VLSI Computational Structures Applied to Fingerprint Image Analysis

Thesis by

Barry Bruce Megdal

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1983

(Submitted February 8, 1983)

Acknowledgements

It is impossible in so brief a space to adequately thank all those who have made this work possible, but the following deserve special mention:

My advisor Carver Mead, for suggesting the area of research, for his numerous helpful criticisms and suggestions, and for his insight and patience.

Jim Kajiya and Lennart Johnsson, whose office doors were always open and who were the source of many good ideas.

Steve Klein, for putting up with seemingly endless discussions of this research, and at the same time providing important creative contributions.

Caltech, Hughes Aircraft Company, Office of Naval Research, and DARPA for providing financial support.

My parents, for providing me with excellent guidance and advice throughout my life, as well as for their unshaking faith in me. There are none better.

My wife, for putting up with the late nights and long hours, and yet still taking the time to provide useful contributions to my research. Without her support and help this thesis would never have existed. Thanks Myrna.

Abstract

Advances in integrated circuit technology have made possible the application of LSI and VLSI techniques to a wide range of computational problems. Image processing is one of the areas that stands to benefit most from these techniques. This thesis presents an architecture suitable for VLSI implementations which enables a wide range of image processing operations to be done in a real-time, pipelined fashion. These operations include filtering, thresholding, thinning and feature extraction.

The particular class of images chosen for study are fingerprints. There exists a long history of fingerprint classification and comparison techniques used by humans, but previous attempts at automation have met with little success. This thesis makes use of VLSI image processing operations to create a graph structure representation (minutia graph) of the inter-relationships of various low-level features of fingerprint images. An approach is then presented which allows derivation of a metric for the similarity of these graphs and of the fingerprints which they represent. An efficient algorithm for derivation of maximal common subgraphs of two minutia graphs serves as the basis for computation of this metric, and is itself based upon a specialized clique-finding algorithm. Results of cross comparison of fingerprints from multiple individuals are presented.

Table of Contents

Chapter 1

Background and Motivation

1.1. Use of VLSI in Image Processing

Image processing algorithms have traditionally been implemented on sequential, general purpose computers. Such machines are not well suited to dealing with large arrays of data, particularly where a given sequence of operations must be repeated many times for each data item. It is not unreasonable to consider dealing with images containing over 10^7 total bits (1000x1000 image with 12 bits per pixel), yet the explicit iterations required to implement even the most trivial transformations on such an image result in very long execution times. Though the fingerprint images dealt with in this work are digitized only to a resolution of 400x400 with 8 bits per pixel, the resulting 1.2 million bits suffice to make the problem evident.

It has been clear for many years that some form of parallel computation is necessary in order to achieve reasonable execution times, particularly in real-time interactive applications such as robotics. For example, a computer controlled manipulator arm equipped with a television camera will be of little use if many minutes are required to process each image frame received from the camera. The use of parallel processors for image transformation is particularly appropriate in such a context, since the human visual system makes use of similar computational techniques

when extracting information from incoming images.

The majority of the special purpose image processing computational structures considered in the literature are of what may be called the "large array" variety. By this it is meant that an attempt is made to produce an array of computational units which is large enough that one unit exists for each pixel in the image. Each such unit is typically connected to various of its neighbors, in order to provide for the communication necessary for operations such as pattern matching, and to allow the input and output of the image itself. Even through the use of modern VLSI techniques it is not possible to provide a computational unit for each pixel in the image, and therefore some means for performing operations on sub-images is required.

A typical instance of such a system, known as CLIP, was designed by Duff at University College, London [Wong79]. The most recent version, currently implemented in NMOS LSI and known as CLIP4, consists of small, special purpose processors (eight per chip) that can be configured in either a rectangular or hexagonal array as desired. Each processor is connected to its immediate neighbors and executes instructions that have been downloaded from a master controller. The operations that can be performed by each processor consist of simple Boolean functions, bit propagation, and single-bit arithmetic. As with all systems of this type, performance tends to be limited both by the need to subdivide the original image into manageable parts, and the overhead encountered in loading the data into the array and reading out the result -- often in a bit serial fashion.

A somewhat different approach is taken in the PICAP system, implemented by Bjorn Kruse of Linkoeping University [Kruse]. PICAP consists of a minicomputer, a special purpose computational unit, and

image I/O devices all resident on a common bus. The computational unit receives commands from the minicomputer over the bus, which specify the operations it is to perform on the image stored within it. The operations are performed on 3x3 neighborhoods in the image, which are extracted through the use of three single image-line buffers, in a manner similar to that used by the "cytocomputers" described below. Fundamental to the functioning of this unit is the concept of a "template match". Particular values (up to 4 bits per pixel, or "don't care") can be specified for each location in the 3x3 template. This template is then in effect scanned over the image, and a specified transformation is applied to the center pixel of the current 3x3 window only if the template matches the actual pixel values in the neighborhood. Along with the definition of the template to be used is included information describing what, if any, rotations of that template are also to be applied. Use of rotations of the template does not require additional passes over the image. Many very powerful image processing functions can be performed based upon such template match primitives, at a rate of about one operation every 2.5 ms for the current PICAP system when operating on 64x64 images. The thinning algorithm described later in this work is based upon a template-matching approach similar to that of PICAP.

The PICAP architecture points the way toward another form of parallel execution image processor known as the "cytocomputer" [Lougheed80]. First proposed in 1976 [Sternberg76], a cytocomputer consists of a serial pipeline of special-purpose processors, each performing a particular operation on small neighborhoods as the image moves by. Pictures "flow" through the pipeline in serial fashion at a constant rate, while shift registers in each of the processors hold three scan lines worth of pixels. As a result,

the values of the pixels in a 3x3 neighborhood are always available to the "neighborhood logic module", which is responsible for applying the appropriate transformation to the neighborhood pixels (Figure 1.1). The constant flow of the image pixels through the pipeline in effect simulates the motion of the 3x3 processing window over the input image, in raster scan fashion.

Cytocomputers have several advantages over large parallel arrays of processors, including low complexity, high bandwidth, and considerable architectural flexibility [Lougheed80]. The chaining of large numbers of physically identical, low cost modules is an ideal solution to the processing of serial data, such as is encountered with raster-scanned images. In addition, such processors are ideal candidates for VLSI implementations, as their inherent serial nature eliminates many problems due to limited pin counts on VLSI packages. Note that this is in direct contrast to the situation for the large-array image processing structures, where pin count may well be an important factor limiting the amount of processing that can be integrated onto one chip.

All of the image processing operations used in this work are implemented on generalized cytocomputer processors, which we call "neighborhood processors". In order to accommodate some of the complex processing necessary for the analysis of fingerprint ridge patterns, many extensions of the basic cytocomputer structures were made.

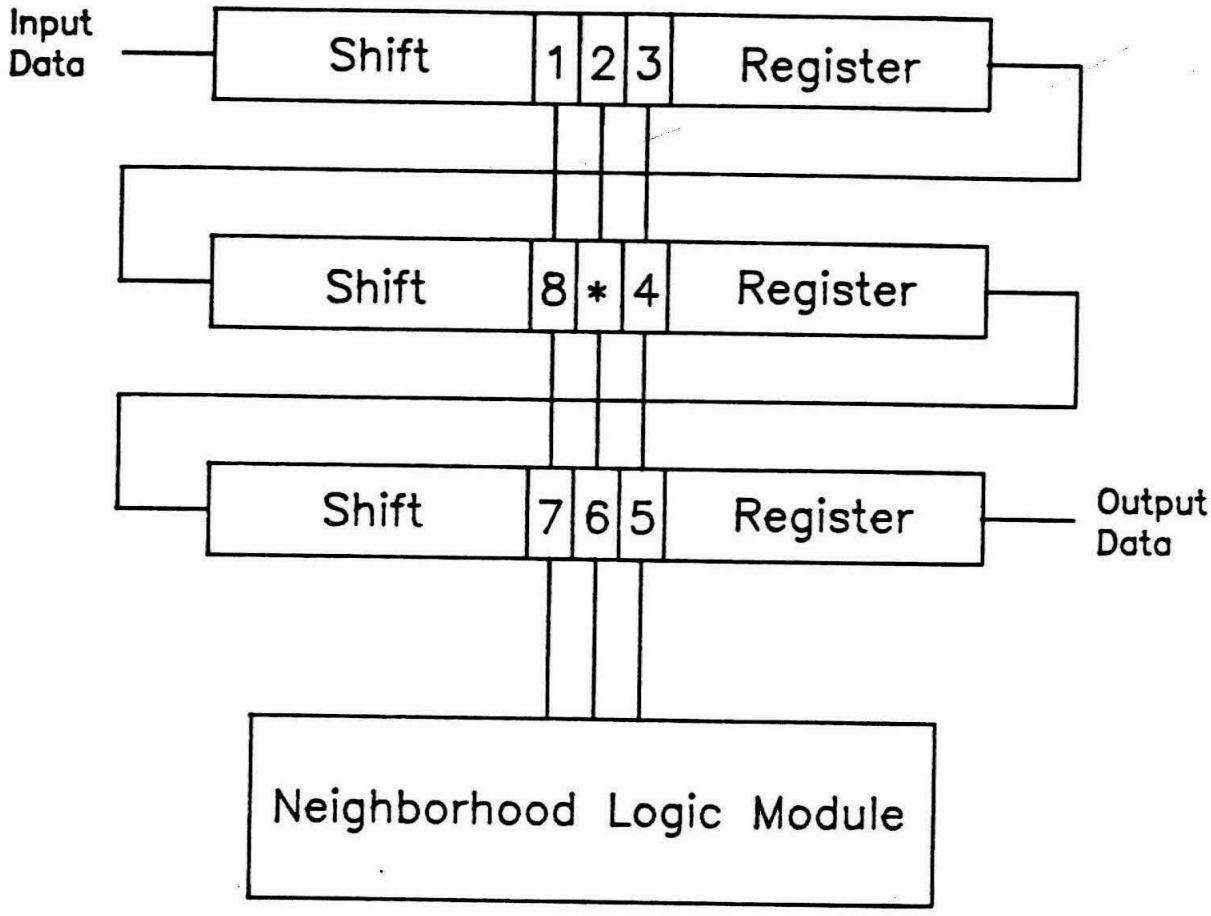


Figure 1.1: Cytocomputer Module

Images are not constrained to be binary (i.e., one bit per pixel), and a bit-slice approach is described which enables straightforward expansion of the "word size" at any point in the pipeline. This can be used to store grey-level information, or to contain labels for features identified in the images. Each pixel in the window is considered to be mapped onto one member of an array of generalized processors, each with the ability to communicate with its neighbors. In effect the cytocomputer structure has been combined with a small rectangular processor array -- an interesting blend of the two image processing paradigms previously discussed.

1.2. Why Fingerprints?

Fingerprint images, considered as a class, are an attractive choice as a test case for the development of VLSI image processing techniques. Though far from trivial in their information content, they are still considerably simpler than an average outdoor scene, with its multitude of complex overlapping objects, and are thus more appropriate for use during initial research into VLSI image processing structures.

In addition, fingerprints and analysis of their properties has been an area of great historical interest, dating back to early cave drawings of finger ridge patterns, though until the recent advent of experimental automated systems, all fingerprint analysis has of course been done by human beings. The need for automated systems becomes obvious when one realizes that the FBI alone is called upon to classify and file over six thousand sets of fingerprint cards each day, and currently has a backlog of over 400,000 unfiled cards.

The possible applications of a reliable automated fingerprint matching and analysis system extend far beyond the classical usage of ten-finger print cards for identification of criminals. One can imagine fingerprints used for control of entry to secure areas, banking identification, and even as a control on the fraudulent use of credit cards. All such applications require a system that is capable of quickly and efficiently verifying identity based upon a single fingerprint image, with very little chance of an incorrect identification.

Previous work in the area of automated fingerprint analysis did not make use of the processing power available through VLSI, and has met with little success. Details of several such attempts are presented in Section 2.1. Much can be learned from the fingerprint analysis methods employed by human beings and, in fact, the computational approach taken in this work has many similarities to those methods.

1.3. Non-automated Fingerprint Matching

Upon looking at a typical fingerprint (Figure 1.2), it is seen that the print consists basically of a series of segments known as ridges, some curved and some fairly straight. These ridges are of widely varying length, quite uniform width, and are packed tightly together over most of the print. The ridges are raised areas on the surface of the friction skin of the hands and feet. This skin is so called because the corrugated surface is effective in improving the ability to grasp objects with minimal slippage.

Figure 1.2: Typical Fingerprint

As can be seen in a good quality fingerprint image, the friction ridges are dotted with small circular openings -- the sweat pores. The perspiration exuded from these pores remains on a surface after it has been touched, forming the latent fingerprints often searched for at crime scenes. Fingerprints form definite patterns, and it is a fundamental premise of fingerprint identification that the friction ridge patterns of an individual do not vary throughout his or her life. In fact, the fingerprint pattern is determined during the third or fourth month of prenatal development, at which time special skin layers known as "primary ridges" begin to form beneath the epidermis. These strips grow and eventually develop into the visible ridge structure. Their pattern of growth is determined by a series of irregularly distributed "pegs" known as dermal papillae, which form at the junction of the epidermis and the dermis [Moenssens71]. In addition, since the dermal papillae are buried quite far beneath the skin surface, any damage to the epidermis will result in the ridge pattern growing back to its original configuration. Only if the injury reaches deep enough to destroy the papillae and interfere with the proper growth of new epidermal cells will a permanent scar form. There is a long history of attempts by criminals to modify their fingerprint patterns by some form of mutilation, such as cutting, sanding, burning, acids, and surgery. Most such attempts have been unsuccessful, as sufficient unmodified ridge structure usually remains to allow identification of the individual. Even destruction of the key core and delta areas used in most classification schemes does not prevent identification because, as will be explained below, single print identification is based upon quite different principles than those used for classification.

The classification system now in widespread use by the FBI and most other law enforcement agencies is the "Henry" system, which has been

modified and extended to allow the FBI to maintain on file a collection of over 200,000,000 sets of prints. This system, which can only produce a classification if a full ten-finger print set is available, makes use of general topological characteristics of the ridge patterns. Some of the major types of patterns are shown in Figure 1.3. The complete classification for a print set consists of a series of letters and numerals, consisting of sections known as primary classification, secondary, subsecondary, major division, final classification, and key. The Henry system is rather ad hoc, having been derived empirically over a period of many decades. For example, in determining the primary classification, even-numbered fingers are considered together, as are the odd-numbered fingers. For the secondary classification level, however, the fingers are grouped together by hand. The major division classification is based solely on the pattern on the thumbs, while the key classification is based upon the little fingers.

Fingerprint cards are filed by classification formula, beginning with the primary. The primary classification yields 1024 possible combinations, but these are not at all evenly distributed. The secondary classification, based mainly upon index finger patterns, is used to further break down the categories. The ordering process is continued through the final and key sequences to obtain greater resolution. With all of this however, there will still be a substantial number of prints with identical classifications if the set being searched is large. In fact, when attempting a match of a print-set with the FBI database, it is typical for the classification formula to isolate the necessary search only down to one or more file cabinets of fingerprint cards, each of which must then be carefully compared to the prints in question by a human expert print matcher.

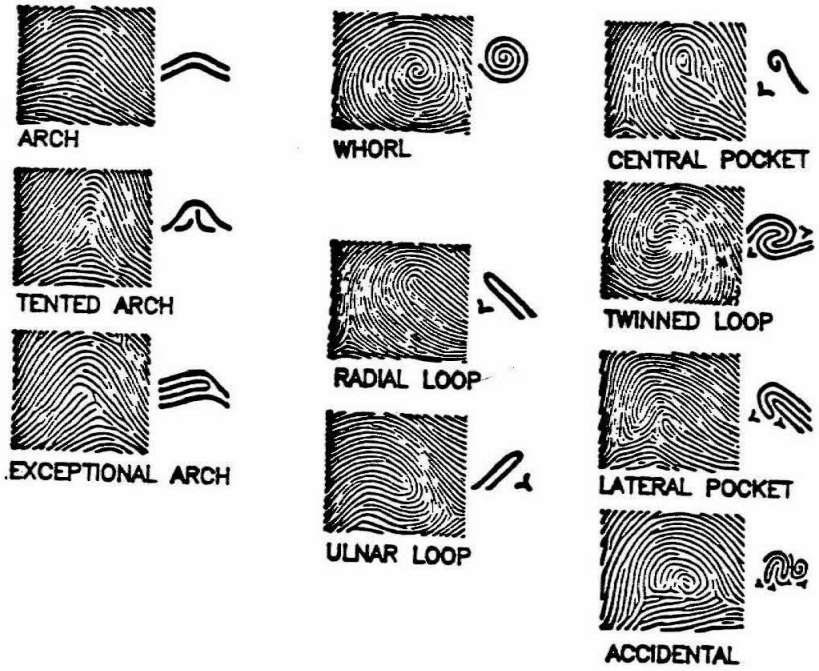


Figure 1.3: Ridge Patterns
[Hankley]

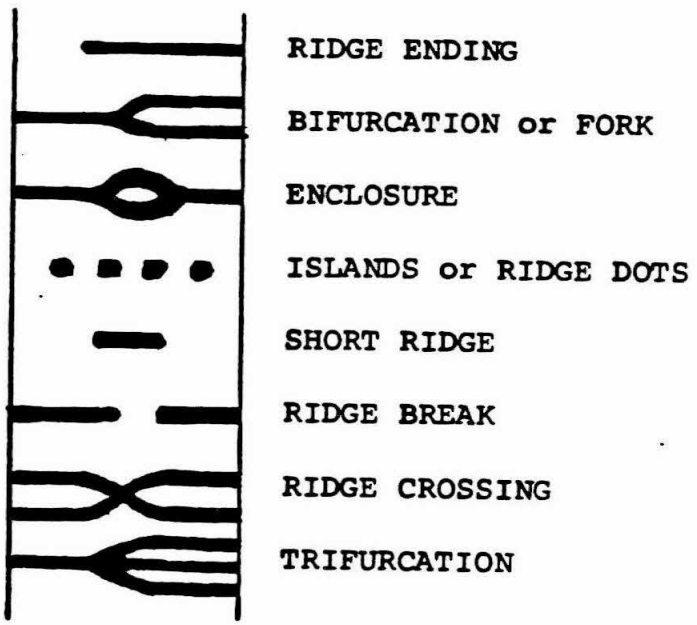


Figure 1.4: Minutiae

Note that, in addition to making use of human judgement to describe complex topological characteristics of the ridge patterns, the Henry system is also less than ideal in that it absolutely requires a complete ten-finger print set to produce a classification formula. Thus it is essentially useless for identification of a single latent print found at the scene of a crime, or for other personal identification applications. It is commonly but incorrectly thought that a criminal can be tracked down based upon a carelessly left fingerprint at a crime scene. A single latent print can only be of use if the set of possible people from which it may have come is limited by other means to a small subset of the FBI's collection (at most several hundred cards). Classification schemes for single fingerprints do exist, but have met with only limited use and success.

As was mentioned, the system used for comparison of individual fingerprints is quite different than the classification system just described. It relies not upon global topology, but upon quantitative and qualitative comparisons of the low-level ridge characteristics, known as minutiae. Minutiae are composed primarily of ridge endings, forks (bifurcations), enclosures, islands or ridge dots, and short ridges. Also sometimes included in the definition [Moenssens71] are ridge breaks, ridge crossings, and trifurcations (Figure 1.4).

A fingerprint image as prepared for exhibition in court is shown in Figure 1.5, with lines drawn to some of the minutiae. When two prints are being compared, it is of course first necessary that they be of the same pattern type. This is not to say that an entire latent print is necessary, but rather that if the pattern type information is available for both prints (i.e. they are complete), then the pattern types must not differ.

[Numbers indicate minutiae]

Figure 1.5: Print Prepared as Exhibit for Presentation in Court [Trauring61]

The key to the comparison process is the type and position of the minutiae. Sufficient minutiae must be the same in both prints, in that they are of the same type, and face in the same direction. In addition, the number of intervening ridges between any two minutiae in one print must be the same as the corresponding number of ridges in the other print. Thus if a fork and a ridge-end are found to be separated by three ridges in one print, and two in another, the minutiae pairs certainly do not match.

Given that it is possible to identify sets of matching minutiae in the prints being compared, the question arises as to exactly how many such matching minutiae are necessary in order to state with a high degree of confidence that the prints came from the same finger. It is generally agreed that ten to twelve "points" (as matching minutiae are called) are sufficient to establish identity. Yet such a guideline is only an approximation, as the rate of occurrence of the various forms of minutiae is not the same. Ridge-endings and bifurcations are the most frequently occurring minutiae [Moenssens71], while islands, enclosures, and trifurcations are rather rare. Thus fewer than ten points can and have been used to establish identity in criminal cases, particularly when the properties of the minutiae involved were in some way unusual. Since the typical fingerprint contains from 50 to 150 identifiable minutiae, the abundance of information available for comparison is encouraging.

Thus we see that fingerprints as they have been employed in the past are of very limited use as a means of identification. The Henry system is cumbersome, difficult to master, and requires a large number of subjective judgments on the part of the person doing the actual classification. The steps in the classification procedure itself are far from logical, and not well

ordered. The frequency of occurrence of each of the possible classifications is not at all uniform, therefore usually requiring tedious manual comparison of hundreds of print sets. In addition, only full ten-finger print sets can be classified in this manner, severely limiting the usefulness of the system as a tool for criminal investigation. Most importantly, the minutiae matching method used for the ultimate comparison of the fingerprint images is totally unrelated to the classification system [Foote74].

The Henry system by its very nature does not lend itself to automation -- nor would one want to use a system which requires complete ten-fingerprint sets as the basis for an automated classification and identification system. Rather, a system more closely related to the minutiae-based comparison approach just described is more desirable, and indeed forms the basis of the work in this thesis.

Chapter 2

Automated Fingerprint Analysis

2.1. Previous Work

Many attempts have been made to automate the fingerprint matching and recognition process. Particularly since the advent of modern digital computers, there has been a widespread feeling that such automation could and should be done. Several previous approaches, none of which have been particularly successful, are described below.

Perhaps the first significant work in this area was done by J.H. Wegstein of the National Bureau of Standards Center for Computer Sciences and Technology. His system was based upon computer matching of clusters or "constellations" of fingerprint minutiae (ridge-ends or forks), 50 to 100 of which exist in a typical print [Wegstein68a]. The positions of the minutiae in the fingerprint image were determined by a human operator and then entered into the program. Also determined were the coordinates of each minutiae point, and the "direction of flow" of the ridge away from that point, as seen in Figure 2.1. The next step was the generation of "constellations", which are groups of minutiae both physically close to one another and having similar flow angles. Each minutia may belong to no more than one constellation, and is eventually considered as a potential focal point for the growth of such a constellation. Each constellation is then

subjected to a coordinate transformation, which moves the origin of the coordinate system to the center of mass of the constellation. The angle of each minutia is then used to classify it as "upward-pointing" or "downward-pointing". The integer coordinates of the minutiae on the new, center-of-mass based coordinate system, along with the bits indicating their direction, form the total descriptor for the constellation. These descriptors, which contain relatively few bits, can then be filed for later comparison with other encoded prints.

In later work [Wegstein69], a somewhat different encoding approach and matching procedure is described. Specifically, each minutia point is labeled with an X, Y, and angle value. The minutiae from the two prints to be compared are then sorted by angle. The comparison process consists of building constellations by selecting candidate pairs of minutiae (one from each print), and verifying that their angles and relative displacements agree within a predefined tolerance. If so, they form the basis for growing a constellation of similar minutiae. If not, new pairings are chosen, and the process continues.

Further refinements of these matching procedures [Wegstein70] enabled a file of encoded fingerprints to be searched using statistical string-matching procedures. Several problems were encountered however, including sensitivity to stretched, twisted, or misaligned prints. As a result, the system was not truly practical. The concept of making use of local groups of features is a sound one however, and is adhered to by one of the encoding systems described later in this work.

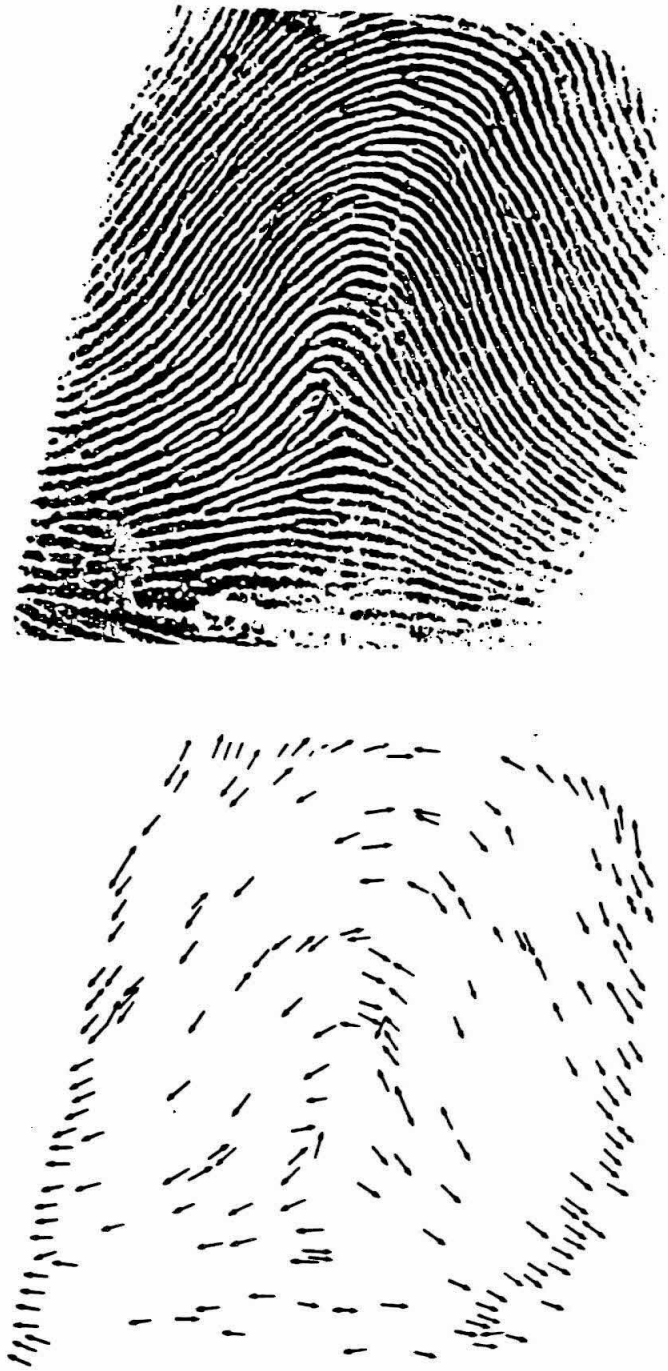


Figure 2.1: Fingerprint With Manually Determined Minutiae Coordinates and Flow Directions. [Foote]

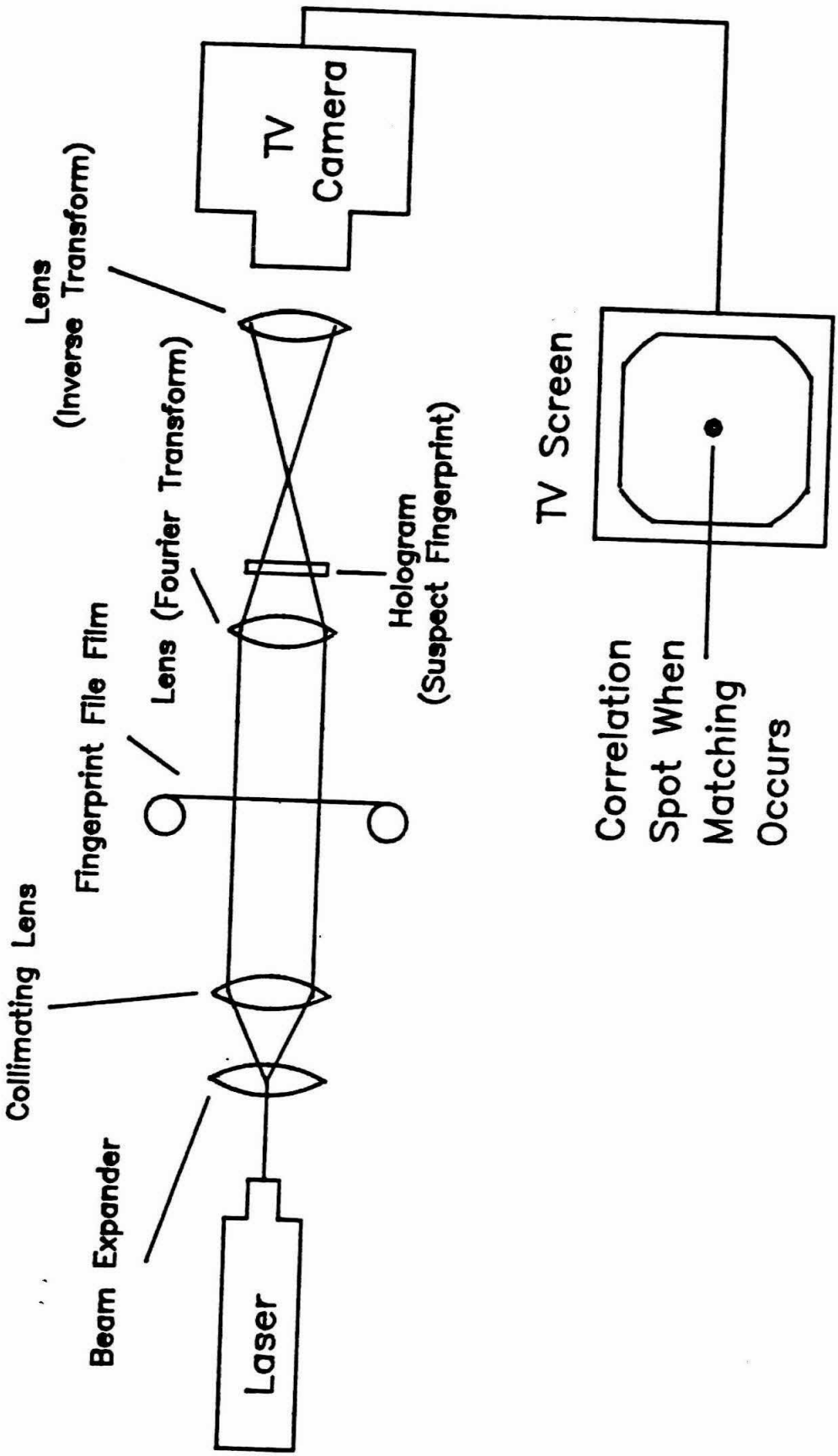


Figure 2.2: Optical Correlation

Holographic and other optical techniques for the comparison of fingerprint images have been used by several investigators. In [Hughes67] it is proposed that coherent optical correlation form the basis for the matching procedure. It is assumed that both the fingerprint image to be identified and the file of candidate matching images are available as photographic transparencies. The unidentified pattern is used as a spatial filter through which the laser light is passed, after first passing through a series of lenses (Figure 2.2). When another pattern is introduced into the light path, a bright central spot will be produced if the two images are identical. Variations in the similarity of the two images result in a pattern that bears less and less resemblance to a small spot as the difference between the prints increases. Systems based on this technique have been proposed by, among others, Hughes, McDonnell Douglas, and Computer Corporation of America.

Optical correlation has serious limitations. Perhaps most importantly, it is required that the information about each filed fingerprint be in the form of a photographic transparency of the entire print, which is far from being a compact form of storage. Contrast this with the system described later in this work, which encodes all the information needed from a single fingerprint in a few hundred bits. Even if the data storage problem is ignored, optical correlation has at least two other drawbacks. First, strong correlation peaks are often obtained for particular displacements of the unknown image with respect to the reference image, though no justification for such a correlation exists. Second, a given unknown image will often correlate rather strongly with reference images which, though they may resemble the unknown image, differ from it in significant ways. Both of these problems are indicative of an approach that relies too heavily on

global properties of the images, and has insufficient sensitivity to the exact positions and shapes of the low-level fingerprint features.

Another class of proposed automatic fingerprint classification systems could be called "syntactic description systems". The major work in this area has been done by W.J. Hankley and J.T. Tou [Hankley]. They recommended a topological encoding with the following features:

(1) The proposed scheme is a single-print system, and thus does not rely on the existence of a complete ten-finger print set.

(2) The system learns the print structure, in the sense that interpretation decisions are based upon previous such decisions.

(3) The interpretation program performs contextual filtering to correct for imperfections such as ridge gaps and contiguous ridges.

(4) The coding scheme is topological, and thus is invariant to distortions caused by rolling or stretching of the skin or translation of the print.

The fingerprint image is input optically, and then quantized both spatially and in terms of intensity. The interpretation program extracts properties of the ridge contour pattern, such as finding the core and describing the shape of a few surrounding ridges. As a result of the core detection and ridge tracing, the general class of the pattern can be determined as one of whorl, plain arch, loop-left, loop-right or tented arch. The next pass, known as the "topological coding program", generates a code sentence to describe finer details of the shape of the core and surrounding ridges. The resulting code sequence can then be used for print classification. Unfortunately, the system as described is not capable of classifying prints to a fine enough level to be used alone for absolute print comparison. Rather, it is useful as an automatic top-level classification

system. [Moayer76] presents a related approach to syntactic encoding. Some form of such a system is probably appropriate for use as a pre-processor for the matching process described in this work, as a means of minimizing overall computation time in "production" fingerprint matchers.

