherwise, *zz_top* is set to a value $Z_{THRESHOLD\_FRACTION}$ of the distance between the mean pixel value $Z\mu_I$ and the maximum pixel $Zmax_I$ value in the image:

$$zz\_top = (1 - Z_{THRESHOLD\_FRACTION}) * Z\mu_I + Z_{THRESHOLD\_FRACTION} * Zmax_I.$$

A.1.4 Integration with Fingerprint Compression

A few modifications were made to integrate the code into the rest of the fingerprint compression process.

- The main routine was converted to a subroutine which was passed an image data structure and the desired block size and returned a thresholded image data structure.
- The function to read an image file was changed to read data from the image data structure.
- The ability to write the thresholded image to a Lucid image file was removed.
- *– Code was added to write out the ridge direction map and to store it in a data structure passed back to the calling routine. The file containing this information is called the block file and is transmitted along with the encoded fingerprint for use during minutiae extraction. Section A.2 below describes the contents of the ridge direction data structure and specifies how that information is written to a file.

A.1.5 Code Speed Up

Finally, a large set of changes was made to improve the processing speed of this algorithm. These changes again involved careful attention to detail to avoid making errors, and frequent checks were made against a validated older version.

- Many explicit casts to float in equations were removed.
- Some parameters passed by reference due to the FORTRAN to C conversion were changed to be passed by value to avoid being accessed via pointers.
- Zero-based indexing was used for loops and for accessing arrays. This frequently removed many references to "*i*–1".
- Variables were added to store intermediate values that are used frequently.
- Where beneficial, pointers were used to access arrays.

A.2 RIDGE DIRECTION MAP

The ridge direction map contains the smoothed edge directions for each 24×24 block in the image. Due to processing constraints, this map must contain an even number of blocks in both the horizontal and vertical directions. When the image size is not a multiple of 48 (2×24), the area that is covered by this map is the largest multiple of 48 that fits inside the image, centered in the entire image area (see figure A-1). Information about this map is stored in a ridge direction data structure and written out to a block file.

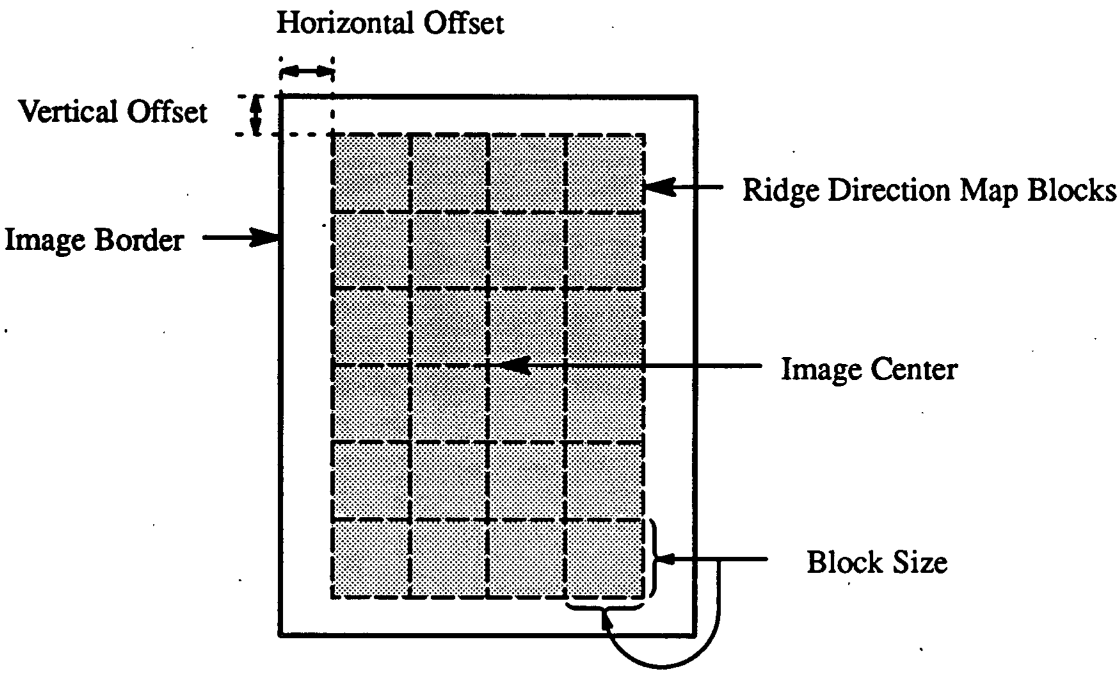


Figure A-1. Blocks Used in Ridge Direction Map

A.2.1 Ridge Direction Data Structure

The ridge direction data structure, *z_blockmap*, contains the following information:

- The horizontal offset (in pixels) of the upper-leftmost block
- The vertical offset (in pixels) of the upper-leftmost block
- Number of blocks horizontally (always a multiple of 2)
- Number of blocks vertically (always a multiple of 2)
- Block size used (blocks are square)
- A 2-dimensional array of the block ridge directions when a valid ridge direction existed, or the type of block when there is no valid ridge direction. There are 16 valid ridge directions, and the types of blocks that can occur when there is no valid ridge direction are: blank block, core/delta block, and "bad" (other) block.

A.2.2 Writing the Block File

At the end of the BHO binarization, the information in the ridge direction data structure is written to a file called the block file. The information in the encoded fingerprint file and the block file together can be used to recreate the fingerprint image and extract valid minutiae points.

The following pseudocode describes an efficient method for encoding this data. In order to make the most efficient use of space, the bit-packing function OUTPUT_STREAM described in section 11.4.3 is used to write bits to the block file.

```
WRITE_BLOCK_FILE[ z_blockmap, block_file ]
    ** Write information about the ridge direction map to block_file
    open block_file for writing
    OUTPUT_STREAM[horizontal offset of z_blockmap, 16]
    OUTPUT_STREAM[vertical offset of z_blockmap, 16]
    OUTPUT_STREAM[number of blocks horizontally in z_blockmap, 16]
    OUTPUT_STREAM[number of blocks vertically in z_blockmap, 16]
    OUTPUT_STREAM[block size used in z_blockmap, 5]
    for each block (bi, bj) in z_blockmap
        if ( block (bi, bj) is blanked out )
            OUTPUT_STREAM[0, 5]
        else if ( block (bi, bj) is bad )
            OUTPUT_STREAM[1, 5]
        else if ( block (bi, bj) is a core/delta block )
            OUTPUT_STREAM[2, 5]
        else if ( block (bi, bj) has a valid direction )
            OUTPUT_STREAM[direction of block (bi, bj) + 3, 5]
    close block_file
    return
```

A.3 SUMMARY

Parameters

ZN = 24 — Height and width (in pixels) of the blocking factor used for the ridge direction map

$Z_{SATURATION\_RATIO}$ = 2.0 — Maximum ratio between pixels at 254 and pixels at 255 for an unsaturated image

$Z_{THRESHOLD\_FRACTION}$ = 0.8 — Fraction of the distance between the mean pixel value and the maximum pixel value in an image used to determine *zz_top*

Input

I — Gray-scale fingerprint image

Output

T	Thresholded fingerprint image
z_blockmap	Ridge direction data structure
block_file	File containing information about ridge directions as well as blocks that should not be used when extracting minutiae

BHO_BINARIZATION[***I***]

** The image ***I*** is thresholded using the modified BHO algorithm to produce image ***T***

1 Run the modified BHO binarization on ***I*** with block size ZN, writing out ***block_file***.

2 **return** (***T***, *z_blockmap*)

APPENDIX B

CURVED RIDGE ENDING REMOVAL

Curved ridge ending removal is a part of ridge cleaning (see section 7). The purpose of this algorithm is to remove curved endings that may lead to a less accurate ridge ending direction estimation. This process is used by ridge cleaning after small offshoot curve removal and before small ridge break connection (see figure 41 in section 7). Only ridge endings that are not connected to other ridges (i.e., not bifurcations) and are not near a bad block as defined by the thresholding process (see Appendix A) are processed by curved ridge ending removal. If a ridge ending is determined to be curved, points are removed until the curved part is removed or an upper limit is reached on the number of points allowed to be removed.

B.1 ALGORITHM DESCRIPTION

It is assumed that *endpoint_map* and the thinned image *T* generated in ridge cleaning and the chamfered image *C* used by ridge cleaning is globally available to this algorithm. The parameter Z_{END_SIZE} specifies that maximum number of points that may be removed from any ridge ending as part of the curved ridge ending removal process. This parameter is also used as part of the curvature and taper calculation for each ridge ending.

Both ends of each ridge that is represented by a curve in the *curve_list* having more than $3 \times Z_{END\_SIZE}$ points are considered. If an endpoint is unconnected and this endpoint is not near a bad block as defined by *z_blockmap*, then that end is considered further. Care is taken to retain ridge endings on the borders of bad blocks in order to prevent the generation of false minutiae during minutiae extraction.

To check the bad block proximity of a ridge endpoint, the endpoint is checked against all the bad blocks defined in *z_blockmap*. If the endpoint is within one pixel of any bad block, it is declared to be near a bad block. Several examples of ridge endpoints near a bad block are shown in figure B-1. In this figure, endpoints of ridges A, B, and C are near the bad block because their endpoints lie within one pixel of the bad block. The endpoint of ridge D is not near the bad block.

Each ridge ending meeting the size and bad block proximity conditions is then checked for amount of curvature and taper. The curvature criterion is measured by selecting three points along the ridge, somewhat equally spaced according to specific criteria, where the first point is the current endpoint of the ridge. This process is illustrated in figure B-2. In this figure, point A is the endpoint of the ridge. Point B is selected as the point that is $Z_{END\_SIZE}+2$ points down from point A. Point C is selected as the first point whose

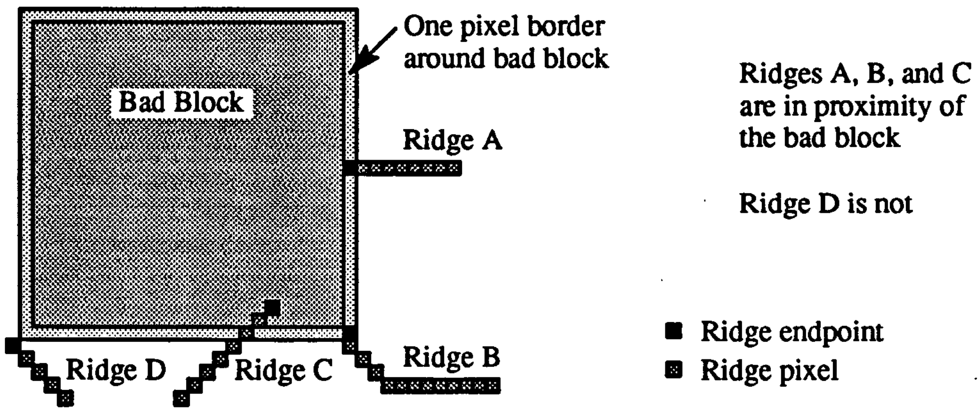
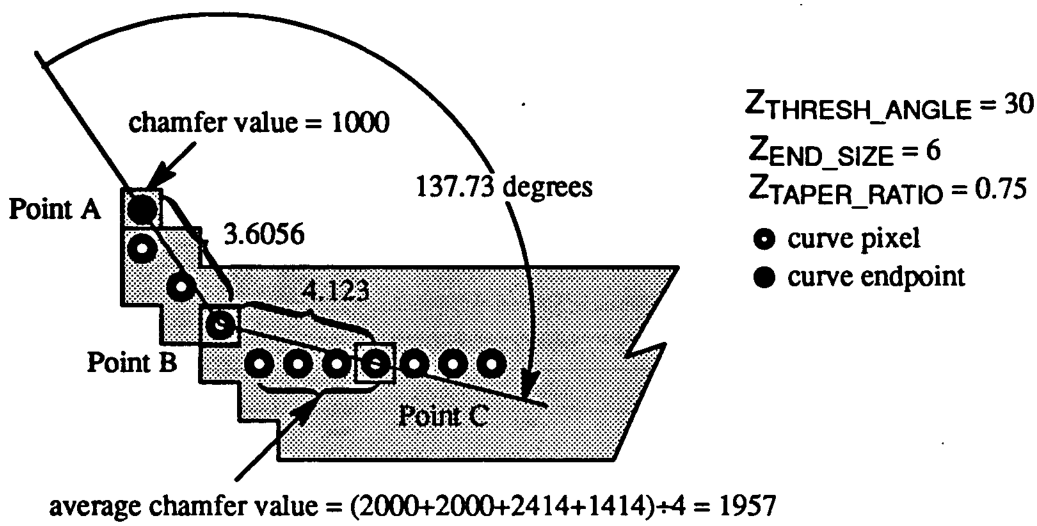


Figure B-1. Proximity of Ridge Endpoints to Bad Block

Euclidean distance from point B is greater than or equal to the Euclidean distance between point A and point B. Point C is defined to be at least $Z_{END\_SIZE} \div 2 + 1$ points down the curve from point B. Once the three points have been selected, the absolute value of the cosine of the angle between the two segments defined by point B to point A and point B to point C is calculated. If this cosine is less than or equal to the cosine of Z_{THRESH_ANGLE}, then the ridge ending is considered to be curved.

A second criterion is checked to estimate the taper of the ridge ending. Occasionally, a ridge ending will not be curved enough to meet the curvature criterion, but will still have a small flip to it caused by an angled ridge edge at the end. The taper criterion is designed to catch these instances. By comparing the ridge width of the endpoint to the average ridge width of a reference section further down the curve, the taper can be estimated. This reference section is defined to be the set of points between point B and point C, not including point B, but including point C. Because chamfer values are directly proportional to ridge widths, they are used in the ridge width comparisons. If the chamfer value of point A is less than or equal to Z_{TAPER_RATIO} times the average chamfer value of the reference section, the ridge ending is considered to have enough curvature to take action.

If either the curvature criterion or the taper criterion is met, point A is moved down the curve by one point, thus marking that previous point for removal. If fewer than Z_{END_SIZE} points have been marked for removal and the curvature or the taper criterion has been met, the process is repeated with the new point A. Otherwise, the process is finished for this ridge ending by removing all points in *curve* from the original endpoint up to point A, and updating the thinned image ***T*** and *endpoint_map*.



if

$|\cos(\theta)| \le \cos(Z_{THRESH\_ANGLE})$

or

chamfer value at point A $\le Z_{TAPER\_RATIO} \times$ average_chamfer value of reference section

then

remove point A from ridge

Figure B-2. Criteria for Removing Curved Ridge Ends

```
CURVED_RIDGE_ENDING_REMOVAL[ curve_list, z_blockmap ]

for each curve in curve_list
    if (number of points in curve > 3 × Z_END_SIZE)
    {
        if ((curve is unconnected at the first endpoint)
            and (the first endpoint is not near a "bad" block in z_blockmap))
            PROCESS_RIDGE_ENDING[ curve, first endpoint of curve]

        if ((curve is unconnected at the last endpoint)
            and (the last endpoint is not near a "bad" block in z_blockmap))
            PROCESS_RIDGE_ENDING[ curve, last endpoint of curve]
    }
end
```

```
PROCESS_RIDGE_ENDING[ z_curve, z_endpoint ]
   ** Assume chamfer image C, thinned image T, and endpoint_map are globally available
   set point z_pointA to z_endpoint
   z_number_points_removed = 0
   z_not_done = TRUE
   ** Process the ridge ending until Z_END_SIZE points have been removed from the
      curved ridge ending, or the curvature and taper criteria indicate that no further action
      should be taken.
   while ( z_number_points_removed ≤ Z_END_SIZE and z_not_done = TRUE )
   {
       set point z_pointB to be Z_END_SIZE ÷ 2 points down z_curve from z_pointA
       set point z_pointC to be Z_END_SIZE ÷ 2 points down z_curve from z_pointB
       z_distanceAB = Euclidean distance between z_pointA and z_pointB
       z_distanceBC = 0
       ** Adjust position of z_pointC so that z_distanceAB is similar to z_distanceBC
       while ( z_distanceBC < z_distanceAB )
           move point z_pointC down the curve by one point
           z_distanceBC = Euclidean distance between z_pointA and z_pointB
       ** Calculate the average chamfer value for section between z_pointB and z_pointC
       z_sum = 0
       for each (z_i, z_j) between z_pointB and z_pointC, z_pointC inclusive,along z_curve
           z_sum = z_sum + C(z_i, z_j)
       z_average_chamfer_value = z_sum / (number of points in the summation)
       if ( DOT_PRODUCT(z_pointC, z_pointB, z_pointA) ≤ cos(Z_THRESH_ANGLE) )
           or ( C(x coordinate of z_pointA, y coordinate of z_pointA)
                  ≤ Z_TAPER_RATIO × z_average_chamfer_value )
       {
           move point z_pointA down the z_curve by one point
           z_number_points_removed = z_number_points_removed + 1
       }
       else
           z_not_done = FALSE
   }
   if number_points_removed > 0
       delete z_endpoint of z_curve from z_endpoint_map
       remove the points from T starting at z_endpoint and up to point z_pointA of z_curve
       remove the points starting at z_endpoint and up to point z_pointA from z_curve
       add new endpoint (z_pointA) of z_curve to z_endpoint_map
   end
```

B.1.1 Summary

Parameters

$Z_{END\_SIZE} = 6$	Maximum number of points that can be removed from a curved ridge end
$Z_{THRESH\_ANGLE} = 30$ degrees	Curvature limit for curved ridge end
$Z_{TAPER\_RATIO} = 0.75$	Tapering ridge width ratio limit for ridge end

Input

curve_list	The list of curves for the live-scan fingerprint
C	Chamfered image calculated as part of ridge thinning (section 5)
T	Thinned image regenerated from *curve_list* and update as the *curve_list* is modified
z_blockmap	Ridge direction data structure
endpoint_map	See definition in section 7.1.1

Output

modified *curve_list*
modified *endpoint_map*
modified *T*

APPENDIX C

BAD BLOCK BLANKING

During the thresholding stage certain blocks are found to contain smudges or other inconsistent fingerprint information. These blocks are labeled as bad blocks but are thresholded, nevertheless, so that artificial ridge endings will not appear at the edge of bad blocks. During the bad block blanking stage, the ridge sections that cross bad blocks are removed so that the encoded file is as small as possible. This process takes place at the end of ridge cleaning, because ridge fragments internal to the bad blocks may contain information useful for cleaning.