The Federal Bureau of Investigation has been actively involved in the development of an automated fingerprint analysis system since 1965 [FBI77]. Due to the large volume of prints with which they deal, any system if it is to be useful, must be capable of extreme refinement in classification. Initial requirements called for development of a machine reader capable of detecting ridge detail (particularly ridge ends and bifurcations) to act as a front-end to the algorithms developed by the National Bureau of Standards and described in section 2.1. The contract was awarded to Cornell Aeronautical Laboratories in 1967, and a prototype reader system was installed in 1972. It is called FINDER, from "FINgerprint reaDER". FINDER is a computer-controlled flying spot scanner based system that is responsible for all processing steps from initial input through identification and enhancement of the extracted minutiae pattern. In 1974 a contract was awarded to the Autonetics Group of Rockwell International for the construction of five advanced versions, to be known as "FINDER II" readers. The FINDER/FINDER II system is designed to accept as input standard 10-finger inked fingerprint cards, which are then optically scanned. The image processing operations that follow are implemented using special-purpose hardware, in an attempt to obtain reasonable processing speed.

As of 1977, work sponsored by the FBI was underway to produce special-purpose hardware implementations of the NBS matching algorithms, in order to complete the implementation of a high-speed automated

classification system. Though the expected completion time was less than two years, no successful implementation yet exists, due partly to some of the limitations inherent in the NBS encoding and matching algorithms. An article appearing in the Los Angeles Times in mid-1981, headlined "FBI Cuts Fingerprint Checks, Citing 400,000-Case Backlog", underscores the problems that would be relieved by a properly functioning automatic processing system. To quote part of that article:

> "Because of a backlog of 400,000 unfiled fingerprint cards, the FBI has informed state agencies it will not make fingerprint checks after Oct. 1 unless they are in reference to criminal cases... One effect of the decision will be that state agencies will no longer be able to have fingerprint checks run on applicants for private security guard positions..."

2.2. Guiding Principles

The process of analyzing a fingerprint image, with the intention of comparing it with a stored representation of another fingerprint image to determine if they are from the same finger, can be approached in many different ways. Several attempts at automating this analysis have already been presented. Though none has been totally successful, careful study of the methods used, and of the problem itself, has resulted in a set of observations about what properties are desirable or even essential in a good fingerprint analysis system. The two systems of encoding fingerprint information described later in this work conform as much as possible to the following principles:

(1) The analysis techniques used must be tolerant of the inevitable noise contamination of the fingerprint images under study. Even with a

high-quality input system, there will always be imperfections in the images obtained (e.g. due to dirt specks on the finger). Also, even the best of input systems cannot prevent changes in the fingerprint image due to actual modification of the ridge structure on the finger, as by scarring. Imperfections in the image can take many other forms, a few of which are illustrated in Figure 2.3. Some of these imperfections are correctable, in that they can be removed by an appropriate pre-processing step. An example of this is the removal of holes in the ridges (i.e. pores), the algorithm for which is described in Section 3.3.3.

Some authors [Hankley] have suggested that small breaks in individual ridges can be removed in a similar manner. Yet direct observation of a sufficient number of fingerprints will show that such breaks sometimes occur naturally, thus calling into question the correctness of their removal. For this reason, no attempt was made to remove breaks in ridges. A related problem appears if an attempt is made to distinguish between ridge-ends and bifurcations. Though they may appear quite different in the original ridge pattern, the removal or addition of a very small number of pixels will transform a bifurcation into a continuous ridge plus a ridge-end, as demonstrated in Figure 2.4. Thus no attempt should be made to distinguish between ridge-ends and bifurcations. Rather, the occurrence of either should simply be considered as a "feature", as is done in this work. This definition is in actuality quite natural, for ridge-ends and bifurcations are really duals of one another -- if one simply considers the white inter-ridge spaces to be ridges, and vice versa, ridge-ends become bifurcations, and bifurcations become ridge-ends.

(2) Maximize the use of local properties of the fingerprint image, rather than global ones. In other words, we would prefer to make statements about the relationship between features (minutiae) which are physically close together on the print, rather than at opposite edges. This is consistent with the practice of human print comparison experts, who tend to match features which are in close proximity, due to the increased probability of having good data integrity in the region between the features. Similar reasons apply to an automated system. Immunity to noise will be much improved if locality is maintained, as destruction of a small region of the print due to noise or scarring will then have a minimal impact on the derived data structure. If locality is not maintained, even very minor changes to small regions can have a profound effect on the result of the encoding. Similarly, immunity to large scale geometric distortions will be improved, as such distortions have less effect when considered over a small region.

(3) Rely as much as possible on descriptions of the topology of the fingerprint, rather than on precise metrics, such as exact position or angle of ridges or minutiae. It is worth noting that the Henry classification system is strongly topological in nature, and though in many ways cumbersome, is the most successful system in use. This is not to suggest that that system or anything similar should be adopted for automated analysis, but rather that a major advantage of the Henry approach should not be ignored. By making use of print topology rather than strict metrics, a great deal of immunity to geometric distortion is gained. Inked fingerprints are particularly susceptible to such distortions, due to differences in the rolling technique used and if possible ink on paper should be avoided as an input mechanism. The

finger-on-prism method used for this work makes geometric distortions due to variables such as finger pressure much less of a problem, but good distortion immunity is nonetheless desirable.

(4) Produce a compact representation of the fingerprint. The original digitized image may contain several million bits of information. Yet it is desired that we be able to store representations of large numbers (millions) of such prints. The representations described below compress the information needed to do the fingerprint matching into at most several hundred bits. This is quite practical for storage of data for many prints. Such a compact representation also aids in the efficient searching of a fingerprint database for a matching print, a procedure of considerable interest to law enforcement agencies, and not now practical with current systems. Though this work was not oriented toward a latent print matching system (but rather toward applications such as identity verification, where only one potential match need be considered), most of the results are applicable to latent print identification. One necessary change would be that no assumptions could now be made regarding the physical orientation of the fingerprint image, and as a result the comparison process would be somewhat slowed.

(5) The system must be capable of rapidly deciding if the fingerprint image presented to it matches the previously encoded and stored representation. It should take no more than 10 to 15 seconds, and preferably less. This speed should easily be obtained by means of the special purpose VLSI computational structures described in this work, and would be impossible if a traditional computer was used.

(6) The system must be accurate. In particular, it must be very unlikely to decide that two fingerprints match when they in reality are from different fingers, as this would be a serious security problem. Somewhat less importantly, the system must not often reject a properly matching print, though occasional instances of such rejection due to defects such as serious scarring are acceptable, and can be dealt with by use of another fingerprint from each individual as a backup.

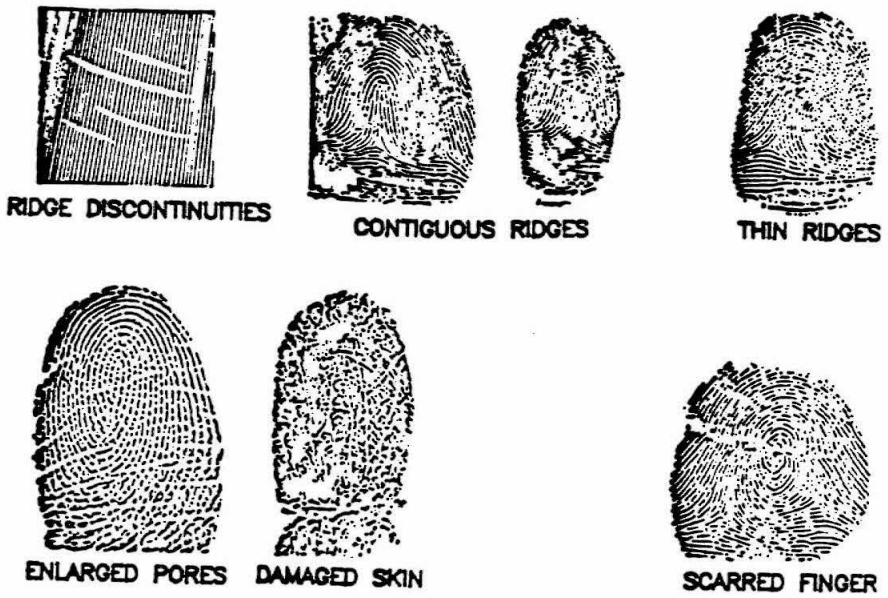


Figure 2.3: Image Imperfections
[Hankley]

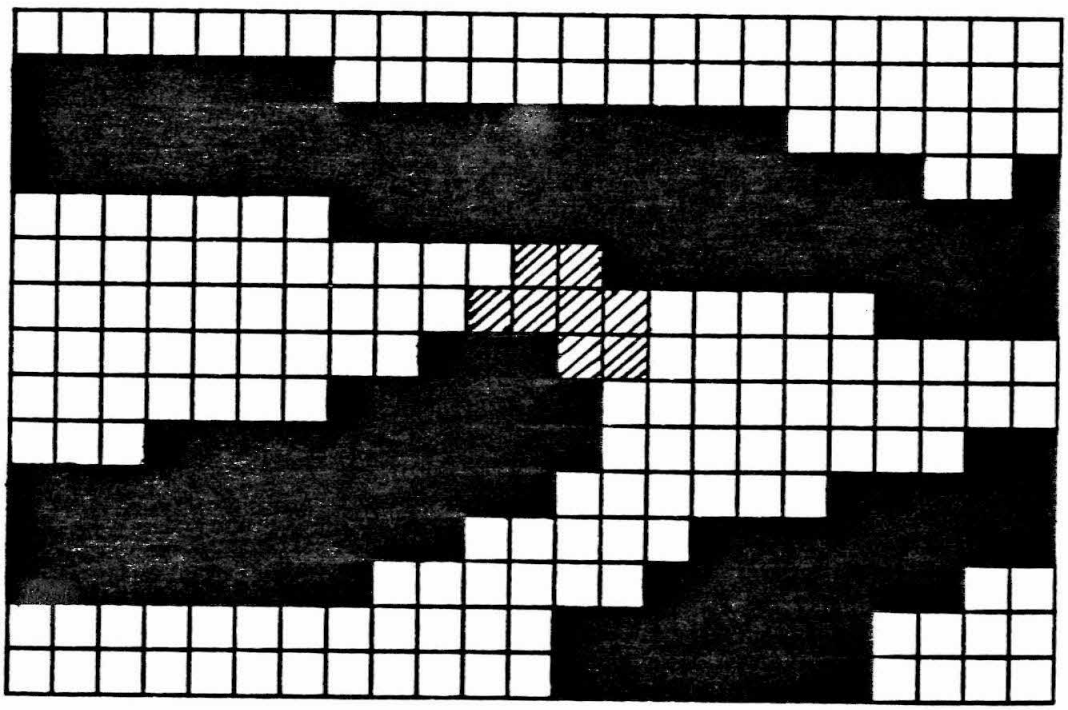


Figure 2.4: Ambiguity in Distinguishing Between Bifurcations and Ridge-Ends

Chapter 3

Image Processing Operations

3.1. Neighborhood Processors

The special-purpose computational elements of the proposed image processing pipeline have in common the ability to access in parallel all of the picture elements within an NxN window into the image. It is necessary that this window be, conceptually at least, scanned across the entire image in raster-scan fashion. This is accomplished not by any motion of the window or its associated processing hardware, but rather by using the shift-register based approach shown in Figure 3.1, where the serialized image is moved through a sequence of N interconnected shift registers, each of length equal to the number of pixels on one scan line of the input image (M). The images considered in this thesis are digitized to a resolution of 400x400, yielding between five and ten pixels across the width of a typical ridge. This has proven to be more than adequate, and results in each of the N shift registers being 400 pixels in length. For the remainder of this work, the notion of "scanning a window over an image" implies serially shifting the bits of the image through the shift registers, so as to simulate a processing window scanning over the image. This architecture is a very natural one when dealing with images that originate with a television camera, due to the inherent raster scan nature of such a medium.

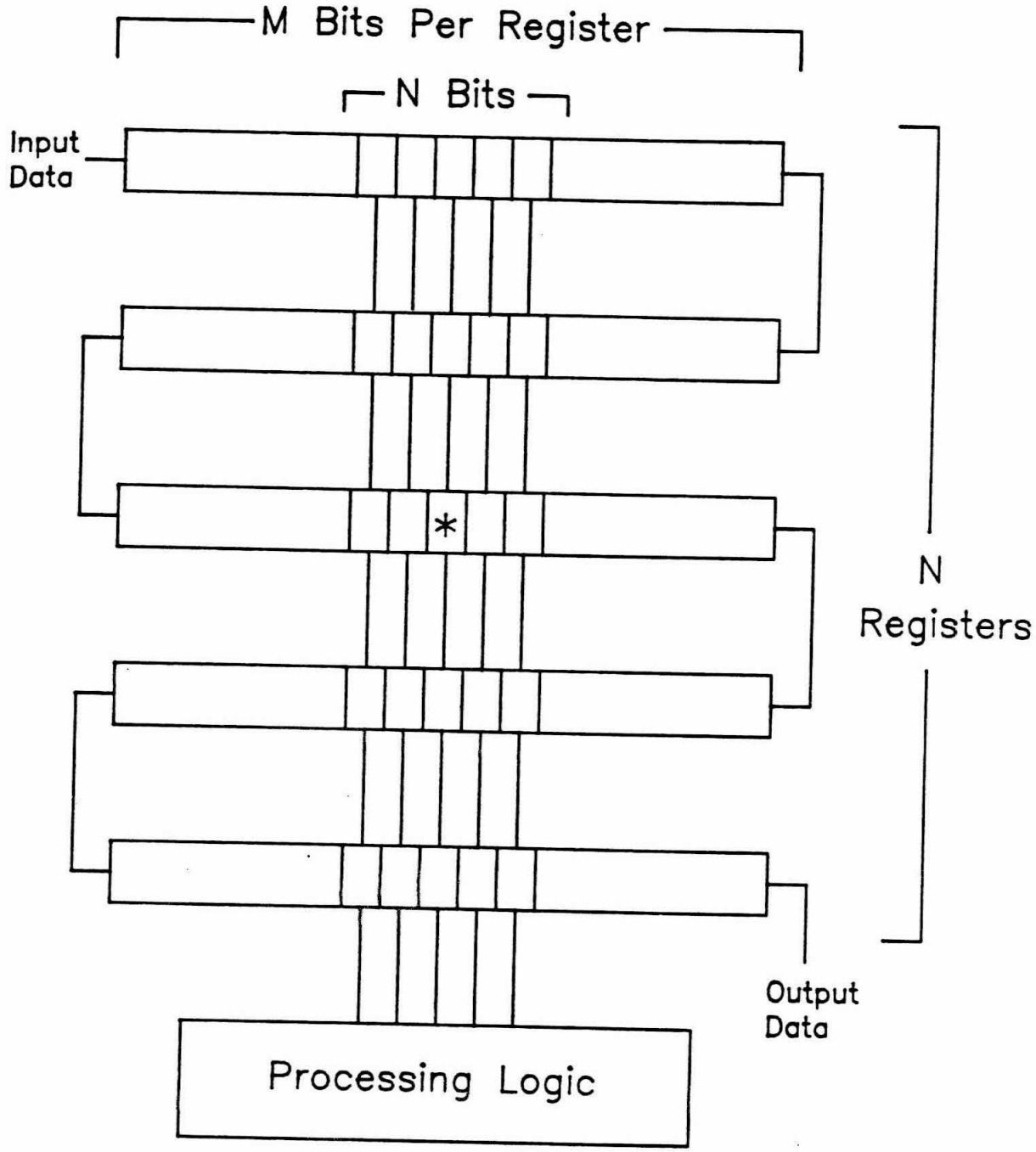


Figure 3.1: Generalized Neighborhood Processor

The choice of the value for N used to implement any particular image processing operation is dictated by the inherent "locality" of the operation itself. In many cases it is sufficient that we have access only to a pixel and its 8 nearest neighbors. This is true for operations such as thinning and ridge numbering, and for these a 3x3 pixel ("Small") window is used. A second class of image processing operations is also used in this work. These include among others, spur removal and ridge adjacency coding, and require that we be able to deal with pixels farther removed from the center of the window than the immediate neighbors. In fact in each of these cases it is necessary that the window size be large enough to span the gap between fingerprint ridges. These operations can typically be implemented with a 15x15 pixel ("Large") window. Thus we see that considerations of over what distance inter-processor communication must occur lead to the observation that only two values of N (3 and 15) are sufficient to implement the wide variety of operations described in this work.

Though these neighborhood processors must each contain MN cells of storage, access is required only to the data in the central N cells of each M-length register, considerably simplifying the design and layout problems involved. For the case of a processor designed to work on single-bit per pixel images, the neighborhood processor becomes essentially a two port device, with serial input and output bit streams. Here, the assumption is that the logic necessary to accomplish the intended operation of the module will be on the same chip, connected to the array of shift-registers by N^2 data wires. Whether or not resident on the same chip as the shift registers, the processing array consists of a single processor for each pixel in the current window, as seen in Figure 3.2.

Since each processor is operating upon a stream of data, the architecture of the neighborhood processors is particularly well suited for connection in a pipeline. The advantage of using a pipeline configuration is that, though many sequential operations may have to be performed on any one image, the pipeline considered as a whole is able to apply all needed operations simultaneously. This is due to the presence within the pipeline of a sequence of images in varying stages of transformation. Also, little or no time is wasted in input/output transfers. This is in marked contrast to the large-array type of image processor often proposed, in which considerable time is spent loading the array with each image frame and removing the result afterwards.

The speed of operation of a pipeline processor is easily quantified. Let us presume that the pipeline contains S stages of neighborhood processors, with window sizes N_1, N_2, ... N_S. The data are shifted through the processors at a rate of one pixel shift per T seconds. The total number of bits of storage in the pipeline is $B=\sum_{i=1}^{S} N_i$. Thus, time BT is taken to fill the pipeline (the "latency" of the pipeline). After the pipeline is full, data emerge from the pipeline at a rate of one pixel per T. The images we are processing each have M^2 pixels, resulting in a time to process one image of M^2T. Even if only one image is to be processed, the latency time is likely to be insignificantly small for images with M large. In the more general case, as for example if we are designing a real-time vision system, images will be following each other through the pipeline in immediate succession, and thus the latency time will be of no consequence.

The neighborhood processor architecture described results in each of the operations in the pipeline being performed on an image in immediate

succession. In particular, suppose that operation B follows operation A in the pipeline. As soon as a particular neighborhood of the image has been operated on by A, it is passed to B. Thus we have the situation that B may be transforming one part of the image while A is working on an earlier part of the same frame. Most often this is not a problem, as for example with operations such as thinning and filtering. But some operations, in particular the ridge numbering processor to be described in a later section, rely on the communication of derived data from one processor in the pipeline to a succeeding one by a path outside the normal data flow. In order to be sure that the earlier processor has been able to complete its operation on a given image (and therefore has finished collecting the information needed by the next processor), we can introduce a delay element between the two stages of the pipeline involved. The introduction of this delay, while increasing the overall latency time somewhat, has no effect on the throughput of the pipeline.

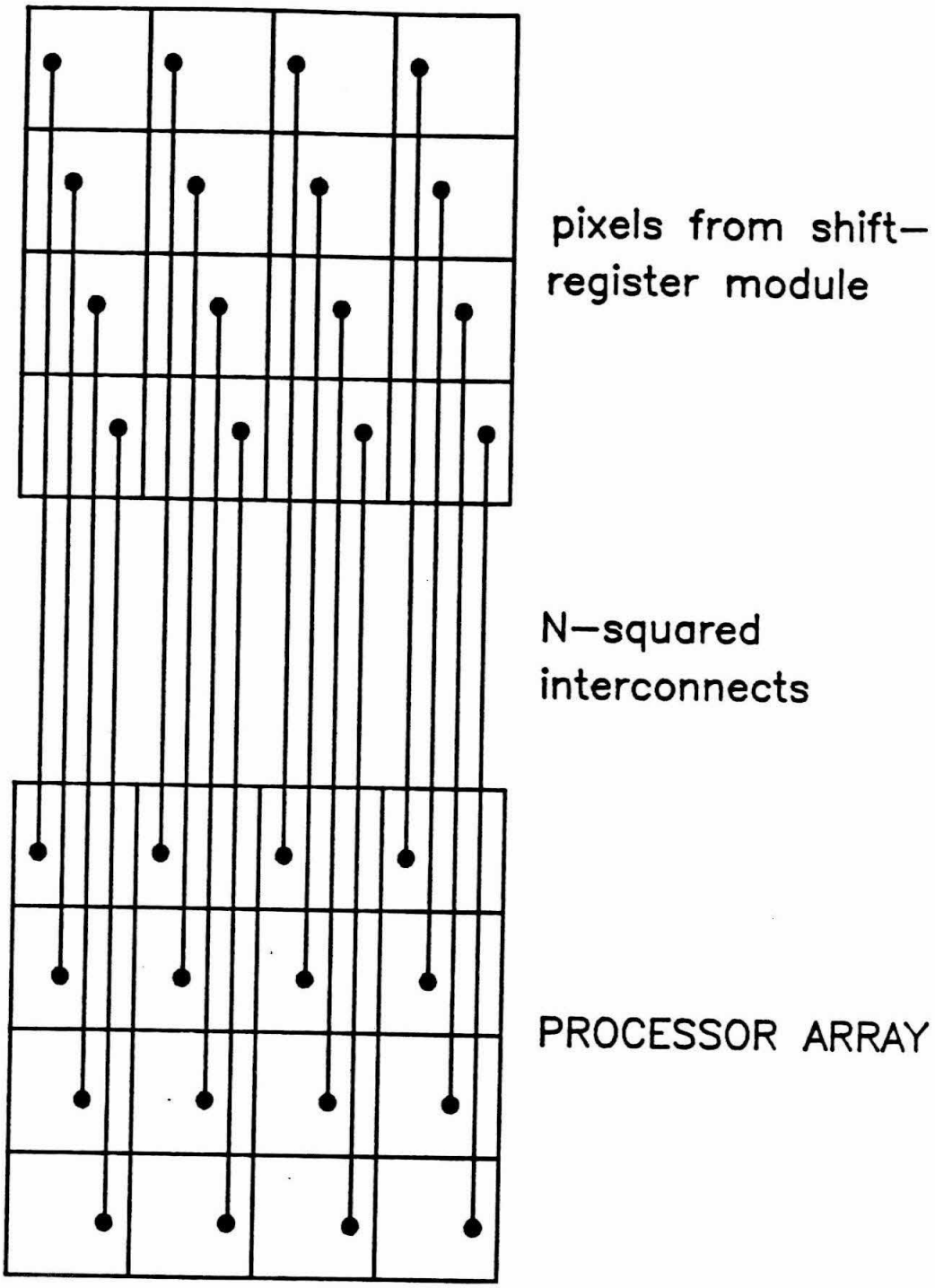


Figure 3.2: Processing Configuration

It is not necessary that the actual logic that performs the computation, based upon the values of the image pixels, reside on the same chip as the shift registers. It may in some cases be preferable to have a standard shift register chip, containing N shift registers, with values from the center N pixels of each register brought off chip. In this way, these N^2 wires can be connected to an arbitrary computation unit, be it random logic, PLA, or even a sufficiently fast microprogrammed processor. Of course, for the Large (i.e. 15x15 pixel) windows it becomes more and more attractive to design a special purpose chip containing both the shift registers and the processing, in order to avoid the needed N^2 wire interface.

Not all of the images being dealt with will have only one bit per pixel -- most will initially contain 8 or more bits per pixel (particularly if color images are involved). Thus each pixel as discussed above really consists of several bits. This added dimension can be handled without the need to design special purpose shift register modules -- one for 1-bit deep images, one for 2-bit deep, etc. By making use of a "bit-slice" approach to the problem, we may parallel any needed number of single-bit shift register modules. For example, if the operation to be performed requires a 3x3 window operating on a 4-bit per pixel image, we would parallel four shift register modules, each containing 3 line-length shift registers (see Figure 3.3). The resulting 36 bits of window data would then be connected to the appropriate processing unit, which could itself be made up of single-bit processing slices interconnected in the conventional manner.

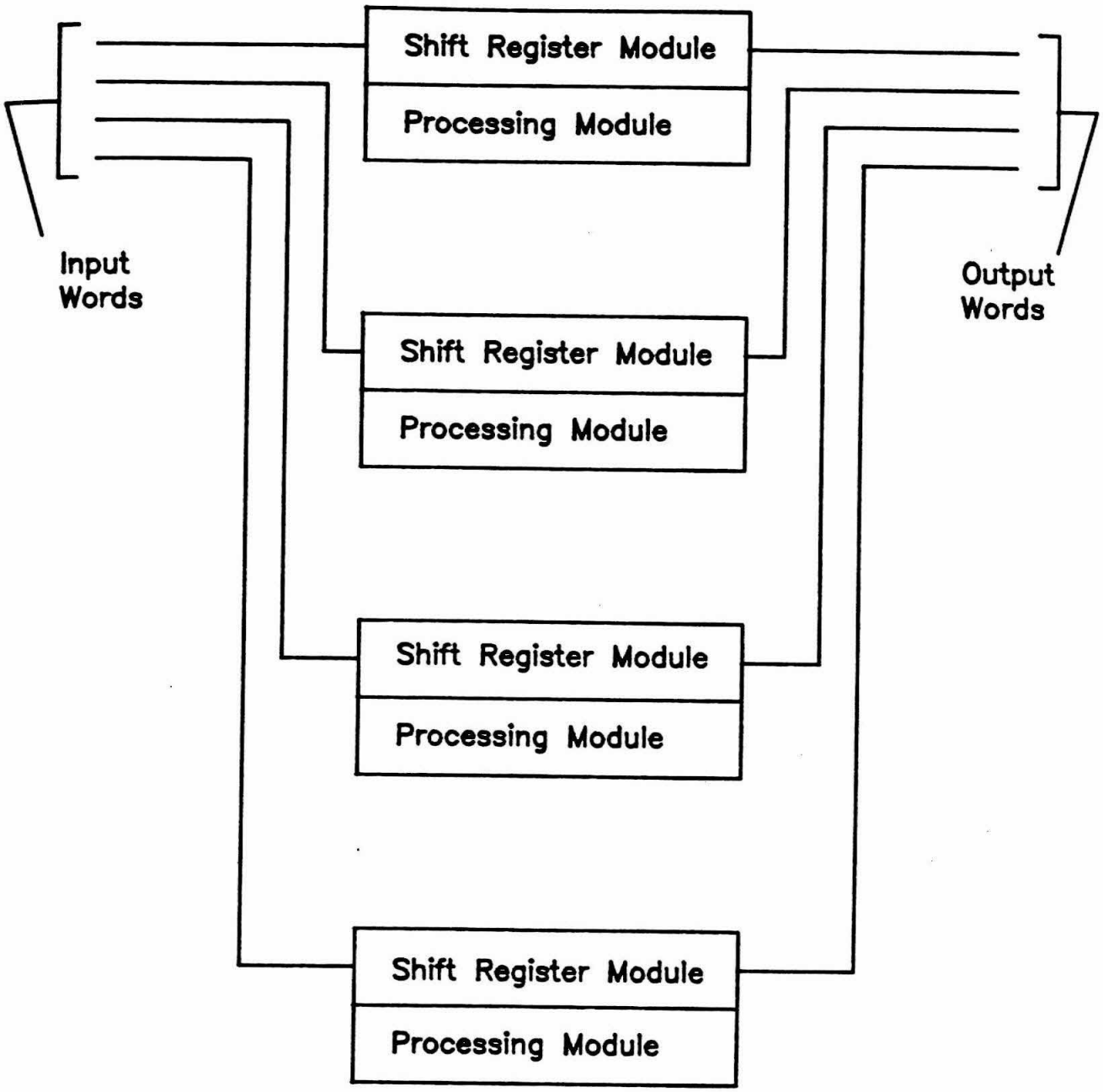


Figure 3.3: Neighborhood Processor Bit Slices (4-bit Words)

Note that multiple-bit per pixel images are not restricted simply to the representation of gray-level or color attributes of the image. Very commonly, in the algorithms to be described later, multi-bit images are used to support labeling. For example the ridge numbering algorithm, though its input is a binary image (a thinned fingerprint), produces as output an 8-bit per pixel image, where the "value" of a particular pixel is really just the assigned label of the ridge to which it belongs. This use of multi-bit images requires the introduction into the pipeline at some locations of special purpose processors. These processors are responsible for image format conversion, from 1-bit per pixel to multi-bit, and vice versa. As an example, the conversion processor used ahead of the ridge numbering step has as its input a stream of pixels of value either 0 or 1. Its output consists of a stream of pixels each 8-bits wide. If the input pixel had value 0 the output pixel has value 0, and similarly for "1" pixels. But now the extra bits in the output stream are available for containing ridge numbers. Thus we may see a pipeline with 8-bit format on the input, 1-bit format after the thinner, followed by several switches from 8-bit to 1-bit and back, depending on the needs of the intermediate processing steps.

Each of the neighborhood processor algorithms described in the remainder of this work was simulated in software using the SIMULA language on a Decsystem-20. Strict adherence to the neighborhood architecture was maintained, even though the resulting simulation software was often far from being an efficient implementation of the desired algorithm. Due to address space limitations, the pixel values were thoroughly compacted into the 36-bit words of the DEC-20, thus requiring significant overhead for access to a particular pixel. This compaction was made transparent to the programmer by use of SIMULA classes which had built into them the data

de-compaction algorithm, as well certain fundamentally useful concepts such as "neighbor" and "window-size". The result was that it was straightforward to write the code to simulate a given neighborhood processor, but the resulting code might require several orders of magnitude more time to transform an image than would the VLSI processor being simulated. Selected examples of the SIMULA code used to produce the results in this work are included in Appendix 2. It should be noted that although as has been discussed above, window processors of only two sizes (3x3 and 15x15 pixels) are necessary to implement all of the processing operations described, windows smaller than 15x15 (e.g. 11x11) were occasionally used in the simulations. This was done in order to minimize computation time, and has no significant effect on the results obtained.

3.2. Composition of Pipeline Stages

In the implementation just described, each function is implemented by a separate stage in the processing pipeline. It is not always necessary that this be true, and we must give consideration to the possibility of combining one or more processing steps in a single pipeline stage. Doing so has the advantage of reducing the number of special-purpose VLSI circuits that must be built into the system, as well as reducing the latency time due to the shortening of the pipeline. In the general case there are many operations which could be merged in this way. To consider a simple example, if one were to have two sequential processing stages that implemented convolutional filters analogous to the low-pass done in this work by the FILTER program, it would be a simple matter to produce a new

set of window weights which would implement in one pass the same filter function as would be obtained by having the data pass through both of the original processing steps. This is really just an example of functional composition, for if we describe the operation performed on the data stream by the first processing stage as F(s), and that performed by the second stage as G(s), the new composite stage must simply implement C(s), where C(s) = G(F(s)). In the case of convolutional filters it would certainly be to one's advantage to do this compression of the pipeline, as the new single stage processor would not be inferior to either of the original two in terms of speed or complexity.

The situation becomes less clear cut when we consider the possibility of combining some of the more complex processing stages used in this work. As will be seen later in this chapter, many of the operations performed make use of "waves" of communication amongst the processors in the window, with the central processor sending out what are essentially requests for information from the other cells, with the result eventually being propagated back to the center for a final decision. While this form of computation does not inherently prevent the combination of successive stages in the pipeline into one, there are some very real practical difficulties encountered if we attempt to do so with the processing operations used in this work.

Perhaps the most serious problem is caused by the varied nature of the operations involved. Though all make use of the same window processor architecture, they differ very much in their inner details. We need only contrast the Pore Removal stage with the Thinning stage which follows it. The fundamental operation of Pore Removal is to determine if a path can be

found from the center of the window to an edge without encountering a ridge pixel. Thinning, in contrast, relies upon successive matching of a number of templates to the pixel pattern contained in the window, with the result of the matching process determining if the center pixel is to be deleted. Though it is not fundamentally impossible to combine two such operations in one window processor, it is also not a combination which appears either natural or obvious. Even the means used to represent the data changes as we move from one stage in the pipeline to the next - at one point a value of "1" will represent the grey level of a pixel, while farther down the pipeline a "1" will mean that the pixel is part of a ridge. At a still later stage a "1" may be a label assigned to a pixel indicating the name of the ridge to which it belongs. Though one would of course be free to modify the data representation used as was needed to aid in the stage compression process, the differences just described are in fact indicative of the deep underlying dissimilarity of the operations.

Even if one were to combine two or more such complex operations in a single window processor, their disparate nature makes any benefits gained from doing so questionable. For unlike the case for the simple convolutional filters, each processing cell would now be called upon to perform significantly more operations per window position than would the original, separate processing stages. As a result, one would expect that either the complexity of each processor would increase, or the speed at which the data could be shifted through would decrease, or both. Any decrease in speed is certainly undesirable, as optimization of throughput was the initial motivation for use of a pipeline structure. Thus we see that composition of successive processing stages is an option to be considered, but only where the similarity of the operations being performed justify it.

Unfortunately, the sequence of operations used in this work is such that little if any benefit would derive from reducing the length of the pipeline.

3.3. The Processing Steps

This section describes the steps involved in processing the fingerprint image from the point at which it is first formatted into an 8-bit per pixel gray-level image on the DEC-20 until the thinned, binary image is ready for the chosen feature extraction algorithm (see Figure 3.4). Lack of a graphics hard-copy device capable of reproducing images with grey-level information forces us to illustrate typical input and output data for the processing steps that follow with 2-level binary images. Nevertheless, the effects of each of the processing steps should be evident.

3.3.1. Spatial Filtering (Smoothing)

The introduction of unwanted noise into the images under consideration is unavoidable. This noise can stem from many sources, such as dirt on the finger, imperfections in the optics, and A/D converter quantization error. The removal of such noise is important in order to prevent confusion later in the processing chain due to the presence of, for example, spurious pixels. Fortunately, such noise is usually concentrated at high spatial frequencies, while the desired fingerprint data is at lower frequencies. In fact, due to the rather regular ridge structure in fingerprint images, the majority of the useful information is at spatial frequencies less than a particular frequency

which we will call F_{cutoff}.

It is clear that F_{cutoff} is determined by the minimum spacing of the ridges in the fingerprint images under consideration, measured in pixels. Though there is some variation of ridge spacing, both within a given print and between individuals, it is still possible to establish a minimum ridge spacing value which will remain correct for a given set of optics. For the purposes of this work the minimum ridge spacing is considered to be approximately 8 pixels, which is equivalent to a spatial frequency of .125 cycles per sample (pixel distance), abbreviated cy/pid.

Thus a low-pass filter properly matched to the desired information in the image will have minimal attenuation at spatial frequencies of .125 cy/pid, with the attenuation increasing for higher frequencies. The filter chosen has a Gaussian frequency domain transfer function, with an attenuation of 6 dB at .25 cy/pid. The transfer function of the filter is shown in Figure 3.5a. The inverse Fourier transform of the transfer function results in the normalized impulse response shown in Figure 3.5b, along with its discrete approximation. The filtering operation is implemented by convolution of the image and the impulse response. As can be seen in Figure 3.5b, the chosen filter cutoff frequency results in an approximated impulse response with only three significant terms, and as a result the necessary convolution is implemented by making use of a Small (3x3) processing window.

This convolution is simulated in the program FILTER, with due care to insure that all operations performed are consistent with the specified architecture for VLSI image processors. Specifically, the 3x3 window of weights shown in Figure 3.6 is "moved across" the entire image, with the

pixel that is currently at the center of the window being replaced in the output image by the weighted average of its neighbors. The weight values are chosen to match the desired impulse response for the filter after appropriate scaling. Using the standard notation for describing the pixels in a 3x3 window shown in Figure 3.7, we may express the filtering process algebraically as:

$$new_{center} := \frac{4old_{center} + 2(old_2 + old_4 + old_6 + old_8) + old_1 + old_3 + old_5 + old_7}{16}$$

See Figure 3.8 for a thresholded version of the image input to the filtering process, and Figure 3.9 for the thresholded result. The reduction in the level of high frequency noise is significant.

3.3.2. Thresholding

Though the fingerprint image is originally digitized to 8-bit (256 grey level) accuracy, this intensity information is needed only for some of the initial image processing steps, such as filtering. At some point it becomes necessary (and desirable from an information storage point of view) to convert the grey-level image to one containing only one bit of information per pixel - i.e., a binary image. This conversion operation is known as thresholding, and is used to define disjoint regions in the image with closed connected boundaries [Castleman79]. In the case of fingerprint images, the regions are of two types, "ridges" and "background". Thresholding is what is known as a "point" operation, in that on a pixel by pixel basis the following rules are applied:

$$G(x,y)<T \rightarrow G(x,y):=0$$

$$G(x,y)\geq T \rightarrow G(x,y):=1$$

where G(x,y) is the grey-level value of the pixel located at (x,y). In other words, all pixels whose grey-level value falls below the threshold are assigned to the background, while any pixels with value at or above the threshold become part of the ridges.

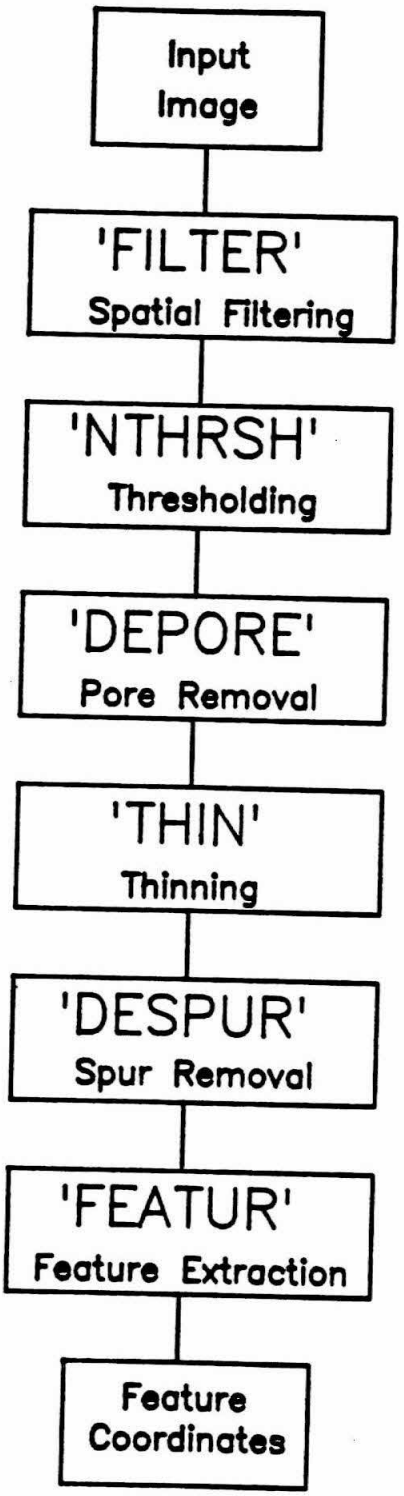


Figure 3.4: Image Processing Operations

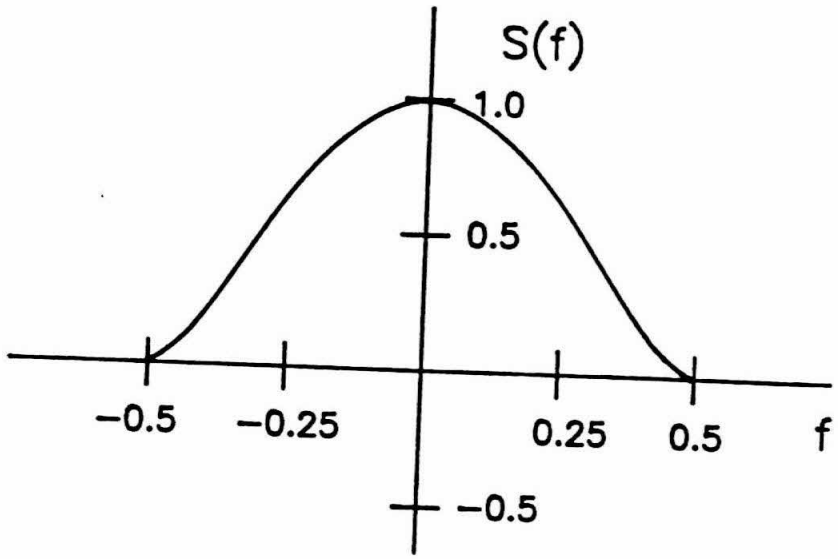


Figure 3.5a: Filter Transfer Function

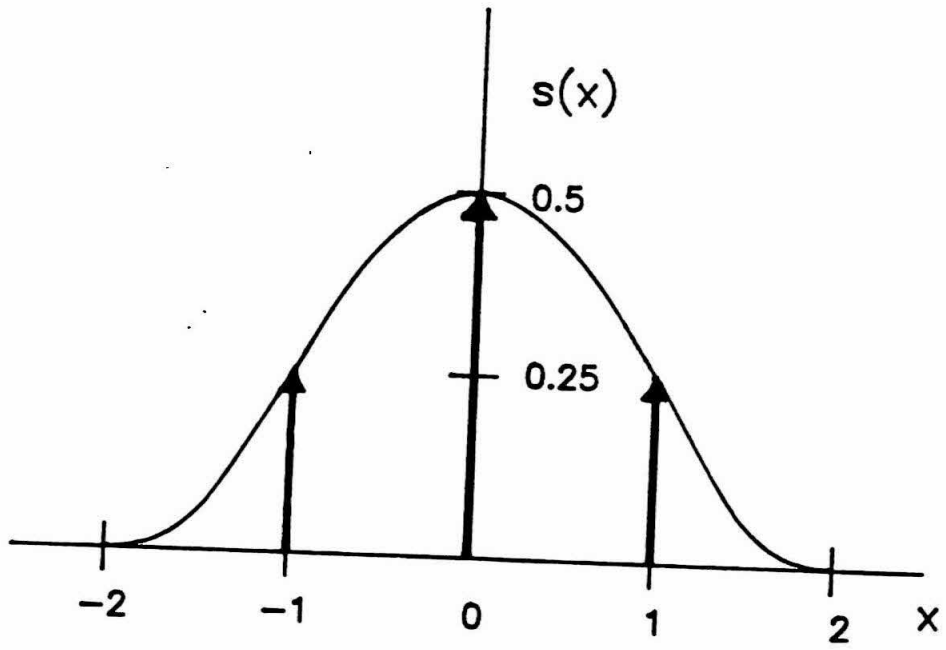


Figure 3.5b: Filter Impulse Response

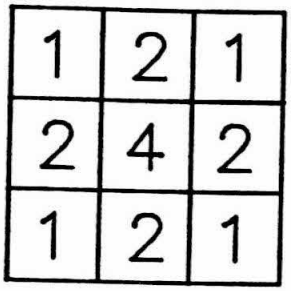


Figure 3.6: Window Weights
for Smoothing Filter

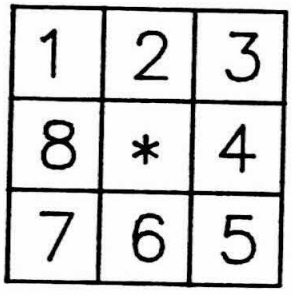


Figure 3.7: Standard Numbering
of Window Pixels

Figure 3.8: Unfiltered Image

Figure 3.9: Filtered Image

In the simplest case, the threshold T is a single grey-level value, which remains constant for the entire image. This approach is known as "global thresholding", and works well only if the image background is of roughly constant intensity, and if the contrast of the objects above the background is also approximately constant. These assumptions are not in general true for the fingerprint images, and as a result global thresholding produces inferior results (Figure 3.10).

A more effective system is known as "adaptive thresholding", whereby the grey-level threshold varies appropriately throughout the image. The key to such a thresholding algorithm is the method used to determine the threshold for each pixel in the image. In this case a method is chosen that is consistent with the neighborhood processor architecture. In particular, a window of sufficient size to guarantee that it is larger than the width of any one ridge (i.e., a Large (15x15) window) is passed over the image. For each position of the window, the histogram of the grey-level values of the N^2 pixels within the window is computed. It is this "window histogram" that is used to decide the threshold value for the pixel at the center of the window, in a manner to be described.

Given the histogram of all of the pixels within the current window, there are two cases to be considered:

(1) The histogram is bimodal (two peaks)

(2) The histogram is unimodal (one peak)

See Figure 3.11 for example histograms. If the histogram is unimodal, then the window must be filled completely with background pixels, for we have assumed that the window is so large as to prevent its being filled solely with ridge pixels. If the histogram is bimodal, then we have both

background and ridge pixels within the window, and must use the shape of the histogram to determine the proper threshold to apply to the pixel at the window center.

Given a bimodal distribution, one common approach to determining the proper threshold position is to use clustering theory (in particular the so-called ISODATA procedure [Duda73]). Unfortunately, such procedures tend to be quite sensitive to the relative heights of the peaks in a bimodal distribution, and often choose the threshold incorrectly. It was for this reason, and for reasons of simplicity in VLSI implementation, that another approach was chosen. If we assume for a moment that the histogram for a particular window is known to be bimodal, we may calculate a threshold as follows: Determine for the histogram the largest and smallest gray-level values for which the number of pixels taking on that value is non-zero. Call these values L and S, respectively. We may then set the threshold a fixed fraction of the distance between L and S. The choice of this fraction is dependent upon knowledge of the relative widths of the peaks typically encountered for the type of images being analyzed. For the fingerprint images, a value of ½ has been found to be suitable. In other words, we are setting the threshold to be in the center of the non-zero portion of the window histogram. See Figure 3.12. The median is used here rather than the weighted mean, in order to minimize the sensitivity of the algorithm to the relative heights of the two peaks in the histogram - such heights being directly related to variables such as finger pressure used when inputting the image.

Of course we would not want to apply the above algorithm to a unimodal histogram, as the threshold would then be placed in the center of

the peak, with meaningless results. Thus we need a method of deciding when the histogram is unimodal. The simplest solution consistent with the assumptions made above is to simply decide based upon the width of the non-zero portion of the histogram - narrow histograms are assumed to be unimodal; wide ones are assumed to be bimodal. It is now necessary to specify a "meta-threshold" T', which is the dividing line between "wide" and "narrow". As a practical matter this value is easily chosen after processing a small number of fingerprint images, and is not critical. Any value in the range of 20-25 has proven adequate for all fingerprint data encountered. So now, rather than choose a fixed grey-level threshold as in the global thresholding approach, we need only choose this meta-threshold. The value of the meta-threshold is determined by general properties of the type of images under consideration (i.e. the spread in pixel gray-levels), and therefore need not be varied for a given configuration of input hardware.

As an improvement to the above approach one can consider a method for automating the choice of T'. For each position of the window on the fingerprint image, we have computed a histogram. Each of these histograms has a particular width of its non-zero portion. If a higher level histogram (meta-histogram) is now made of the widths of the window histograms, we can now analyze the meta-histogram using techniques similar to those being discussed to decide upon the meta-threshold. Obviously it would be pointless to recurse infinitely in this decision process, and thus the meta-threshold is conveniently chosen as the halfway point in the meta-histogram. Unfortunately the computation of such a meta-histogram is a global operation, and as such does not fit well into the proposed processing architecture. Thus the simpler approach outlined above was used in these simulations.

Figure 3.10: Globally Thresholded

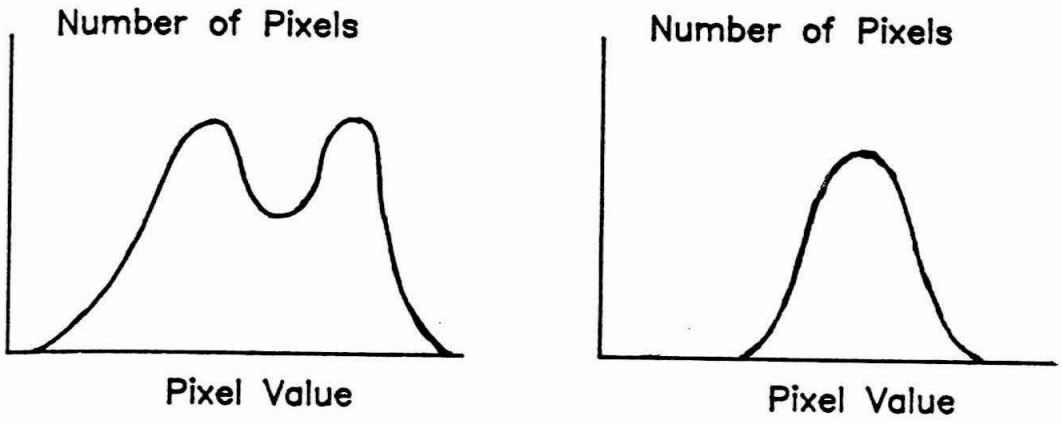


Figure 3.11: Example Histograms

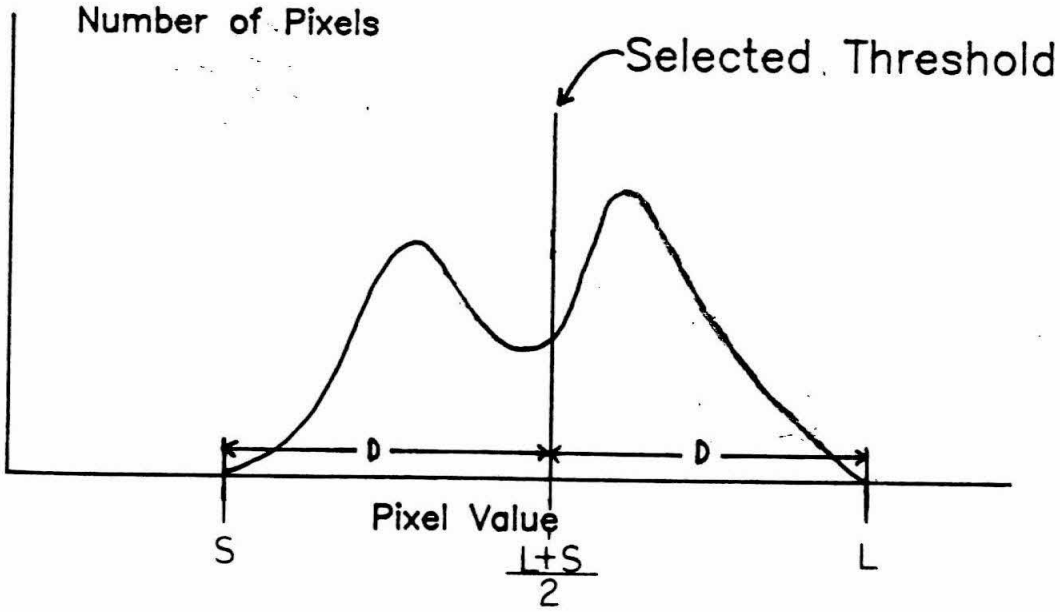


Figure 3.12: Threshold Determination

The simulation of the method described above works quite well, but requires substantial amounts of computer time due to the need to compute the histogram of a 15x15 window for each point in a 400x400 image. In order to minimize the time necessary for the simulation, the histogram is computed differentially, relying on the fact that the window moves only by one pixel each time. Thus only a small number of the 225 entries in the histogram must be updated. However the situation is even better when we consider the VLSI implementation of the thresholding algorithm. Because of the use of the $\frac{L+S}{2}$ method described above for determining the threshold for each window position, it is not necessary that a complete histogram be calculated at all. Rather, each processor cell in the 15x15 array need only pass information about the maximum and minimum grey-level values encountered toward the center of the window (i.e., propagate the values of L and S toward the center). When the center processor has received this information from all the other cells, it can verify that L-S is greater than the meta-threshold (i.e. the histogram is bimodal), and then, if its grey-level value is less than $\frac{L+S}{2}$ change itself into a background pixel, or if its grey-level value is greater change to a ridge pixel. The time necessary for this to occur is proportional to the linear dimensions of the window used.

Compare Figure 3.10, thresholded using a global thresholder, with Figure 3.13, for which the adaptive thresholder simulation was used.

3.3.3. Pore Removal

The resolution of the optical system used to input the fingerprint images is such that upon close inspection one can see small white areas

that lie within the ridges of the fingerprint. These pinhole-like areas are skin pores, which appear at irregular intervals along each ridge. These pores are visible in Figure 3.9, which is the thresholded version of a filtered image. It is desirable to remove the pores before the image is thinned, as otherwise the resultant skeleton will contain small, closed circles at the pore sites (see Figure 3.14). These circles are topologically equivalent to pairs of ridge bifurcations, and thus would tend to interfere with any feature extraction scheme that made use of bifurcations.

The program used to do pore removal is called DEPORE, and simulates a moving-window type of image processor, using a Large (i.e. 15x15 pixel) window. This windowsize was chosen based upon observation of the sizes of the largest pores encountered. In order to be removed, a pore must fit entirely within the window. Yet one would not want to make the window overly large when doing simulations, as the CPU time required increases quickly for large windows. Note that this is a limitation of the simulation, not of the computing structure proposed. As will be explained below, VLSI implementations of this algorithm are possible for which execution time increases no worse than linearly in the window dimensions (i.e. proportional to the square-root of the number of processors present).

A pore is defined as a region of white (background) pixels completely enclosed by a black (ridge) region. Before proceeding with the description of the pore removal algorithm, it is necessary to first consider the issue of connectivity of pixels within regions. Given a pixel, we may speak of it having two different sets of neighbors: its "4-neighbors" which are the pixels immediately above, below, to the right, and to the left of the pixel, and its "8-neighbors" which consist of its 4-neighbors plus its four diagonal

neighbors (see Figure 3.15). We define two pixels to be "4-connected" if they are 4-neighbors of each other. Similarly, two pixels are "8-connected" if they are 8-neighbors of each other.

For purposes of pore removal, we will use the concept of 4-connectedness when dealing with background pixels, and 8-connectedness when dealing with ridge pixels. Figure 3.16 demonstrates the reason for this choice. The circular white region is completely surrounded by the black region, and is considered a pore. If we had not defined the connectivity of the ridge and background pixels in this way, the circular white region would be connected to the remainder of the background, and would no longer be considered a pore.

The DEPORE program is implemented using the following algorithm:

```
  FOR each position of the window DO

BEGIN
    IF center pixel of window is a background pixel THEN

    BEGIN
      IF Is-A-Pore-Pixel?(CENTER PIXEL) THEN
          set to black all pixels connected to the center pixel,
          as well as the center pixel (i.e., fill in the pore);
  END;
END;


BOOLEAN PROCEDURE Is-A-Pore-Pixel?(PIXEL);

  IF PIXEL is a ridge pixel THEN Is-A-Pore-Pixel?:=TRUE ELSE
    !if run into a ridge pixel, then we may be in a pore;

BEGIN
    IF we are at the edge of the window THEN Is-A-Pore-Pixel:=FALSE
  !we ran into edge of window, so may not be in a pore;

  ELSE BEGIN
       mark this pixel as connected to center pixel;

          IF Is-A-Pore-Pixel?(left neighbor) AND
          Is-A-Pore-Pixel?(right neighbor) AND
            Is-A-Pore-Pixel?(upper neighbor) AND
        Is-A-Pore-Pixel?(lower neighbor) THEN

          Is-A-Pore-Pixel?:=TRUE;
        !if all the 4-neighbors are in a pore then so are we;
     END;
END;
```

Thus we can see that the boolean procedure Is-A-Pore-Pixel? is invoked recursively, beginning with the pixel at the center of the window, and spreading outward in an ever-expanding "decision wave". If at any point this wave encounters the edge of the window, then we know that the background region containing the center pixel is not completely surrounded by ridge pixels (at least within the confines of the window), and we simply go on to the next window position. If, however, only ridge pixels are

encountered, then the center pixel of the window must be inside a pore region; thus it and all of the pixels that have been marked as connected to it are transformed into ridge pixels, thereby filling in the pore.

An efficient VLSI implementation of this algorithm would not use the above described recursive form of the algorithm. Rather, it would be preferable to use an array of small processors, each connected to its 4-neighbors. The determination of whether the center pixel of the window is in a pore is then a simple matter of the center processor asking each of its 4-neighbors "are you connected to a pixel at the edge of the window?". Each of the neighbors in turn asks this question of its neighbors, etc., until all processors in the window have been queried. When this questioning process encounters a ridge pixel, the question is not further propagated. If no answers of "yes" are received back at the center processor, then the center pixel of the window is obviously completely surrounded by ridge pixels, and is therefore within a valid pore. At this point a command to "change all questioned pixels to ridge pixels" (fill in the pore) can be sent out. Note that this method requires time proportional to the linear size of the window, while the recursive form used in the simulation requires, at best, time proportional to the window area.

The immediate result of pore removal can be seen in the image of Figure 3.17.

Figure 3.13: Adaptively Thresholded

Figure 3.14: Thinned (No Pore Removal)

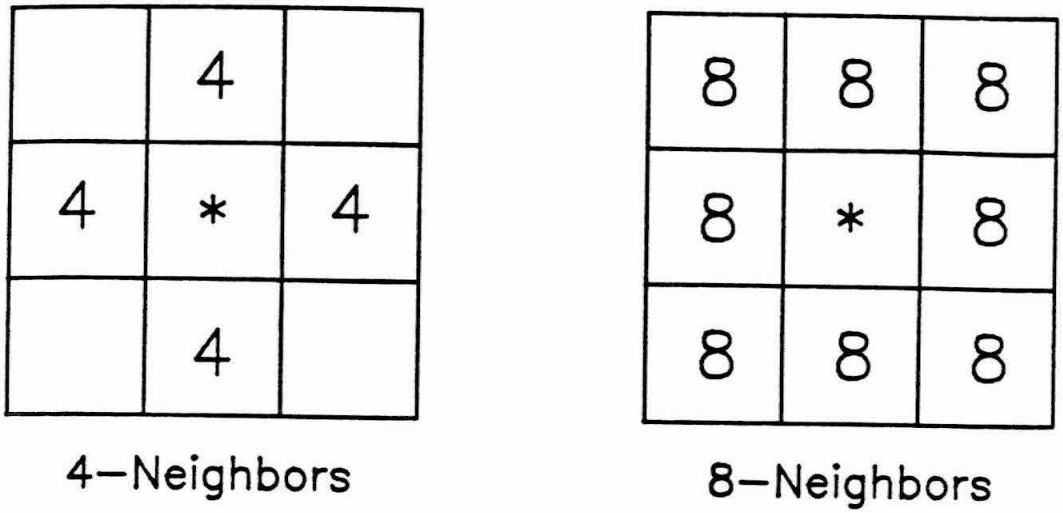


Figure 3.15: Neighbor Connectivity

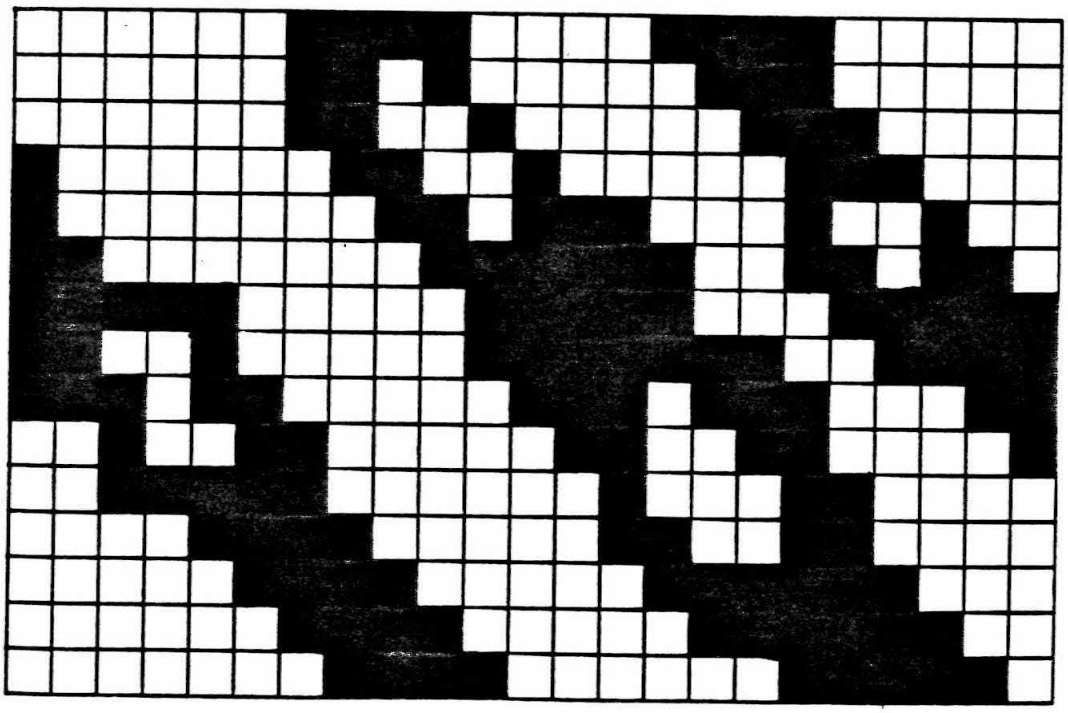


Figure 3.16: Pore Detail

Figure 3.17: Result of Pore Removal

3.3.4. Thinning

The amount of pressure that can be used when making a fingerprint impression, whether with ink or on an optical encoding device, may vary from several ounces to as much as ten pounds without significant loss of important feature information [Moenssens71]. Yet over this range of finger pressures the one parameter that will vary substantially is the apparent width of the ridges on the print. For this reason, and for reasons of data compaction, it is desirable to remove the ridge width information from the fingerprint images -- i.e., to "thin" them.

The subject of thinning has been studied widely in the image processing and pattern recognition literature, with mixed results. "Thinning" can be described intuitively as "transforming a figure into a set of pixels which has unit thickness everywhere and still retains significant information of both a topological and geometric nature" [Arcelli81]. Yet, this definition points to no one "correct" algorithm. In fact, dozens have been proposed, differing both in approach and results. Two common approaches are "skeletonization", which means finding an approximation to the medial axis of the objects in question [Pavlidis80], and "shrinking", which means the reduction of a set of pixels to its smallest topological equivalent.

Perhaps the most important consideration when choosing a thinning algorithm is the behavior of the algorithm in the presence of the unavoidable noise in the image. In the case of fingerprints we are dealing with objects (ridges) which may be considered "nearly thinned" due to their long, narrow shape --but which also have significant variations in local outline. The width of a given ridge may vary by a factor of two or more within a short distance, as well as having many small-scale indentations and

protuberances. Unfortunately, many thinning algorithms are quite sensitive to such variations and will produce a thinned image that is replete with small "spurs" -- many only a few pixels in length. One such popular algorithm, which attempts to simplistically distinguish interior from exterior pixels, and then strip away the exterior pixels [Wong79], was applied to a typical fingerprint image. The result (Figure 3.18) is clearly unsatisfactory.

Such sensitivity to small details in the image is not merely a question of "good" vs. "bad" implementations of thinning algorithms, but rather is directly related to the definition chosen for "thinning". An entire range of algorithms are possible, from a pure medial axis derivation (which will follow the most minor variations in the contour), to a less sensitive, but more intuitive thinner. In fact, an algorithm has been described [Arcelli81] which by variation of a single parameter spans the entire range, making use of local curvature measures to determine the significance of variations in the shape of an object. In order to make use of such curvature information, this (and many other) thinning algorithms require the extraction of the contour of the objects in the image as a preliminary stage to the actual thinning operation. Though contour extraction can be done in a wide variety of ways, none seems particularly well suited to the neighborhood processor architecture we desire to use. As a result, an alternative thinning algorithm was implemented.

The method used consists of two parts. First, a connectivity preserving shrinking transformation is applied to the image by use of a Small (3x3) window neighborhood processor (program THIN). This transformation is a modified form of an algorithm described by Kruse, and assumes that the

objects are 8-connected. Though a form of shrinking algorithm, and therefore rather immune to the noise problems described above, some spurious segments are still produced. The second part of the thinning process (program DESPUR) is designed to remove virtually all of these spurs, and will be described below.

The shrinking transformation actually makes use of several passes of neighborhood processors over the input image. The need for multiple passes by somewhat different neighborhood processors is not a problem, as each of the special purpose processors may simply be placed one after another in a pipeline, each modifying the image in its turn. The image that is to be thinned is represented in a one-bit per pixel format, having already passed through the thresholding step. Yet at various points in the thinning pipeline, the image will be represented using multiple bits per pixel, in order that pixels may be given labels other than "ridge" and "background". These labels are temporary only, and are for use by one of the subsequent processing steps. This need for multiple bits per pixel (typically no more than two) is easily met by paralleling single bit window processing "slices", as described in the Section 3.1.

The fundamental operation used in the shrinking transformation is that of a template match. We specify a pattern of values for the cells in the 3x3 window and, as the window is "moved over" the image, if the pixels in the image match the template, the center pixel is re-labeled as specified by the particular transformation in use. The image is assumed to originally consist of only pixels with values 0 (background) and 1 (ridge). A value of -1 in a template is a "don't care", and matches any pixel value.

Figure 3.18: Improperly Thinned Image

The first two templates applied to the image are shown in Figure 3.19a. Template A marks "potentially deletable" pixels with the value 2, while template B deletes any pixel marked as deletable by A which has the correct pattern of neighbors. In particular, no pixel labeled "2" is deleted unless it has a neighboring ridge pixel, thus serving to preserve the 8-connectivity of the ridges. Note that, though for each of templates A and B only a single pattern is shown, all eight rotations of the specified template are used in trying to find a match with the pixels in the window. Each of these templates (or rather sets of eight templates) is passed over the image multiple times, until no further change occurs. The number of passes necessary is bounded above by the largest expected ridge thickness (in pixels). Therefore it is not necessary to iterate until no change is observed -- rather, we can simply place a sufficient number of thinning stages sequentially in the pipeline. No harm is done if an already thinned portion of the image is processed again, as template B will not delete the pixels of a completely thinned ridge. The number of thinning stages need only be sufficient to guarantee complete processing of the thickest ridges. The resulting image is then an approximately thinned version of the input image. Specifically, the resulting thinned pattern may be two pixels in width at some points, a condition remedied by latter passes.

To obtain unit width of the thinned ridges, we apply the sixteen templates shown in Figure 3.19b, in order. Only two basic patterns are involved, with all eight of their rotations, but each of the rotations is explicitly shown. See Figure 3.20 for an example of an image that has been thinned by templates A and B (with their rotations), and Figure 3.21 for the same image after thinning down to unit ridge width.