C.1 ALGORITHM DESCRIPTION

The bad block blanking process works as a two stage process. The first stage follows each ridge in the fingerprint structure and removes segments that cross bad blocks. The segments that need to be removed may be at the beginning, end, or middle of a curve; each of these cases requires slightly different handling, including potentially modifying, removing, and adding curves. The second stage is needed for cleaning, since during the process of removing segments that cross bad blocks, a curve that enters a bifurcation may be removed. (This removal changes the bifurcation where three curves intersect into a location where only two curves intersect.) The second stage of bad block blanking, therefore, identifies this condition and joins the two curves in question.

C.1.1 Removing Curve Segments

In the first stage, each curve in *curve_list* is processed in turn. During processing, the *endpoint_map* is modified when an endpoint from the incoming curve list is removed. The modified *endpoint_map* will be used in the second stage to identify the locations where exactly two curves intersect.

Each curve is traversed from beginning to end while searching for the first two contiguous curve points that fall outside of a bad block. If there is no such location on the curve, then the entire curve is deleted from *curve_list* and the curve endpoints are removed from the *endpoint_map*. (See curve *d* in figures C-1 and C-2 for an example of such a curve.) If, on the other hand, the curve does contain two contiguous points outside a bad block, then the location of these first two good points determines whether the beginning of the curve must be removed.

If the first two good points are not the first two points in the curve, then the first endpoint (which is in or next to a bad block) is removed from the *endpoint_map* and the curve points

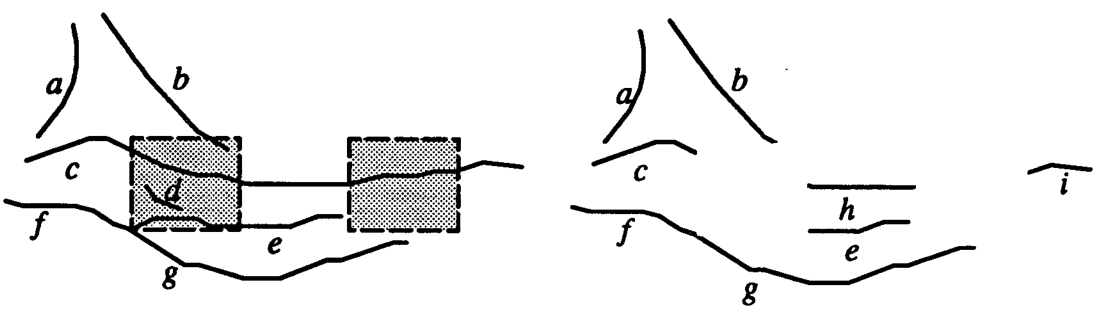


Figure C-1. Curve List Before and After the First Stage of Bad Block Blanking.

prior to the first two good points are removed from the curve. (For example, see curve *e* in figure C-1 when traversed from left to right.) After any initial bad block points are removed, the modified curve is treated the same as any curve that begins in a good block.

When the first two points in a curve are in a good block, the curve must be searched for any later sections that might enter a bad block. First, the curve is searched for the first point in a bad block. If there is no such point (the most common occurrence), then there are no further changes to this curve and processing of the next curve begins. (Curves *a*, *f*, and *g* in figure C-1 fall in this category, as well as curve *e* after the removal of its initial bad block section.) If a bad block point is found, the last endpoint of this curve is removed from the *endpoint_map* and the current curve is modified to end just prior to the first bad block point. (This happens to curves *b* and *c* in the example.)

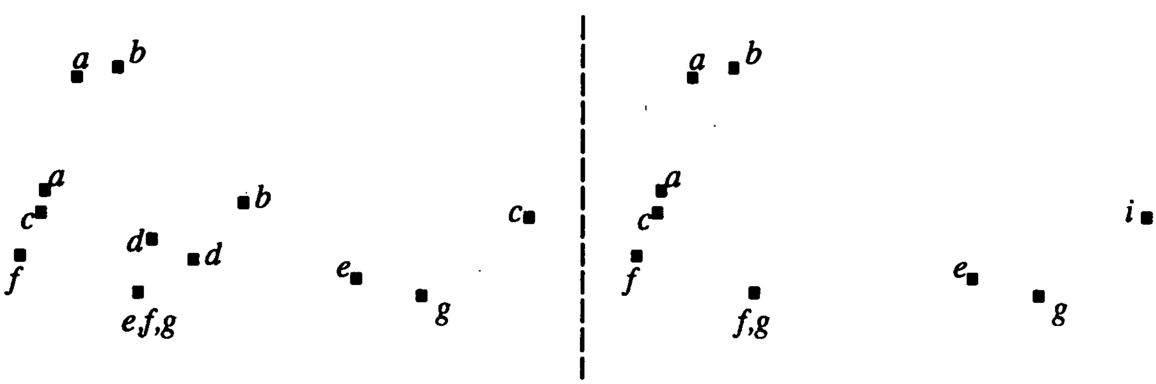


Figure C-2. Endpoint Map Before and After the First Stage of Bad Block Blanking.

The bad block segment is then followed until it ends (two contiguous points are found in a good block). If the bad block continues until the end of the original curve, no further processing is needed (see curve *b* in the example). Otherwise, a new curve must be created to contain the next good curve segment. (From curve *c*, first curve *h* and then curve *i* are created in the example.) This new curve is added to *z_new_curve_list*, which temporarily stores all new curves. If the end of this new curve is the same as the end of the original curve, then the second endpoint must be added back to the *endpoint_map* (for example, at the far end of curve *i*). Otherwise, if more of the original curve points remain, the process described in this paragraph is repeated until the entire curve has been traversed.

Once all the curves in *curve_list* have been checked for bad block sections, the new curves that were generated and stored in *z_new_curve_list* are moved into *curve_list*.

C.1.2 Joining Curves at Lost Bifurcations

The last stage of bad block blanking is to check each of the curve endpoints to ensure that no two-curve intersections exist. Recall that two-curve intersections occur when one curve of a bifurcation has been removed by the process described above. For each curve, an examination of the *endpoint_map* at each of the endpoints shows how many curves touch the endpoint. If the value is two at either endpoint, then the two curves that meet at this point are combined. For example, figure C-2 shows that curves *e*, *f*, and *g* share and endpoint prior to bad block blanking. After the first stage, the portion of curve *e* that intersects curves *f* and *g* has been removed, so the *endpoint_map* shows only *f* and *g* sharing that endpoint. Therefore, curves *f* and *g* create a two-curve intersection and must be joined to form the curve *j* shown in Figure C-3.

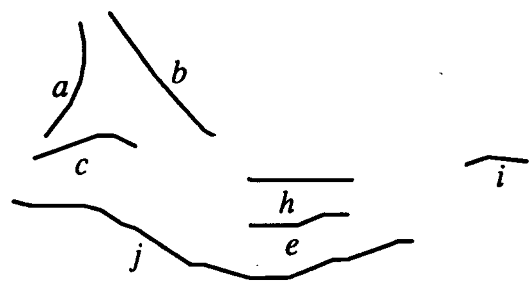


Figure C-3. Curve List After Last Stage of Bad Block Blanking

C.2 SUMMARY

Input

curve_list	The list of curves for the live-scan fingerprint
z_blockmap	Ridge direction data structure
endpoint_map	See definition in section 7.1.1

Output

modified *curve_list*

BAD_BLOCK_BLANKING[*curve_list, z_blockmap*]

```
** Note that endpoint_map is globally accessible
z_new_curve_list = EMPTY
for each z_curve in curve_list
{
    z = 0
    z_num_points = number of points in z_curve
    Z_POINT = points in z_curve (in order)

    do                              ** Search for two contiguous points in good blocks
    {
        z = z + 1
        while ((z ≤ z_num_points) and (Z_POINT(z+1) ∈ bad block))
            z = z + 1
    } while ((z < z_num_points) and (Z_POINT(z) ∈ bad block))

    if (z ≥ z_num_points)           ** Did not find two contiguous points in good blocks
        delete endpoints of z_curve from endpoint_map
        delete z_curve from curve_list

    else                            ** Found two contiguous points in good blocks
    {
        if (z > 1)
        {
            ** The first good segment is not at the beginning of z_curve, so
               remove the initial endpoint from the endpoint_map and the
               initial bad points from z_curve
            delete first endpoint of z_curve from endpoint_map
            delete Z_POINT(1) through Z_POINT(z–1) from z_curve
        }
```

```
        ** The initial segment must now be good, so find the end of it
        while ((z ≤ z_num_points) and (Z_POINT(z) ∈ good block))
            z = z + 1
        ** z is now the index of the point after the last good point found
        if (z ≤ z_num_points)
        {
            ** Only part of the curve is good. Keep the first good segment on the list
            delete the last endpoint of z_curve from endpoint_map
            delete Z_POINT(z) through Z_POINT(z_num_points) from z_curve

            ** Now search for any other good sections that might exist
            while (z ≤ z_num_points)
            {
                ** Scan to beginning of next good section
                while ((z ≤ z_num_points) and (Z_POINT(z) ∈ bad block))
                    z = z + 1
                z_first_point = z

                ** Find end of this good segment
                while ((z ≤ z_num_points) and (Z_POINT(z) ∈ good block))
                    z = z + 1

                ** Check the length of the good segment
                if ((z – z_first_point) > 1)
                {
                    ** A valid segment with at least two pixels,
                        so create a new curve for it
                    create z_new_curve with points Z_POINT(z_first_point)
                        through Z_POINT(z–1)
                    put z_new_curve on z_new_curve_list
                    if (z > z_num_points)
                        add last endpoint of z_new_curve to endpoint_map
                }
            }
        }
    }
}

** Add newly created curves to original curve list
for each z_curve in z_new_curve_list
    add z_curve to curve_list
```

```
** Curves that entered a bifurcation may have been removed. Wherever this happened,
   connect the remaining curves
for each z_curve in curve_list
    ** Consider each curve in order of appearance on the list of curves, so that any
       curve that is added to the end of the list will also be considered
    if (first endpoint of z_curve is shared by exactly one other curve)
        JOIN_CURVES[z_curve, other curve]                        ** Section 7.1.3.1
    else if (second endpoint of z_curve is shared by exactly one other curve)
        JOIN_CURVES[z_curve, other curve]                        ** Section 7.1.3.1

** Return the updated curve list
return curve_list
```

APPENDIX D

PARTITIONING FOR NEIGHBORHOOD AVERAGE RIDGE WIDTHS

The algorithms for Pore Filling (section 4) use the average ridge width in the neighborhood of each pore candidate. Rather than calculate the average ridge width in neighborhoods centered on each candidate, which would be computationally expensive, the average ridge width is found for fixed regions across the fingerprint image. The average ridge width in the neighborhood of a pore candidate is then approximated by the average ridge width in the fixed region in which it lies.

D.1 ALGORITHM DESCRIPTION

The $R \times C$ (rows $\times$ columns) fingerprint image is partitioned into R_P sections vertically and C_P sections horizontally (figure D-1). Each resulting $R/R_C \times C/C_P$ rectangle is used as a neighborhood for the average ridge width calculation. The parameter values used during development and testing of the Pore Filling algorithms are given in section 4.2. The values of R_P and C_P were chosen to evenly partition the image so that the resulting neighborhoods were roughly 60×60, thus covering large enough portions of the fingerprint to yield meaningful average ridge widths. To allow for a range of fingerprint image sizes, an algorithm was developed to choose the number of horizontal and vertical sections in an image that would most closely partition the image into 60×60 pixel regions.

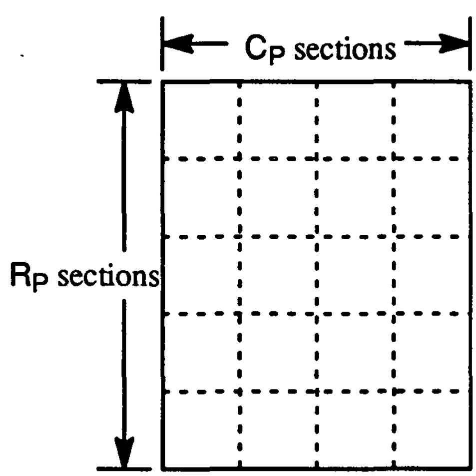


Figure D-1. Partitioning of Fingerprint Image for Neighborhood Average Ridge Width Calculation

The procedure to choose R_P, the number of sections vertically in the image, is the same as the procedure to find C_P, the number of sections horizontally. Therefore, in the following discussion, the image height or width is referred to as zz_{image_size}. First, zz_{image_size} is divided by the desired section size $Z_{DESIRED\_SECTION\_SIZE}$. If the result of this division is an integer, then that integer is the number of sections (R_P or C_P) in the given image dimension. Otherwise, $Z_{DESIRED\_SECTION\_SIZE}$ is alternately incremented and decremented (up to a maximum of $Z_{DELTA\_SECTION\_SIZE}$ from its original value) to find a section size that divides zz_{image_size} evenly. If such a section size is not found, then the section size that results in the smallest remainder from the division is chosen. Finally, zz_{image_size} is divided by the resulting section size to obtain the number of sections (R_P or C_P). Typical values of the section size and number of sections obtained for various image dimensions are given in table D-1.

Table D-1. Partitions for Typical Image Sizes

height (width)	number of sections R_P (C_P)	pixels per section
440	8	55
450	9	50
480	8	60
512	8	64
600	10	60
640	10	64
750	15	50
800	16	50

D.2 SUMMARY

The parameter values used during development and testing of the algorithms described in this section, as well as the input and output variables, are listed below.

Parameters

** $Z_{DELTA\_SECTION\_SIZE} = 20$ Maximum variation in the height or width of a fingerprint section

$Z_{DESIRED\_SECTION\_SIZE} = 60$ Desired height and width of a fingerprint section

Input

zz_{image_size} Image height or width

Output

Number of sections into which the input dimension should be partitioned

```
FIND_BEST_PARTITION[ zz_image_size ]
    ** zz_image_size is either the image width or height
    ** Returns the number of sections into which this dimension should be partitioned
    if ((zz_image_size mod Z_DESIRED_SECTION_SIZE) = 0)        ** mod is the modulus operator
        ** Can form an integer number of sections of Z_DESIRED_SECTION_SIZE
        return (zz_image_size / Z_DESIRED_SECTION_SIZE)
    else
        ** Try zz_section_size within Z_DESIRED_SECTION_SIZE ± Z_DELTA_SECTION_SIZE
        zz_best_remainder = Z_DESIRED_SECTION_SIZE
        for zz from 1 to Z_DELTA_SECTION_SIZE
        {
            ** Try zz_section_size > Z_DESIRED_SECTION_SIZE
            zz_section_size = Z_DESIRED_SECTION_SIZE + zz
            zz_remainder = (zz_image_size mod zz_section_size)
            if (zz_remainder = 0)
                return (zz_image_size / zz_section_size)
            else
                if (zz_remainder < zz_best_remainder)
                    ** This partitioning is the best so far, so save it
                    zz_best_remainder = zz_remainder
                    zz_best_section_size = zz_section_size

            ** Try zz_section_size < Z_DESIRED_SECTION_SIZE
            zz_section_size = Z_DESIRED_SECTION_SIZE − zz
            zz_remainder = (zz_image_size mod zz_section_size)
            if (zz_remainder = 0)
                return (zz_image_size / zz_section_size)
            else
                if (zz_remainder < zz_best_remainder)
                    ** This partitioning is the best so far, so save it
                    zz_best_remainder = zz_remainder
                    zz_best_section_size = zz_section_size
        }

    ** No zz_section_size was found to yield an integral partition, so return the best one found
    return (zz_image_size / zz_best_section_size)
```

APPENDIX E

PSEUDOCODE FUNCTION CALL TREE

This appendix contains the pseudocode function call tree for the Flat Live-Scan Searchprint Compression and Decompression algorithms. The main functions, SEARCHPRINT_COMPRESSION and SEARCHPRINT_DECOMPRESSION, are given as flowcharts instead of as pseudocode routines. The functions are listed in the order that they appear in the document.