0	X	1
0	1	1
0	X	1

(all 8 rotations)

(X ==> "don't care")

If template matches, center pixel is relabeled "2"

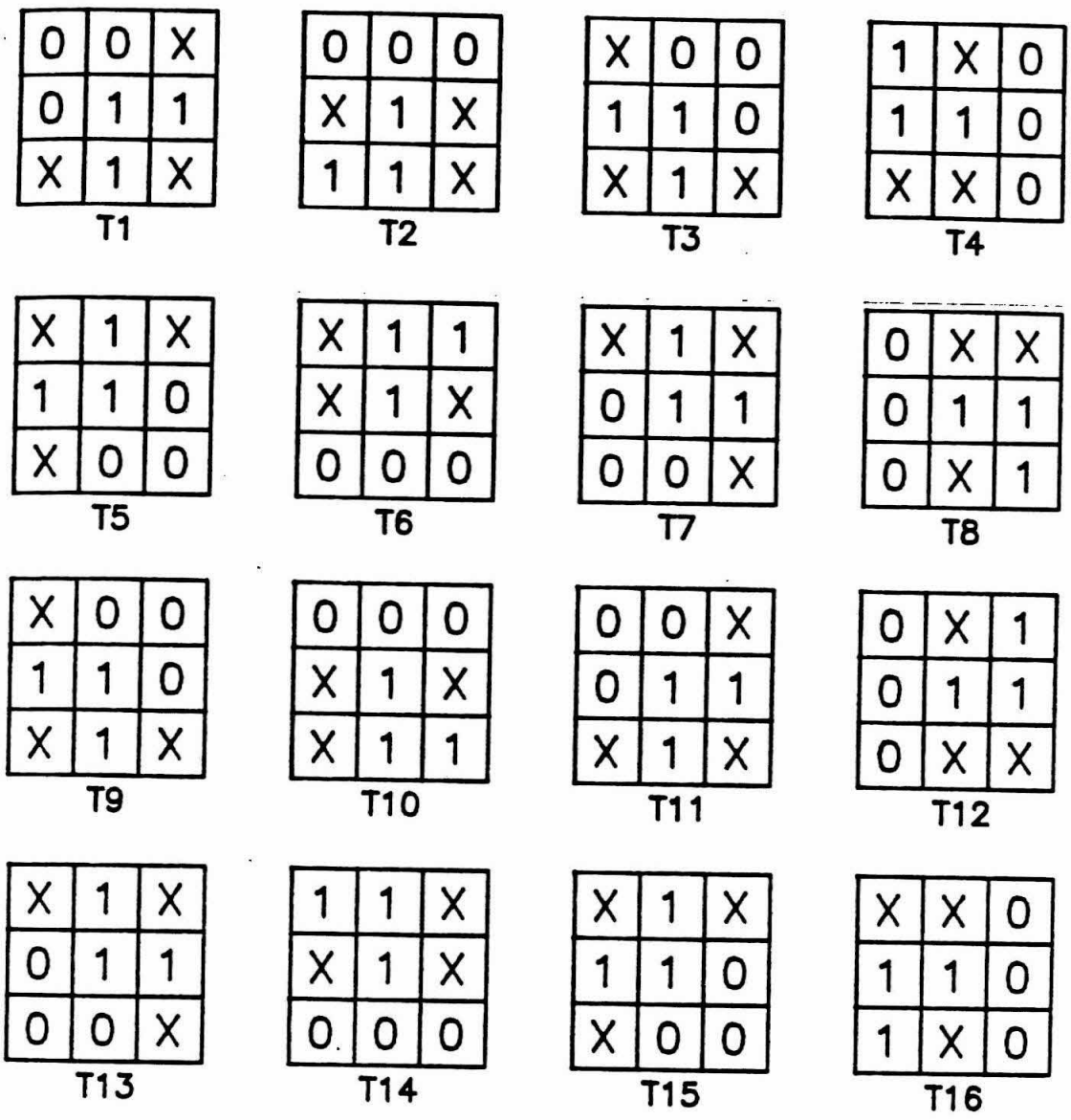
0	X	X
0	2	1
0	X	X

(all 8 rotations)

(X ==> "don't care")

If template matches, center pixel is set to "0"

Figure 3.19a: Initial Thinning Templates



(X ==> "don't care")

(If template matches neighborhood
then center pixel set to 0)

Figure 3.19b: Templates Used to Thin Image to Single-pixel Line Width

Figure 3.20: Thinned to 2 Pixels Wide

Figure 3.21: Thinned to 1 Pixel Wide

The simulation of the operation of this pipeline of window processors requires, not surprisingly, a substantial amount of CPU time. This is due to the fact that each of the 32 templates involved must be compared to the nine pixels in the window for each of the 160,000 window positions in the 400x400 image. With a correct VLSI implementation however, the template match process is performed by a small set of gates (or a PLA) which serve to decode the nine input pixel values and decide the correct output value for the center pixel. So we see that the VLSI pipeline implementation will certainly be capable of real-time image thinning.

The small spurious strings of pixels remaining in the image when thinned by the above process are undesirable, both from an aesthetic point of view, and in order to avoid unnecessary complication of the feature extraction processes to come later. Fortunately, we can make use of our a priori knowledge of the geometric properties of fingerprint ridges to allow the removal of these "spurs". Specifically, we know that the vast majority of fingerprint ridges are much longer than they are wide. Thus if we were to remove ridges (and branches of ridges) whose lengths were less than a particular threshold, we would have minimal impact on valid ridge structure, yet would remove essentially all of the spurs. A convenient value for this threshold is on the order of the typical inter-ridge spacing, and values from 10 to 15 were used successfully.

Before presenting the details of the spur removal algorithm, it is necessary to define two special types of pixels that occur frequently in fingerprint images. These are what we will call "ridge-end" and "fork" pixels. A ridge-end pixel is simply a pixel that occurs at the extreme end of a ridge or a branch of a ridge (see Figure 3.22a). Defined precisely, a

ridge-end pixel is a pixel that has exactly one neighbor (as always, we use the concept of 8-neighbors for ridges). A fork pixel is a pixel positioned at the intersection of two or more ridge branches. Though the concept of a fork pixel is intuitively straightforward, a precise definition is complicated by the varied forms encountered in real images. As can be seen in Figure 3.22b, valid fork pixels may have from three to five neighbors, in various configurations.

The exact definition of a fork pixel requires the concept of "transitions" in the neighborhood of a pixel. The number of transitions present in the neighborhood of a pixel is defined to be the number of times a change takes place from "ridge" to "background" or vice versa as the 8 neighbors of the pixel are traversed in order (for example in the sequence 1, 2, 3, 4, 5, 6, 7, 8, 1) (see Figure 3.7 for the standard neighbor numbering scheme). In other words, if we consider the eight neighbors of a pixel to form a circle of pixels surrounding it, we simply start at any one of the neighbor pixels and move around the circle, counting ridge-to-background and background-to-ridge transitions, until we arrive back at the starting neighbor. The total number of transitions counted is the "number of transitions in the neighborhood of the pixel". Given this definition, it can now be stated that a fork pixel is any pixel that has 3, 4, or 5 neighbors, and has at least 6 transitions in its neighborhood.

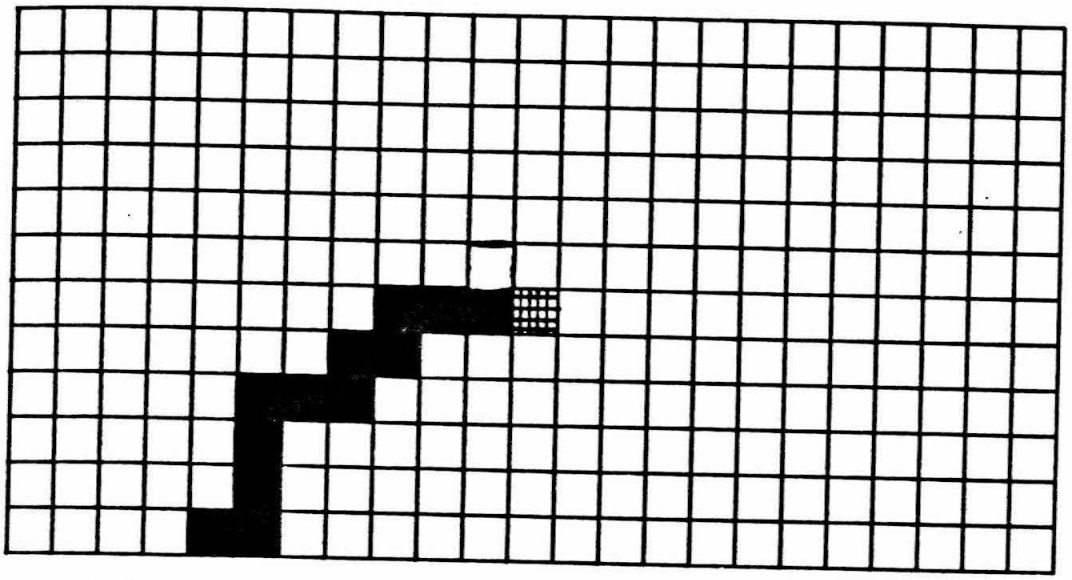


Figure 3.22a: Ridge–end Pixel

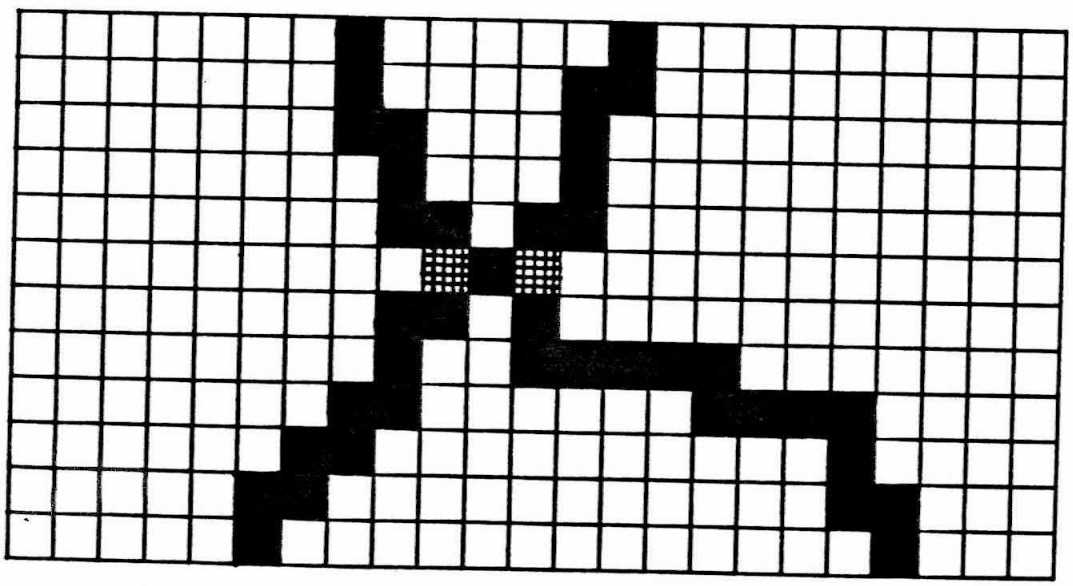
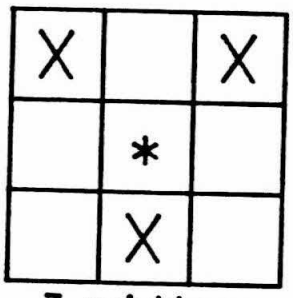
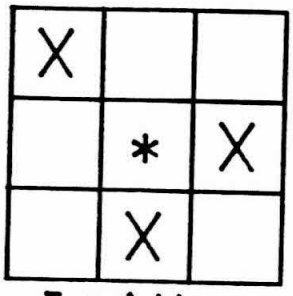


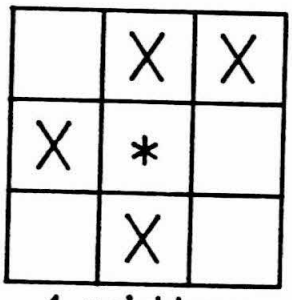
Figure 3.23: Dual Fork–pixels



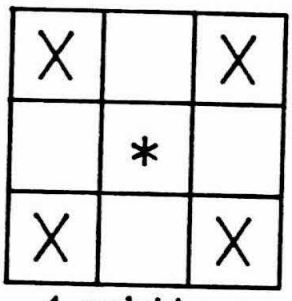
3 neighbors
6 changes



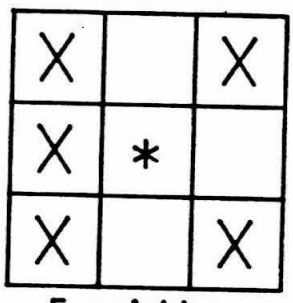
3 neighbors
6 changes



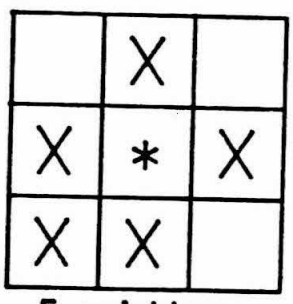
4 neighbors
6 changes



4 neighbors
8 changes



5 neighbors
6 changes



5 neighbors
6 changes

Figure 3.22b: Typical Fork Pixels

The algorithm for spur removal uses a neighborhood processor with a Large (15x15) window. This window size was chosen based upon the length of the longest ridge or ridge branch which we want considered a spur and therefore removed. If a smaller size (such as 11) was wanted for the maximum spur length, then the window size would be reduced accordingly, as was done in some of the examples. The window is scanned over the image, until a position of the window is found such that the center pixel is a ridge-end, as defined above. A ridge-end pixel has only one neighbor, and we now begin following the ridge by looking at this neighbor. This ridge following process is continued, until one of four possible termination conditions occur:

(1) The edge of the window is encountered. In this case we presume that we were following a long ridge (i.e., not a spur), and we move on to the next window position (Figure 3.24a).

(2) A fork pixel (as defined above) is encountered. We now mark for deletion all the pixels along the ridge we have been following, up to but not including the fork pixel itself (deleting the fork pixel would likely break into two parts the ridge to which the current spur is attached) (Figure 3.24b) .

(3) A pixel is encountered which has more than two neighbors, but is not a fork pixel. In this case all the pixels along the ridge we have been following are marked for deletion, including the pixel with more than two neighbors (Figure 3.24c).

(4) Another ridge-end pixel is encountered. We clearly have been following a short, isolated ridge, so we mark all the pixels on it for deletion. (Figure 3.24d).

The pixels marked for deletion by the above process will actually be deleted during another pass over the image (i.e., by the next processor in the pipeline).

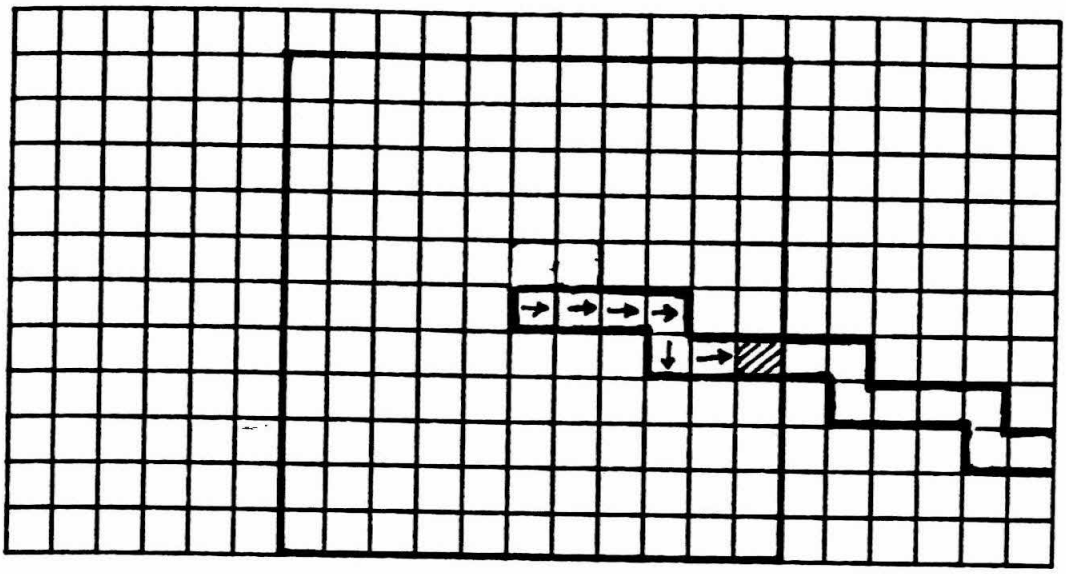


Figure 3.24a: Edge of Window Encountered During Spur Removal

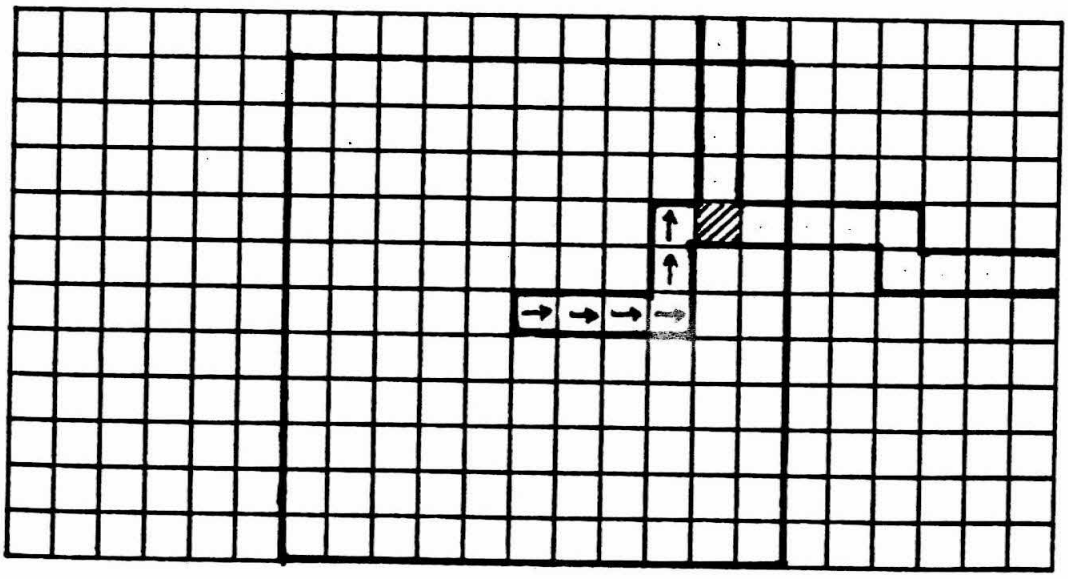


Figure 3.24b: Fork Encountered During Spur Removal

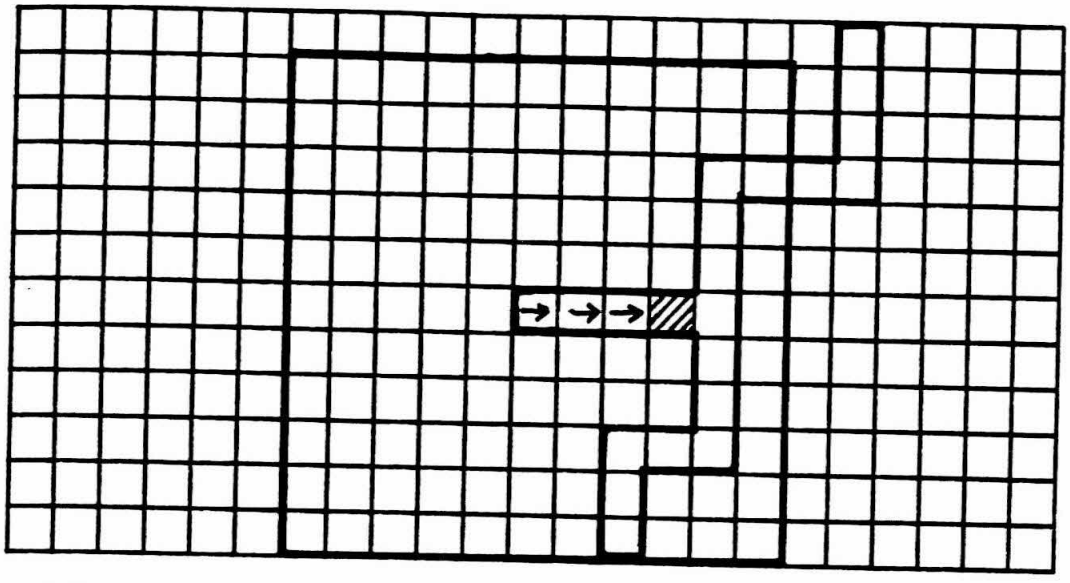


Figure 3.24c: Multi–neighbor Pixel Encountered During Spur Removal

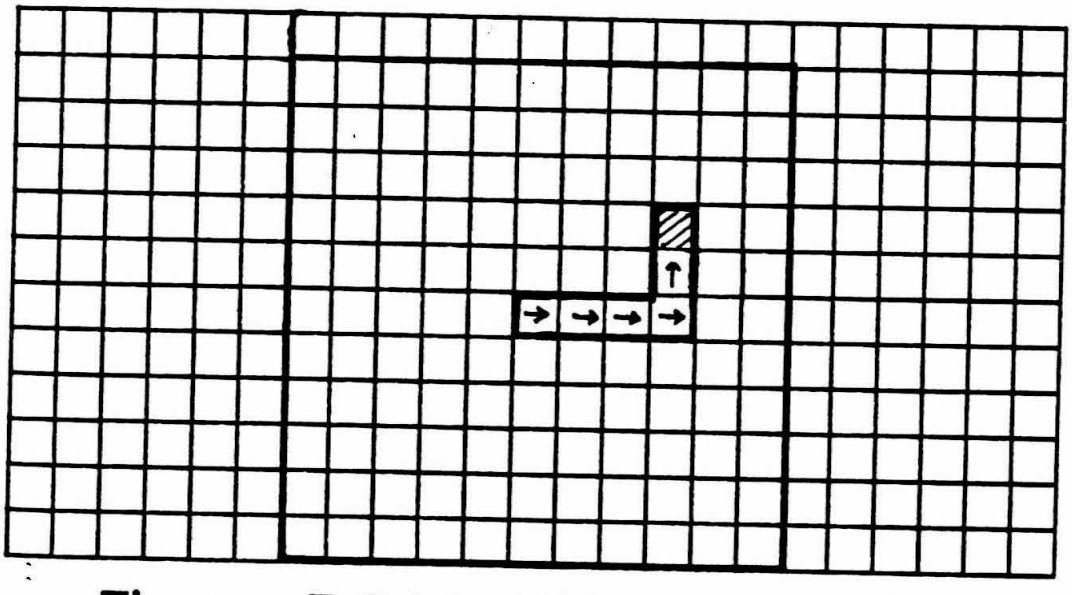


Figure 3.24d: Ridge–end Pixel Encountered During Spur Removal

The SIMULA implementation of the algorithm just described made use of its ability to random access the pixels within the processing window. The correct VLSI implementation would have no need for such random access, as a separate processor would be mapped onto each pixel in the window, with each of these processors connected to its eight neighbor processors. By making use of the pixel values passed to it by its neighbors, each processor can quickly determine if its pixel is a ridge-end or a fork pixel. When the center processor detects that its pixel is indeed a ridge-end, it passes a message to the appropriate one of its neighbors, and the process continues as described above. When a processor elsewhere in the window is passed a message specifying it as being on the ridge currently being followed, it checks to see if its pixel meets one of the termination conditions. If not, it passes the message on to the correct one of its neighbors. If so, based upon which of the termination conditions was encountered, it passes a message back along the chain of processors in the reverse direction previously used, specifying which, if any, pixels should be deleted.

See Figure 3.25 for an example of a thinned image from which spurs have been removed.

The principle "defect" remaining in the image of Figure 3.25 is the occasional bridge between two adjacent ridges. Such bridges are not always spurious, since this type of "H-shaped" ridge pattern does occur naturally, and as a result no attempt was made to remove them. The thinned image is now suitable for the feature extraction processing, where the final data compression will occur. Two approaches to the creation of a compact graph representation of the information in the fingerprint are considered below.

Figure 3.25: Image After Spur Removal

Chapter 4

Ridge Adjacency Graphs

4.1. Introduction

The next step in the fingerprint analysis process is the creation of a graph representation of key information in the print image. Many choices regarding exactly what information should be used are possible. Two are considered in this work -- "ridge adjacency graphs" and "minutiae graphs". This chapter considers the details of the construction and significance of the first of these, the ridge adjacency graphs. In this encoding, each ridge in the print is mapped onto a node in the resulting graph, with links being placed between two nodes if and only if the two corresponding ridges are "adjacent" - i.e. if they have only background space separating them for at least some non-trivial part of their length. This representation for fingerprints was not used to produce the results of this work, but is presented here because the algorithms involved are instructive in demonstrating applications of the neighborhood processing architecture.

4.2. Ridge Numbering

The first step necessary in the construction of the adjacency graph for a given fingerprint image is the assignment of a unique numeric label to

each ridge in the print. The simplest approach to the assignment of these labels is to scan a Small (3x3) window over the image and, as unlabeled ridges are encountered, assign them the next unused ridge number. Only a Small window is needed because the only information necessary to decide the correct ridge number to assign to a pixel are the current ridge numbers (if any) of its 8 immediate neighbors. Though the input to this ridge numbering step (program RIGNUM) is a 1-bit per pixel image, it is first converted to 8-bit per pixel format, so that the storage of values other than 0 or 1 can be used to assign numerical labels to the ridges. An additional complexity is introduced by the fact that, though the ridges are the entities we want to label, the actual labels are carried on each pixel. It is therefore necessary that we determine if any of the neighbors of a given pixel are already labeled before choosing a label for the pixel in question.

The most complex problem appears because we are assuming that the image is being processed in strict raster-scan order. Thus it is possible (and in fact quite likely) that at some point in the labeling process we will encounter an irreconcilable conflict - i.e., a pixel which has neighbors with two or more different ridge numbers. The simplest case in which this occurs is that of the "V-shaped" ridge, as seen in Figure 4.1. Let us assume that, as the image is being scanned raster-wise, we are assigning ridge numbers of "5" to the pixels on the left arm of the V, and "10" to the pixels on the right arm. Of course there is no way to know that both of these arms are part of the same ridge -- at least not until the pixel at the apex of the "V" is reached. It now becomes necessary to assign a ridge number to this pixel. Should it be "5"? Or "10"? And what about the fact that we now have two halves of a ridge which are assigned different labels? Clearly some renumbering of the ridge pixels will be required.

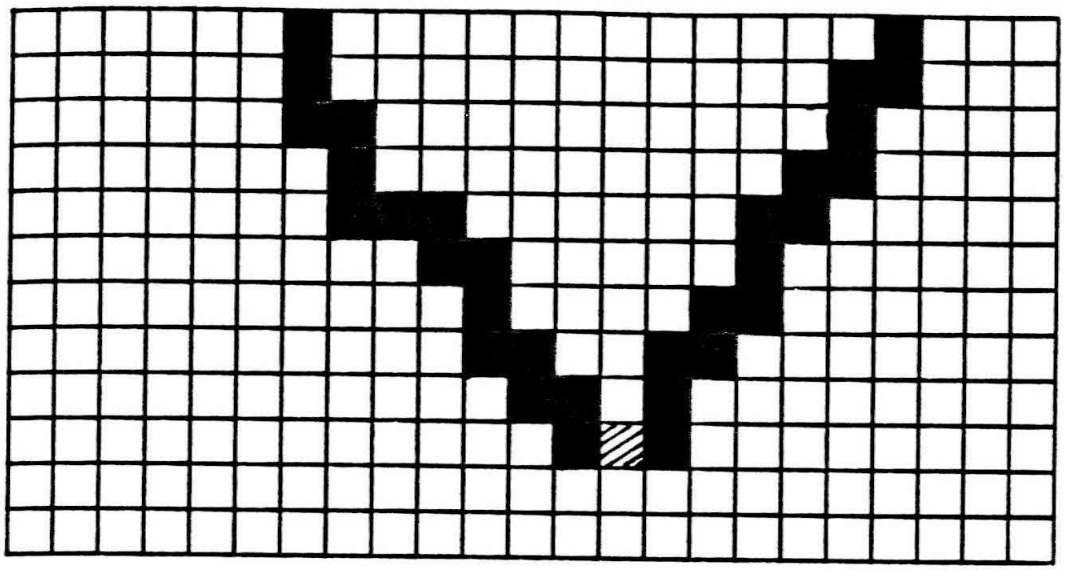


Figure 4.1: "V-shaped" Ridge

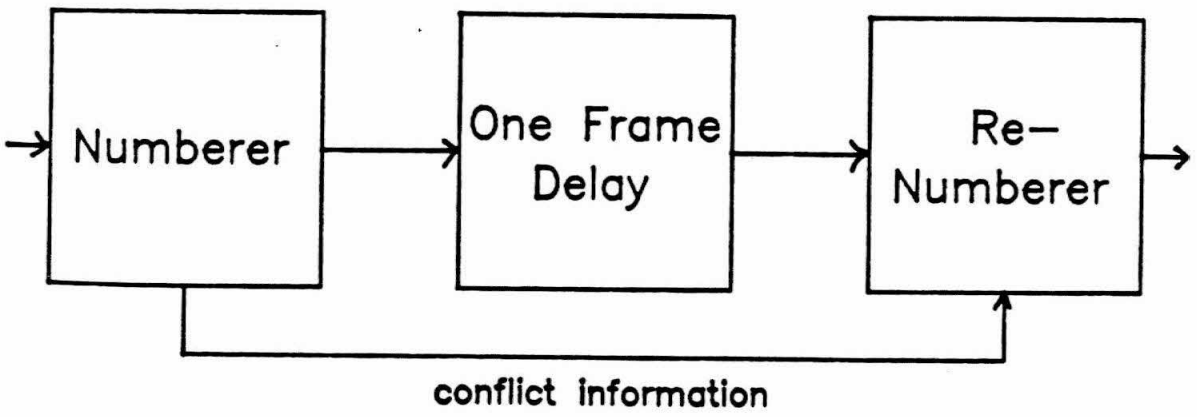


Figure 4.2: Ridge Numbering Units

In order to accomplish this renumbering while still adhering to the pipeline processing architecture it is necessary to introduce two special purpose processing units to the pipeline (see Figure 4.2). Last in the flow sequence is the "Renumbering Unit", which has the responsibility of resolving conflicts, such as that just described, by renumbering the pixels in the image as needed. In this case, the Renumbering Unit could change the label of all pixels labeled "10" to "5", thus removing the inconsistency. In general, the Renumbering Unit will maintain a table (called the "replacement table") indicating what transformation is to be applied to the labels of the ridge pixels as they pass through. The default action is, of course, not to change the label at all.

The information used to produce the replacement table must be acquired by the renumbering unit from a previous stage in the pipeline, specifically from the unit doing the actual assignment of ridge numbers (the "Numberer"). This occurs via the special purpose data connection shown. The algorithm used by the Numberer to label ridges and generate data for the Renumbering Unit is as follows:

```
FOR each position of the 3x3 processing window DO

BEGIN
  IF the center pixel in the window has no labeled neighbors

  THEN Label the center pixel with next available ridge number;

  ELSE BEGIN
       IF the center pixel has one or more labeled neighbors, all of
         which have the same label

       THEN Label the center pixel with the same label as
          its neighbors;

       ELSE BEGIN
            IF the center pixel has 2 or more labeled neighbors
              any two of which do not have the same label

            THEN Label the center pixel with the smallest of
               the labels held by its neighbors, and report
               the details of the conflict encountered to the
               Renumbering Unit over the special data path;
          END;
     END;
END;
```

Thus, as the conflicts in ridge numbering (and therefore the requisite renumberings) are found, they are reported to the Renumbering Unit, which uses this information to construct the replacement table. Since we are assuming that the only internal storage present in the Numberer is the usual 3 scan lines of shift register (for processors using 3x3 windows), it is necessary to introduce the Delay unit between the Numberer and the Renumberer. This is due to the fact that we will not have found all of the conflicts until the entire image has finished passing through the Numberer. If no Delay were present, the Renumbering Unit would be attempting to renumber parts of the image before all details of the necessary transformations were known. Thus, we introduce a delay of one image time into the pipeline, so as to allow all conflicts in an image to be found and

sent to the Renumbering Unit before actual renumbering of that image commences. Of course, while the Renumbering unit is being sent conflict information about the image currently being numbered, it must be in the process of renumbering the preceding image frame. Thus we require double buffering of the replacement table in the Renumbering Unit -- the ability to input data for one table, while simultaneously making use of the previous version of the table. In addition, we also require that the Renumbering Unit have the ability to follow simple chains of renumbering instructions while creating the replacement table. For example, if an entry already exists in the table specifying that all ridges numbered "7" are to be renumbered with "5", and we then find a conflict causing us to specify that all ridges numbered "5" are to be renumbered "2", it is necessary that the 7→5 rule be modified also, to now read 7→2.

The ridge numbering system as described considers all branches of a ridge (i.e. parts of a ridge separated by forks) as belonging to the same ridge. If desired, it would be straightforward to modify the algorithm so that such branches would be considered separate ridges, and assigned unique labels. The simulation of the ridge numbering process was implemented to treat all branches of a ridge as having the same label. The result of ridge numbering a typical (manually thinned) fingerprint image can be seen in Figure 4.3, where the ridge numbers have been added next to their respective ridges.