SEARCHPRINT_COMPRESSION — Figure 2

Calls: BHO_BINARIZATION
IMAGE_CLEANING
PORE_FILLING
RIDGE_THINNING
CURVE_EXTRACTION
RIDGE_CLEANING
RIDGE_SMOOTHING
CALCULATE_CHORD_POINTS
CURVE_SORTING
ENCODE_FINGERPRINT

SEARCHPRINT_DECOMPRESSION — Figure 3

Calls: DECODE_CURVE_LIST
B-SPLINE

IMAGE_CLEANING — Section 3

Called by: SEARCHPRINT_COMPRESSION

Calls: SPUR_REMOVAL

SPUR_REMOVAL — Section 3.2.1

Called by: IMAGE_CLEANING

Calls: PROCESS_CANDIDATE_SPUR_PIXEL

PROCESS_CANDIDATE_SPUR_PIXEL — Section 3.2.1

Called by: SPUR_REMOVAL
PROCESS_CANDIDATE_SPUR_PIXEL

Calls: PROCESS_CANDIDATE_SPUR_PIXEL

PORE_FILLING — Section 4.1

Called by: SEARCHPRINT_COMPRESSION

Calls: REMOVE_SMALL_PORES
REMOVE_LARGE_PORES

REMOVE_SMALL_PORES — Section 4.1.1

Called by: PORE_FILLING

Calls: FOUR-CONNECTED_COMPONENTS — Reference [3]
PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS
AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH

REMOVE_LARGE_PORES — Section 4.1.2.2

Called by: PORE_FILLING

Calls: PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS
AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH
LARGE_PORE_TEST

LARGE_PORE_TEST — Section 4.1.2.3

Called by: REMOVE_LARGE_PORES

Calls: SEARCH_EDGE_FOR_MINIMIZING_PIXEL

SEARCH_EDGE_FOR_MINIMIZING_PIXEL — Section 4.1.2.4

Called by: LARGE_PORE_TEST

PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS — Section 4.1.3

Called by: REMOVE_SMALL_PORES
REMOVE_LARGE_PORES

Calls: RIDGE_THINNING
AVERAGE_SECTION_RIDGE_WIDTH

AVERAGE_SECTION_RIDGE_WIDTH — Section 4.1.3

Called by: PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS

AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH — Section 4.1.3

Called by: REMOVE_SMALL_PORES
REMOVE_LARGE_PORES

RIDGE_THINNING Section 5.1

Called by: PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS
SEARCHPRINT_COMPRESSION

Calls: CHAMFER
DETECT_LOCAL_MAXIMA
FOLLOW_RIDGE

CHAMFER Section 5.1.1

Called by: RIDGE_THINNING

DETECT_LOCAL_MAXIMA Section 5.1.2

Called by: RIDGE_THINNING

FOLLOW_RIDGE Section 5.1.3

Called by: RIDGE_THINNING
FOLLOW_RIDGE

Calls: FOLLOW_RIDGE

CURVE_EXTRACTION Section 6.1

Called by: SEARCHPRINT_COMPRESSION

Calls: CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES
EXTRACT_CURVES

CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES Section 6.1.1.1

Called by: CURVE_EXTRACTION

Calls: APPLY_MASKS

APPLY_MASKS Section 6.1.1.2

Called by: CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES

EXTRACT_CURVES Section 6.1.2

Called by: CURVE_EXTRACTION

Calls: INITIALIZE_AND_FOLLOW_CURVE
FOLLOW_TO_DO_LIST

INITIALIZE_AND_FOLLOW_CURVE Section 6.1.2.1

Called by: EXTRACT_CURVES

Calls: FOLLOW

FOLLOW Section 6.1.2.2

Called by: INITIALIZE_AND_FOLLOW_CURVE
FOLLOW_TO_DO_LIST
FOLLOW

Calls: COUNT_NEIGHBORS_FOR_FOLLOWING
FOLLOW
FIND_POSSIBLE_BRANCHES
INITIALIZE_BRANCHES

COUNT_NEIGHBORS_FOR_FOLLOWING Section 6.1.2.3

Called by: FOLLOW

FIND_POSSIBLE_BRANCHES Section 6.1.2.3

Called by: FOLLOW

INITIALIZE_BRANCHES Section 6.1.2.5

Called by: FOLLOW

FOLLOW_TO_DO_LIST Section 6.1.2.6

Called by: EXTRACT_CURVES

Calls: FOLLOW

RIDGE_CLEANING Section 7.1

Called by: SEARCHPRINT_COMPRESSION

Calls: PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS_CURVE
SMALL_OFFSHOOT_CURVE_REMOVAL
CURVED_RIDGE_ENDING_REMOVAL
SMALL_RIDGE_BREAK_CONNECTION
SMALL_RIDGE_CONNECTION_REMOVAL
SMALL_RIDGE_SEGMENT_REMOVAL
BAD_BLOCK_BLANKING

PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS_CURVE Section 7.1

Called by: RIDGE_CLEANING

RIDGE_SECTION_AVERAGE_RIDGE_WIDTH Section 7.1.2

Called by: CONNECTION_SCORING_FUNCTION
SMALL_RIDGE_CONNECTION_REMOVAL

SMALL_OFFSHOOT_CURVE_REMOVAL Section 7.1.3

Called by: RIDGE_CLEANING

Calls: JOIN_CURVES

JOIN_CURVES — Section 7.1.3.1

Called by: SMALL_OFFSHOOT_CURVE_REMOVAL
BAD_BLOCK_BLANKING

SMALL_RIDGE_BREAK_CONNECTION — Section 7.1.4

Called by: RIDGE_CLEANING

Calls: CONNECTION_SCORING_FUNCTION
CONNECT_CURVES

CONNECTION_SCORING_FUNCTION — Section 7.1.4.1

Called by: SMALL_RIDGE_BREAK_CONNECTION

Calls: RIDGE_SECTION_AVERAGE_RIDGE_WIDTH

CONNECT_CURVES — Section 7.1.4.2

Called by: SMALL_RIDGE_BREAK_CONNECTION

SMALL_RIDGE_CONNECTION_REMOVAL — Section 7.1.5

Called by: RIDGE_CLEANING

Calls: RIDGE_SECTION_AVERAGE_RIDGE_WIDTH
DOT_PRODUCT

DOT_PRODUCT — Section 7.1.5

Called by: SMALL_RIDGE_CONNECTION_REMOVAL

SMALL_RIDGE_SEGMENT_REMOVAL — Section 7.1.6

Called by: RIDGE_CLEANING

RIDGE_SMOOTHING — Section 8.2

Called by: SEARCHPRINT_COMPRESSION

CALCULATE_CHORD_POINTS — Section 9.2

Called by: SEARCHPRINT_COMPRESSION

Calls: LINE_FITTING

LINE_FITTING — Section 9.2

Called by: CALCULATE_CHORD_POINTS
LINE_FITTING

Calls: LINE_FITTING

CURVE_SORTING Section 10.1

Called by: SEARCHPRINT_COMPRESSION

Calls: SELECTIVE_PROCESSING
CYCLIC_PROCESSING

SELECTIVE_PROCESSING Section 10.1.1

Called by: CURVE_SORTING

Calls: SEARCH_FOR_THE_BEST-FIT_CURVE
RESULTS_CHECKING

SEARCH_FOR_THE_BEST-FIT_CURVE Section 10.1.1.1

Called by: SELECTIVE_PROCESSING

Calls: DISTANCE_COMPARISON

DISTANCE_COMPARISON Section 10.1.1.2

Called by: SEARCH_FOR_THE_BEST-FIT_CURVE
SEARCH_FOR_THE_BEST_INSERTION_LOCATION

Calls: MAX_BITS
SUM_BITS
SUM_DISTANCE

MAX_BITS Section 10.1.1.2

Called by: DISTANCE_COMPARISON

Calls: NUM_BITS

SUM_BITS Section 10.1.1.2

Called by: DISTANCE_COMPARISON

Calls: NUM_BITS

SUM_DISTANCE Section 10.1.1.2

Called by: DISTANCE_COMPARISON

NUM_BITS Section 10.1.1.2

Called by: MAX_BITS
SUM_BITS
LINKAGE_COMPARISON
RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE

RESULTS_CHECKING Section 10.1.1.3

Called by: SELECTIVE_PROCESSING

CYCLIC_PROCESSING Section 10.1.2

Called by: CURVE_SORTING

Calls: SEARCH_FOR_THE_BEST_INSERTION_LOCATION
RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE

SEARCH_FOR_THE_BEST_INSERTION_LOCATION Section 10.1.2.1

Called by: CYCLIC_PROCESSING

Calls: DISTANCE_COMPARISON
LINKAGE_COMPARISON

LINKAGE_COMPARISON Section 10.1.2.2

Called by: SEARCH_FOR_THE_BEST_INSERTION_LOCATION

Calls: NUM_BITS
IS_SMALL

IS_SMALL Section 10.1.2.2

Called by: LINKAGE_COMPARISON

RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE Section 10.1.2.3

Called by: CYCLIC_PROCESSING

Calls: NUM_BITS

ENCODE_FINGERPRINT Section 11.4

Called by: SEARCHPRINT_COMPRESSION

Calls: CALCULATE_RELATIVE_DISTANCES
DETERMINE_FINGERPRINT_DATA_PROPERTIES
ENCODE_CURVE_LIST

CALCULATE_RELATIVE_DISTANCES Section 11.4.1

Called by: ENCODE_FINGERPRINT

Calls: DETERMINE_CURVE_DELTA_OFFSETS
DETERMINE_CURVE_JUMP_OFFSETS

DETERMINE_CURVE_DELTA_OFFSETS Section 11.4.1

Called by: CALCULATE_RELATIVE_DISTANCES

DETERMINE_CURVE_JUMP_OFFSETS Section 11.4.1

Called by: CALCULATE_RELATIVE_DISTANCES

DETERMINE_FINGERPRINT_DATA_PROPERTIES Section 11.4.2

Called by: DETERMINE_FINGERPRINT

Calls: DETERMINE_CURVE_SIGN_MONOTONICITY
GENERATE_HISTOGRAM
DETERMINE_WORD_SIZES

GENERATE_HISTOGRAM Section 11.4.2

Called by: DETERMINE_FINGERPRINT_DATA_PROPERTIES

DETERMINE_WORD_SIZES Section 11.4.2

Called by: DETERMINE_FINGERPRINT_DATA_PROPERTIES

DETERMINE_CURVE_SIGN_MONOTONICITY Section 11.4.2

Called by: DETERMINE_FINGERPRINT_DATA_PROPERTIES

Calls: SIGN

SIGN Section 11.4.2

Called by: DETERMINE_CURVE_SIGN_MONOTONICITY
ENCODE_JUMP

ENCODE_CURVE_LIST Section 11.4.3

Called by: ENCODE_FINGERPRINT

Calls: ENCODE_HEADER
OUTPUT_STREAM
ENCODE_CURVE_DELTAS
ENCODE_JUMP

ENCODE_HEADER Section 11.4.3

Called by: ENCODE_CURVE_LIST

Calls: ENCODE_WORD_SIZES
OUTPUT_STREAM

ENCODE_WORD_SIZES Section 11.4.3

Called by: ENCODE_HEADER

Calls: OUTPUT_STREAM

OUTPUT_STREAM Section 11.4.3

Called by: ENCODE_CURVE_LIST
ENCODE_HEADER
ENCODE_WORD_SIZES
ENCODE_USING_WORD_SIZES
ENCODE_JUMP_REFERENCE_END
ENCODE_SIGN
ENCODE_CURVE_DELTAS

ENCODE_USING_WORD_SIZES Section 11.4.3

Called by: ENCODE_JUMP
ENCODE_CURVE_DELTAS

Calls: OUTPUT_STREAM

ENCODE_JUMP Section 11.4.3

Called by: ENCODE_CURVE_LIST

Calls: ENCODE_JUMP_REFERENCE_END
ENCODE_USING_WORD_SIZES
ENCODE_SIGN
SIGN

ENCODE_JUMP_REFERENCE_END Section 11.4.3

Called by: ENCODE_JUMP

Calls: OUTPUT_STREAM

ENCODE_SIGN Section 11.4.3

Called by: ENCODE_JUMP
ENCODE_CURVE_DELTAS

Calls: OUTPUT_STREAM

ENCODE_CURVE_DELTAS Section 11.4.3

Called by: ENCODE_CURVE_LIST

Calls: ENCODE_USING_WORD_SIZES
OUTPUT_STREAM
ENCODE_SIGN

DECODE_CURVE_LIST Section 12.1

Called by: SEARCHPRINT_DECOMPRESSION

Calls: DECODE_HEADER
INPUT_STREAM
DECODE_CURVE_DELTAS
DECODE_JUMP
APPLY_CURVE_DELTA_OFFSETS
APPLY_JUMP_OFFSETS

DECODE_HEADER Section 12.1

Called by: DECODE_CURVE_LIST

Calls: DECODE_WORD_SIZES
INPUT_STREAM

DECODE_WORD_SIZES Section 12.1

Called by: DECODE_HEADER

Calls: INPUT_STREAM

INPUT_STREAM Section 12.1

Called by: DECODE_CURVE_LIST
DECODE_HEADER
DECODE_WORD_SIZES
DECODE_JUMP_REFERENCE_END
DECODE_USING_WORD_SIZES
DECODE_SIGN_FOR_VALUE
DECODE_CURVE_DELTAS
DECODE_SIGN

DECODE_JUMP Section 12.1

Called by: DECODE_CURVE_LIST

Calls: DECODE_JUMP_REFERENCE_END
DECODE_USING_WORD_SIZES
DECODE_SIGN_FOR_VALUE

DECODE_JUMP_REFERENCE_END Section 12.1

Called by: DECODE_JUMP

Calls: INPUT_STREAM

DECODE_USING_WORD_SIZES Section 12.1

Called by: DECODE_JUMP
DECODE_CURVE_DELTAS

Calls: INPUT_STREAM

DECODE_SIGN_FOR_VALUE Section 12.1

Called by: DECODE_JUMP

Calls: INPUT_STREAM]

DECODE_CURVE_DELTAS Section 12.1

Called by: DECODE_CURVE_LIST

Calls: DECODE_USING_WORD_SIZES
INPUT_STREAM
DECODE_SIGN
APPLY_SIGN_TO_VALUE

DECODE_SIGN Section 12.1

Called by: DECODE_CURVE_DELTAS

Calls: INPUT_STREAM

APPLY_SIGN_TO_VALUE Section 12.1

Called by: DECODE_CURVE_DELTAS

APPLY_JUMP_OFFSET Section 12.1

Called by: DECODE_CURVE_LIST

APPLY_CURVE_DELTA_OFFSETS Section 12.1

Called by: DECODE_CURVE_LIST

B-SPLINE Section 13.1

Called by: SEARCHPRINT_DECOMPRESSION

Calls: A
B
C
D

A Section 13.1

Called by: B-SPLINE

B Section 13.1

Called by: B-SPLINE

C Section 13.1

Called by: **B-SPLINE**

D Section 13.1

Called by: **B-SPLINE**

BHO_BINARIZATION Appendix A

Called by: **SEARCHPRINT_COMPRESSION**

Calls: **WRITE_BLOCK_FILE**

WRITE _BLOCK_FILE Appendix A

Called by: **BHO_BINARIZATION**

CURVED_RIDGE_ENDING_REMOVAL Appendix B

Called by: **RIDGE_CLEANING**

Calls: **PROCESS_RIDGE_ENDING**

PROCESS_RIDGE_ENDING Appendix B

Called by: **CURVED_RIDGE_ENDING_REMOVAL**

BAD_BLOCK_BLANKING Appendix C

Called by: **RIDGE_CLEANING**

Calls: **JOIN_CURVES**

FIND_BEST_PARTITION Appendix D

Used to set parameters in **PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS**

APPENDIX F

LISTS OF CONSTANTS, PARAMETERS, AND VARIABLES

This appendix contains separate tables for constants, parameters, and variables that are used in the pseudocode in this document. Each table contains the name of the item, the pseudocode function that refers to it, and the section where the pseudocode resides. The parameter and variable lists also contain a brief description of the item, while the constant list is preceded by a table showing the constant groupings within which the values must be distinct.