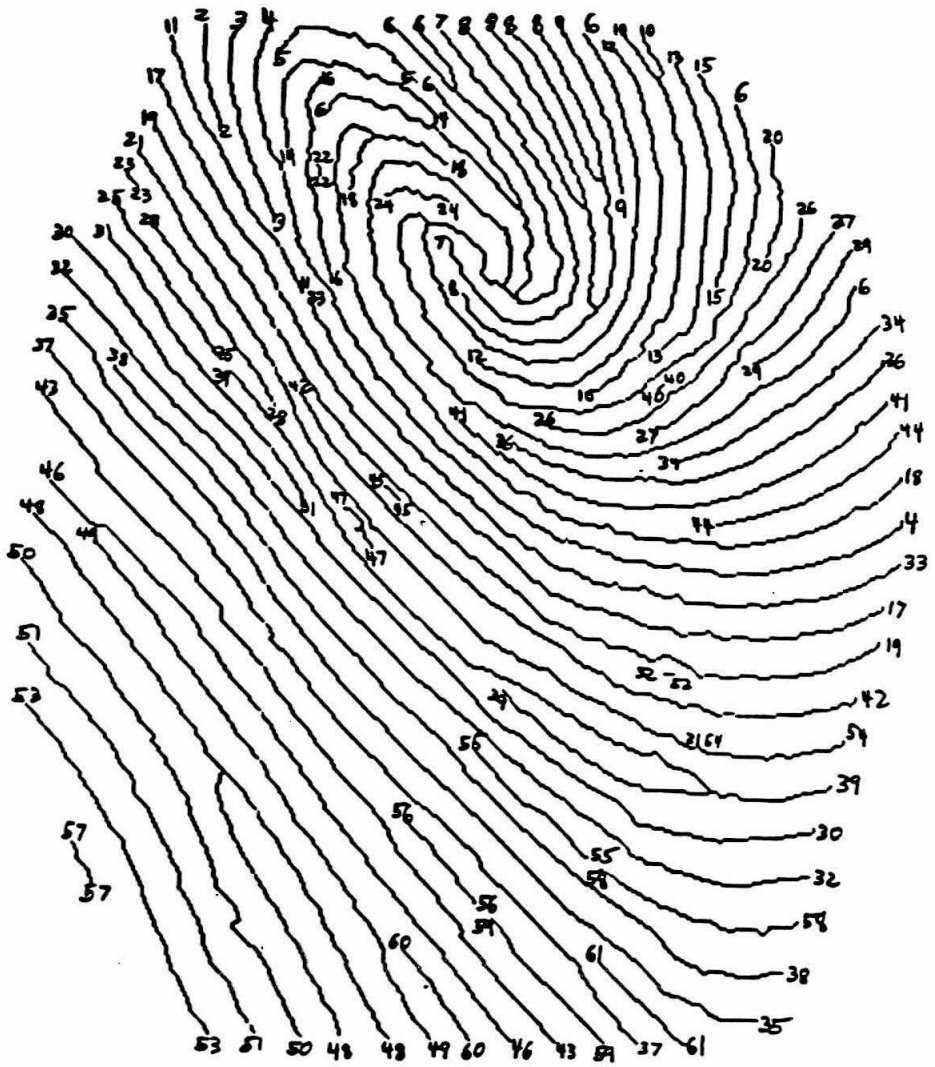


Figure 4.3: Thinned Fingerprint With Ridgenumbers Indicated (manually thinned)

4.3. Ridge Adjacency

As was mentioned above, the adjacency graph representation for a fingerprint maps each ridge in the print onto a node in the graph, and places a link between the two nodes if the corresponding ridges are anywhere adjacent. Figure 4.4a is a fragment of a fingerprint ridge pattern. As examples, ridges A and C are adjacent, as are ridges E and F. Ridges B and G are not adjacent, nor are E and C. The corresponding adjacency graph for the print fragment is shown in Figure 4.4b. The motivation for the choice of this representation stems from several areas. Perhaps most important is the fact that such a representation is purely topological -- i.e. no distance measurements are used. Thus the representation is very much immune to the type of plastic distortions that can be encountered in fingerprint images. This immunity to distortion will be demonstrated in a later section. In addition, by encoding the fingerprint pattern in terms of ridge adjacency we are in many ways imitating the methods used by human fingerprint experts to describe and compare details of prints. Human experts will make statements such as "we have here a short, curved ridge which is next to a long ridge ending in a bifurcation". Rather than mention distances, use is made of the natural coordinate system of the fingerprint: namely the large numbers of adjacent, roughly parallel ridges.

Of course, by limiting ourselves to making statements only about immediate adjacency of ridges, we are imposing a somewhat arbitrary restriction. Indeed it is certainly meaningful to incorporate into our graph structure the fact that in Figure 4.4a not only is ridge D adjacent to ridge B and ridge B adjacent to ridge E, but also that ridge D "overlaps ridge E, but at a distance of 2 ridge spacings". Such an encoding makes even better use

of the natural coordinate system. In fact, this process can be continued almost arbitrarily far. We can say for example that ridge F is also adjacent to D, because they in some sense overlap. The only limitation on how "far" (measured in terms of number of ridge spacings) two ridges can be apart and still be considered to overlap is that our intuitive notion of when two ridges do overlap begins to break down when they are far apart, particularly in the presence of significant ridge curvature.

It is instructive to consider what the situation would be like if all ridges were straight line segments, rather than being arbitrarily curved. We would then have a picture similar to that of Figure 4.5. It now becomes much simpler to think in terms of "overlap" of ridges. In fact, if we consider each ridge to be mapped onto a segment of the real line, then we may say that two ridges overlap if and only if the segments of the real line they represent have a non-zero intersection. Thus ridges T and U overlap, as do X and Y. W and Z do not overlap. Note that we apply no restriction here as to the "distance" between ridges. So, ridges T and Y overlap, even though several other ridges are "between" them.

4.4. Interval and Circular-Arc Graphs

The arrangement of ridges (segments) in Figure 4.5 and the graph that would result is identical to what is known as an "interval graph". Specifically, given a set of intervals on the real line, if we map each of the intervals onto the node of a graph, and connect two nodes if and only if their corresponding intervals overlap, the resulting graph is an interval graph. The invention and study of interval graphs was motivated by a

biological application concerning the fine structure of genes [Benzer59]. In particular, the problem was to decide whether or not the subelements of the genes are linked together in a linear fashion. For certain microorganisms there exist both a standard form and mutants. The mutants result from the standard form by alteration of some connected part of the genetic structure. It can be determined experimentally whether or not the blemished parts of two mutant genes intersect. Given a large number of such mutants, together with the information about when blemished portions of pairs of mutants intersect, the goal is to determine whether this information is compatible with a linear model of the gene [Fulkerson65]. In other words, given that we represent the blemished, possibly overlapping sections of the genes as intervals, if the resulting graph is indeed an interval graph, then the data is consistent with a linear model for the gene.

Since their invention, interval graphs have been applied to a variety of areas, ranging far afield from genetics. For example, in a recent paper [Ohtsuki79], interval graphs are used to generate efficient MOS VLSI implementations of logic equations by simplifying the process of optimum placement of gates along wiring tracks. In addition to their application to diverse research areas, interval graphs have been well studied from a more theoretical point of view - see [Gilmore64] and [Lekkerkerker62].

Interval graphs have many properties that distinguish them from more general graphs. Perhaps most important and useful of these is the fact that algorithms are known that allow the determination in linear time of whether or not two given interval graphs are isomorphic [Lueker79]. This is in remarkable contrast to the situation for general graphs, where

determination of isomorphism is quite difficult (actually the problem is known to be in NP, but has yet to be proven to be in either P or to be NP-complete [Lubiw81] [Johnson81]). Determination of strict isomorphism is not practical as a means for comparison of graphs representing fingerprint images. This is because the definition of isomorphism is too restrictive, in that two graphs are either isomorphic or not -- there is no middle ground. For the purposes of fingerprint comparison, we must consider "fuzzier" versions of isomorphism in order to gain the requisite immunity to noise and distortions.

Returning to the construction of ridge adjacency graphs for fingerprint images, we see that if we allow our definition of adjacency to include ridges that are not immediately next to one another, the resulting graph will be similar to an interval graph. Unfortunately, the curving nature of typical fingerprint ridges prevents us from discussing overlap of ridges spaced apart by arbitrary distances. Thus we must set a limit for the maximum "distance" (measured in ridge spacings) allowable between two ridges said to be adjacent. If this adjacency threshold is set to one, we have the simple immediate adjacency graphs discussed initially.

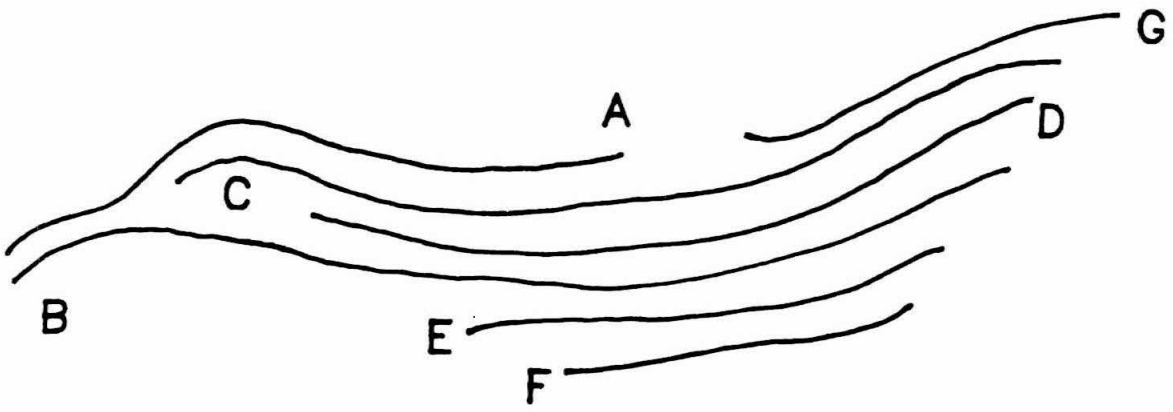


Figure 4.4a: Typical Ridge Fragments

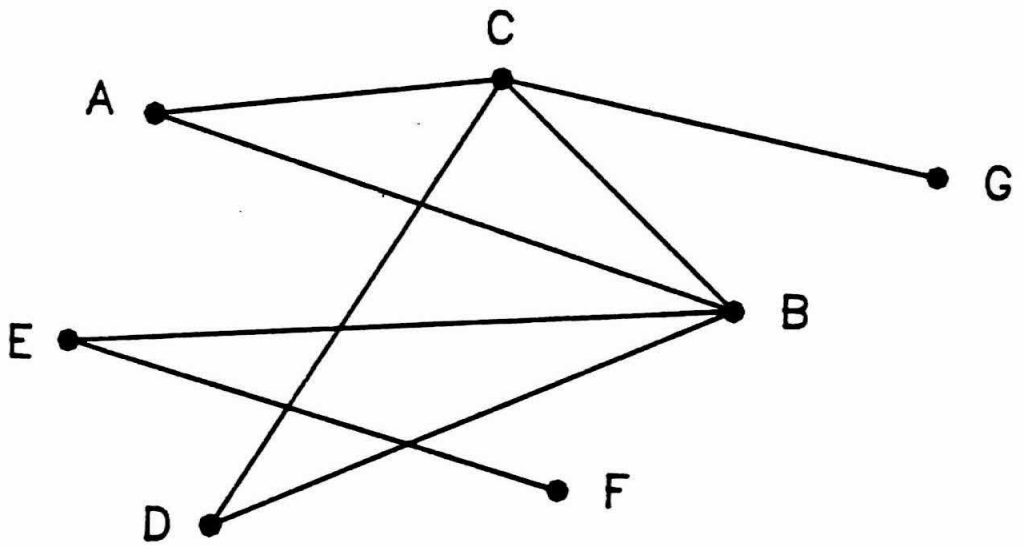


Figure 4.4b: Adjacency Graph

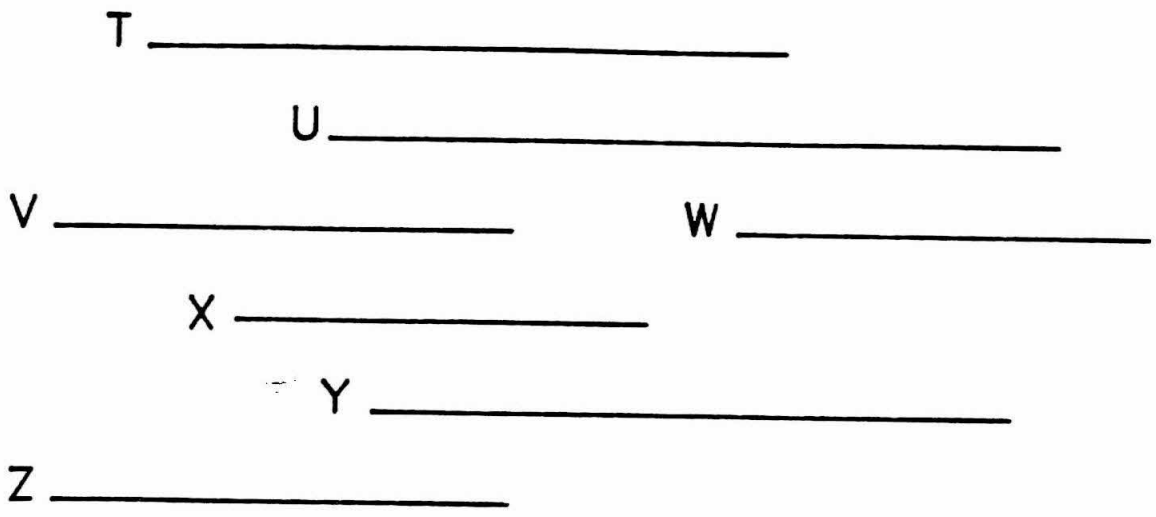


Figure 4.5: Some Straight Ridges
(encode as interval graph)

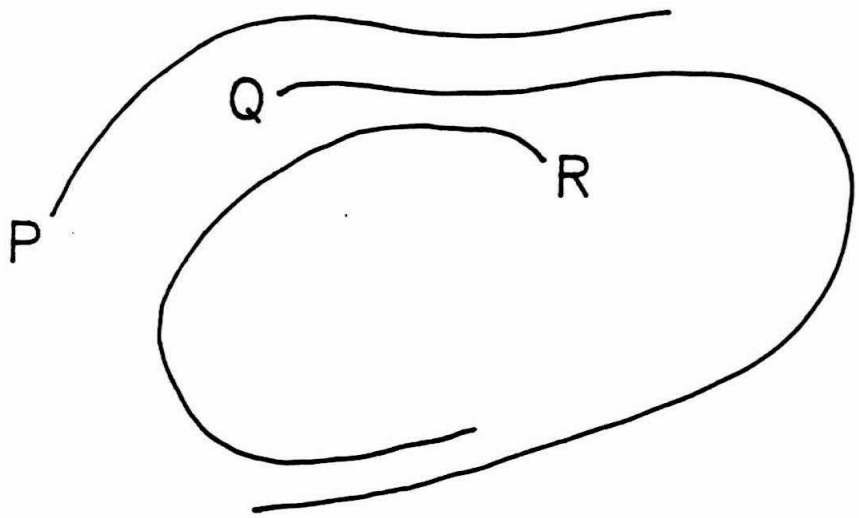


Figure 4.6: Some Very Curved Ridges
(encode as circular–arc graph)

An additional complication is introduced by the possible ridge configuration shown in Figure 4.6. The same ridge curvature which requires that we use a small finite value of the above mentioned threshold, also allows for a situation that is impossible in a standard interval graph, namely a pair of ridges that overlap in two non-contiguous regions. Though the ridge configuration shown is rather pathological and not likely to often occur in actual fingerprints, the implications of such an occurrence must be considered. A generalization of interval graphs known as "circular-arc graphs" exists, and is relevant in this situation. A circular-arc graph is produced according to almost the identical rules used for interval graphs, with the exception that the intervals involved are now arcs around the circumference of a circle, rather than intervals on the real line. The configuration seen in Figure 4.6 is quite legal for circular-arc graphs. Circular-arc graphs are not as well studied as interval graphs, though some work has been done in the area [Tucker] [Tucker74] [Garey80]. In general, the time complexity of operations on circular-arc graphs is greater than for the same operation on interval graphs, despite their apparent similarity [Johnson82]. It should be noted that interval graphs are mentioned here as analogies only - the results derived both for them and for circular arc graphs cannot be directly applied to adjacency graph structures.

4.5. Determining Adjacency

The adjacency graphs used to describe fingerprints in this work were all derived on the basis of immediately neighboring ridges, in order to maintain compatibility with the desired neighborhood processor

architecture. The fundamental operation used to determine ridge adjacency is that of "growing" a vector from a given point on a ridge in a direction approximately perpendicular to the local slope of the ridge at that point. As the growth of this vector proceeds, we are constantly checking for a "collision" with any other ridge. Such a collision would indicate adjacency of the ridges in question. Figure 4.8 shows a number of vectors being grown in the manner described. The vectors can be grown from ridge points as the points are encountered while the image is being scanned raster-wise. Note that although there are two possible perpendicular directions for vector growth from each ridge pixel, only one of these directions need be used. In the figure only the more "downward" of the two vectors has been grown.

This method, although guaranteeing that all existing ridge adjacencies will be found, is rather expensive computationally, and generates much redundant information. The basis for a significantly better approach is shown in Figure 4.9. Here we are only growing vectors from the endpoints of each fingerprint ridge, rather than from all the points of the ridge. The fact that ridges are defined to be continuous curves allows us to make this very useful simplification. Unfortunately, as can be seen in the figure, it is now no longer sufficient to grow vectors in only one of the two possible perpendicular directions. With vector growth in one direction only, the adjacencies of the pair AB was not found. Thus the correct approach (shown in Figure 4.10) grows vectors only from the ridge ends, but in both possible directions.

The input image to the ridge adjacency window processor is assumed to be a multi-bit per pixel image, with each pixel location containing a value

identifying the ridge to which the pixel belongs. The output is a list of ridge adjacencies or, in other words, a list of the links in the adjacency graph. Thus it is at this point in the processing pipeline that we use for the last time a neighborhood processor. The remainder of the processing will deal not with images but with graphs. The original input image contained approximately 1.2 million bits of information. The output of the adjacency processor is only a few hundred bits, yet contains sufficient information to permit accurate fingerprint matching.

The determination of adjacency proceeds as follows: For each position of the window on the input image, the center pixel (processor) of the window determines whether or not it is mapped onto a ridge-end pixel. If not, nothing is done. If so, then the slope of the end of the ridge in the immediate area of the ridge pixel is calculated. Two vectors are then grown out from the ridge-end pixel in the two directions that are approximately perpendicular to the slope of the end region of the ridge. The vectors are grown until they encounter either the edge of the window (nothing is done), or another ridge (we report a ridge adjacency pair). Thus the two operations that must be supported by the window processor are slope determination and vector growth, each of which will now be considered in detail.

Fundamental to the concept of ridge adjacency as just described is the notion of "in which direction to look" for a neighboring ridge. Intuitively, the correct direction is in some sense perpendicular to the end of the ridge in question. Yet the nature of the small scale variations in ridge structure makes it essential that more than two or three pixels at the end of the ridge be used in the slope determination procedure. The method chosen

makes use of a number of pixels at the end of the ridge equal to one-half of the size of the window in use. Specifically, for each of the last N pixels along the ridge (not including the ridge-end pixel itself), we compute:

$$\frac{1}{N}\sum_{n=1}^{N}\Delta X_n$$

and

$$\frac{1}{N}\sum_{n=1}^{N}\Delta Y_n$$

where ΔX_n and ΔY_n are the X and Y displacements of the n^{th} point along the ridge measured with respect to the ridge-end pixel. The results are, in fact, the X and Y coordinates of a "mean point", and our slope estimate is simply the slope of the segment connecting this mean point with the ridge-end (see Figure 4.11). So the slope estimate is

$$slope = \frac{\sum_{n=1}^{N}\Delta Y_n}{\sum_{n=1}^{N}\Delta X_n}$$

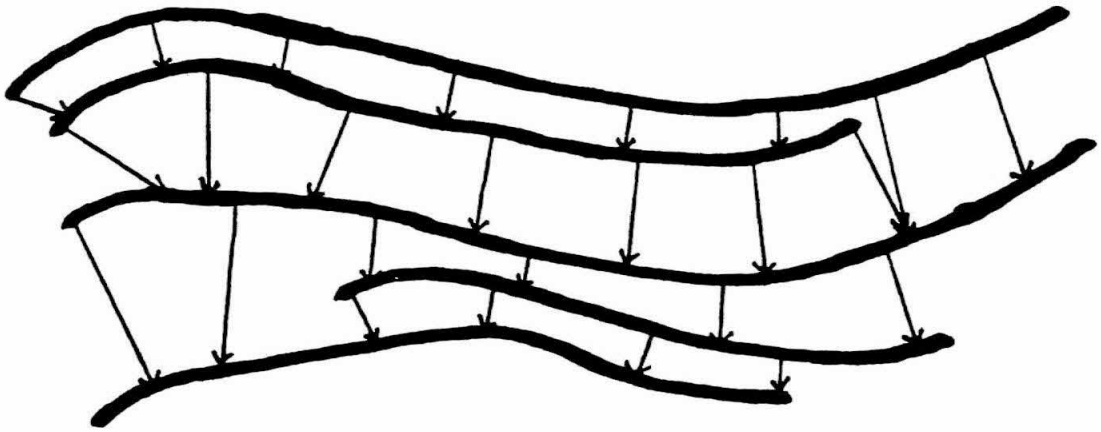


Figure 4.8: Vector Growth

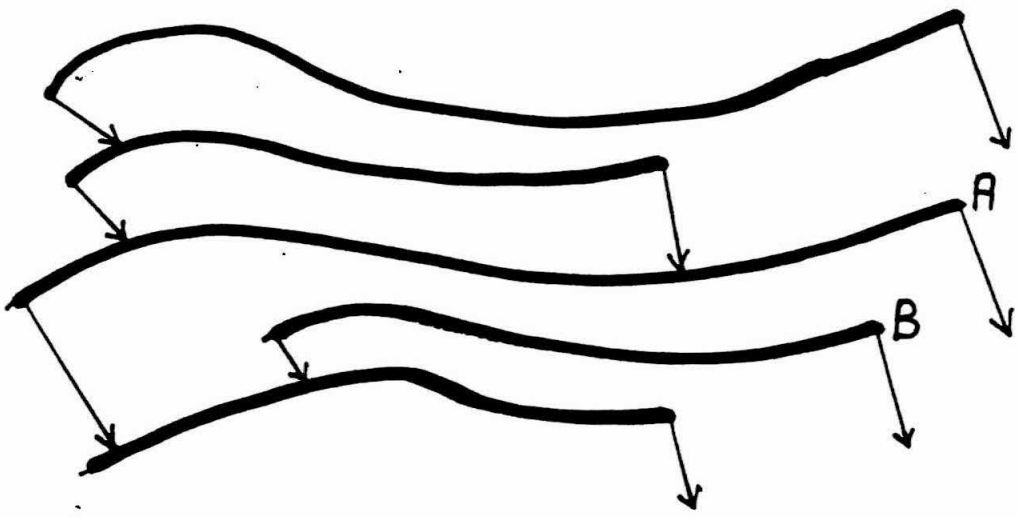


Figure 4.9: Growth Only From Ridge—ends

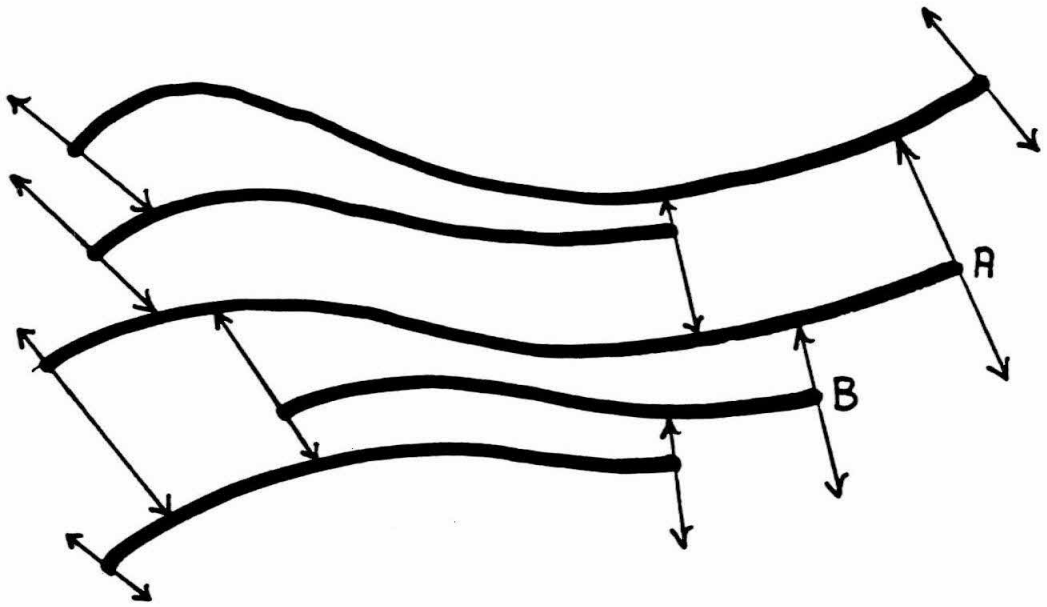


Figure 4.10: Better Approach For Vector Growth

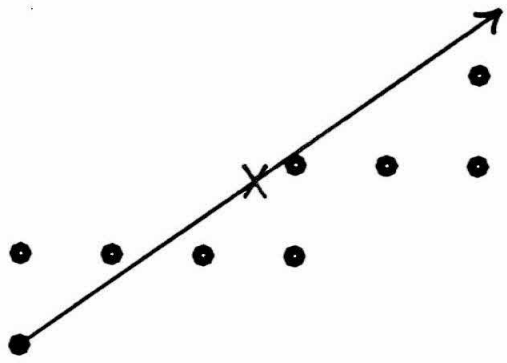


Figure 4.11: Estimation of Ridge Slope

It is not difficult to see how a VLSI implementation would handle the slope computation. Beginning with the processor at the ridge-end pixel, each processor simply passes to its neighbor along the ridge a message containing both information about the coordinates within the window of the processor sending the message, as well as a counter of how many pixels have been traversed. In effect, a "running average" of the coordinates of the pixels traversed is kept. When such a message finally reaches a processor which cannot pass the information along to an appropriate neighbor, due to having reached the edge of the window, sufficient information exists to compute the slope of the ridge end. The slope information can then be sent back to the originating ridge-end pixel to be used to determine the correct directions for vector growth.

The algorithm used to generate the vectors for determining ridge adjacency is the so-called non-symmetric Digital Differential Analyzer (DDA) [Newman73]. Using this approach, one coordinate (X or Y) is identified as the direction of greatest change per step of the vector growth process. This coordinate is then always incremented (or decremented as appropriate) by 1, while to the other coordinate value is added the slope of the desired segment (a fractional value). The resulting value is then rounded to the nearest integer, and the next pixel in the vector is placed at that point. Assuming X is the coordinate of greatest change, we have:

$$Y_0 = Y_{start}$$

$$Y_n = Y_{n-1} + \frac{\Delta Y}{\Delta X}$$

$$Y_{new} = int\,(Y_n + 1/2)$$

The process continues in this manner for as long as desired. The

determination of which of the two coordinates is "changing faster" is made based upon the sign of the slope of the ridge-end, as well as whether the slope value calculated is greater or less than 1. Lines which are closer to vertical than horizontal have Y as the fast-changing variable, and vice versa. If the slope calculated for the vector to be grown (which is of course the reciprocal of the slope of the end of the ridge) is greater than 1, it is inverted before use, so that the fractional increment used in the DDA calculation is always less than one.

The number of bits used to represent the fractional values determines the overall accuracy obtained in the generated vectors. In the software simulation of the vector generation process, three bits were used to represent the desired slope of the vector and two bits were used to represent the fractional part of the slowly changing coordinate. In addition, one other bit is used to specify whether X or Y is the more rapidly changing variable. The use of 3 bits to specify slope gives an angular resolution of approximately eleven degrees in the vector generation process, which proves to be quite adequate. A VLSI implementation of the vector generation process would make use of exactly the same data representation. Thus the center processor in the window, after receiving information on the slope of the ridge-end as described above, would begin the vector growth process by passing messages containing the necessary six bits of state information to the correct two of its neighbors. The remainder of the vector growth would then consist of each processor doing the simple computation described above to determine to which of its neighbors to pass on the updated information. This of course continues until the window edge or another ridge is encountered – such an event generating an "adjacent ridges found" message as part of the output of this pipeline stage.

The process described above was simulated (program GRAPH), and the results of deciding ridge adjacency for a typical fingerprint image (Figure 4.3) can be seen in the graph representation of Figure 4.14. In the representation used for the adjacency graph, the first number on each line represents the label of a particular ridge, while the numbers following the colon are other ridges that were found to be adjacent to the original ridge.

An improved method for determining the adjacency of fingerprint ridges is what we will call "circular growth", and was suggested by Carver Mead. In this method, rather than attempting to determine the slope of a line fit to the last few pixels at the end of each ridge so that vectors can be grown in the two perpendicular directions, we instead "grow" an ever-expanding circle of pixels surrounding the ridge-end pixel. As this circle of pixels contacts neighboring ridges we gain the required adjacency information. Specifically, the algorithm makes use of a Large (15x15) window, as the window must at least span the distance between ridges. As the window is "scanned" across the fingerprint image, the center processor in the array (i.e. the one mapped onto the center pixel of the window) is continually checking to see if the pixel currently positioned there is a ridge-end pixel. If so, the center processor begins the growth of the circular "wave" by sending a message to each of its 8-neighbors, who in turn do the same. If at any point in this process another ridge is encountered, this information is propagated back to the center processor.

This circular growth approach overcomes one of the more serious limitations of the previously described vector growth method, which is the performance in the face of a situation such as that in Figure 4.15. Here we have two ridges which overlap only marginally, if at all. Whether the vector

growth algorithm would determine them to be adjacent depends upon the fine details of the vector generation process, such as limited angular resolution, and thus is not straightforward to predict. In addition, it is likely that rather minor geometric distortions would affect the adjacency determinations. Circular growth on the other hand produces a consistent adjacency decision.

Circular growth is much more consistent with the stated philosophy underlying this work, in that it decides adjacency almost completely based upon pure topological considerations, as contrasted with vector growth which makes use of metrics such as line slope - and has as a result problems with image distortion. As is clear from the above description of the circular growth algorithm, a VLSI implementation would be straightforward, and indeed simpler than that required for the vector growth method.

```
 2:    3  11
 3:    2   4  11
 4:    5   3  14  16  33  11  18
 5:    4  14   6
 6:   16  18  24  15  13  29  10  27   9   7   6   5  12  14  20   8  26  36  **
41  34  40
 7:    6   8
 8:    8   9   7   6
 9:    6   8
10:    6  10  13  12
11:    2   3  17   4
12:    6  10
13:    6  10  15
14:    5  16   6   4
15:    6  13
16:    6  14  22   4  18
17:   11  33  19
18:    6  44  41  16  22   4  18
19:   52  45  42  21  17
20:   26   6  27
21:   42  39  47  28  23  19
22:   16  18
23:   25  30  21
24:    6
25:   23  28  31
26:   20  40   6  27
27:    6  26  29  20
28:   25  39  21
29:    6  27
30:   31  39  32  23
31:   30  25  39
32:   58  55  38  35  30
33:   17   4
34:   36   6
35:   38  61  37  32
36:   34  41   6
37:   61  59  56  43  35
38:   35  55  58  32
39:   31  28  47  30  21  54  39
40:   26   6
41:   18  36   6  44
42:   52  21  45  54  19
43:   56  59  46  37
44:   18  41
45:   42  19
46:   60  43  49  48
47:   39  21
48:   49  48  50  46
49:   48  60  46
50:   48  51
51:   53  50
52:   42  19
53:   51  57
54:   39  42
55:   38  32
56:   43  37
57:   53
58:   38  32
59:   43  37
60:   49  46
61:   37  35
```

Figure 4.14: Derived Adjacency Graph

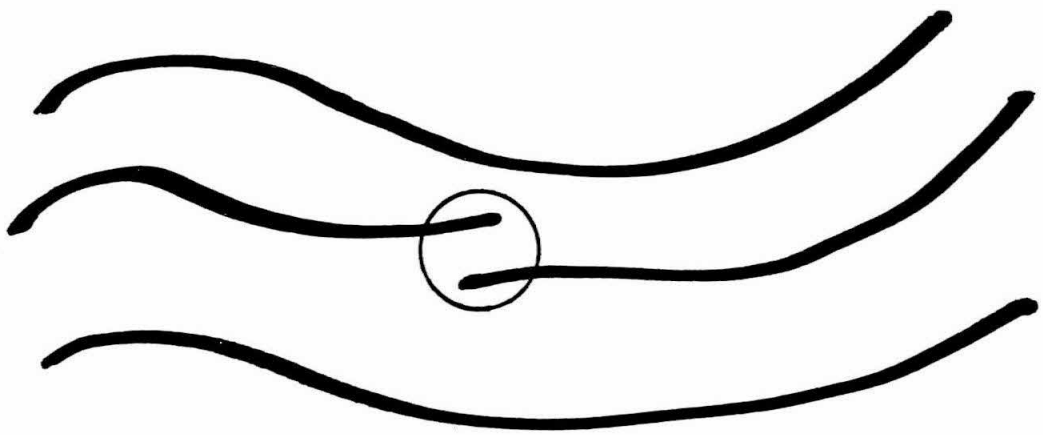


Figure 4.15: Marginal Ridge Overlap

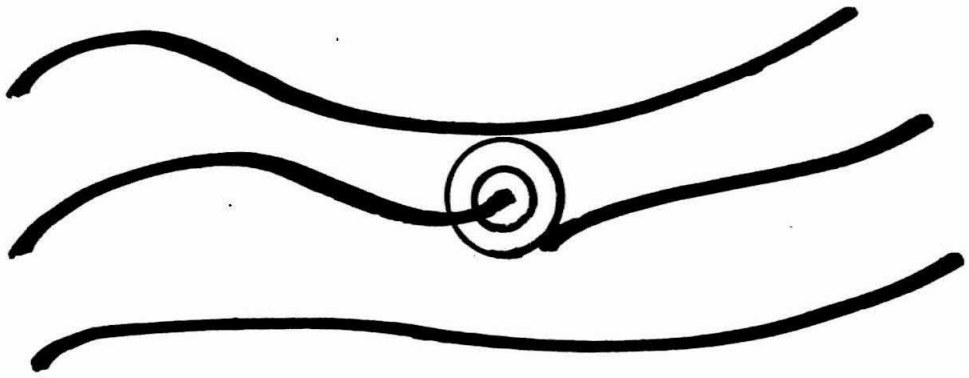


Figure 4.16: Circular Growth

4.6. Immunity of Adjacency Graph Encoding to Geometric Distortion

Central among the possible differences that may be encountered between a stored "reference" fingerprint and a print with which it is being compared are those due to geometric distortions of the ridge patterns. This distortion may be due to a number of causes, the most likely being the use of excessive finger pressure in recording the fingerprint image. As was mentioned elsewhere, acceptable fingerprint images are obtained with finger pressures ranging from several ounces to over ten pounds. Yet the resiliency and elasticity of the finger and the skin can cause problems. For example, if finger pressure is applied in such a way that the angle between the body of the finger and the recording surface is far from normal, distortion of the image will result. See Figure 4.17 for an example of two inked fingerprint impressions from the same finger that show significant differences due to pressure distortion.

Fortunately, much of the pressure distortion seen in inked fingerprint impressions like those of Figure 4.17 is due to variations in the rolling motion used to record such prints. This problem is much less likely to occur with the optical system used for this work, as it merely requires that a finger be placed lightly against a glass surface without rolling. Nevertheless, it is desirable that a fingerprint encoding system be immune to at least local distortions in the print image. Lack of such distortion immunity has been the downfall of many of the proposed automatic recognition systems, particularly those that attempt to make use of global measurements of the position of features within a print.

Both encoding systems described in this work have sufficient immunity to local distortions. The "minutiae graph" approach (described in Chapter

5) obtains this immunity by not making use of distance measurements between features that are separated by more than a specified small distance. In this way, severe distortions or disruptions in one area of a print will have minimal effect on the information contained in other areas. The "adjacency graph" approach has outstanding immunity to even very severe distortions, whether they be global or local, due to the topological nature of the adjacency graph encoding. In other words, the concept of distance is not used at all - rather we speak of a ridge being "adjacent to" other ridges. Adjacency of ridges is a property of fingerprint images which remains unaffected by any reasonable geometric distortion.

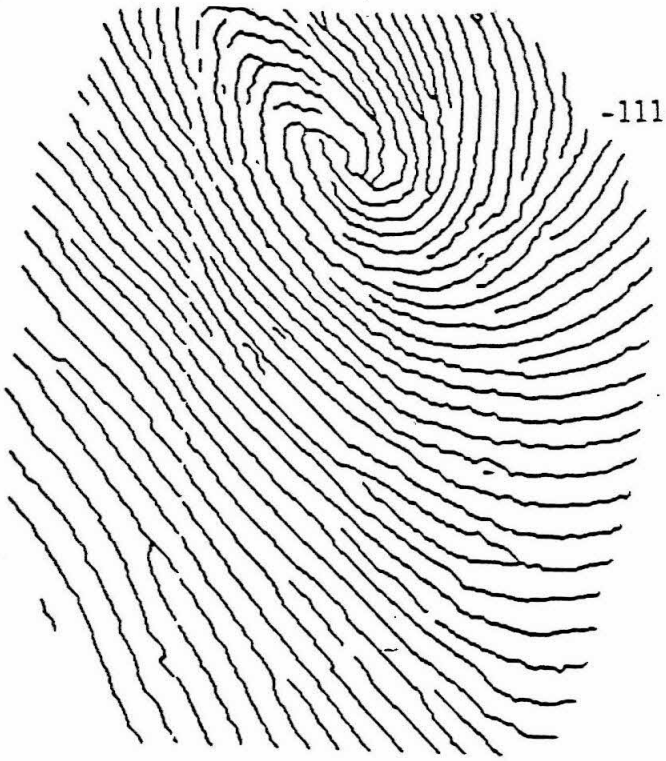
In order to demonstrate the excellent distortion immunity of the adjacency graph encoding, a fingerprint image was artificially distorted. Figure 4.18 shows a typical thinned fingerprint image, along with the same image after having been transformed by the equations

$$x := x + 5 \sin\frac{x}{10}$$

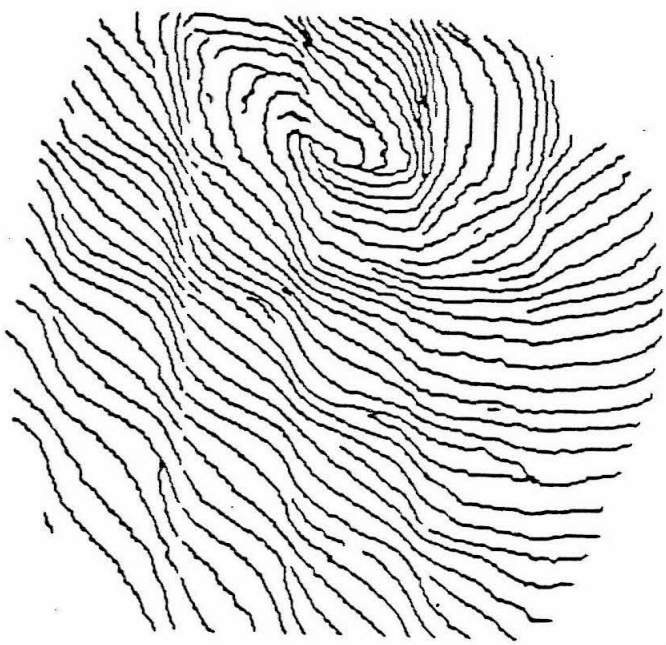
$$y := 0.8y + 3 \cos\frac{y}{13}$$

Figure 4.19 is a representation of the adjacency graphs derived from the undistorted and distorted prints. The number at the beginning of each line is the number given to a ridge in the fingerprint. Following the colon are the numbers of all ridges which are connected (i.e. adjacent) to the first ridge. The graph derived from the original print has 61 nodes (ridges), and 104 links (adjacencies). The distortion process resulted in a net change to the graph of only 3 links removed and 1 link added. This is excellent performance, particularly since the distortions used are greater than any likely to be encountered with the present optical input system.

Figure 4.17: Pressure Distortion
Comparison of two inked fingerprints taken from the same finger using different amounts of pressure.
[Moenssens71]



ORIGINAL



DISTORTED

Figure 4.18: Original and Artificially Distorted Fingerprints

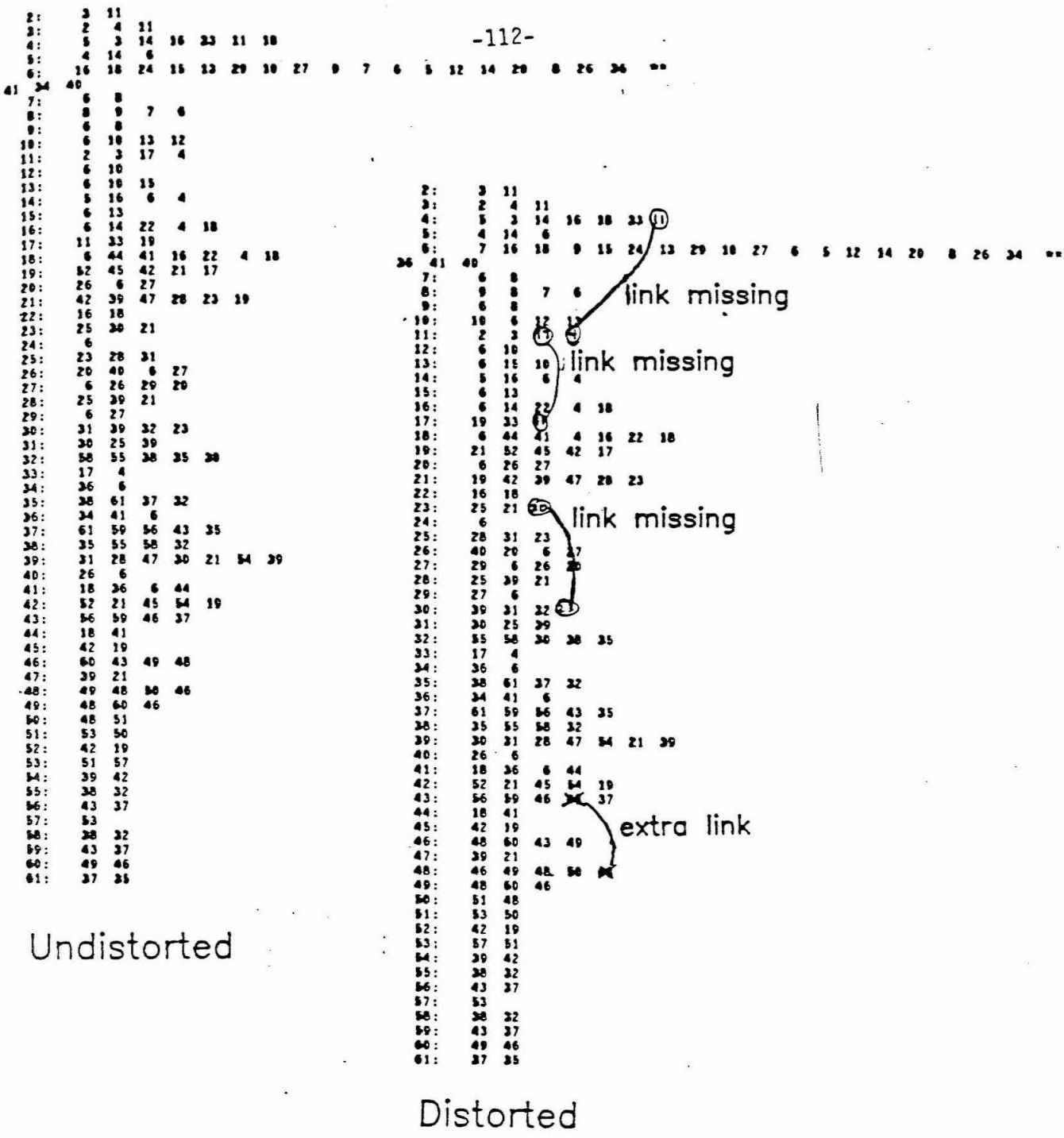


Figure 4.19: Adjacency Graphs for Undistorted and Distorted Prints

In addition to immunity to geometric distortions, we must also consider the behavior of any chosen encoding system when faced with the noise typically present in a fingerprint image. One of the most common manifestations of this noise will be small, random breaks in ridges, where no such breaks exist in the original fingerprint. Note that it is essentially impossible to remove such accidental breaks with a post-processor as we do with pores, as some breaks occur naturally in fingerprints, and there is no way to distinguish the noise-induced from the genuine breaks. With the adjacency encoding system just described, an extra break added to a ridge will result in the two ridge fragments being labeled as individual ridges and therefore being mapped onto two distinct nodes in the graph representation. Similarly, the links which would have been attached to the node representing the original, unbroken ridge will now be split between the two new nodes which have replaced the single node in the graph. The method ultimately used to compare the resulting graphs must therefore have a reasonable degree of immunity to such splitting of nodes.

Chapter 5

Feature Extraction and Minutia Graphs

5.1. Feature Extraction

Comparison of fingerprints by human experts is not done by the purely topological Henry system used for classification of ten-finger sets, but rather by a system based upon descriptions of the relationships of the low-level features of the prints, known as minutiae. The graph encoding to be described also makes use of selected ones of these features, along with the distances separating them.

The specific features used in the minutia graphs are ridge-ends and ridge-forks (bifurcations). These were chosen because they are the most fundamental of the entire class of minutiae, and in fact all of the others (ridge breaks, trifurcations, etc.) can be considered as combinations of these two. Though it is necessary for the feature extraction process to separately identify forks and ridge-ends, beyond this stage no attempt is made to distinguish between them, due to their inherent similarity as was explained earlier. Doing so would seriously compromise the noise immunity of the resultant encoding. Thus the basic step involved in the process of creating a minutia graph representation for a given fingerprint image is the identification and extraction of the ridge-ends and forks from a properly processed and thinned image.

The identification of exactly which pixels are to be considered forks was discussed in detail in the section on spur removal. To summarize, a "fork pixel" is any pixel with 3, 4, or 5 neighbors, that has at least 6 on-to-off or off-to-on transitions around its immediate neighborhood. See Figure 3.22b for examples. The definition of a ridge-end pixel is somewhat simpler, as a ridge-end pixel is simply any pixel in the image with exactly one neighbor (Figure 3.22a). It is thus straightforward to produce a final stage in our image processing pipeline which is responsible for extracting the row and column coordinates of each of the features in the image. The input to this stage is the binary image output from the spur removal process, and it produces as a result a list of ordered pairs of row and column coordinates. Note that this is truly the final stage in the pipeline, for it is at this point that the data are finally converted to a form different from that of a sampled image. It is also at this point that the desired data compression has occurred, for what was originally some sixteen-million bits of information has now been reduced to approximately one-hundred ordered pairs of small integers - a compression by four orders of magnitude.

The architecture of the processing stage responsible for this feature extraction (simulated by the program 'FEATUR') is straightforward. A 3x3 window, with a limited amount of processing power at each cell is required. The ridge-end detection function requires only that the center cell be able to ascertain when it has exactly one neighbor. We will assume that the center cell receives a single bit from each of the peripheral cells in the window indicating whether or not that neighbor is "on". The sum of these single bit quantities will then be the "number of neighbors" (NON) parameter needed.

Fork detection also makes use of the calculated NON value. By the previous definition, in order for the center pixel to be a potential fork, we must have NON in the range of 3 through 5, inclusive. Given this, it must then be determined how many transitions from on to off and vice versa occur in the surrounding eight pixels. This is easily done in a manner similar to the NON calculation: Each neighbor pixel (processor) reports a change (a value of 1) if it is different than the next pixel around the neighborhood in say the clockwise direction. It reports no change (a value of 0) if its state is the same as that of the next pixel. The sum of these values (computed by the center processor) is the "number of changes" (NOC) in the neighborhood of the center pixel. If NOC > 6, then our candidate fork pixel is indeed a valid fork pixel.

The result of this feature extraction process can be seen in Figure 5.1, in which the location of each identified feature has been indicated by a dot in an image, as well as in Figure 5.2, which is a partial list of the actual output of the feature extraction stage, namely the coordinates of the features found. The fingerprint from which these data are derived is shown in Figure 5.3.

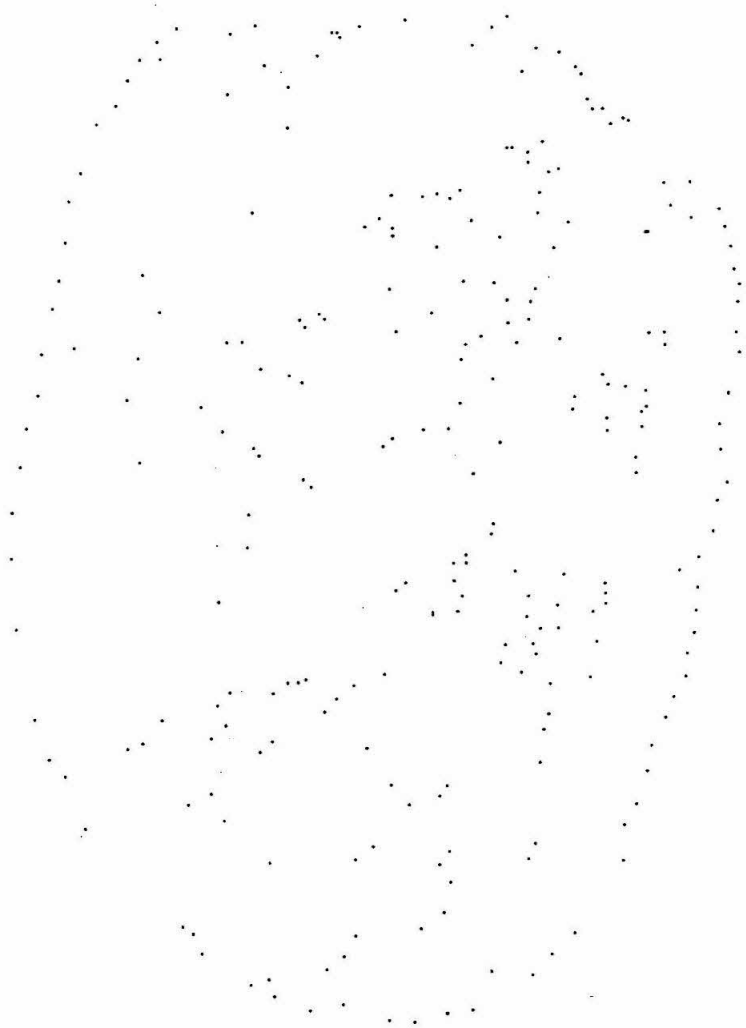


Figure 5.1: Locations of Features

#	Row	Col
1	12	244
2	13	121
3	14	183
4	17	162
5	23	263
6	26	171
7	28	147
8	35	288
9	39	290
10	42	170
11	45	297
12	51	270
13	53	256
14	53	258
15	55	264
16	59	264
17	62	276
18	63	272
19	69	237
20	69	317
21	70	210
22	70	228
23	71	222
24	71	268
25	72	233
26	75	155
27	78	319

Figure 5.2: Typical Feature Coordinates

Figure 5.3: The Original Fingerprint

When looking at Figure 5.1, the "halo" surrounding the main body of features is quite obvious. It is made up of features which are really spurious. They consist of ridge-ends generated by the fact that we are examining a limited section of the fingerprint, with the resultant cut-off at the edges of the image. The inclusion of such features is undesirable, as they are not present on the finger itself, and their position will vary amongst prints taken from the same finger as the pressure and angle used to record the print changes. Removal of these features is accomplished by a modification to the feature extraction processing stage which serves to identify such "edge features".

The size of the window used is increased from the minimum 3x3 window necessary for the basic feature identification process just described, to a size that is substantially larger than the largest inter-ridge spacing expected. The basis for the identification and removal of the spurious features is the observation that essentially all of the features which appear at the outer edges of the print are in fact spurious. The definition of "being at the edge" used in the algorithm is simply that at least a ninety-degree sector of the extended neighborhood of the feature in question contains no "on" (i.e. ridge) pixels. For sufficiently large neighborhood sizes, any such pixel cannot be in the interior of the print. See Figure 5.4 for examples of typical situations.

In order to remain consistent with the standard neighborhood processor architecture in use, it is desirable to make use of a standard square neighborhood, rather than the idealized round neighborhoods implied above. Figure 5.5 is a representation of the approach used. A square neighborhood of size 2L+1 (L an integer) is divided into eight

overlapping areas, known as "half-edge regions". If at least two half-edge regions are found which do not contain any ridge pixels, than the feature at the center of the region is rejected as being too close to the edge of the fingerprint, and therefore spurious. The parameters L and W in Figure 5.5 are chosen based upon the observation that features closer to the edge of the print than L-W ($\frac{\sqrt{2}}{2}$(L-W) worst case) will be deleted. In addition, both L and W should be greater than the largest ridge spacing. Values ranging from L=8, W=4 to L=16, W=10 have been employed, with excellent results, and very little sensitivity to the exact values chosen. Figure 5.6 is the set of extracted features after removal of the spurious ones, and can be compared with Figure 5.1 in order to see that indeed only the spurious features at the image edge have been removed.

Figure 5.4: Feature Neighborhoods

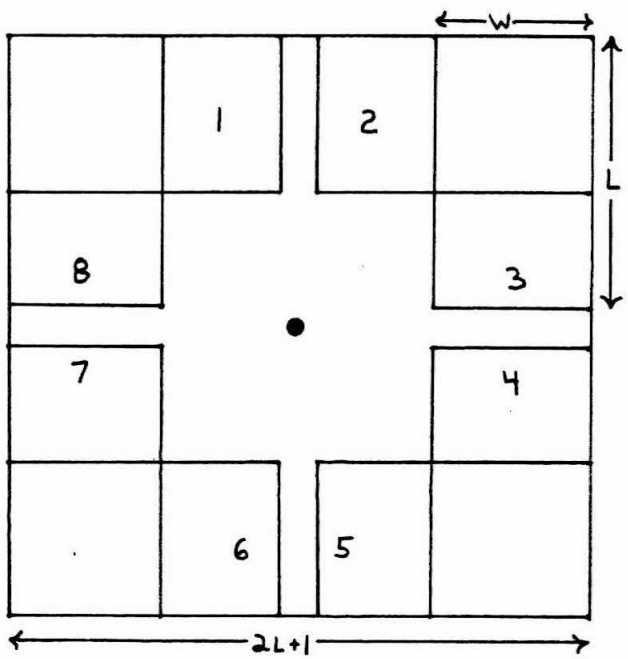


Figure 5.5: The "Half-Edge Regions"

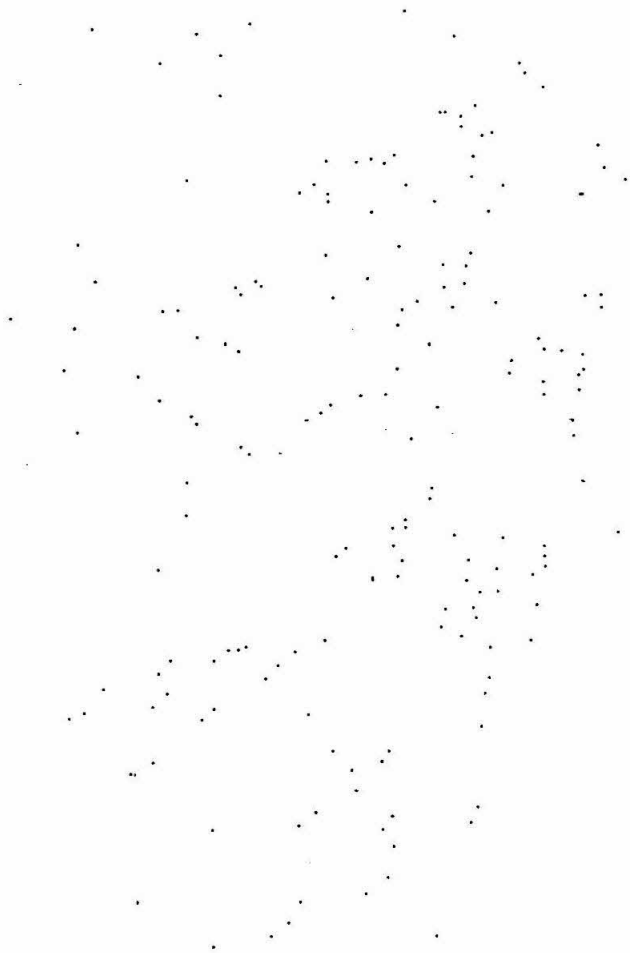


Figure 5.6: Valid Feature Locations

5.2. Minutia Graphs

After identification of the features in a particular print is complete, the next task is creation of the appropriate representation for the information contained in the print. The previously discussed adjacency graphs are one such representation. Minutia graphs are a second, and in many ways more effective representation. Where in the adjacency graph each node was mapped onto a single ridge in the original fingerprint, with minutia graphs each node corresponds to a single feature (ridge-end or fork) in the print. The links between the nodes are created based upon the "distance" between any two nodes (features). Distance is used here in a general sense, as many specific distance definitions may be used and will be discussed below. Two nodes in the minutia graph are connected by an edge if and only if the "distance" between them is less than a fixed value (the "local-region size").

One obvious choice as a definition of "distance" would simply be to count the number of intervening ridges between any two features under consideration. For example, the ridge-ends in Figure 5.7 can be said to be at a distance of "3" from each other, as there are clearly three intervening ridges. This method of specifying distance is quite appealing, as it makes use of what one might call the "natural coordinate system" of any fingerprint, namely the ridge pattern. Unfortunately, this coordinate system is really rather one-dimensional, since the ridges do not form a "cross-hatch" pattern, but are instead collections of roughly parallel curves. As a result, situations such as that shown in Figure 5.8 occur, where the line connecting two features is essentially parallel to the local direction of the ridges, and determination of a unique (or accurate) value for the "distance" becomes impossible.

The solution is to make use of some form of geometric distance between the features, measured in, for example, units of pixels. Because the inter-ridge spacing in a fingerprint can be considered to be roughly constant, there is considerable analogy between this approach and the ridge-counting system. The primary distinction between the two methods is in the area of immunity to plastic distortion of the fingerprint image. As has been discussed, such distortion is often encountered, particularly in prints obtained by inked transfer. The ridge-counting definition of distance is more immune to such distortions, since the number of intervening ridges is (at least in the ideal case) not affected by distortion. However, in the minutia graph encoding, distances are only used between features that are in the same "local region", i.e. that are close to each other in the print. As a result, immunity to most global distortions remains excellent. Similarly, any distortions that are locally severe will affect only a limited part of the data structure, and therefore not greatly interfere with matching of the prints.

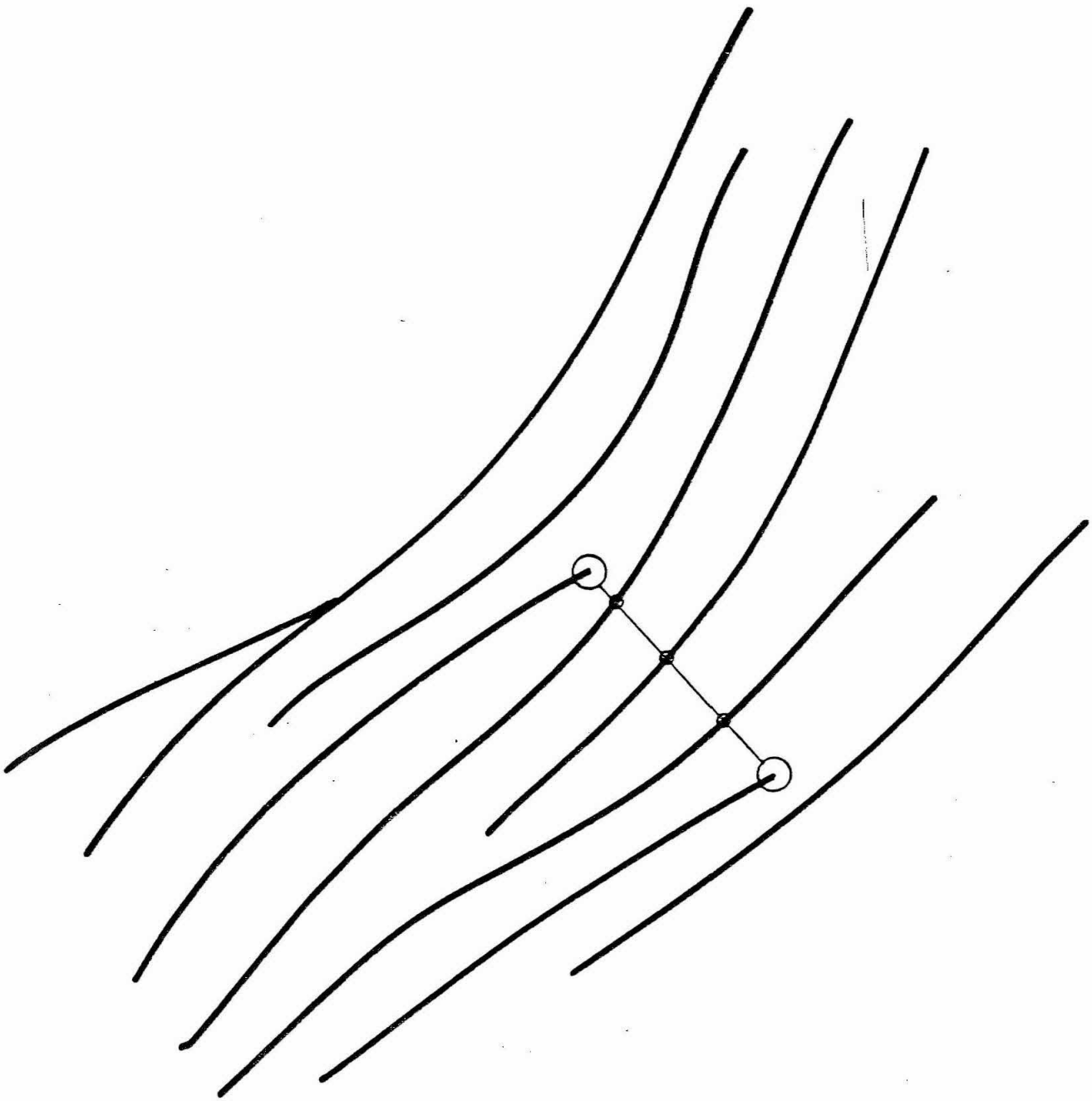


Figure 5.7: Ridge Counting

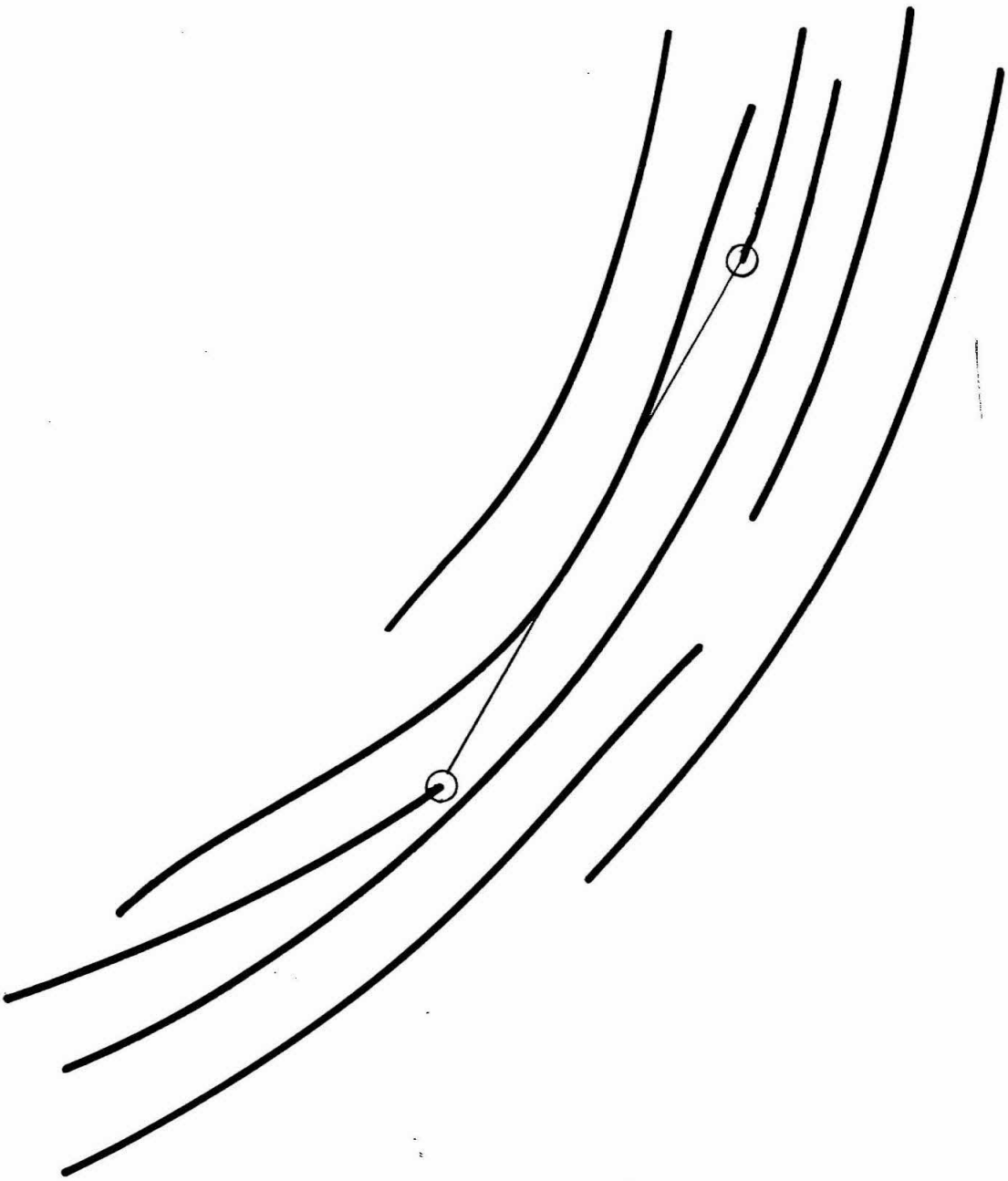


Figure 5.8: Ridge Counting Ambiguity

It should be noted that the method used by human fingerprint experts to describe the similarities between two prints also makes much use of inter-feature distances. It is in fact quite common for such an expert when testifying in court to make a statement describing two features in terms of the number of intervening ridges. Thus it can be seen that the minutia graph representation is quite consistent with the stated goals for a good fingerprint representation, in that it makes use primarily of local information, has good distortion immunity, and bears certain similarities to methods successfully in use by humans.

As a further refinement of the minutia graph concept, rather than computing the actual geometric distance between two features in question, we simply compute and keep the Δx and Δy values (i.e. x_2-x_1 and y_2-y_1), complete with sign. Thus we are in effect measuring not only the distance between the two features, but also the angle (slope) of the line connecting them. This additional information proves useful in the comparison process, as it serves to further "prune" the search tree when doing node by node matching of the minutia graphs. An added benefit is the simplicity of computing Δx and Δy rather than the true distance, as no square root is involved. See Figure 5.9 for a typical pair of features with Δx and Δy indicated. Use of Δx and Δy rather than actual distance results in the matching procedure not being independent of the relative rotation of the two fingerprint images. This is acceptable however, as we are assuming throughout this work that the images are approximately aligned before we begin (as would be the case with, for example, fingerprint-controlled security systems).