Table F-1. Constant Groupings

WHITE, BLACK

FALSE, TRUE

TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, BOTTOM_RIGHT, LEFT, TOP, RIGHT, BOTTOM

MONOTONIC_BOTH, MONOTONIC_DX, MONOTONIC_DY, NON_MONOTONIC

FIRST_ENDPOINT, LAST_ENDPOINT

POSITIVE, NEGATIVE

BACKGROUND, RIDGE, LOCAL_MAXIMUM

Table F-2. List of Constants

Constant	Function	Section
BACKGROUND	DETECT_LOCAL_MAXIMA	5.1.2
BACKGROUND	FOLLOW_RIDGE	5.1.3
BIFURCATION	EXTRACT_CURVES	6.1.2
BIFURCATION	FOLLOW	6.1.2.2
BIFURCATION	FOLLOW_TO_DO_LIST	6.1.2.6
BIFURCATION	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1
BIFURCATION	INITIALIZE_BRANCHES	6.1.2.5
BLACK	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3
BLACK	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1
BLACK	DYNAMIC_THRESHOLDING	2.2
BLACK	EXTRACT_CURVES	6.1.2
BLACK	FIND_POSSIBLE_BRANCHES	6.1.2.3
BLACK	FOLLOW	6.1.2.2
BLACK	FOLLOW_TO_DO_LIST	6.1.2.6
BLACK	PROCESS_CANDIDATE_SPUR_PIXEL	3.2.1
BLACK	REMOVE_LARGE_PORES	4.1.2.2
BLACK	REMOVE_SMALL_PORES	4.1.1
BLACK	SPUR_REMOVAL	3.2.1
BOTTOM	FOLLOW_RIDGE	5.1.3
BOTTOM_LEFT	FOLLOW_RIDGE	5.1.3
BOTTOM_RIGHT	FOLLOW_RIDGE	5.1.3
CONTINUE_FIRST_STAGE	RESULTS_CHECKING	10.1.1.3
CONTINUE_FIRST_STAGE	SELECTIVE_PROCESSING	10.1.1
EMPTY	EXTRACT_CURVES	6.1.2
EMPTY	FIND_POSSIBLE_BRANCHES	6.1.2.3
EMPTY	FOLLOW_TO_DO_LIST	6.1.2.6

Table F-2. List of Constants (continued)

Constant	Function	Section
EMPTY	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1
EMPTY	INITIALIZE_BRANCHES	6.1.2.5
EMPTY	BAD_BLOCK_BLANKING	Appendix C
FALSE	APPLY_MASKS	6.1.1.2
FALSE	CONNECT_CURVES	7.1.4.2
FALSE	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1
FALSE	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2
FALSE	DISTANCE_COMPARISON	10.1.1.2
FALSE	FOLLOW_RIDGE	5.1.3
FALSE	LINKAGE_COMPARISON	10.1.2.2
FALSE	REMOVE_SMALL_PORES	4.1.1
FALSE	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1
FALSE	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1
FALSE	SMALL_RIDGE_BREAK_CONNECTION	7.1.4
FIRST_ENDPOINT	APPLY_JUMP_OFFSET	12.2
FIRST_ENDPOINT	DECODE_JUMP_REFERENCE_END	12.2
FIRST_ENDPOINT	DETERMINE_JUMP_OFFSET	11.4.1
FIRST_ENDPOINT	ENCODE_JUMP_REFERENCE_END	11.4.3
FIRST_ENDPOINT	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1
FIRST_ENDPOINT	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1
FIRST_STAGE_FINISHED	RESULTS_CHECKING	10.1.1.3
ILLEGAL_CONNECTION	CONNECTION_SCORING_FUNCTION	7.1.4.1
LARGE_PORE_CANDIDATE	LARGE_PORE_TEST	4.1.2.3
LARGE_PORE_CANDIDATE	REMOVE_LARGE_PORES	4.1.2.2
LAST_ENDPOINT	APPLY_JUMP_OFFSET	12.2
LAST_ENDPOINT	DECODE_JUMP_REFERENCE_END	12.2

Table F-2. List of Constants (continued)

Constant	Function	Section
LAST_ENDPOINT	Determine_Jump_Offset	11.4.1
LAST_ENDPOINT	Encode_Jump_Reference_End	11.4.3
LAST_ENDPOINT	Results_Checking	10.1.1.3
LAST_ENDPOINT	Search_For_The_Best-Fit_Curve	10.1.1.1
LAST_ENDPOINT	Search_For_The_Best_Insertion_Location	10.1.2.1
LEFT	Follow_Ridge	5.1.3
LOCAL_MAXIMUM	Detect_Local_Maxima	5.1.2
LOCAL_MAXIMUM	Follow_Ridge	5.1.3
LOCAL_MAXIMUM	Follow_Ridge	5.1.3
LOCAL_MAXIMUM	Ridge_Thinning	5.1
MAX_OFFSET_POSSIBLE	Results_Checking	10.1.1.3
MAX_OFFSET_POSSIBLE	Search_For_The_Best-Fit_Curve	10.1.1.1
MONOTONIC_BOTH	Decode_Curve_Deltas	12.2
MONOTONIC_BOTH	Determine_Curve_Sign_Monotonicity	11.4.2
MONOTONIC_BOTH	Encode_Curve_Deltas	11.4.3
MONOTONIC_DX	Decode_Curve_Deltas	12.2
MONOTONIC_DX	Determine_Curve_Sign_Monotonicity	11.4.2
MONOTONIC_DX	Encode_Curve_Deltas	11.4.3
MONOTONIC_DY	Decode_Curve_Deltas	12.2
MONOTONIC_DY	Determine_Curve_Sign_Monotonicity	11.4.2
MONOTONIC_DY	Encode_Curve_Deltas	11.4.3
NEGATIVE	Apply_Sign_To_Value	12.2
NEGATIVE	Decode_Sign	12.2
NEGATIVE	Encode_Sign	11.4.3
NEGATIVE	Sign	11.4.2
NON_MONOTONIC	Decode_Curve_Deltas	12.2

Table F-2. List of Constants (continued)

Constant	Function	Section
NON_MONOTONIC	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2
NON_MONOTONIC	ENCODE_CURVE_DELTAS	11.4.3
NOT_VALID	LARGE_PORE_TEST	4.1.2.3
NOT_VALID	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4
NULL	CYCLIC_PROCESSING	10.1.2
NULL	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3
NULL	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1
NULL	SELECTIVE_PROCESSING	10.1.1
POSITIVE	DECODE_SIGN	12.2
POSITIVE	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2
POSITIVE	ENCODE_SIGN	11.4.3
POSITIVE	SIGN	11.4.2
POSSIBLE	FIND_POSSIBLE_BRANCHES	6.1.2.3
REPEAT_FIRST_STAGE	RESULTS_CHECKING	10.1.1.3
REPEAT_FIRST_STAGE_SEARCH	SELECTIVE_PROCESSING	10.1.1
RIDGE	CHAMFER	5.1.1
RIDGE	FOLLOW_RIDGE	5.1.3
RIGHT	FOLLOW_RIDGE	5.1.3
SEED	FIND_POSSIBLE_BRANCHES	6.1.2.3
SEED	FOLLOW	6.1.2.2
SEED	INITIALIZE_BRANCHES	6.1.2.5
TRUE	APPLY_MASKS	6.1.1.2
TRUE	CONNECT_CURVES	7.1.4.2
TRUE	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1
TRUE	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2
TRUE	DISTANCE_COMPARISON	10.1.1.2

Table F-2. List of Constants (continued)

Constant	Function	Section
TRUE	FOLLOW_RIDGE	5.1.3
TRUE	LINKAGE_COMPARISON	10.1.2.2
TRUE	PORE_FILLING	4.1
TRUE	REMOVE_SMALL_PORES	4.1.1
TRUE	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3
TRUE	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1
TRUE	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1
UNDEFINED_DIRECTION	RIDGE_THINNING	5.1
WHITE	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1
WHITE	CREASE_TRIMMING	3.1.1
WHITE	DYNAMIC_THRESHOLDING	2.2
WHITE	FOLLOW	6.1.2.2
WHITE	FOLLOW_TO_DO_LIST	6.1.2.6
WHITE	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1
WHITE	PROCESS_CANDIDATE_SPUR_PIXEL	3.2.1
WHITE	REMOVE_LARGE_PORES	4.1.2.2
WHITE	REMOVE_SMALL_PORES	4.1.1
ZERO	DECODE_SIGN	12.2
ZERO	DECODE_SIGN_FOR_VALUE	12.2
ZERO	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2
ZERO	SIGN	11.4.2

Table F-3. List of Parameters

Parameter	Function	Section	Description
$A_{COLINEAR}$ = 45 degrees	CONNECTION_SCORING_FUNCTION	7.1.4.1	Angular limit for colinearity
$A_{PARALLEL}$ = 45 degrees	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Angular limit for parallelism of neighboring ridges
$A_{SEGMENT}$ = 60 degrees	CONNECTION_SCORING_FUNCTION	7.1.4.1	Angular limit for colinearity with a small segment
$A_{STRAIGHT}$ = 90 degrees	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Angular limit for straightness
ALLOWABLE_RESIDUE	LINE_FITTING	9.2	Smallest acceptable perpendicular distance between the curve segment and the chord segment
$BITS_{HUFFMAN\_INDEX}$ = 2	ENCODE_HEADER	11.4.3	The number of bits used to represent the sign monotonicity type index that is assigned to a particular Huffman symbol

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$BITS_{IMAGE\_SIZE} = 16$	ENCODE_HEADER	11.4.3	The number of bits used to represent the dimensions of the image
$BITS_{IMAGE\_SIZE} = 16$	DECODE_HEADER	12.2	The number of bits used to represent the dimensions of the image
$BITS_{MINIMUM\_NUMBER\_OF\_DELTA} = 2$	DECODE_HEADER	12.2	The number of bits used to represent the minimum number of deltas of any curve of the curve list
$BITS_{MINIMUM\_NUMBER\_OF\_DELTA} = 2$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The number of bits used to represent the minimum number of deltas of any curve of the curve list

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$BITS_{MINIMUM\_NUMBER\_OF\_DELTA} = 2$	ENCODE_HEADER	11.4.3	The number of bits used to represent the minimum number of deltas of any curve of the curve list
$BITS_{NUMBER\_OF\_WORD\_SIZES} = 2$	DECODE_WORD_SIZES	12.2	The number of bits used to represent the number of word sizes in a word_size coding scheme
$BITS_{NUMBER\_OF\_WORD\_SIZES} = 2$	ENCODE_WORD_SIZES	11.4.3	The number of bits used to represent the number of word sizes in a word_size coding scheme
$BITS_{NUMBER\_OF\_CURVES} = 11$	DECODE_CURVE_LIST	12.2	The number of bits used to represent the number of curves in the fingerprint curve list

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$BITS_{NUMBER\_OF\_CURVES} = 11$	ENCODE_CURVE_LIST	11.4.3	The number of bits used to represent the number of curves in the fingerprint curve list
$BITS_{WORD\_SIZE} = 4$	DECODE_WORD_SIZES	12.2	The number of bits used to represent a word size in a word_size coding scheme
$BITS_{WORD\_SIZE} = 4$	ENCODE_WORD_SIZES	11.4.3	The number of bits used to represent a word size in a word_size coding scheme
$BITS_{X\_COORDINATE} = 9$	DECODE_CURVE_LIST	12.2	The number of bits used to represent an absolute x-coordinate in the live-scan fingerprint image (based on the width of the image)

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$BITS_{X\_COORDINATE} = 9$	ENCODE_CURVE_LIST	11.4.3	The number of bits used to represent an absolute x-coordinate in the live-scan fingerprint image (based on the width of the image)
$BITS_{Y\_COORDINATE} = 10$	DECODE_CURVE_LIST	12.2	The number of bits used to represent an absolute y-coordinate in the live-scan fingerprint image (based on the height of the image)
$BITS_{Y\_COORDINATE} = 10$	ENCODE_CURVE_LIST	11.4.3	The number of bits used to represent an absolute y-coordinate in the live-scan fingerprint image (based on the height of the image)

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$C = 450$	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of columns in the fingerprint image
$C_P = 9$	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of horizontal sections in the partition of the fingerprint image used to calculate average ridge widths
$C_{\%} = 25\%$	RESULTS_CHECKING	10.1.1.3	Maximum percentage of curves that can exist in the *unsorted_list* before the cyclic processing stage will begin if SEARCH_FOR_THE_BEST-FIT_CURVE fails
$D_{CYCLIC} = 64$	CYCLIC_PROCESSING	10.1.2	Initial filter value assigned to each curve upon entering the cyclic processing stage

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$D_{SELECT} = 128$	SELECTIVE_PROCESSING	10.1.1	Initial value assigned to the filter variable upon entering the selective processing stage
$E_{MAX} = 50$	LARGE_PORE_TEST	4.1.2.3	The maximum distance for a search along a ridge edge, in pixels
$E_{MAX} = 50$	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The maximum distance for a search along a ridge edge, in pixels
$F_{DOUBLY\_CONNECTED} = 2.25$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Maximum length of a doubly connected curve in terms of the average of its neighboring end sections' average ridge widths

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$F_{OFFSHOOT\ CURVE} = 2.0$	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Length of the smallest allowable singly connected curve in terms of $ridge\_width_{fingerprint}$
$F_{RIDGE\_BREAK} = 1.0$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Maximum length of a possibly connectable ridge break in terms of $ridge\_width_{fingerprint}$
$F_{UNCONNECTED\_CURVE} = 5.0$	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Length of the smallest allowable unconnected curve in terms of $ridge\_width_{fingerprint}$

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$H = 5$	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	When choosing a ridge edge pixel to minimize the distance to a point, a pixel is considered to minimize this distance if no ridge edge pixel within H pixels yields a smaller distance
$L_{DOUBLY\_CONNECTED} = 20$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Maximum length of a doubly connected curve to be considered for removal
$L_{MAX} = 10$	REMOVE_LARGE_PORES	4.1.2.2	Maximum ratio between the white area of a large pore candidate and the average ridge width in its neighborhood

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$LU_{MAX} = 15$	LARGE_PORE_TEST	4.1.2.3	Maximum distance to the left of, or up from, an initial pore pixel to its enclosing ridge edge, in pixels
$MAX_{OFFSET} = 601$	RESULTS_CHECKING	10.1.1.3	The larger of the width and height of the image, plus one
$MAX_{OFFSET} = 601$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The larger of the width and height of the image, plus one
$N = 30$	B-SPLINE	13.2	Height and width (in pixels) of the pixel neighborhood window
$N = 9$	DYNAMIC_THRESHOLDING	2.2	Height and width (in pixels) of the pixel neighborhood window
$P_{INIT} = 6$	SELECTIVE_PROCESSING	10.1.1	Initial value for the penalty variable

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$P_{MAX} = 2.5$	LARGE_PORE_TEST	4.1.2.3	Maximum ratio between the pore and ridge widths of a candidate and the average neighborhood ridge width in the large pore model
$P_{MIN} = 3.0$	LARGE_PORE_TEST	4.1.2.3	Minimum ratio between the width of a pore candidate and the ridges to either side of it in the large pore model
$R = 600$	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of rows in the fingerprint image
$R_P = 10$	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of vertical sections in the partition of the fingerprint image used to calculate average ridge widths

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$RADIUS_{DEFAULT}$ = 81	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Default search radius for the small ridge break connection algorithm
$RIDGE\_SIZE_{MIN}$ = 5	CONNECTION_SCORING_FUNCTION	7.1.4.1	Minimum length of a curve allowed to be used in calculating colinearity
$RIDGE\_SIZE_{MIN}$ = 5	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Minimum length of a curve allowed to be used in calculating colinearity
S = 15	IS_SMALL	10.1.2.2	Limit used to test whether one insertion linkage has more small offsets than another insertion linkage

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
S = 15	LINKAGE_COMPARISON	10.1.2.2	Limit used to test whether one insertion linkage has more small offsets than another insertion linkage
$S_{VERTICAL\_RUN} = 1/2\ l_{WIDTH}$	CREASE_TRIMMING	3.1.1	Width of sampled *vertical_run*
$SIZES_{DELTAS} = 2$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	Maximum number of word sizes allowed for encoding the deltas of curves
$SIZES_{JUMPS} = 3$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	Maximum number of word sizes allowed for encoding the jumps between curves
$SIZES_{NUM\_DELTAS} = 2$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	Maximum number of word sizes allowed for encoding the number of deltas in curves