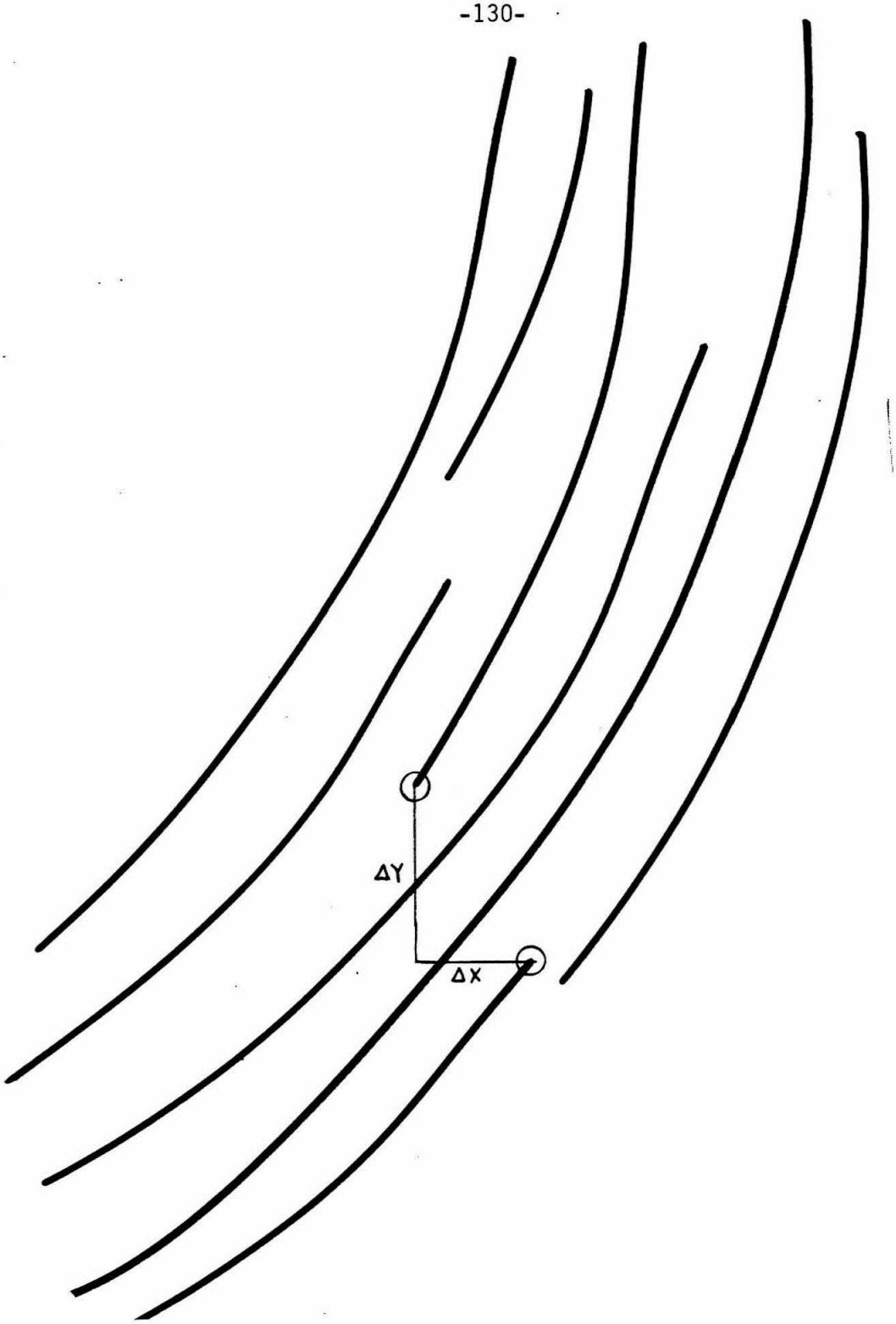


Figure 5.9: Measuring Inter-Feature Distances

One of the most important factors in the selection of a print representation is the question of noise immunity - i.e., how sensitive is the representation to the inevitable broken ridges, dark patches, and other disturbance of the ridge pattern? It is in this area that the minutia graph approach shows a significant advantage over the previously described adjacency graph encoding. When using the adjacency graphs, a break in a single ridge results in what was once a single node in the graph being "split" into two new nodes, with the links that were connected to the original node being distributed between the two new nodes. These links may extend arbitrarily to other points in the graph, since many fingerprints have single ridges which "wander" through most of the print area (i.e. a very long ridge). A break in such a ridge has serious effects on the entire adjacency graph representation. This property that a small, very localized error introduced into the data can cause global changes in the graph is obviously undesirable.

With minutia graphs however, defects such as blotches and ridge breaks serve only to add spurious features (nodes) to the graph, with links only to other, physically near nodes. These additional nodes are easily ignored by the graph matching procedures used (and to be explained below). Thus though adjacency graphs are superior with regard to immunity to geometric distortion, the locality properties of minutia graphs make them a better choice, particularly when a low-distortion (non-inked) input mechanism is used. It is for these reasons that the remainder of this work will deal primarily with results obtained by use of the minutia graph encoding. Further details of that encoding will become apparent as the method used to do the graph matching is explained in the next chapter. But first the general issue of comparison of non-identical, yet similar graphs

needs to be explored...

Chapter 6

Graph Comparison

6.1. Introduction

Given that a particular pair of fingerprints have been encoded into a form of graph representation such as the adjacency or minutia graphs already described, the key problem now becomes determination of "how similar" the graphs, and therefore the original fingerprints, are. Consideration of that question leads directly into subjects such as Fuzzy Theory as well as relatively unexplored areas of graph theory dealing with partial isomorphisms. But first, we will consider an idealized case.

Suppose that at some time in the past, an image of a particular finger has been taken and encoded into a minutia graph as described in the last chapter. We are now presented with another image of a finger, and must determine if it is from the same finger as the original. So we proceed to process the new image, producing a second minutia graph. Even if we assume that no changes have occurred in the finger itself, and that no noise or distortion was introduced anywhere in the process, it is likely that whatever labelings we have for the nodes in the two graphs are not identical, due simply to differences in finger orientation. So the problem becomes: Given two graphs, determine if there exists an adjacency preserving one-to-one correspondence between the vertices [Harary69].

This is the much-studied "isomorphism problem" in graph theory. Graph isomorphism has application in widely varied fields such as chemistry, switching theory, and information retrieval.

It was stated above that the isomorphism problem consisted of finding a one-to-one correspondence between the vertices of the two graphs in question. Beyond that is the more important issue of finding a GOOD (i.e. efficient) algorithm for determining if two graphs are indeed isomorphic.

6.2. Isomorphism Determination

One way of demonstrating the inherent difficulty of the isomorphism problem is examination of some typical graphs, such as those in Figure 6.1. Though they seem quite different from one another, all three are, in fact, simply different ways of drawing the identical graph. Thus we see that the usual "line and dot" representation of a graph can be quite misleading. Another common representation of a graph is the adjacency matrix form, as shown in Figure 6.2. Here, each row and column of a matrix is labeled with the name of a node in the graph, and the entries in the matrix are 1 if and only if the pair of nodes corresponding to the row and column of the entry are linked. Unfortunately, this representation is also often misleading, as there are as many adjacency matrix representations of a given graph as there are unique labeling of the nodes in the graph. See Figure 6.3 for an alternate labeling and adjacency matrix for the graph of Figure 6.2. Nevertheless, as we will see below, the adjacency matrix representation can be quite useful.

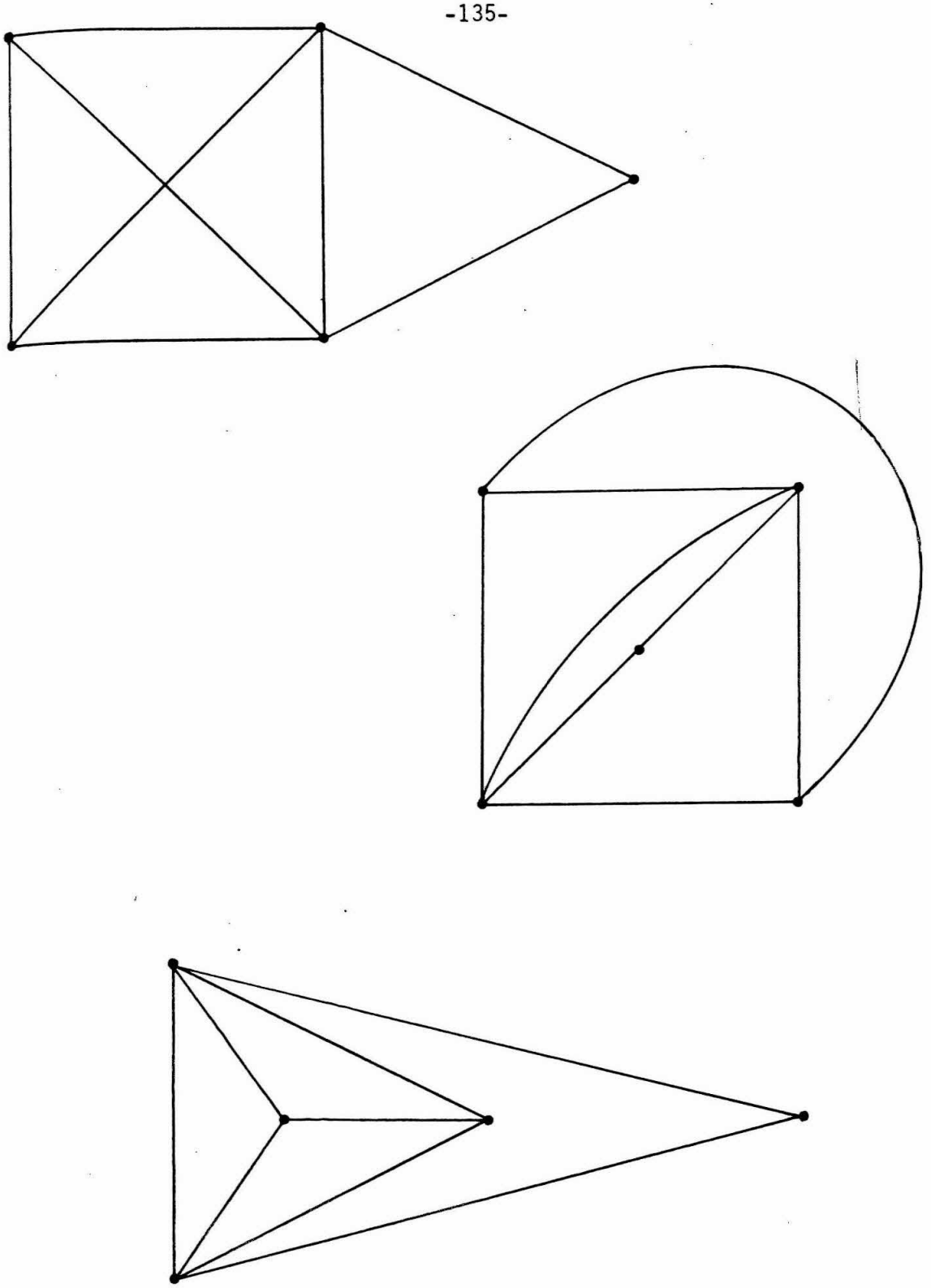
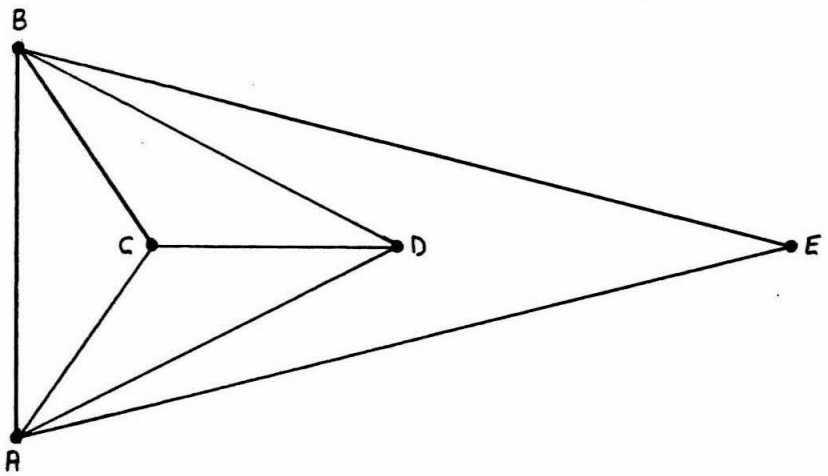
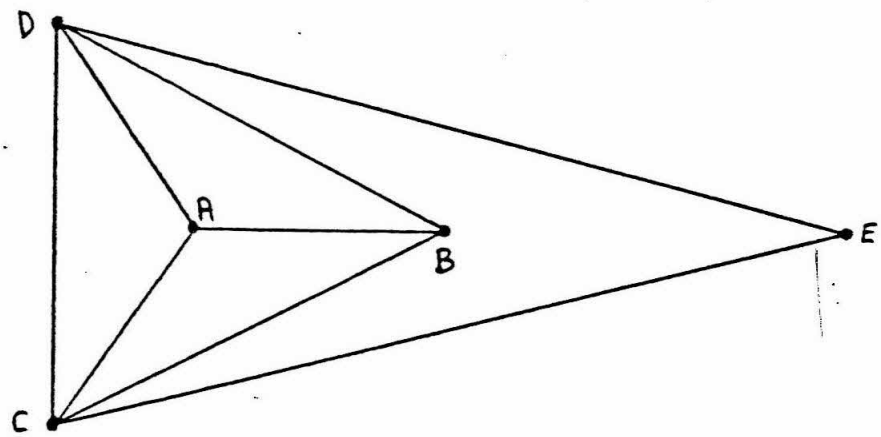


Figure 6.1: Three Isomorphic Graphs



	A	B	C	D	E
A	0	1	1	1	1
B	1	0	1	1	1
C	1	1	0	1	0
D	1	1	1	0	0
E	1	1	0	0	0

Figure 6.2: Graph and Its Adjacency Matrix



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	1	0
C	1	1	0	1	1
D	1	1	1	0	1
E	0	0	1	1	0

Figure 6.3: Same Graph With New Labeling

The difficulty in determining isomorphism of two given graphs is not one of finding an algorithm that will do the determination, but rather one of finding an algorithm that will do so in a short enough period of time (the 'good' algorithm mentioned above). For if execution time were of no consequence, we could simply try all possible labelings for the nodes in each graph, and then compare the resulting adjacency matrices until either a match was found or all possible labelings had been exhausted - clearly a very time-consuming process. More specifically, let us assume that we have two graphs, G_1 and G_2, each having n vertices. The nodes of each graph can then be labeled with the integers 1, 2,...,n. For each such labeling, there exists a unique adjacency matrix, as previously described. Further assume that an arbitrary labeling is used for graph G_1. If G_1 and G_2 are indeed isomorphic, then there must exist a labeling of G_2 which results in the same adjacency matrix as that for G_1. Thus we need only try all possible labelings of G_2 (there are n! of them), comparing the resulting adjacency matrices for each such labeling. As there are n^2 elements in each adjacency matrix, the comparison operation must take $O(n^2)$ operations. Thus the entire process of determining if there exists an identical pair of adjacency matrices (and that the graphs are therefore isomorphic), would take $O(n^2 n!)$ operations. Unfortunately, this requires time that is more than exponential in the number of nodes in the graphs, and is therefore impractical for graphs with more than a few nodes. What is of course desired is an algorithm that requires only a polynomial amount of time, and the search for such algorithms is closely tied to the general area of computational complexity.

A brief foray into the terminology of complexity theory seems appropriate. A problem is said to be in P, if it can be solved in polynomial

time on a deterministic, one-tape Turing machine. Such a problem is equivalently solvable in polynomial time on a conventional Von Neumman-style computer. A problem is said to be in NP if it can be solved in polynomial time on a one-tape non-deterministic Turing machine. A non-deterministic Turing machine may be informally described as a Turing machine that allows simultaneous evaluation of all paths in a search tree. Though it would seem that a non-deterministic Turing machine should be inherently more powerful than a deterministic one, this is in fact an open question ("does P=NP?"), and has been the subject of considerable attention. A special class of problems in NP exists, known as the NP-complete problems. They have the property that if any of the NP-complete problems are in P, than all problems in NP are in P (i.e., P=NP). It is generally believed that P≠NP. Many famous graph theoretical problems have been shown to be NP-complete, including k-clique, Hamiltonian cycle, and vertex cover.

The graph isomorphism problem is in NP, but as no polynomial time algorithm has yet to be found, it is not known whether it is in P. More interestingly, it has yet to be shown that graph isomorphism is NP-complete. It is thus in a rather special class, known as the Isomorphism Complete problems [Lubiw81], which includes chordal graph isomorphism, acyclic digraph isomorphism , and others. It is possible (though considered unlikely) that the Isomorphism Complete problems are in P, while the NP-complete problems are not. A recent proof [Ladner75] shows that if P≠NP, than there must exist problems in NP which are neither in P nor NP-complete. The Isomorphism Complete problems may well fall into this category, thus explaining the lack of success in trying to prove the isomorphism problem NP-complete.

Though no algorithms have been found that can be proven to solve the isomorphism problem in polynomial time, many algorithms have been developed for specific applications. Such algorithms are often reasonably efficient if certain assumptions are made limiting the generality of the graphs to which they are applied. For example, efficient isomorphism algorithms are known if the graphs involved are interval graphs (linear time, [Lueker79]]), or planar graphs (linear, [Hopcroft74]).

An alternative approach is to make use of heuristic algorithms for deciding isomorphism - in other words, an algorithm that will not always produce an answer in reasonable time, but does so often enough as to still be useful. The first work in this area was done by Unger, with the creation of a program known as GIT [Unger64]. GIT uses a variety of processes to narrow the search for an isomorphism, beginning with the obvious step of comparing the number of nodes in each of the two graphs under consideration. Of course isomorphism is impossible unless both have identical numbers of nodes. The program then classifies nodes in terms of the number of arcs entering and leaving the node. Consider the set $\{A_1, A_2, A_3\}$ of all nodes in G_1 having exactly one arc leaving the node, and the similarly defined set $\{B_1, B_2, B_3\}$ for graph G_2. Clearly any mapping of nodes in G_1 to nodes in G_2 being considered must map elements of the first set of nodes only onto elements of the second. This classification serves to reduce the number of mappings that must be considered considerably below the original n!.

Other functions can be computed for each node, and used to speed up the search for a possible isomorphism. Three of those used in GIT are:

(1) The number of nodes that can be reached from a given node by means of paths not longer than a set length.

(2) A binary-valued function that is equal to 1 on those nodes which are included in circuits (closed paths) of length n.

(3) The number of nodes from which a given node can be reached by means of paths not longer than a set length.

Brute force isomorphism testing is, as was mentioned, time-consuming. Unger calculates that at a rate of one-millisecond per node-pairing trial it would take about one hour to compare two 10-node graphs, and more than 40 years for 15-node graphs. GIT was observed to routinely match 24-node graphs of a type particularly difficult for the algorithms chosen in about 2 minutes. "Random" 24-node graphs typically were processed much faster. Though the true "worst cases" for GIT were likely not found, the improvement over a brute force approach is apparent.

GIT is but the earliest example of the type of approach to isomorphism testing that has come to be called a "Vertex Classification" algorithm [Read72]. The most common such algorithms use what is known as iterative vertex classification. Given a graph with set of vertices V, consider an ordered collection of subsets V_1, V_2, V_3 ... V_k, where the ordering is determined by the value of some function like those just enumerated. In general these functions describe each node in terms of basic adjacency information (such as the number of incoming arcs). To each vertex x we assign a list $a_1, a_2, \ldots, a_k$, where each a_i is the number of vertices in subset V_i adjacent to x. We now use these lists to partition each of the sets of vertices V_i into subsets, each subset consisting of all vertices with a given list. Each of these subsets can then be ordered lexically based upon the

corresponding list, thus obtaining a refinement of the collection of subsets.

This process may be repeated continually as long as further refinement is obtained, at which time a new function may be used to continue the process. If at any point all of the subsets are sufficiently small, than the product of all of the permutations of the elements of each subset $[\prod_i p_i!]$ will be small enough to allow exhaustive enumeration of all of the possible mappings.

Of course in the general case the refinement will not proceed far enough to allow exhaustive enumeration as an efficient alternative. It is then necessary to consider what approach to use from that point. Let us assume that the vertex classification algorithm has been applied to both graphs, resulting in the two partitionings $V_1, V_2, \ldots, V_k$ and $W_1, W_2, \ldots, W_k$. We first verify that for all j, $|V_j| = |W_j|$, as this is an obvious necessary condition for isomorphism. Further we observe that if the graphs are indeed isomorphic, than the mapping between the vertices must be such that vertices in V_i are mapped only onto vertices in W_i, for all i.

If we consider all possible mappings between nodes of the two graphs, under the constraints imposed by the partitionings, we get a tree structure such as that shown in Figure 6.4. This was derived from the graph in the figure, using the degree of each node as the classification function. Searching this tree is the obvious next step. In fact, most attempts to produce isomorphism algorithms appropriate for a given application are no more than elaborations on one of two standard tree search methods - depth-first or breadth-first.

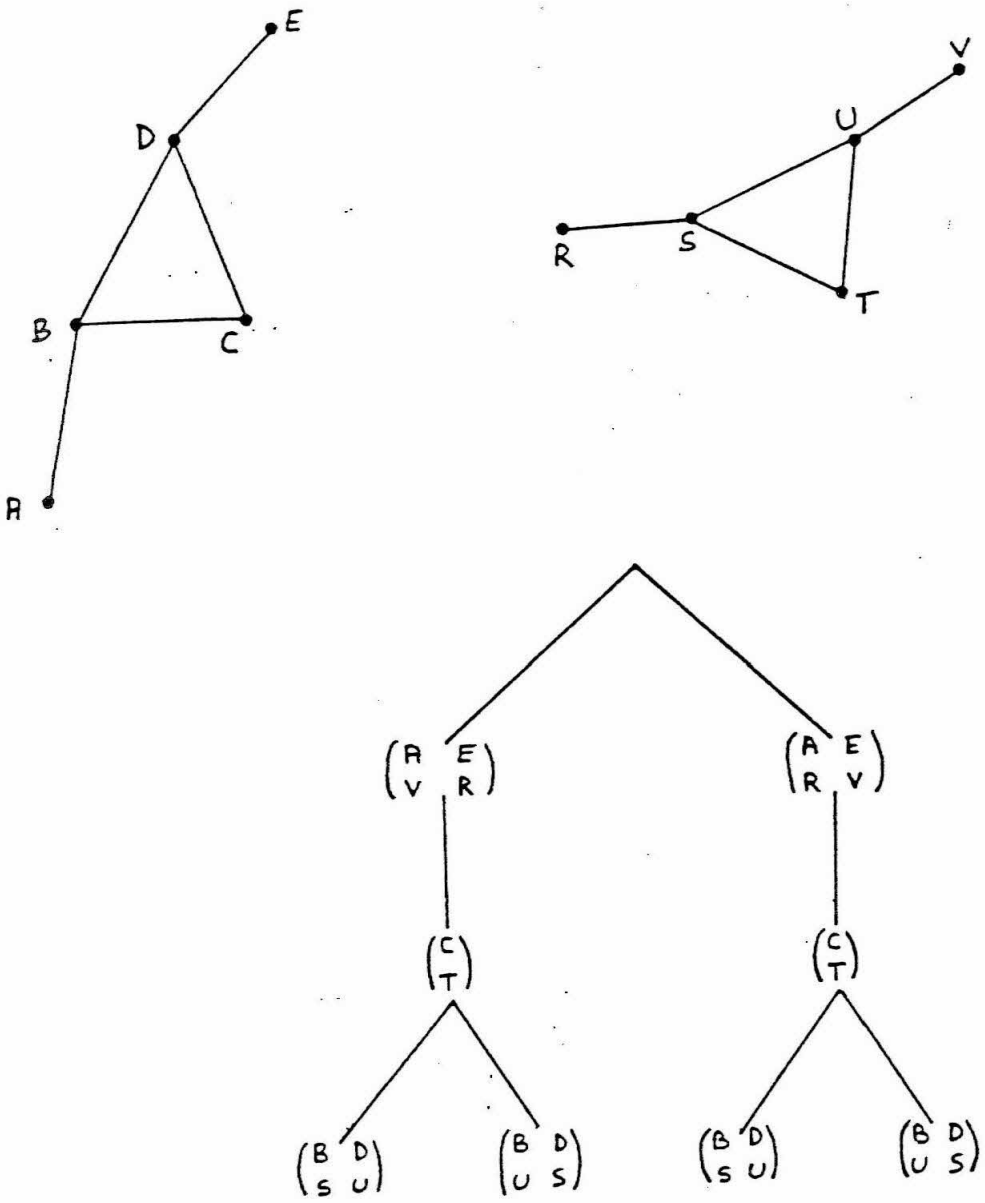


Figure 6.4: Node Mapping Tree

Breadth-first tree search proceeds by choosing an arbitrary vertex x in V_i. This vertex is then mapped onto all vertices in W_i. In other words, all mappings of x onto vertices of the other graph are tried, within the constraints imposed by the existent partitionings. For each such mapping, it may be possible to refine the vertex partitionings and if so this is done. When one graph is so further partitioned, it is necessary that the second graph exhibit identical behavior, else they are not isomorphic. The process continues either until such evidence for non-isomorphism is found, or all isomorphisms are produced. This breadth-first search method is most useful if the two graphs involved are NOT isomorphic, as this is likely to be determined rather early. Unfortunately, in the case where the graphs is indeed isomorphic, it may take considerable time to demonstrate that fact.

The depth-first search also begins with the arbitrary pairing of two vertices, one in V_i, and one in W_i. Based upon this pairing, the vertex partitioning is refined, and a new pairing is arbitrarily chosen. Thus we proceed "down" the search tree, continuing either until an isomorphism is produced, or it becomes clear that the set of choices made so far cannot lead to an isomorphism. If an isomorphism is found, we are of course done, and often in very little time. In the case where we reach a "dead-end", it is necessary to backtrack, and choose a new set of pairings. The behavior here is quite the opposite of the breadth-first search - if the graphs are isomorphic we will likely soon finish, but if they are not it may require considerable time to determine that fact. Such backtracking procedures have been widely discussed in the graph isomorphism literature. In [Schmidt76], the distance matrix representation of the graphs are used in order to produce an initial partitioning of the vertices, after which a backtracking procedure is used to determine isomorphism. In [Berztiss73],

the algorithm is based upon a linear formula representation of the graph. An excellent example of a backtracking isomorphism algorithm is [Corneil70], where considerable effort is devoted to producing a good partitioning of the vertices, and as a result a polynomial bound is claimed for the time required. Though a counter-example to their claim has been found, the algorithm as presented is still quite efficient for "typical" graphs.

6.3. Beyond Isomorphism

In this work a backtracking procedure is also used, with the vertex partitioning based upon the labels of both the nodes and edges in the graph. The need for an efficient backtracking procedure is made even more important by the fact that the "maximal common subgraph" problem being treated here is in fact NP-complete [Garey79], and therefore likely to be harder than the strict isomorphism problems so far discussed.

The above consideration of algorithms for determining isomorphism of two given graphs cannot be directly applied to the fingerprint matching problem, no matter which of the above discussed graph representations for a print is used, primarily because the very concept of "isomorphism" of two graphs is far too inflexible to deal with the omnipresent corruption of the fingerprint images due to various forms of noise and distortion. Two graphs are either isomorphic or they are not -- no middle ground is considered. If we encode two fingerprints from the same finger into their minutia graph representations, the graphs obtained will be quite similar. Perhaps there will be a number of missing or extraneous ridges or nodes in one graph as compared to the other - but the majority of the nodes and edges will be the

same. We need to be able to recognize the inherent similarity of such resultant graphs if we are to have any hope of using graph encodings to compare fingerprints.

Using the terminology of conventional set theory, given a graph G, one can divide the universe of graphs into two sets: the set Y containing all graphs isomorphic to G, and the set N of all graphs which are not isomorphic to G. When presented with a graph H, one can, by algorithms such as those discussed above, determine to which of the two sets (Y or N) H belongs - and it must belong to one or the other. The property which here decides membership in the set Y is "isomorphism to the graph G", and since isomorphism is a Boolean function, membership in G is well-defined.

When comparing non-identical graphs, the situation is best described as a problem in the field of Fuzzy Set theory [Kandel79]. Rather than using isomorphism as the criterion for set membership, we use an as yet undefined function S, which we shall call the "similarity" of two graphs. For any two graphs A and B, assume that we can compute S(A,B), such that $0 \leq S(A,B) \leq 1$. Let the set Y' contain graphs g_1, g_2, ... as elements. Each element of the set has associated with it a characteristic function f_i, where $f_i = S(G,g_i)$. In other words, each element of the set is "tagged" with a value in the range 0 to 1, indicating the "grade of membership" of the element in the set. The set Y' as so defined is a fuzzy set, with membership function S.

If we further assume that the function S is in some sense a measure of the "alikeness" of its operands, then it is clear that elements of Y' with membership values close to 1 are graphs which are more like G, while graphs with values close to 0 are quite different than G. Values near .5

imply some intermediate situation. Given the existence of such a function S, we have a powerful tool for using graph structures to represent "real-world" objects, and in particular fingerprint images. No longer need we limit discussions to "are graphs A and B isomorphic?", but can now quantitatively discuss "how similar are graphs A and B?".

The definition and method of determination of the function S is one of the major results of this work, and will be considered in detail. Though the function described has been derived specifically to be applied to minutia graph encodings of fingerprint images, it is in fact applicable to a wide variety of problems making use of labeled graphs.

Very little work has been done in the area of comparison of similar, but non-isomorphic graph structures. Some of the earliest was by Edward Sussenguth [Sussenguth65], who was interested in the problem of producing an efficient chemical information retrieval system. In a typical such system, the chemist formulates his request as the structural diagram of a chemical, which is then searched for in a large library of compounds. There are (and were at that time) many systems which would, given a diagram for a complete molecule, determine if that molecule already existed in the database, and retrieve any existing information associated with it. This is simply the already described graph isomorphism problem. The more generally useful system is, however, one that allows the specification of an arbitrary fragment of a compound. The system will then return with information regarding which chemical compounds in the library contain the specified fragment as an integral part. In graph terminology, we are asking the system to locate those graphs (compounds) in the library that contain a subgraph isomorphic to the specified graph (fragment), where a subgraph of

a graph G is defined as a graph having all of its nodes and edges present in G [Harary69]. Sussenguth's system relied on calculation of characteristic functions for each node of the graphs, based upon connectivity with adjacent nodes. A matching procedure was then used to pair nodes based upon their characteristic functions. The usual matching rules (e.g. the requirement that two nodes must have the same number of links) were suitably modified to handle the subgraph isomorphism case.

Later, A. Roger Meetham [Meetham68] also used a graph theoretic approach to finding structural similarities in organic molecules. He assumed that each molecule had been encoded as a graph, with the nodes labeled with the atom type (e.g., C, H, Cl...), and the links labeled with bond type (single, double, or triple). The algorithm was limited to tree-like molecules (no rings), and required time exponential in the number of atoms. This work was unique though, in that it considered a more difficult case than had Sussenguth: Given the graph representation of two molecules, find the molecular fragments (subgraphs) that are shared by the two original molecules. Such subgraphs are known as common subgraphs. His algorithm, though ultimately inefficient, did work acceptably well for small (i.e. less than 10-15 atom) molecules.

We have thus identified three isomorphism-related graph problems, which can be listed in order of increasing difficulty as:

(1) Graph isomorphism: is G_1 isomorphic to G_2?

(2) Subgraph isomorphism: is any subgraph of G_2 isomorphic to G_1?

(3) (Maximal) Common subgraphs: find the (maximal) isomorphic graphs g_1 and g_2 such that g_1 is a subgraph of G_1, and g_2 is a subgraph of G_2.

The computational complexity of problem (1) has already been discussed. Problems (2) and (3) have been shown to be NP-complete, and are computationally equivalent to the well known CLIQUE problem [Garey79].

6.4. Maximal Common Subgraphs

Maximal common subgraph (MCS) derivation is an excellent choice as the algorithm for comparison of graph representations of both fingerprint and other images. The errors most commonly encountered when dealing with the minutia graph representation consist of missing or extraneous links or nodes. Though such an encoding is relatively immune to even major geometric distortion, minor distortions can still result in two nodes moving far enough apart that they will not be linked in the minutia graph. Extraneous links can also be created by a similar mechanism. Use of maximal common subgraphs for comparison is ideal in such cases, as the only consequence of the changes in the graphs is likely to be a small reduction in the size of the common subgraphs found.

MCS determination is a quite general means for comparison of graphs, and has application in any area where graph representations are used. It has been mentioned in the artificial intelligence field as a method for describing the relationships between objects in a scene [Barrow76] [Ambler73]. Unfortunately, being an NP-complete problem, it is not currently possible to produce an algorithm that will be efficient in all cases. It is however possible to write heuristic algorithms which, though exponential in the worst case, are efficient for many or even most sets of

input data.

The procedure used for finding the maximal common subgraphs of a given pair of graphs will first be presented in a general manner, with latter explanation of the specializations built into the algorithm to optimize performance with fingerprints and minutia graphs. The algorithms presented here for transforming the maximal common subgraph problem into a clique-finding problem are modifications of those presented in [Levi73].

Determination of the maximum common subgraphs of a given pair of input graphs can be done via a very straightforward (albeit inefficient) two-step method. Assume we wish to find all common subgraphs of order k (i.e., having k nodes) of two graphs G_1 and G_2, of order m and n respectively. Step 1 is to derive sets of subgraphs of order k from each of G_1 and G_2. Step 2 is to test all such pairs for isomorphism. If step 2 is simply implemented by comparing all possible permutations of the graph adjacency matrices, deriving common subgraphs of order k requires $\frac{m!n!}{k!(m-k)!(n-k)!}$ matrix comparisons. The times thereby required are absurd, and a better algorithm, such as that now to be described, is clearly needed.

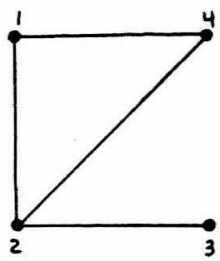
The initial part of the algorithm is basically concerned with transformation of the MCS problem into an equivalent clique-finding problem. We assume that the two input graphs are $G_1(N_1, E_1)$ and $G_2(N_2, E_2)$, where N_i and E_i are the nodes and edges of graph G_i. The graphs are each specified by their adjacency matrices M_1(i,j) and M_2(i,j), said matrices having m^2 and n^2 elements, respectively. The minutia graphs under consideration are labeled graphs, and this is reflected in their adjacency

graph representation. The value of the element M(i,j) is the label of edge (n_i, n_j), while the value of the elements M(i,i) is the label of node n_i.

In order to make the process clearer, we will consider a small example. Figure 6.5 shows two graphs, G_1 and G_2 of order m and n respectively, and their associated adjacency matrices M_1 and M_2. As can be seen, the graphs involved are quite small, G_1 having 4 nodes, and G_2 having 3. Since the graphs are unlabeled, the entries in the adjacency matrices are straightforward, with a 1 implying an edge between the corresponding nodes, and a 0 implying no edge. An edge connecting a node to itself (if allowed), would be represented by a 1 in the appropriate diagonal entry of the matrix.