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$T_{OFFSET} = 40$	CREASE_TRIMMING	3.1.1	Number of rows below the crease where trimming begins
W	RIDGE_SMOOTHING	8.2	The window size constant for the smoothing window
$W_{DOUBLY\_CONNECTED} = 0.95$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Maximum average ridge width of the doubly connected curve in terms of the average of its neighboring end sections average ridge widths
$W_{MAX} = 8.0$	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Maximum width of a ridge for the average ridge width calculation, in pixels

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$W_{MIN} = 1.4$	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Minimum width of a ridge for the average ridge width calculation, in pixels
$Z_{DELTA\_SECTION\_SIZE} = 20$	FIND_BEST_PARTITION	D.2	Maximum variation in the width or height of a fingerprint section
$Z_{DESIRED\_SECTION\_SIZE} = 60$	FIND_BEST_PARTITION	D.2	Desired width and height of a fingerprint section
$Z_{END\_SIZE} = 6$	CURVED_RIDGE_ENDING_REMOVAL	B.1	Maximum number of points that can be removed from a curved ridge ending
$Z_{END\_SIZE} = 6$	PROCESS_RIDGE_ENDING	B.1	Maximum number of points that can be removed from a curved ridge ending

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$Z_{LENGTH\_OFFSHOOT} = 5.0$	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Length of the smallest allowable singly connected curve in terms of the local average ridge width
$Z_{LENGTH\_UNCONNECTED} = 10.0$	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Length of the smallest allowable unconnected curve in terms of the local average ridge width
$Z_{SATURATION\_RATIO} = 2.0$	BHO_BINARIZATION	A.1.3	Maximum ratio between pixels at 254 and pixels at 255 for an unsaturated image
$Z_{TAPER\_RATIO} = 0.75$	PROCESS_RIDGE_ENDING	B.1	Tapering ridge width ratio limit for curved ridge ending
$Z_{THRESH\_ANGLE} = 30$ degrees	PROCESS_RIDGE_ENDING	B.1	Curvature limit for curved ridge ending

Table F-3. List of Parameters (continued)

Parameter	Function	Section	Description
$Z_{\text{THRESHOLD\_FRACTION}} = 0.8$	BHO_BINARIZATION	A.1.3	Fraction of the distance between the mean pixel value and the maximum pixel value in an image used to determine *zz_top*
$Z_{\text{WIDTH\_OFFSHOOT}} = 0.65$	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Width of the smallest allowable singly connected curve in terms of the local average ridge width
$Z_{\text{WIDTH\_UNCONNECTED}} = 0.65$	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Width of the smallest allowable unconnected curve in terms of the local average ridge width
ZN = 24	BHO_BINARIZATION	App A	Height and width (in pixels) of blocks

Table F-4. List of Variables

Variable	Function	Section	Description
μ_{column}	RIDGE_SMOOTHING	8.2	Average column coordinate of the points currently within the smoothing window
μ_I	DYNAMIC_THRESHOLDING	2.2	Image overall mean pixel value
μ_{row}	RIDGE_SMOOTHING	8.2	Average row coordinate of the points currently within the smoothing window
$\mu_{vertical\_run}$	CREASE_TRIMMING	3.1.1	Mean of the sampled ***vertical_run*** lengths
μ_{window}	DYNAMIC_THRESHOLDING	2.2	Mean pixel value of a pixel's neighborhood window
$\sigma_{vertical\_run}$	CREASE_TRIMMING	3.1.1	Standard deviation of the sampled ***vertical_run*** lengths
a	APPLY_CURVE_DELTA_OFFSETS	12.2	Index in *curve* of point prior to one currently being considered

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
a	APPLY_JUMP_OFFSET	12.2	Index in *curve_list* of curve prior to one currently being considered
a	CALCULATE_RELATIVE_DISTANCES	11.4.1	Index in *curve_list* of curve prior to one currently being considered
a	CHAMFER	5.1.1	Candidate chamfer value
a	CONNECT_CURVES	7.1.4.2	One of two curves to be connected into one curve
a	CONNECTION_SCORING_FUNCTION	7.1.4.1	An endpoint that is on one side of a potential small ridge break
a	DECODE_CURVE_LIST	12.2	Index in *curve_list* of curve prior to one currently being considered
a	DECODE_JUMP	12.2	Index in *curve_list* of curve prior to one currently being considered
a	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	Index in *curve* of point prior to one currently being considered

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
a	DETERMINE_JUMP_OFFSET	11.4.1	Index in *curve_list* of curve prior to one currently being considered
a	DOT_PRODUCT	7.1.5	A point
a	ENCODE_JUMP	11.4.3	Index in *curve_list* of curve prior to one currently being considered
a	JOIN_CURVES	7.1.3.1	One of two curves to be concatenated into one curve
a	RIDGE_SMOOTHING	8.2	Point on the front of the smoothing window
a	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	One of two curves that shares an endpoint with the curve being considered
a	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Endpoint initiating search for small ridge break
a	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A curve overlapping the curve being considered for being a small ridge connection

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
after_curve	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	Curve in *sorted_list* before which the first curve in *unsorted_list* best fits in *sorted_list*
$after\_curve_{reference\_end\_flag}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The reference end flag of *after_curve*
$angle\_score_a$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Value related to the angle of change traversed from ref_a through endpoint ***a*** to endpoint ***b***
$angle\_score_b$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Value related to the angle of change traversed from ref_b through endpoint ***b*** to endpoint ***a***
area_vector	REMOVE_SMALL_PORES	4.1.1	Vector of areas corresponding to labels in ***LABEL_IMAGE***
a_x	APPLY_CURVE_DELTA_OFFSETS	12.2	The x coordinate of point ***a***
a_x	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The y coordinate of point ***a***

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
a_y	APPLY_CURVE_DELTA_OFFSETS	12.2	The y coordinate of point ***a***
a_y	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The y coordinate of point ***a***
b	APPLY_CURVE_DELTA_OFFSETS	12.2	Index in *curve* of point currently being considered
b	APPLY_JUMP_OFFSET	12.2	Index in *curve_list* of curve currently being considered
b	CALCULATE_RELATIVE_DISTANCES	11.4.1	Index in *curve_list* of curve currently being considered
b	CHAMFER	5.1.1	Candidate chamfer value
b	CONNECT_CURVES	7.1.4.2	One of two curves to be connected into one curve
b	CONNECTION_SCORING_FUNCTION	7.1.4.1	An endpoint that is on one side of a potential small ridge break
b	DECODE_CURVE_LIST	12.2	Index in *curve_list* of curve currently being considered
b	DECODE_JUMP	12.2	Index in *curve_list* of current curve being considered

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
b	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	Index in *curve* of point currently being considered
b	DETERMINE_JUMP_OFFSET	11.4.1	Index in *curve_list* of curve currently being considered
b	DOT_PRODUCT	7.1.5	A point
b	ENCODE_JUMP	11.4.3	Index in *curve_list* of curve currently being considered
b	JOIN_CURVES	7.1.3.1	One of two curves to be concatenated into one curve
b	RIDGE_SMOOTHING	8.2	Point on the back of the smoothing window
b	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	One of two curves that shares an endpoint with the curve being considered
b	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Candidate endpoint for being part of small ridge break

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
b	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A curve overlapping the curve being considered for being a small ridge connection
b_x	APPLY_CURVE_DELTA_OFFSETS	12.2	The x coordinate of point *b*
b_x	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The x coordinate of point *b*
b_y	APPLY_CURVE_DELTA_OFFSETS	12.2	The y coordinate of point *b*
b_y	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The x coordinate of point *b*
before_curve	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	Curve in *sorted_list* after which the first curve in *unsorted_list* best fits in *sorted_list*
$before\_curve_{reference\_end\_flag}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The reference end flag of *before_curve*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
best_insertion_linkage	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The best insertion linkage found for placing the first curve of *unsorted_list* into *sorted_list*
$\textit{best\_insertion\_linkage}_{\textit{from\_endpoint\_offset}}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The offset of the "from" side of the best insert linkage
$\textit{best\_insertion\_linkage}_{\textit{to\_endpoint\_offset}}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The offset of the "to" side of the best insert linkage
$\textit{best\_insertion\_linkage}_{\textit{x\_offset\_from}}$	CYCLIC_PROCESSING	10.1.2	The x offset of the "from" side of the best insert linkage
$\textit{best\_insertion\_linkage}_{\textit{x\_offset\_from}}$	LINKAGE_COMPARISON	10.1.2.2	The x offset of the jump of *best_insertion_linkage* from first curve of *unsorted_list* to *curve* of *sorted_list*
$\textit{best\_insertion\_linkage}_{\textit{x\_offset\_to}}$	CYCLIC_PROCESSING	10.1.2	The x offset of the "to" side of the best insert linkage

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$\mathit{best\_insertion\_linkage}_{x\_offset\_to}$	LINKAGE_COMPARISON	10.1.2.2	The x offset of the jump of *best_insertion_ linkage* from *curve* of *sorted_list* to first curve of *unsorted_list*
$\mathit{best\_insertion\_linkage}_{y\_offset\_from}$	CYCLIC_PROCESSING	10.1.2	The y offset of the "from" side of the best insert linkage
$\mathit{best\_insertion\_linkage}_{y\_offset\_from}$	LINKAGE_COMPARISON	10.1.2.2	The y offset of the jump of *best_insertion_ linkage* from first curve of *unsorted_list* to *curve* of *sorted_list*
$\mathit{best\_insertion\_linkage}_{y\_offset\_to}$	CYCLIC_PROCESSING	10.1.2	The y offset of the "to" side of the best insert linkage
$\mathit{best\_insertion\_linkage}_{y\_offset\_to}$	LINKAGE_COMPARISON	10.1.2.2	The y offset of the jump of *best_insertion_ linkage* from *curve* of *sorted_list* to first curve of *unsorted_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
best_insertion_location	CYCLIC_PROCESSING	10.1.2	Curve in *sorted_list* having after which *first_curve* of *unsorted_list* should be placed
best_insertion_location	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	Curve in *sorted_list* after which the first curve in *unsorted_list* best fits in *sorted_list*
best_insertion_location	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Location in *sorted_list* that the first curve in *unsorted_list* is to be placed
best_jump	DISTANCE_COMPARISON	10.1.1.2	Best jump found so far in the sorting process
best_jump	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The best jump to and point in *unsorted_list* from the last point in *sorted_list*
$best\_jump_x$	RESULTS_CHECKING	10.1.1.2	The x offset of the best jump found by the sorting process

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
best_jump_from_unsorted_curve$_x$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x offset of the best jump from first curve of *unsorted_list* to *curve_plus_one* of *sorted_list*
best_jump_to_unsorted_curve$_x$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x offset of the best jump from *curve* of *sorted_list* to first curve of *unsorted_list*
best_jump$_x$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The x value of *best_jump*
best_jump$_y$	RESULTS_CHECKING	10.1.1.2	The y offset of the best jump found by the sorting process
best_jump$_y$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The y value of *best_jump*
best_numbits	LINKAGE_COMPARISON	10.1.2.2	Number of bits required to represent the largest offset magnitude in *best_insertion_linkage*
best_quantity_smalls	LINKAGE_COMPARISON	10.1.2.2	Number of offsets in *best_insertion_linkage* that are less than or equal to S

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
best_score	CREASE_TRIMMING	3.1.1	The largest score encountered while searching for the crease
$best\_score_a$	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	The largest score of the endpoints in the candidate list
bi	RIDGE_CLEANING	7.1	Block row index
bi	WRITE_BLOCK_FILE	A.2.2	Block row index
bits	DETERMINE_WORD_SIZES	11.4.2	The number of bits calculated for some combination of word sizes
$bits_{min}$	DETERMINE_WORD_SIZES	11.4.2	The minimum number of bits
bj	RIDGE_CLEANING	7.1	Block colmn index
bj	WRITE_BLOCK_FILE	A.2.2	Block colmn index
block_file	BHO_BINARIZATION	A.3	File containing the ridge direction data structure
block_file	WRITE_BLOCK_FILE	A.2.2	File containing the ridge direction data structure
branch	INITIALIZE_BRANCHES	6.1.2.5	A leg of a bifurcation
c	CHAMFER	5.1.1	Candidate chamfer value

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
c	CONNECT_CURVES	7.1.4.2	Curve being generated by connecting two curves across a small ridge break
C	DETECT_LOCAL_MAXIMA	5.1.2	Chamfered image
c	DOT_PRODUCT	7.1.5	A point
c	JOIN_CURVES	7.1.3.1	Curve being generated by joining two curves
C	RIDGE_CLEANING	7.1	Chamfer image
C	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	Chamfered image
C	RIDGE_THINNING	5.1	Chamfered image
c	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A curve overlapping the curve being considered for being a small ridge connection
C	CURVED_RIDGE_ENDING_REMOVAL	B.1	Chamfered image
C_{bottom}	CREASE_TRIMMING	3.1.1	The row index corresponding to the estimated bottom of the flexion crease
C_{center}	CREASE_TRIMMING	3.1.1	The row index corresponding to the estimated center of the flexion crease

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
c_{max}	CHAMFER	5.1.1	A very large integer value to indicated that a pixel has not been yet processed
candidate	LARGE_PORE_TEST	4.1.2.3	White region containing P_o, i.e., the pore candidate
candidate_list	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	List of candidate endpoints that may be part of a small ridge break
CHAMFER	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Chamfered fingerprint image
CHAMFER	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Chamfered fingerprint image
check_list	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	A list of candidate endpoints that are to be checked for being the mutually best small ridge break
chord	LINE_FITTING	9.2	The line passing through *first_endpoint* and *second_endpoint*
closest_curve	RESULTS_CHECKING	10.1.1.2	Curve found to be closest to last point in *sorted_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
closest_curve	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Curve found to be closest to last point in *sorted_list*
column	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Column index of the ***RIDGE_WIDTH_ARRAY***
column	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Column index of the ***RIDGE_WIDTH_ARRAY***
columns_per_section	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Number of columns in a fingerprint image section
columns_per_section	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of columns in a fingerprint image section
component_max	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	Number of bits necessary to represent largest offest

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
connection_score	CONNECTION_SCORING_FUNCTION	7.1.4.1	Value indicating the relative possibility that a pair of endpoints is part of a small ridge break
count	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Number of ridge points in the current fingerprint image section
current_insertion_linkage	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Linkage between curves in *sorted_list* and the first curve of *unsorted_list* that is currently being considered
$current\_insertion\_linkage_{from\_jump}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The jump of the current insertion linkage from the first curve of *unsorted_list* to *curve_plus_one* in *curve_list*
$current\_insertion\_linkage_{to\_jump}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The jump of the current insertion linkage from curve in *curve_list* to first curve of *unsorted_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$\mathit{current\_insertion\_linkage}_{x\_offset\_from}$	LINKAGE_COMPARISON	10.1.2.2	The x offset of the jump of *current_insertion_linkage* from the first curve of *unsorted_list* to *curve* of *sorted_list*
$\mathit{current\_insertion\_linkage}_{x\_offset\_to}$	LINKAGE_COMPARISON	10.1.2.2	The x offset of the jump of *current_insertion_linkage* from *curve* of *sorted_list* to first curve of *unsorted_list*
$\mathit{current\_insertion\_linkage}_{y\_offset\_from}$	LINKAGE_COMPARISON	10.1.2.2	The x offset of the jump of *current_insertion_linkage* from first curve of *unsorted_list* to *curve* of *sorted_list*
$\mathit{current\_insertion\_linkage}_{y\_offset\_to}$	LINKAGE_COMPARISON	10.1.2.2	The y offset of the jump of *current_insertion_linkage* from *curve* of *sorted_list* to first curve of *unsorted_list*
current_jump	DISTANCE_COMPARISON	10.1.1.2	Jump currently being compared against *best_jump*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
current_jump	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Jump between endpoints in *last_curve* and *curve*
current_jump	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The jump between the first curve in *unsorted_list* to a curve from *sorted_list*
$\mathit{current\_jump}_x$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Absolute difference between the x coordinates of endpoints in *last_curve* and *curve*
$\mathit{current\_jump}_x$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x offset of *current_jump*
$\mathit{current\_jump}_y$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Absolute difference between the y coordinates of endpoints in *last_curve* and *curve*
$\mathit{current\_jump}_y$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y offset of *current_jump*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
current_numbits	LINKAGE_COMPARISON	10.1.2.2	Number of bits required to represent the largest offset magnitude in *current_insertion_linkage*
current_quantity_smalls	LINKAGE_COMPARISON	10.1.2.2	Number of offsets in *current_insertion_linkage* that are less than or equal to S
curve	APPLY_CURVE_DELTA_OFFSETS	12.2	One curve in the *curve_list*
curve	B-SPLINE	13.2	Current curve being processed
curve	CALCULATE_CHORD_POINTS	9.2	The fingerprint curve currently being processed
curve	CALCULATE_RELATIVE_DISTANCES	11.4.1	One curve in the *curve_list*
curve	CYCLIC_PROCESSING	10.1.2	Curve from *unsorted_list* currently being considered
curve	DECODE_CURVE_DELTAS	12.2	One curve in the *curve_list*
curve	DECODE_CURVE_LIST	12.2	One curve in the *curve_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve	Decode_Jump_Reference_End	12.2	One curve in the *curve_list*
curve	Determine_Curve_Delta_Offsets	11.4.1	One curve in the *curve_list*
curve	Determine_Curve_Sign_Monotonicity	11.4.2	One curve in the *curve_list*
curve	Determine_Fingerprint_Data_Properties	11.4.2	One curve in the *curve_list*
curve	Curved_Ridge_Ending_Removal	B.1	A curve under consideration for having a curved ridge ending
curve	Encode_Curve_Deltas	11.4.3	One curve in the *curve_list*
curve	Encode_Curve_List	11.4.3	One curve in the *curve_list*
curve	Encode_Jump_Reference_End	11.4.3	One curve in the *curve_list*
curve	Extract_Curves	6.1.2	A curve being extracted from a fingerprint image
curve	Follow_To_Do_List	6.1.2.6	A curve being extracted from a fingerprint image
curve	Initialize_and_Follow_Curve	6.1.2.1	A curve being extracted from a fingerprint image

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve	INITIALIZE_BRANCHES	6.1.2.5	A curve being extracted from a fingerprint image
curve	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	The curve along which the ridge section resides
curve	RIDGE_SMOOTHING	8.2	The curve that is in the process of being smoothed
curve	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Curve currently being considered
curve	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Curve in *sort_list* being considered for being an insertion point
curve	SELECTIVE_PROCESSING	10.1.1	Curve that is closest to the center of the fingerprint image
curve	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	The curve being considered for being a small offshoot curve
curve	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Curve being considered for being part of a small ridge break

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A curve under consideration for being a small ridge connection
curve	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	A curve under consideration for being a small ridge segment
$curve_{first\_endpoint\_x}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The x coordinate of first endpoint in *curve*
$curve_{first\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the first endpoint of the *curve* from *sorted_list*
$curve_{first\_endpoint\_y}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The y coordinate of first endpoint in *curve*
$curve_{first\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the first endpoint of the *curve* from *sorted_list*
$curve_{last\_endpoint\_x}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The x coordinate of last endpoint in *curve*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$curve_{last\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the last endpoint of *curve* from *sorted_list*
$curve_{last\_endpoint\_y}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The y coordinate of last endpoint in *curve*
$curve_{last\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the last endpoint of *curve* from *sorted_list*
$curve_a$	CONNECTION_SCORING_FUNCTION	7.1.4.1	One of two curves that are potentially part of a small ridge break
$curve_b$	CONNECTION_SCORING_FUNCTION	7.1.4.1	One of two curves that are potentially part of a small ridge break
$curve_{filter\_value}$	CYCLIC_PROCESSING	10.1.2	Filter value for *curve*
curve_2	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1	The second half of a curve being extracted from a fingerprint image
curve_list	BAD_BLOCK_BLANKING	C.2	List of curves that represent the fingerprint

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve_list	B-SPLINE	13.2	List of curves that represent the fingerprint
curve_list	CALCULATE_CHORD_POINTS	9.2	List of curves that represent the fingerprint
curve_list	CALCULATE_RELATIVE_DISTANCES	11.4.1	List of curves representing the fingerprint
curve_list	CONNECT_CURVES	7.1.4.2	List of curves representing the fingerprint
curve_list	DECODE_CURVE_LIST	12.2	List of curves representing the fingerprint
curve_list	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	List of curves representing the fingerprint
curve_list	CURVED_RIDGE_ENDING_REMOVAL	B.1	List of curves representing the fingerprint
curve_list	ENCODE_CURVE_LIST	11.4.3	List of curves representing the fingerprint
curve_list	ENCODE_FINGERPRINT	11.4	List of curves representing the fingerprint
curve_list	JOIN_CURVES	7.1.3.1	List of curves representing the fingerprint

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve_list	RIDGE_CLEANING	7.1	List of curves representing the fingerprint
curve_list	RIDGE_SMOOTHING	8.2	List of curves representing the fingerprint
curve_list	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	List of curves representing the fingerprint
curve_list	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	List of curves representing the fingerprint
curve_list	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	List of curves representing the fingerprint
curve_list	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	List of curves representing the fingerprint
curve_plus_one	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The curves following *curve* in *sorted_list*
$curve\_plus\_one_{first\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the first endpoint of *curve_plus_one* from *sorted_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$curve\_plus\_one_{first\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the first endpoint of *curve_plus_one* from *sorted_list*
$curve\_plus\_one_{last\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the last endpoint of *curve_plus_one* from *sorted_list*
$curve\_plus\_one_{last\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the last endpoint of *curve_plus_one* from *sorted_list*
curve_set	CURVE_EXTRACTION	6.1	Set of curves extracted from fingerprint image
curve_set	EXTRACT_CURVES	6.1.2	Set of curves extracted from fingerprint image
curve_set	FOLLOW_TO_DO_LIST	6.1.2.6	Set of curves extracted from fingerprint image
$curve\_sign_x$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	The sign of the first $delta_x$ in curve
$curve\_sign_y$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	The sign of the first $delta_y$ in curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
curve_sign_monotonicity	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	The monotonicity type, or sign fluctuations, determined for the delta offsets of a particular curve
curve_sign_monotonicity	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The monotonicity type, or sign fluctuations, determined for the delta offsets of a particular curve
d	CHAMFER	5.1.1	Candidate chamfer value
d	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A curve overlapping the curve being considered for being a small ridge connection
delta	DECODE_CURVE_DELTAS	12.2	The relative distance between two adjacent points within a curve
delta	ENCODE_CURVE_DELTAS	11.4.3	The relative distance between two adjacent points within a curve

rTable F-4. List of Variables (continued)

Variable	Function	Section	Description
$delta_{minimum\_per\_curve}$	DECODE_CURVE_DELTAS	12.2	The minimum number of deltas per curve for all curves in *curve_list*
$delta_{minimum\_per\_curve}$	DECODE_HEADER	12.2	The minimum number of deltas per curve for all curves in *curve_list*
$delta_{minimum\_per\_curve}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The minimum number of deltas per curve for all curves in *curve_list*
$delta_x$	APPLY_CURVE_DELTA_OFFSETS	12.2	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	DECODE_CURVE_DELTAS	12.2	The relative distance in the x direction of two adjacent points within a curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$delta_x$	DECODE_HEADER	12.2	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	ENCODE_CURVE_DELTAS	11.4.3	The relative distance in the x direction of two adjacent points within a curve
$delta_x$	ENCODE_HEADER	11.4.3	The relative distance in the x direction of two adjacent points within a curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$delta_y$	APPLY_CURVE_DELTA_OFFSETS	12.2	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	DECODE_CURVE_DELTAS	12.2	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	DECODE_HEADER	12.2	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	DETERMINE_CURVE_DELTA_OFFSETS	11.4.1	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The relative distance in the y direction of two adjacent points within a curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$delta_y$	ENCODE_CURVE_DELTAS	11.4.3	The relative distance in the y direction of two adjacent points within a curve
$delta_y$	ENCODE_HEADER	11.4.3	The relative distance in the y direction of two adjacent points within a curve
delta_count	DECODE_CURVE_DELTAS	12.2	The number of deltas in curve less $delta_{minimum\_per\_curve}$
delta_count	ENCODE_CURVE_DELTAS	11.4.3	The number of deltas in curve less $delta_{minimum\_per\_curve}$
direction	FOLLOW_RIDGE	5.1.3	Pixel direction to previous pixel of ridge following
direction	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The search direction: either clockwise or counterclockwise

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
distance	CONNECTION_SCORING_FUNCTION	7.1.4.1	Euclidean distance between two endpoint that are on either side of a potential small ridge break
e	CHAMFER	5.1.1	Candidate chamfer value
$edge_{left}$	CREASE_TRIMMING	3.1.1	The left edge of the fingerprint impression
$edge_{right}$	CREASE_TRIMMING	3.1.1	The right edge of the fingerprint impression
$endpoint_0$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	An endpoint of the curve being considered for being a small ridge connection
$endpoint_1$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	An endpoint of the curve being considered for being a small ridge connection
$endpoint_a$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The endpoint of a curve overlapping the curve being considered for being a small ridge connection

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$endpoint_b$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The endpoint of a curve overlapping the curve being considered for being a small ridge connection
$endpoint_c$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The endpoint of a curve overlapping the curve being considered for being a small ridge connection
$endpoint_d$	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The endpoint of a curve overlapping the curve being considered for being a small ridge connection
endpoint_flag	RESULTS_CHECKING	10.1.1.2	Flag indicating whether a jump originated from the first or last endpoint in a curve
endpoint_flag	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Flag indicating whether a jump originated from the first or last endpoint in a curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
endpoint_flag	SELECTIVE_PROCESSING	10.1.1	Flag indicating whether a jump originated from the first or last endpoint in a curve
endpoint_flag_one	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Temporary endpoint flag for *curve* from *sorted_list*
endpoint_flag_two	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Temporary endpoint flag for the first curve on *unsorted_list*
endpoint_map	BAD_BLOCK_BLANKING	C.2	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
endpoint_map	CONNECT_CURVES	7.1.4.2	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
endpoint_map	CURVED_RIDGE_ENDING_REMOVAL	B.1	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
endpoint_map	JOIN_CURVES	7.1.3.1	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
endpoint_map	RIDGE_CLEANING	7.1	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
endpoint_map	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
endpoint_map	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
endpoint_map	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	A representation of curve endpoints allowing efficient search for endpoints near a specified coordinate
f	CHAMFER	5.1.1	Candidate chamfer value
first_curve	CYCLIC_PROCESSING	10.1.2	First curve in *unsorted_list*
first_curve	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The first curve of *unsorted_list*
$\mathit{first\_curve}_{filter\_value}$	CYCLIC_PROCESSING	10.1.2	Filter value for *first_curve*
$\mathit{first\_curve}_{filter\_value}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The filter value of the first curve in *unsorted_list*
$\mathit{first\_curve}_{filter\_value}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Filter value for first curve in *unsorted_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$first\_curve_{first\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the first endpoint of the first curve in *unsorted_list*
$first\_curve_{first\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the first endpoint of the first curve in *unsorted_list*
$first\_curve_{last\_endpoint\_x}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The x coordinate of the first endpoint of the first curve in *unsorted_list*
$first\_curve_{last\_endpoint\_y}$	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The y coordinate of the first endpoint of the first curve in *unsorted_list*
$first\_curve_{reference\_end\_flag}$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The reference end flag of *first_curve*
first_endpoint	LINE_FITTING	9.2	An endpoint of the current chord
first_pt	APPLY_JUMP_OFFSET	12.2	First point in curve *b*
first_pt	DETERMINE_JUMP_OFFSET	11.4.1	First point in curve *b*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$first\_pt_x$	APPLY_JUMP_OFFSET	12.2	The x coordinate of first point in curve *b*
$first\_pt_x$	DETERMINE_JUMP_OFFSET	11.4.1	The x coordinate of the first point in curve *b*
$first\_pt_y$	APPLY_JUMP_OFFSET	12.2	The y coordinate of first point in curve *b*
$first\_pt_y$	DETERMINE_JUMP_OFFSET	11.4.1	The y coordinate of the first point in curve *b*
flag	DECODE_JUMP_REFERENCE_END	12.2	Flag value to decode
flag	DECODE_SIGN	12.2	Flag value to decode
flag	DECODE_SIGN_FOR_VALUE	12.2	Flag value to decode
from_endpoint_offset	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The jump from *first_curve* to *after_curve*
$from\_endpoint\_offset_x$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The x offset of the jump from *first_curve* to *after_curve*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$from\_endpoint\_offset_y$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The y offset of the jump from *first_curve* to *after_curve*
g	CHAMFER	5.1.1	Candidate chamfer value
h	CHAMFER	5.1.1	Candidate chamfer value
height	EXTRACT_CURVES	6.1.2	Height of fingerprint image
height	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Height of fingerprint image
height	REMOVE_LARGE_PORES	4.1.2.2	Height of fingerprint image
height	REMOVE_SMALL_PORES	4.1.1	Height of fingerprint image
$height_C$	DETECT_LOCAL_MAXIMA	5.1.2	Height of chamfer image ***C***
$height_I$	CHAMFER	5.1.1	Height of image ***I***
histogram	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	A histogram of differences of number of deltas of each curve less $delta_{minimum\_per\_curve}$
histogram	DETERMINE_WORD_SIZES	11.4.2	Frequency distribution of values

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
histogram	GENERATE_HISTOGRAM	11.4.2	Frequency distribution of values
horizontal_run	CREASE_TRIMMING	3.1.1	The longest run of consecutive white pixels for every row
i	A	13.1	Array index for x coordinates
i	APPLY_MASKS	6.1.1.2	Row index
i	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Row index
i	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Row index
i	B	13.2	Array index for x coordinates
I	BHO_BINARIZATION	A.3	Gray-scale fingerprint image
i	RIDGE_CLEANING	7.1	Row index
i	B-SPLINE	13.2	Array index of current curve
i	C	13.2	Array index for x coordinates
i	CHAMFER	5.1.1	Row index
I	CHAMFER	5.1.1	Gray-scale fingerprint image
i	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1	Row index
i	CREASE_TRIMMING	3.1.1	Row index
I	CREASE_TRIMMING	3.1.1	Gray-scale fingerprint image

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
i	D	13.2	Array index for x coordinates
i	DETECT_LOCAL_MAXIMA	5.1.2	Row index
i	DYNAMIC_THRESHOLDING	2.2	Row index
I	DYNAMIC_THRESHOLDING	2.2	Gray-scale fingerprint image
i	EXTRACT_CURVES	6.1.2	Row index
i	FOLLOW_RIDGE	5.1.3	Row index
I	IMAGE_CLEANING	3	Gray-scale fingerprint image
i	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1	Row index
i	LARGE_PORE_TEST	4.1.2.3	Row index
i	LINE_FITTING	9.2	Loop index
i	PROCESS_CANDIDATE_SPUR_PIXEL	3.2.1	Row index
I	PROCESS_CANDIDATE_SPUR_PIXEL	3.2.1	Gray-scale fingerprint image
i	REMOVE_LARGE_PORES	4.1.2.2	Row index
i	REMOVE_SMALL_PORES	4.1.1	Row index
i	RIDGE_THINNING	5.1	Row index
I	RIDGE_THINNING	5.1	Gray-scale fingerprint image
i	SPUR_REMOVAL	3.2.1	Row index
I	SPUR_REMOVAL	3.2.1	Gray-scale fingerprint image
IMAGE	APPLY_MASKS	6.1.1.2	Fingerprint image to which masks are being applied

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
IMAGE	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1	Fingerprint image in which ridges is being converted to single-pixel width
IMAGE	CURVE_EXTRACTION	6.1	Fingerprint image from which curves are being extracted
IMAGE	EXTRACT_CURVES	6.1.2	Fingerprint image from which curves are being extracted
IMAGE	FOLLOW	6.1.2.2	Fingerprint image from which curves are being extracted
IMAGE	FOLLOW_TO_DO_LIST	6.1.2.6	Fingerprint image from which curves are being extracted
IMAGE	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1	Fingerprint image from which curves are being extracted
IMAGE	PORE_FILLING	4.1	Fingerprint image for which pores are being filled
IMAGE	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Fingerprint image from which neighborhood ridge widths are being extracted

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
IMAGE	Remove_Large_Pores	4.1.2.2	Fingerprint image from which large pores are being removed
IMAGE	Remove_Small_Pores	4.1.1	Fingerprint image from which small pores are being removed
j	Apply_Masks	6.1.1.2	Column index
j	Average_Neighborhood_Ridge_Width	4.1.3	Column index
j	Average_Section_Ridge_Width	4.1.3	Column index
j	Ridge_Cleaning	7.1	Column index
j	B-Spline	13.1	Array index of current point in current curve
j	Calculate_Chord_Points	9.2	First index
j	Chamfer	5.1.1	Column index
j	Convert_to_Single_Pixel_Wide_Ridges	6.1.1.1	Column index
j	Crease_Trimming	3.1.1	Column index
j	Detect_Local_Maxima	5.1.2	Column index
j	Dynamic_Thresholding	2.2	Column index
j	Extract_Curves	6.1.2	Column index
j	Follow_Ridge	5.1.3	Column index
j	Initialize_and_Follow_Curve	6.1.2.1	Column index
j	Large_Pore_Test	4.1.2.3	Column index
j	Line_Fitting	9.2	First index
j	Process_Candidate_Spur_Pixel	3.2.1	Column index
j	Remove_Large_Pores	4.1.2.2	Column index

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
j	REMOVE_SMALL_PORES	4.1.1	Column index
j	RIDGE_THINNING	5.1	Column index
j	SPUR_REMOVAL	3.2.1	Column index
jump	DETERMINE_JUMP_OFFSET	11.4.1	The relative distances from the endpoint of one curve and the first endpoint of the next consecutive curve
jump	MAX_BITS	10.1.1.2	Jump for which the largest number of bits is being calculated
jump	SUM_BITS	10.1.1.2	Jump for which the sum of the bits is being calculated
jump	SUM_DISTANCE	10.1.1.2	Jump for which the sum of the distances is being calculated
$jump_x$	APPLY_JUMP_OFFSET	12.2	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$jump_x$	DECODE_HEADER	12.2	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_x$	DECODE_JUMP	12.2	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_x$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$jump_x$	DETERMINE_JUMP_OFFSET	11.4.1	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_x$	ENCODE_HEADER	11.4.3	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_x$	ENCODE_JUMP	11.4.3	The relative distance in the x direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_x$	MAX_BITS	10.1.1.2	The x offset of *jump*
$jump_x$	SUM_BITS	10.1.1.2	The x offset of *jump*
$jump_x$	SUM_DISTANCE	10.1.1.2	The x offset of *jump*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$jump_y$	APPLY_JUMP_OFFSET	12.2	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_y$	DECODE_HEADER	12.2	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_y$	DECODE_JUMP	12.2	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$jump_y$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_y$	DETERMINE_JUMP_OFFSET	11.4.1	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_y$	ENCODE_HEADER	11.4.3	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$jump_y$	ENCODE_JUMP	11.4.3	The relative distance in the y direction of the endpoint of one curve and the first endpoint of the next consecutive curve
$jump_y$	MAX_BITS	10.1.1.2	The y offset of *jump*
$jump_y$	SUM_BITS	10.1.1.2	The y offset of *jump*
$jump_y$	SUM_DISTANCE	10.1.1.2	The y offset of *jump*
k	CALCULATE_CHORD_POINTS	9.2	Last index
k	CREASE_TRIMMING	3.1.1	Temporary row index
k	LINE_FITTING	9.2	Last index
LABEL_IMAGE	REMOVE_SMALL_PORES	4.1.1	Fingerprint image with labeled white regions
last_curve	SELECTIVE_PROCESSING	10.1.1	Curve at the end of *sorted_list*
$last\_curve_{first\_endpoint\_x}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The x coordinate of first endpoint in *last_curve*
$last\_curve_{first\_endpoint\_y}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The y coordinate of first endpoint in *last_curve*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$last\_curve_{last\_endpoint\_x}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The x coordinate of last endpoint in *last_curve*
$last\_curve_{last\_endpoint\_y}$	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	The y coordinate of last endpoint in *last_curve*
$last\_curve_{reference\_end\_flag}$	RESULTS_CHECKING	10.1.1.2	reference end flag for the last curve in *sorted_list*
last_point	COUNT_NEIGHBORS_FOR_FOLLOWING	6.1.2.2	Last point on a curve
last_point	FIND_POSSIBLE_BRANCHES	6.1.2.3	Last point on a curve
last_point	FOLLOW	6.1.2.2	Last point on a curve
last_point	FOLLOW_TO_DO_LIST	6.1.2.6	Last point on a curve
last_point	INITIALIZE_BRANCHES	6.1.2.5	Last point on a curve
$length_{LW}$	DETERMINE_WORD_SIZES	11.4.2	The largest number of bits needed to represent any element of the histogram
$length_{SW}$	DETERMINE_WORD_SIZES	11.4.2	The largest number of bits needed to represent any element of the histogram

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$length_{zero}$	DETERMINE_WORD_SIZES	11.4.2	For three word size case, number of bits needed to represent the value zero
m	LINE_FITTING	9.2	Boundary of points with same residue
m	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The distance between *P* and *Q* or *Q'*
magnitude	DECODE_USING_WORD_SIZES	12.2	Absolute value
magnitude	ENCODE_USING_WORD_SIZES	11.4.3	Absolute value
magnitude	GENERATE_HISTOGRAM	11.4.2	Absolute value
mask	APPLY_MASKS	6.1.1.2	A mask from a mask set
mask_set	APPLY_MASKS	6.1.1.2	One of *nub_mask_set* or *topology_mask_set*
max_I	DYNAMIC_THRESHOLDING	2.2	Maximum pixel value over entire image
$max_{vertical\_run}$	CREASE_TRIMMING	3.1.1	Maximum of the sampled ***vertical_run*** lengths

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
max_best_bits	DISTANCE_COMPARISON	10.1.1.2	Largest number of bits necessary to represent the magnitude of the x or y offset of *best_jump*
max_best_bits	RESULTS_CHECKING	10.1.1.2	Largest number of bits necessary to represent the magnitude of the x or y offset of *best_jump*
max_current_bits	DISTANCE_COMPARISON	10.1.1.2	Largest number of bits necessary to represent the magnitude of the x or y offset of *current_jump*
max_offset	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Limit for largest offset
max_offset	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Limit for largest offset
max_offset	SELECTIVE_PROCESSING	10.1.1	Limit for largest offset
maximum_number_of_word_sizes	DETERMINE_WORD_SIZES	11.4.2	The maximum number of word sizes allowable
min_I	DYNAMIC_THRESHOLDING	2.2	Minimum pixel value over entire image

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$monotonic_x$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	Constant positive or negative sign values in the x coordinate
$monotonic_y$	DETERMINE_CURVE_SIGN_MONOTONICITY	11.4.2	Constant positive or negative sign values in the y coordinate
n	INPUT_STREAM	12.2	Number of bits
n	NUM_BITS	10.1.1.2	Value for which the number of bits necessary to represent it is being calculated
n	OUTPUT_STREAM	11.4.3	Number of bits
n	PROCESS_CANDIDATE_SPUR_PIXEL	3.2.1	Number of pixels neighboring the current pixel $I(i, j)$ whose value equals BLACK
n	RIDGE_SMOOTHING	8.2	Current size of the smoothing window
n	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	Pixel counter
n_neighbors	COUNT_NEIGHBORS_FOR_FOLLOWING	6.1.2.2	Number of neighbors to be followed from a curve point

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
n_neighbors	FOLLOW	6.1.2.2	Number of neighbors to be followed from a curve point
n_past_min	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	Number of pixels past the current Q_{min} that the search has proceeded
neighbor	COUNT_NEIGHBORS_FOR_FOLLOWING	6.1.2.2	A neighbor of a curve point
neighbor	FIND_POSSIBLE_BRANCHES	6.1.2.3	A neighbor of a curve point
neighbor	FOLLOW	6.1.2.2	A neighbor of a curve point
neighbor_1	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1	The first neighbor of an initial curve point
neighbor_2	INITIALIZE_AND_FOLLOW_CURVE	6.1.2.1	The second neighbor of an initial curve point
nub_mask_set	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1	Set of masks for nub removal
num_regions	REMOVE_SMALL_PORES	4.1.1	Number of white regions in fingerprint image

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
number_of_pixels	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	The number of pixels in ridge section on *curve* between P_{start} and P_{end}
number_of_points_in_curve	B-SPLINE	13.2	Number of points in current curve
number_of_word_sizes	DETERMINE_WORD_SIZES	11.4.2	The calculated number of word sizes allowable
p	CONNECT_CURVES	7.1.4.2	Point in curve being appended
p	JOIN_CURVES	7.1.3.1	Point in curve being appended.
P	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The fixed pixel to which this routine minimizes the distance along a ridge edge
P_c	LARGE_PORE_TEST	4.1.2.3	Center of area of large pore candidate
P_{ccw1}	LARGE_PORE_TEST	4.1.2.3	Ridge pixel on side 1 of large pore candidate in counterclockwise direction

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
P_{ccw2}	LARGE_PORE_TEST	4.1.2.3	Ridge pixel on side 2 of large pore candidate across ridge from P_{ccw1}
P_{cw1}	LARGE_PORE_TEST	4.1.2.3	Ridge pixel on side 1 of large pore candidate in clockwise direction
P_{cw2}	LARGE_PORE_TEST	4.1.2.3	Ridge pixel on side 2 of large pore candidate across ridge from P_{cw1}
P_e	LARGE_PORE_TEST	4.1.2.3	Initial edge pixel of ridge surrounding large pore candidate
$P_{e,left}$	LARGE_PORE_TEST	4.1.2.3	Left ridge edge pixel of large pore candidate
$P_{e,up}$	LARGE_PORE_TEST	4.1.2.3	Top ridge edge pixel of large pore candidate
P_{end}	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	The ending point of a ridge section
P_o	LARGE_PORE_TEST	4.1.2.3	Initial pixel of large pore candidate

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
P_{p1}	LARGE_PORE_TEST	4.1.2.3	Ridge edge pixel on side 1 of large pore candidate closest to P_c
$P_{p1,ccw}$	LARGE_PORE_TEST	4.1.2.3	Ridge edge pixel on side 1 of large pore candidate closest to P_c in counterclockwise direction from P_e
$P_{p1,cw}$	LARGE_PORE_TEST	4.1.2.3	Ridge edge pixel on side 1 of large pore candidate closest to P_c in clockwise direction from P_e
P_{p2}	LARGE_PORE_TEST	4.1.2.3	Ridge edge pixel on side 2 of large pore candidate closest to P_c
P_{start}	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	The starting point of a ridge section
P_{temp}	LARGE_PORE_TEST	4.1.2.3	Temporary pixel for large pore candidate calculations

rTable F-4. List of Variables (continued)

Variable	Function	Section	Description
p_distance	LINE_FITTING	9.2	Perpendicular distance from the *point* to the *chord* connecting the endpoints
peak_score	CREASE_TRIMMING	3.1.1	The score proportional to the area of each peak in the ***row_score***
penalty_size	DISTANCE_COMPARISON	10.1.1.2	Limit for the penalty test
penalty_size	RESULTS_CHECKING	10.1.1.2	Limit for the penalty test
penalty_size	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	Limit for the penalty test
penalty_size	SELECTIVE_PROCESSING	10.1.1	Limit for the penalty test
pixel	REMOVE_LARGE_PORES	4.1.2.2	An image pixel
pixel_set_to_white	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1	Flag
point	LINE_FITTING	9.2	Point on *curve* currently being considered
possible_branches	FIND_POSSIBLE_BRANCHES	6.1.2.3	Set of possible branches from a curve point
possible_branches	FOLLOW	6.1.2.2	Set of possible branches from a curve point

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
possible_branches	INITIALIZE_BRANCHES	6.1.2.5	Set of possible branches from a curve point
previous_point	COUNT_NEIGHBORS_FOR_FOLLOWING	6.1.2.2	Next to last point on a curve
previous_point	FIND_POSSIBLE_BRANCHES	6.1.2.3	Next to last point on a curve
previous_point	FOLLOW	6.1.2.2	Next to last point on a curve
previous_residue	LINE_FITTING	9.2	Previous largest residue value
Q	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	A white pixel on a ridge edge that serves as a starting point for the search
Q_{min}	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The minimizing white pixel on a ridge edge from the search
Q'	SEARCH_EDGE_FOR_MINIMIZING_PIXEL	4.1.2.4	The current white pixel on a ridge edge in the search
$radius_{searcha}$	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Search limit for finding candidate endpoints for small ridge breaks
RAW_THIN	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Raw (not final) thinned fingerprint image

rTable F-4. List of Variables (continued)

Variable	Function	Section	Description
RAW_THIN	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Raw (not final) thinned fingerprint image
ref_a	CONNECTION_SCORING_FUNCTION	7.1.4.1	The point that is $section\_size_a$ points down $curve_a$ from endpoint *a*
ref_a	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The point that is $reference\_length_h$ points down curve *a* from its overlapping endpoint
ref_b	CONNECTION_SCORING_FUNCTION	7.1.4.1	The point that is $section\_size_a$ points down $curve_a$ from endpoint *a*
ref_b	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The point that is $reference\_length_h$ points down curve *b* from its overlapping endpoint
ref_c	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The point that is $reference\_length_h$ points down curve *c* from its overlapping endpoint

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
ref_d	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	The point that is $reference\_length_h$ points down curve *d* from its overlapping endpoint
ref_pt	APPLY_JUMP_OFFSET	12.2	Reference end point of curve ***a***
ref_pt	DETERMINE_JUMP_OFFSET	11.4.1	First point in curve ***a***
ref_pt_x	APPLY_JUMP_OFFSET	12.2	The x coordinate of reference end point of curve ***a***
ref_pt_x	DETERMINE_JUMP_OFFSET	11.4.1	The x coordinate of the first point in curve ***a***
ref_pt_y	APPLY_JUMP_OFFSET	12.2	The y coordinate of reference end point of curve ***a***
ref_pt_y	DETERMINE_JUMP_OFFSET	11.4.1	The y coordinate of the first point in curve ***a***
reference_length	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Size of the desired curve reference section
residue	LINE_FITTING	9.2	Distance from *point* to chord

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
reverse_flag	RESULTS_CHECKING	10.1.1.2	Boolean indicating whether the current curve being added to *sorted_list* needs its point order reversed
reverse_flag	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	Boolean indicating whether the current curve being added to *sorted_list* needs its point order reversed
reverse_flag	SELECTIVE_PROCESSING	10.1.1	Boolean indicating whether the current curve being added to *sorted_list* needs its point order reversed
reverse_flag_one	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Temporary reversal flag for *curve* from *sorted_list*
reverse_flag_two	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	Temporary reversal flag for the first curve on *unsorted_list*
$ridge\_width_a$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Average ridge width of the curve section on $curve_a$

rTable F-4. List of Variables (continued)

Variable	Function	Section	Description
$ridge\_width_{ave}$	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	The resulting average ridge width value for the ridge section on *curve* between P_{start} and P_{end}
$ridge\_width_b$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Average ridge width of the curve section on $curve_b$
$ridge\_width_{fingerprint}$	RIDGE_CLEANING	7.1	Average width of all ridges in fingerprint
$ridge\_width_{fingerprint}$	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Average width of all ridges in fingerprint
$ridge\_width_{fingerprint}$	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Average width of all ridges in fingerprint
RIDGE_WIDTH_ARRAY	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Array of average section ridge widths
RIDGE_WIDTH_ARRAY	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Array of average section ridge widths
rotation	APPLY_MASKS	6.1.1.2	A rotation for a nub mask
row	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Row index of the ***RIDGE_WIDTH_ARRAY***

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
row	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Row index of the ***RIDGE_WIDTH_ARRAY***
row_score	CREASE_TRIMMING	3.1.1	The row score proportional to the white region around each *horizontal_runi*
rows_per_section	AVERAGE_NEIGHBORHOOD_RIDGE_WIDTH	4.1.3	Number of rows in a fingerprint image section
rows_per_section	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Number of rows in a fingerprint image section
s_{column}	RIDGE_SMOOTHING	8.2	A running sum of column coordinates along a section of a curve being smoothed
s_{row}	RIDGE_SMOOTHING	8.2	A running sum of row coordinates along a section of a curve being smoothed
same_residue	LINE_FITTING	9.2	Boundary of points with same residue

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
saved_endpoint_flag_one	CYCLIC_PROCESSING	10.1.2	Endpoint flag for the curve on the "from" side of the candidate insertion point
saved_endpoint_flag_one	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The endpoint flag for *curve* of the *best_insertion_linkage*
saved_endpoint_flag_one	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The endpoint flag for *curve* of the *best_insertion_linkage*
saved_endpoint_flag_two	CYCLIC_PROCESSING	10.1.2	Endpoint flag for the curve on the "to" side of the candidate insertion point
saved_endpoint_flag_two	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The endpoint flag for *first_curve* of the *best_insertion_linkage*
saved_endpoint_flag_two	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The endpoint flag for *first_curve* of the *best_insertion_linkage*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
saved_reverse_flag_one	CYCLIC_PROCESSING	10.1.2	Reversal flag for the curve on the "from" side of the candidate insertion point
saved_reverse_flag_one	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The reversal flag for *curve* of the *best_insertion_linkage*
saved_reverse_flag_one	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The reversal flag for *curve* of the *best_insertion_linkage*
saved_reverse_flag_two	CYCLIC_PROCESSING	10.1.2	Reversal flag for the curve on the "to" side of the candidate insertion point
saved_reverse_flag_two	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The reversal flag for *first_curve* of the *best_insertion_linkage*
saved_reverse_flag_two	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	The reversal flag for *first_curve* of the *best_insertion_linkage*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
score	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Value indicating the relative possibility that a pair of endpoints is part of a small ridge break
second_endpoint	LINE_FITTING	9.2	An endpoint of the current chord
section_size	CONNECTION_SCORING_FUNCTION	7.1.4.1	Size of the desired curve end section
$section\_size_a$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Size of end section for curve ***a***
$section\_size_b$	CONNECTION_SCORING_FUNCTION	7.1.4.1	Size of end section for curve ***b***
seed_index	EXTRACT_CURVES	6.1.2	Index for labeling branch seed curves
seed_index	INITIALIZE_BRANCHES	6.1.2.5	Index for labeling branch seed curves
sign	APPLY_SIGN_TO_VALUE	12.2	Sign of value
sign	ENCODE_SIGN	11.4.3	Sign of value
$sign_x$	DECODE_CURVE_DELTAS	12.2	The x coordinate sign
$sign_y$	DECODE_CURVE_DELTAS	12.2	The y coordinate sign
smooth_curve	RIDGE_SMOOTHING	8.2	The resulting curve that is in the process of being smoothed

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
smooth_curve_list	RIDGE_SMOOTHING	8.2	List of curves representing the fingerprint that have been smoothed
sorted_list	CURVE_SORTING	10.1	List of curves after having been processed by sorting
sorted_list	CYCLIC_PROCESSING	10.1.2	List of curves after having been placed by sorting
sorted_list	RESULTS_CHECKING	10.1.1.2	List of curves after having been placed by sorting
sorted_list	SEARCH_FOR_THE_BEST_INSERTION_LOCATION	10.1.2.1	List of curves after having been placed by sorting
sorted_list	SELECTIVE_PROCESSING	10.1.1	List of curves after having been placed by sorting
spline_x	B-SPLINE	13.2	Calculated x coordinate for current iteration
spline_y	B-SPLINE	13.2	Calculated y-coordinate for current iteration

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
status	FOLLOW_RIDGE	5.1.3	Boolean value indicating end condition of ridge following
status	SELECTIVE_PROCESSING	10.1.1	Boolean indicating the completion of this stage of curve sorting
status	SMALL_RIDGE_BREAK_CONNECTION	7.1.4	Boolean value indicating status of the search for small ridge breaks
sum	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Sum of ridge widths in section
sum	CREASE_TRIMMING	3.1.1	The estimation of white area in the fingerprint image surrounding a particular row
sum	RIDGE_SECTION_AVERAGE_RIDGE_WIDTH	7.1.2	Sum of chamfer values along a ridge section
sum_best_bits	DISTANCE_COMPARISON	10.1.1.2	Sum of the bits necessary to represent the magnitudes of the x and y offsets of *best_jump*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
sum_best_distance	DISTANCE_COMPARISON	10.1.1.2	Sum of the magnitudes of the x and y offsets of *best_jump*
sum_current_bits	DISTANCE_COMPARISON	10.1.1.2	Sum of the bits necessary to represent the magnitudes of the x and y offsets of *current_jump*
sum_current_distance	DISTANCE_COMPARISON	10.1.1.2	Sum of the magnitudes of the x and y offsets of *current_jump*
SW	DETERMINE_WORD_SIZES	11.4.2	Short word size in number of bits
T	BHO_BINARIZATION	A.3	Thresholded fingerprint image
t	B-SPLINE	13.2	B-Spline correction coefficient
t	CREASE_TRIMMING	3.1.1	Threshold of ***vertical_run*** lengths
T	CURVED_RIDGE_ENDING_REMOVAL	B.1	Thinned fingerprint image
T	DYNAMIC_THRESHOLDING	2.2	Thresholded fingerprint image

Table F-4. List of Variables (continued).

Variable	Function	Section	Description
T	FOLLOW_RIDGE	5.1.3	Thinned fingerprint image
T	JOIN_CURVES	7.1.3.1	Thinned fingerprint image
T	RIDGE_THINNING	5.1	Thinned fingerprint image
T	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Thinned fingerprint image
T	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Thinned fingerprint image
T	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Thinned fingerprint image
t_{lower}	DYNAMIC_THRESHOLDING	2.2	Absolute lower limit for thresholding
t_{max}	CREASE_TRIMMING	3.1.1	Maximum threshold of ***vertical_run*** lengths
t_{upper}	DYNAMIC_THRESHOLDING	2.2	Absolute upper limit for thresholding
temp_chord_points	CALCULATE_CHORD_POINTS	9.2	Temporary ordered list of chord points
temp_chord_points	LINE_FITTING	9.2	Temporary ordered list of chord points
to_do	FOLLOW_TO_DO_LIST	6.1.2.6	List of branch seed curves yet to be followed

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
to_endpoint_offset	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The jump from *before_curve* to *first_curve*
$to\_endpoint\_offset_x$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The x offset of the jump from *before_curve* to *first_curve*
$to\_endpoint\_offset_y$	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	The y offset of the jump from *before_curve* to *first_curve*
topology_mask_set	CONVERT_TO_SINGLE_PIXEL_WIDE_RIDGES	6.1.1.1	Set of masks for non-topology-changing pixel removal
total	DETERMINE_WORD_SIZES	11.4.2	The total number of elements in the histogram
$total_{SW}$	DETERMINE_WORD_SIZES	11.4.2	The total number of elements to be represented with a short word
$total_{zero}$	DETERMINE_WORD_SIZES	11.4.2	The number of elements in the histogram equal to zero

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
unsorted_list	CURVE_SORTING	10.1	List of curves representing the fingerprint that has not been processed by sorting
unsorted_list	CYCLIC_PROCESSING	10.1.2	List of curves representing the fingerprint that has not been placed by sorting
unsorted_list	RESULTS_CHECKING	10.1.1.2	List of curves representing the fingerprint that has not been placed by sorting
unsorted_list	RESULTS_CHECKING_AND_INSERTION_OF_UNSORTED_CURVE	10.1.2.3	List of curves representing the fingerprint that has not been placed by sorting
unsorted_list	SEARCH_FOR_THE_BEST-FIT_CURVE	10.1.1.1	List of curves representing the fingerprint that has not been placed by sorting

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
unsorted_list	SELECTIVE_PROCESSING	10.1.1	List of curves representing the fingerprint that has not been placed by sorting
value	APPLY_SIGN_TO_VALUE	12.2	Value being tested
value	DECODE_SIGN_FOR_VALUE	12.2	Value being tested
value	INPUT_STREAM	12.2	Value being read from input stream
value	IS_SMALL	10.1.2.2	An offset value that is being compared to S
value	OUTPUT_STREAM	11.4.3	Value being written to output stream
value	SIGN	11.4.2	Value being tested
vertical_run	CREASE_TRIMMING	3.1.1	The longest run of consecutive white pixels for every column
w	AVERAGE_SECTION_RIDGE_WIDTH	4.1.3	Ridge width at current pixel
w	SMALL_RIDGE_CONNECTION_REMOVAL	7.1.5	Average ridge width of the neighboring reference sections
w_a	LARGE_PORE_TEST	4.1.2.3	Average ridge width in a neighborhood of a pixel

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
w_a	REMOVE_LARGE_PORES	4.1.2.2	Average ridge width in a neighborhood of a pixel
w_a	REMOVE_SMALL_PORES	4.1.1	Average ridge width in a neighborhood of a pixel
w_{ccw}	LARGE_PORE_TEST	4.1.2.3	Width of ridge in counterclockwise direction from large pore candidate
$w_{current}$	RIDGE_SMOOTHING	8.2	Current size of the smoothing window
w_{cw}	LARGE_PORE_TEST	4.1.2.3	Width of ridge in clockwise direction from large pore candidate
w_p	LARGE_PORE_TEST	4.1.2.3	Width of large pore candidate
w_r	LARGE_PORE_TEST	4.1.2.3	Width of ridge to side of large pore candidate
width	EXTRACT_CURVES	6.1.2	Width of fingerprint image
width	PREPARE_AVERAGE_NEIGHBORHOOD_RIDGE_WIDTHS	4.1.3	Width of fingerprint image

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
width	REMOVE_LARGE_PORES	4.1.2.2	Width of fingerprint image
width	REMOVE_SMALL_PORES	4.1.1	Width of fingerprint image
$width_C$	DETECT_LOCAL_MAXIMA	5.1.2	Width of chamfer image ***C***
$width_I$	CHAMFER	5.1.1	Width of image ***I***
word_size	DECODE_USING_WORD_SIZES	12.2	One word size in *word_sizes*
word_size	DECODE_WORD_SIZES	12.2	One word size in *word_sizes*
word_size	ENCODE_USING_WORD_SIZES	11.4.3	One word size in *word_sizes*
word_sizes	DECODE_USING_WORD_SIZES	12.2	The calculated number of word sizes allowable
word_sizes	DECODE_WORD_SIZES	12.2	The calculated number of word sizes allowable
word_sizes	ENCODE_USING_WORD_SIZES	11.4.3	The calculated number of word sizes allowable
word_sizes	ENCODE_WORD_SIZES	11.4.3	The calculated number of word sizes allowable
$word\_sizes_{deltax}$	DECODE_CURVE_DELTAS	12.2	The calculated word sizes for the $delta_x$

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$word\_sizes_{deltax}$	DECODE_HEADERS	12.2	The calculated word sizes for the $delta_x$
$word\_sizes_{deltax}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The calculated word sizes for the $delta_x$
$word\_sizes_{deltax}$	ENCODE_CURVE_DELTAS	11.4.3	The calculated word sizes for the $delta_x$
$word\_sizes_{deltax}$	ENCODE_HEADERS	11.4.3	The calculated word sizes for the $delta_x$
$word\_sizes_{deltay}$	DECODE_CURVE_DELTAS	12.2	The calculated word sizes for the $delta_y$
$word\_sizes_{deltay}$	DECODE_HEADER	12.2	The calculated word sizes for the $delta_y$
$word\_sizes_{deltay}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The calculated word sizes for the $delta_y$
$word\_sizes_{deltay}$	ENCODE_CURVE_DELTAS	11.4.3	The calculated word sizes for the $delta_y$
$word\_sizes_{deltay}$	ENCODE_HEADER	11.4.3	The calculated word sizes for the $delta_y$
$word\_sizes_{jumpx}$	DECODE_HEADER	12.2	The calculated word sizes for the $jump_x$

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$word\_sizes_{jumpx}$	DECODE_JUMP	12.2	The calculated word sizes for the $jump_x$
$word\_sizes_{jumpx}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The calculated word sizes for the $jump_x$
$word\_sizes_{jumpx}$	ENCODE_HEADER	11.4.3	The calculated word sizes for the $jump_x$
$word\_sizes_{jumpx}$	ENCODE_JUMP	11.4.3	The calculated word sizes for the $jump_x$
$word\_sizes_{jumpy}$	DECODE_HEADER	12.2	The calculated word sizes for the $jump_y$
$word\_sizes_{jumpy}$	DECODE_JUMP	12.2	The calculated word sizes for the $jump_y$
$word\_sizes_{jumpy}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The calculated word sizes for the $jump_y$
$word\_sizes_{jumpy}$	ENCODE_HEADER	11.4.3	The calculated word sizes for the $jump_y$
$word\_sizes_{jumpy}$	ENCODE_JUMP	11.4.3	The calculated word sizes for the $jump_y$

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
$word\_sizes_{num\_deltas}$	DECODE_CURVE_DELTAS	12.2	The calculated word sizes for the number of deltas per curve
$word\_sizes_{num\_deltas}$	DECODE_HEADER	12.2	The calculated word sizes for the number of deltas per curve
$word\_sizes_{num\_deltas}$	DETERMINE_FINGERPRINT_DATA_PROPERTIES	11.4.2	The calculated word sizes for the number of deltas per curve
$word\_sizes_{num\_deltas}$	ENCODE_CURVE_DELTAS	11.4.3	The calculated word sizes for the number of deltas per curve
$word\_sizes_{num\_deltas}$	ENCODE_HEADER	11.4.3	The calculated word sizes for the number of deltas per curve
x	A	13.2	Array of x coordinates in current curve
x	B	13.2	Array of x coordinates in current curve
x	B-SPLINE	13.2	Array of x coordinates in current curve

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
x	C	13.2	Array of x coordinates in current curve
x	CALCULATE_CHORD_POINTS	9.2	An array which holds *x* coordinate information for *curve*
x	D	13.2	Array of x coordinates in current curve
x	LINE_FITTING	9.2	An array which holds *x* coordinate information for *curve*
x_coordinate	B-SPLINE	13.2	Calculated B-Spline x coordinate
y	B-SPLINE	13.2	Array of y coordinates in current curve
y	CALCULATE_CHORD_POINTS	9.2	An array which holds *y* coordinate information for *curve*
y	LINE_FITTING	9.2	An array which holds *y* coordinate information for *curve*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
y_coordinate	B-SPLINE	13.2	Calculated B-Spline y coordinate
z	BAD_BLOCK_BLANKING	C.2	Index
$Z\mu_I$	BHO_BINARIZATION	A.1.3	Image overall mean pixel value
$Zmax_I$	BHO_BINARIZATION	A.1.3	Maximum pixel value over entire image
z_average_chamfer_value	PROCESS_RIDGE_ENDING	B.1	Average chamfer value for last half of reference section
z_blockmap	BAD_BLOCK_BLANKING	C.2	Ridge direction map
z_blockmap	BHO_BINARIZATION	A.3	Ridge direction map
z_blockmap	RIDGE_CLEANING	7.1	Ridge direction map
z_blockmap	CURVED_RIDGE_ENDING_REMOVAL	B.1	Ridge direction map
z_blockmap	PROCESS_RIDGE_ENDING	B.1	Ridge direction map
z_blockmap	RIDGE_THINNING	5.1	Ridge direction map
z_blockmap	WRITE_BLOCK_FILE	A.2.2	Ridge direction map
z_curve	BAD_BLOCK_BLANKING	C.2	One curve in the *curve_list*

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
z_curve	PROCESS_RIDGE_ENDING	B.1	One curve in the *curve_list*
z_distanceAB	PROCESS_RIDGE_ENDING	B.1	Euclidean distance between z_point_A and z_point_B
z_distanceBC	PROCESS_RIDGE_ENDING	B.1	Euclidean distance between z_point_B and z_point_C
z_endpoint	PROCESS_RIDGE_ENDING	B.1	One endpoint of the curve being processed
z_first_point	BAD_BLOCK_BLANKING	C.2	Index of curve point
z_local_ridge_width	SMALL_OFFSHOOT_CURVE_REMOVAL	7.1.3	Average of the local average ridge widths at the unconnected endpoint and at the midpoint of *curve*
z_local_ridge_width	SMALL_RIDGE_SEGMENT_REMOVAL	7.1.6	Average of the local average ridge widths at the endpoints of *curve*
z_new_curve	BAD_BLOCK_BLANKING	C.2	New curve structure
z_new_curve_list	BAD_BLOCK_BLANKING	C.2	Temporary list of new curves
z_not_done	PROCESS_RIDGE_ENDING	B.1	Flag indicating loop status

Table F-4. List of Variables (continued)

Variable	Function	Section	Description
z_number_points_removed	PROCESS_RIDGE_ENDING	B.1	Counter of current number of points marked for removal
z_num_points	BAD_BLOCK_BLANKING	C.2	Number of points in a curve
Z_POINT	BAD_BLOCK_BLANKING	C.2	Temporary array of points in a curve
z_point_A	PROCESS_RIDGE_ENDING	B.1	First point of reference section
z_point_B	PROCESS_RIDGE_ENDING	B.1	Midpoint of reference section
z_point_C	PROCESS_RIDGE_ENDING	B.1	Last point of reference section
z_sum	PROCESS_RIDGE_ENDING	B.1	Sum of the chamfer values for last half of reference section
zz	FIND_BEST_PARTITION	D.2	Index
$zz_{best\_remainder}$	FIND_BEST_PARTITION	D.2	Size of best-case remainder section, in pixels
$zz_{best\_section\_size}$	FIND_BEST_PARTITION	D.2	Size of best-case section, in pixels
zz_{image_size}	FIND_BEST_PARTITION	D.2	Image width or height
$zz_{remainder}$	FIND_BEST_PARTITION	D.2	Size of remainder section, in pixels
$zz_{section\_size}$	FIND_BEST_PARTITION	D.2	Size of section, in pixels

Table F-4. List of Variables (Concluded)

Variable	Function	Section	Description
zz_top	BHO_BINARIZATION	A.1.3	Absolute upper threshold