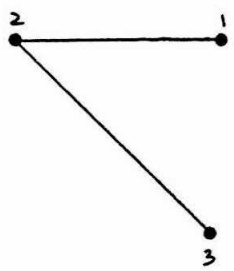
We next construct what will be called the "Node Correspondence Table", N(i,j). This matrix, shown in Figure 6.6, has m rows, corresponding to the nodes of G_1, and n columns, corresponding to the nodes of G_2. The element N(i,j) is concerned with the possible mapping of the i'th node of G_1 (n_i^1) onto the j'th node of G_2 (n_j^2), for the purposes of finding isomorphic common subgraphs. The entries in the table are binary, with a 1 indicating that a mapping between the nodes is possible, and a 0 indicating that it is not. As can be seen in Figure 6.6, in this case all entries in the table are 1, implying that no restrictions exist on node mappings - an obvious result of the fact that the original graphs G_1 and G_2 were unlabeled. However, if we were concerned with matching of molecules, then the graphs would have each node labeled with the type of its corresponding atom. Were this the case, it would be clear that there is no reason to consider the possibility of mapping say a carbon atom onto an oxygen atom, and zeroes would be placed appropriately in the Node Correspondence Table.

G_1



	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

G_2



	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

Figure 6.5: Example Graphs

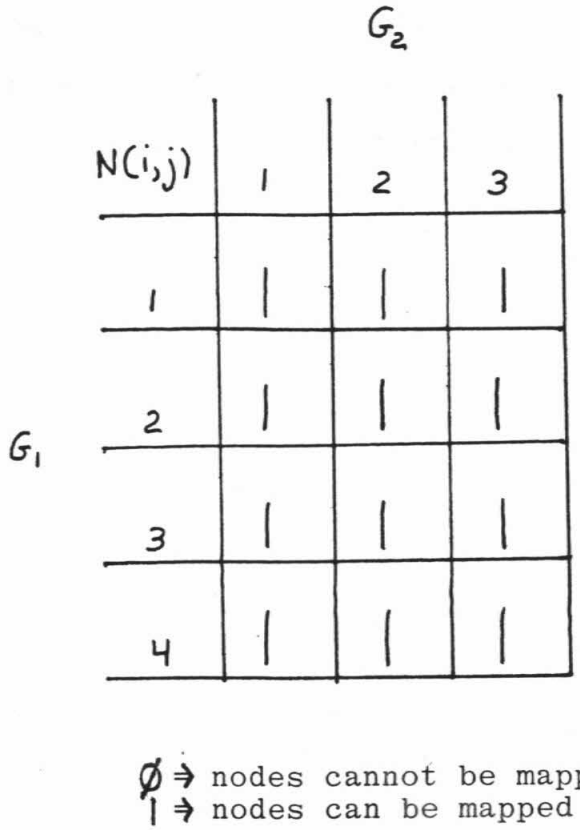


Figure 6.6: Node Correspondence Table

Each of the non-zero entries (i,j) in the Node Correspondence Table thus represents a possible mapping of a node in G_1 onto a node in G_2. Though each such mapping is legal when considered individually, an important further reduction in allowable mappings can be made by considering the mappings (i.e. the entries in N) in pairs. As an example, Figure 6.7a shows a pair of node mappings between G_1 and G_2. In this case, we have mapped node 1 of G_1 (n_1^1) onto node 2 of G_2 (n_2^2), and node 4 of G_1 (n_4^1) onto node 3 of G_2 (n_3^2). Figure 6.7b shows another pair of mappings, specifically node 1 of G_1 (n_1^1) onto node 2 of G_2 (n_2^2), and node 3 of G_1 (n_3^1) onto node 1 of G_2 (n_1^2). There is an obvious difference between these two pairs of mappings. The pair $n_1^1 \rightarrow n_2^2$, $n_4^1 \rightarrow n_3^2$ is a legitimate pair of mappings. The second pair $n_1^1 \rightarrow n_2^2$, $n_3^1 \rightarrow n_1^2$ is invalid, since nodes 1 and 3 are not linked in G_1, yet nodes 2 and 1 are linked in G_2. As a result we cannot consider the second pair of mappings, as they are not consistent with the adjacency-preserving clause in the definition of a subgraph.

In general, given a pair of graphs $G_1(N_1, E_1)$ and $G_2(N_2, E_2)$, a pair of node mappings ($N_i^1 \rightarrow N_k^2$, $N_j^1 \rightarrow N_l^2$) is "compatible" only if one of the following two conditions is met:

1) (n_i^1, n_j^1) is in E_1, and
   (n_k^2, n_l^2) is in E_2 or

2) (n_i^1, n_j^1) is not in E_1, and
   (n_2^2, n_l^2) is not in E_2

In other words, a pair of node mappings is compatible only if the nodes concerned are either linked in both G_1 and G_2, or are not linked in both G_1 and G_2.

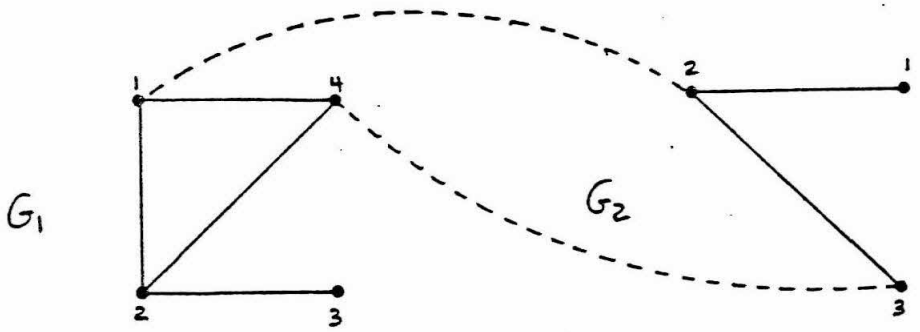


Figure 6.7a: A Legal Mapping

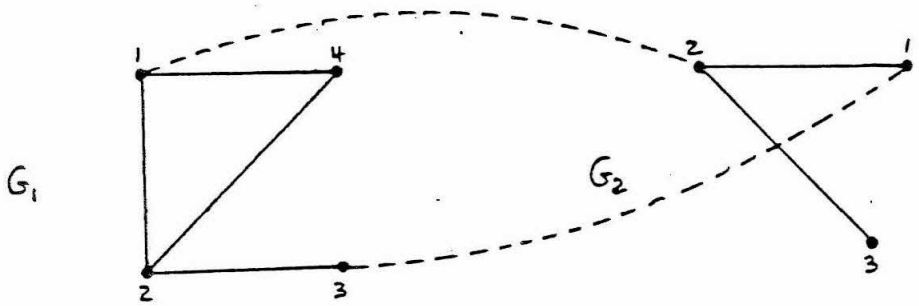


Figure 6.7b: An Invalid Mapping

Given this definition, we can now create a list of incompatible pairs of node mappings. Figure 6.8 is that list for the example under consideration. It is divided into two halves, depending upon whether the incompatibility derives from condition 1 or condition 2 above. The first item on the list, for example, is ((1,1),(2,3)). Its presence on the list indicates that the mappings of G_1 node 1 to G_2 node 1 and G_1 node 2 to G_2 node 3 are not compatible. For the graphs given there are 16 entries in the list of incompatible pairs.

As the next step, we create the "Compatibility Table". This table is a matrix of maximum size mn by mn (i.e. m^2 n^2 total elements), which has as its elements all valid pairs of node mappings. As can be seen in Figure 6.9, the labels along the edges of the matrix are the non-zero entries of the Node Correspondence Table. In the example shown, since the Node Correspondence Table has no zero entries, the Compatibility Table is of maximum size. Each entry in the Compatibility Table C(p,q) is a bit specifying whether the mapping p (a cell in the Node Correspondence Table) is "compatible" with the mapping q (another cell from that table), under the definition that

Two cells in the Node Correspondence Table are not compatible if:

(1) They occupy the same row or same column in the table, or

(2) They are in the list of incompatible pairs.

Two cells that occupy the same row or the same column in the Node Correspondence Table are not compatible due to the need to maintain uniqueness of the node mappings - e.g. if we map node 1 of G_1 onto node 3 of G_2, we cannot also map node 1 of G_1 onto a different node of G_2. Similarly, if we map node 1 of G_1 onto node 3 of G_2, we cannot also map a

different node of G_1 onto node 3 of G_2.

((1,1) , (2,3))
((1,3) , (2,1))
((1,1) , (4,3))
((1,3) , (4,1))
((2,1) , (3,3))
((2,3) , (3,1))
((2,1) , (4,3))
((2,3) , (4,1))

((1,1) , (3,2))
((1,2) , (3,1))
((1,2) , (3,3))
((1,3) , (3,2))
((3,1) , (4,2))
((3,2) , (4,1))
((3,2) , (4,3))
((3,3) , (4,2))

Figure 6.8: Incompatible Mapping Pairs

$[G_1 \rightarrow G_2]$

$C(i_{ij})$	(1,1) 1→1	(1,2) 1→2	(1,3) 1→3	(2,1) 2→1	(2,2) 2→2	(2,3) 2→3	(3,1) 3→1	(3,2) 3→2	(3,3) 3→3	(4,1) 4→1	(4,2) 4→2	(4,3) 4→3
(1,1) 1→1		O	O	O	|	⊙	O	⊙	|	O	|	⊙
(1,2) 1→2			O	|	O	|	⊙	O	⊙	|	O	|
(1,3) 1→3				⊙	|	O	|	⊙	O	⊙	|	O
(2,1) 2→1					O	O	O	|	⊙	O	|	⊙
(2,2) 2→2						O	|	O	|	|	O	|
(2,3) 2→3							⊙	|	O	⊙	|	O
(3,1) 3→1								O	O	O	⊙	|
(3,2) 3→2									O	⊙	O	⊙
(3,3) 3→3										|	⊙	O
(4,1) 4→1											O	O
(4,2) 4→2												O
(4,3) 4→3												

| ⇒ compatible pair
⊙⇒ not compatible; on incompatible pairs list
O⇒ not compatible; due to uniqueness requirement

Figure 6.9: Compatibility Table

In the Compatibility Table shown in Figure 6.9 the cells that are zero due to both of the above compatibility criteria are indicated. The remainder of the cells are of value one, indicating pairs of compatible mappings. In fact the matrix is symmetric, and thus values are indicated only in one-half of the table.

6.5. C-graphs

Each row and column of the Compatibility Table is labeled with the node mapping which it represents, and we may consider the entire Compatibility Table to be the adjacency matrix representation of a new undirected graph, which we will call the C-graph. Each node in the C-graph is labeled with a mapping of a node in G_1 onto a node in G_2. Two nodes in the C-graph are linked if and only if the mappings that they represent are compatible, under the definition given above. The C-graph for the example under consideration is shown in Figure 6.10, and has 12 nodes and 20 edges.

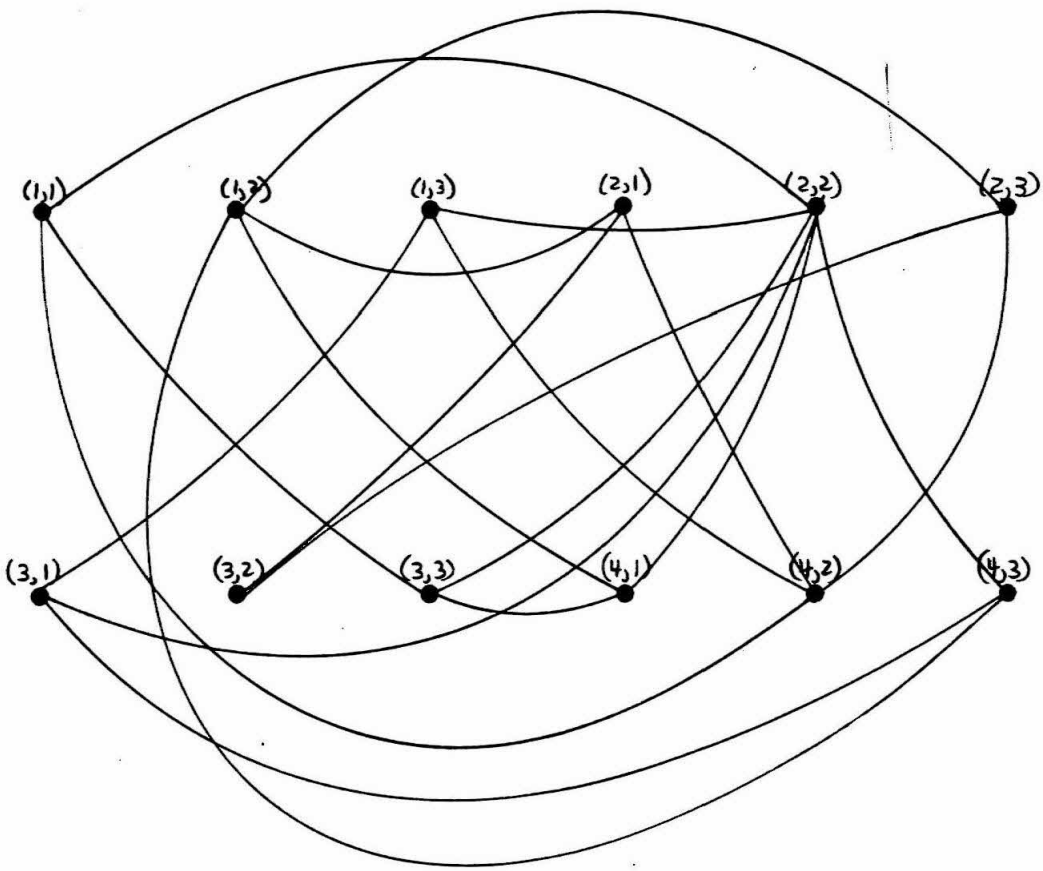


Figure 6.10: C—graph

The usefulness of the C-graph representation becomes apparent when we consider the significance of the cliques of the C-graph. A clique of a graph is defined as a maximal complete subgraph of that graph [Harary69]. In other words, it is a set of nodes, each of which is connected (linked) to each of the other nodes in the set. In addition, no other node of the graph may be added to the set without violating the connectedness criteria (i.e., the set is maximal). A clique of the C-graph can now be interpreted to mean a maximal set of mappings of nodes from G_1 to G_2, all of which are mutually compatible. Since, by the definition of compatibility given above, any set of mutually compatible mappings is equivalent to a common subgraph of G_1 and G_2, a clique of the C-graph is in fact equivalent to a maximal common subgraph of G_1 and G_2.

Figure 6.11 is a list of all 14 cliques containing two or more nodes present in the C-graph of Figure 6.10. Thus we see that there are 14 non-trivial maximal common subgraphs of G_1 and G_2. Of these, the four largest contain three nodes each, and the mappings which they imply are shown in detail in Figure 6.12. Though it is not apparent from this simple example, the transformation of the problem of finding maximal common subgraphs into that of clique finding is of great benefit, as powerful algorithms can be used to attack the clique problem, as will be detailed below.

(2,2) (3,1) (1,3)

(2,2) (3,1) (4,3)

(2,2) (1,1) (3,3)

(2,2) (3,3) (4,1)

(1,2) (2,1)

(1,2) (2,3)

(1,2) (4,1)

(1,2) (4,3)

(2,1) (3,2)

(2,1) (4,2)

(2,3) (3,2)

(2,3) (4,2)

(4,2) (1,1)

(4,2) (1,3)

Figure 6.11: Cliques Found

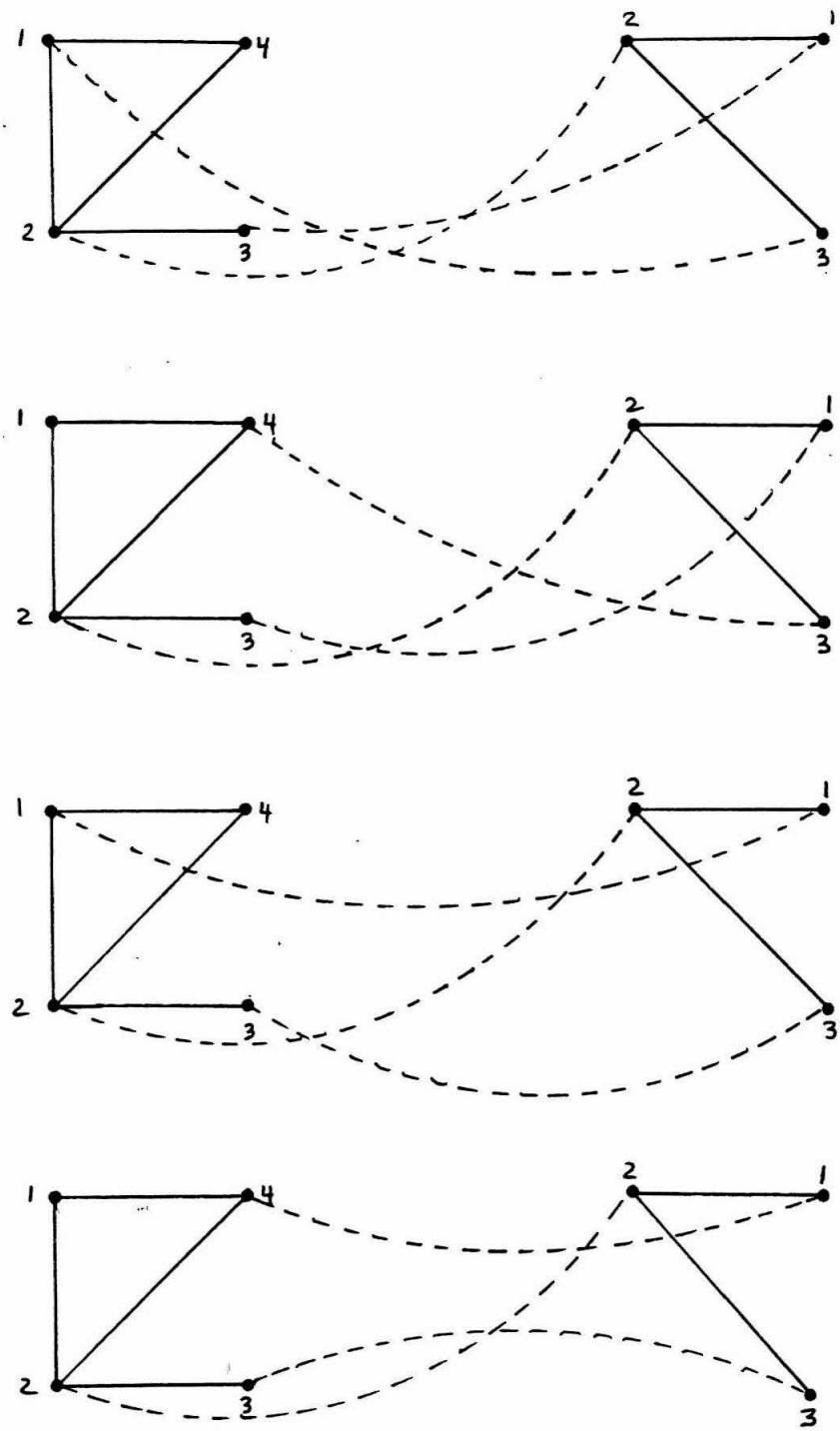


Figure 6.12: Mappings Implied by Largest Cliques

One obvious concern with the C-graph construction is simply that of size. Assuming that, as before, we begin with two unlabeled graphs G_1 and G_2, having m and n nodes respectively, the Node Correspondence Table will be a matrix of size m by n, and therefore have a total of mn entries. The resulting Compatibility matrix will be of size mn by mn, and though symmetric, will still require the storage of $\frac{m^2n^2}{2}$ bits. For input graphs in which the number of nodes is in the range of 50 to 200, as is true for the minutia graphs used for fingerprint analysis, this quantity of information is unmanageable. Fortunately, it is not necessary to explicitly create the Node Correspondence and Compatibility matrices. In fact, little more space is needed than that required to store the x and y coordinate data for all of the features in the two input graphs. The minutia graph adjacency matrices and the entries in the Compatibility matrix can then be computed as needed by the clique-finding algorithm, as follows.

The coordinates of the features in the input graphs are stored in four arrays, G1X($nodenum_1$), G1Y($nodenum_1$), G2X($nodenum_2$), and G2Y($nodenum_2$), containing the x and y coordinates for each of the features in G_1 and G_2 respectively. The subsequent creation of the minutia graph adjacency matrices and the Node Correspondence Table need not be done at all. Rather, we create a translation table which will, when given the (i,j) cell in the Compatibility Table for which we desire the value, allow us to directly look up the corresponding node-numbers in the input graphs, and therefore have access to the x and y coordinates of the node. Note that if all entries in the Node Correspondence Table were 1, the construction of this translation table would be straightforward (and superfluous). In the general case (and for comparing fingerprints in particular), it is necessary

to create this translation table by the following algorithm:

```
Initialize Counter;
FOR i:=1 to m DO
  FOR j:=1 to n DO
    BEGIN
      IF G1X(i) is close enough to G2(j)
      AND G1Y(i) is close enough to G2Y(j)
                    !the nodes are in the same local region;
      THEN BEGIN
             Increment Counter;
          g1coeff[Counter]:=i;
          g2coeff[Counter]:=j;   !fill translation table;
       END;
  END;
```

The definition of "close enough" is based upon the local region size in use. The resultant arrays g1coeff and g2coeff are used in the following manner. Given a cell (i,j) in the Compatibility Table (and the corresponding mapping of a pair of nodes from G_1 into G_2) , g1coeff(i) will be the nodenumber of the first node of the pair in G_1, while g1coeff(j) will be the nodenumber of the second node of the pair in G_1. Similarly, g2coeff(i) is the nodenumber of the first node of the two in G_2, and g2coeff(j) is the nodenumber of the second. Thus if we need, for example, the x coordinate of the second node in G_2 being mapped by the Compatibility Table cell (i,j), we need only compute G2X(g2coeff(j)).

Now let us assume that we desire to know the state of the (i,j) cell of the Compatibility Table. By the previous discussion, this cell represents the following two node mappings:

g1coeff(i) → g2coeff(i) and

g1coeff(j) → g2coeff(j)

For compactness, let

a_i = g1coeff(i)

a_j = g1coeff(j)

b_i = g2coeff(i)

b_j = g2coeff(j)

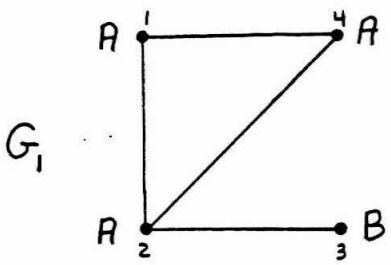
We may then determine the state of the C(i,j) cell by the following algorithm:

```
IF a_i = a_j OR b_i = b_j  THEN C(i,j)=0
          ! mapping must be unique;
 ELSE BEGIN
      Compute G1ΔX, G1ΔY, G2ΔX, G2ΔY;  !the inter node deltas;

      IF G1ΔX< local_region_size AND G1ΔY< local_region_size
      THEN linked_in_1 := TRUE; !nodes in G1 are in same local
            !region (and therefore linked in minutia graph);

      IF G2ΔX< local_region_size AND G2ΔY< local_region_size
      THEN linked_in_2 := TRUE; !nodes in G2 are in same local
            !region (and therefore linked in minutia graph);


      IF (linked_in_1 AND NOT linked_in_2)
        OR (NOT linked_in_1 AND linked_in_2) THEN C(i,j):=0
          !if nodes are in same region in only one of the two
          !input graphs then the mapping pairs are not
          !compatible;
      ELSE BEGIN
            IF (NOT linked_in_1 AND NOT linked_in_2)
            THEN C(i,j)=1  !nodes are not in same region in either
                      !graph, so all we can do is say the
                      !mappings are compatible;
            ELSE BEGIN
                  IF ( |G1ΔX−G2ΔX| ≤ matching_criterion AND
                       |G1ΔY−G2ΔY| ≤ matching_criterion)
                  THEN C(i,j):=1  !the inter-node distances
                    !and angles are close enough;
                  ELSE C(i,j):=0; !linked in both, but the
                    !distances and angles don't match well enough;
                END;
        END;
  END;
```

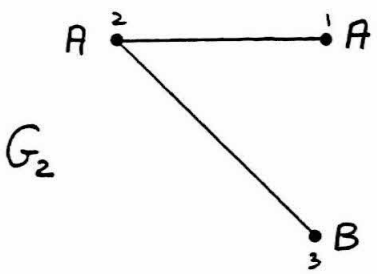
Though complex, the above algorithm executes quite rapidly, and thus it poses no problem to compute each of the C(i,j) elements repeatedly, as they are needed. Use of this algorithm solves the space requirements problem, but excessive time would still be required were it not for the labeling of the nodes and edges of the minutia graphs.

To demonstrate the advantages of such labeling, let us re-consider the previous example, but this time with labels added to the nodes of both

graphs. The labels used will separate the nodes into two categories, type "A", and type "B", as shown in Figure 6.13. The resulting Node Correspondence Table (Figure 6.14) now has zero entries in five of its twelve cells, as we cannot map a type A node onto a type B, and vice versa. The existence of these disallowed mappings reduces the size of both the list of incompatible pairs, and most importantly, the Compatibility Table, as shown in Figure 6.15. The size of this table has been reduced from the previous 72 ($\frac{12^2}{2}$) entries to less than 25. As a result, once incompatible pairs are eliminated, we obtain a considerably simpler C-graph, which is shown in Figure 6.16 along with the list of all its non-trivial cliques. Note that there now exist only two maximal maximum common subgraphs (size three in this case), and the node mappings which they imply are indicated in Figure 6.17. The algorithms used to compare minutia graphs in this work make use of labels on both the nodes and edges, resulting in even more improvement in speed. Specifically, the nodes are labeled with a rough approximation of the X-Y position of the corresponding features in the print image, while the edges are labeled with the approximate angle and direction of the features with respect to one another. Much more detail on the labeling technique used is presented in Section 6.7.



	(A) 1	(A) 2	(B) 3	(A) 4
(A) 1	0	1	0	1
(A) 2	1	0	1	1
(B) 3	0	1	0	0
(A) 4	1	1	0	0



	(A) 1	(A) 2	(B) 3
(A) 1	0	1	0
(A) 2	1	0	1
(B) 3	0	1	0

Figure 6.13: Example With Labels

G_2

G_1 \ N(i,j)	(A) 1	(A) 2	(B) 3
(A) 1	1	1	0
(A) 2	1	1	0
(B) 3	0	0	1
(A) 4	1	1	0

0 ⇒ nodes cannot be mapped
1 ⇒ nodes can be mapped

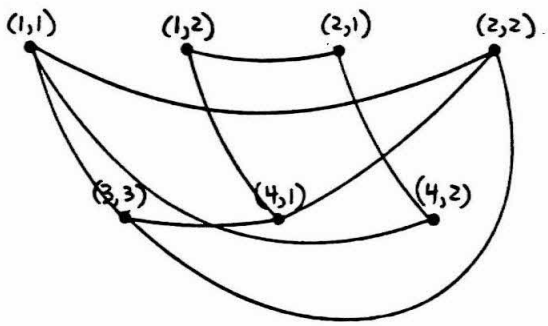
Figure 6.14: Node Correspondence Table (with Labels)

$[G_1 \rightarrow G_2]$

$C(i,j)$	$(1,1)$ $1\rightarrow1$	$(1,2)$ $1\rightarrow2$	$(2,1)$ $2\rightarrow1$	$(2,2)$ $2\rightarrow2$	$(3,3)$ $3\rightarrow3$	$(4,1)$ $4\rightarrow1$	$(4,2)$ $4\rightarrow2$
$(1,1)$ $1\rightarrow1$		0	0	1	1	0	1
$(1,2)$ $1\rightarrow2$			1	0	Θ	1	0
$(2,1)$ $2\rightarrow1$				0	Θ	0	1
$(2,2)$ $2\rightarrow2$					1	1	0
$(3,3)$ $3\rightarrow3$						1	Θ
$(4,1)$ $4\rightarrow1$							0
$(4,2)$ $4\rightarrow2$							

1 ⇒ compatible pair
Θ ⇒ not compatible - on incompatible pairs list
0 ⇒ not compatible - due to uniqueness requirement

Figure 6.15: Compatibility Table (with Labels)



(2,2) (1,1) (3,3)

(2,2) (3,3) (4,1)

(1,2) (2,1)

(1,2) (4,1)

(1,2) (4,3)

(2,1) (4,2)

(4,2) (1,1)

(Cliques)

Figure 6.16: C—graph (with Labels)

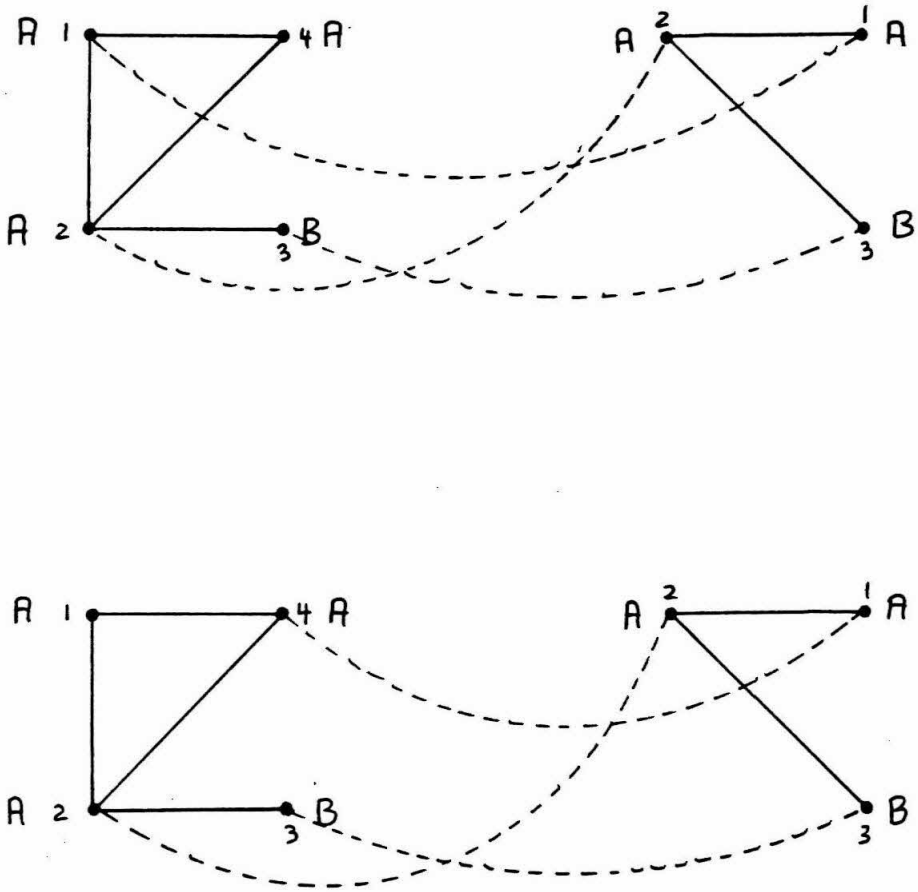


Figure 6.17: Mappings Implied by Largest Cliques (with Labels)

6.6. Finding Cliques

The problem of finding cliques of a given graph in a reasonably efficient manner has received only a small amount of attention in the computing literature. In 1965, Moon and Moser [Moon65] derived several results constraining the number and size of cliques that may be found in particular types of graphs. However none of that work is of use in finding a good algorithm - it simply serves to quantify the worst case possible. In the early 1970's, work done by Augustson and Minker in the field of automating information retrieval systems made use of clique-finding algorithms to do cluster analysis on databases [Augustson70]. The best algorithm available at that time was the so-called "Bierstone algorithm" [Mulligan72], which attempts to simultaneously "grow" cliques from various starting points in the graph. Though useful, the time required per clique found is large, and grows ever larger as the size of the input graph increases.

The algorithm used for clique-finding in this work is based upon an algorithm due to Bron and Kerbosch [Bron73]. In the current implementation, it has the advantage of requiring an essentially constant amount of computation time per clique found, independent of the number of nodes in the input graph. The core of the algorithm is a recursively defined function which is used to extend the size of the clique currently under consideration, and if necessary backtrack and make new selections of nodes for membership in the clique.

Three sets of nodes are kept as the algorithm progresses:

(1) COMPSUB, which is the current complete (though not necessarily maximal) subgraph under consideration. At each step in the process, COMPSUB will either be extended by the addition of a new node, or have

a node removed as a result of a backtracking step. The nodes added and removed are collected in the other two sets, which together consist of all nodes such that each is connected to all nodes in COMPSUB.

(2) CANDIDATES, which is the set of all nodes which will, at some future point in the search process, be used as extensions of COMPSUB.

(3) NOT, which consists of all nodes which have already been used as extensions of COMPSUB at some point in the past, and are now excluded from such use in the future.

There are two criteria that must be satisfied before we can declare at any point in the search process that a valid clique has been found. First, the set CANDIDATES must be empty. This is obvious, for if there was a node remaining in CANDIDATES, the set COMPSUB could be extended by the addition of that node, and therefore COMPSUB was not a maximal complete subgraph (i.e., not a clique). Second, the set NOT must also be empty. This criterion is less obvious, but necessary. By the definition of NOT, any nodes in it were in the past used as an extension of the current COMPSUB, and all nodes in it are connected to each member of COMPSUB. Thus if a node remains in NOT, the current configuration of COMPSUB was at some point in the past contained in another configuration, and is therefore not maximal, and not a clique.

The search portion of the algorithm proceeds by invoking the recursive function mentioned above, which, given the current configuration of COMPSUB as input, produces all extensions of COMPSUB possible given the current set of CANDIDATES. This process can be separated into steps, as follows:

(1) Select a node from CANDIDATES, and add it to COMPSUB.

(2) Create new sets CANDIDATES and NOT, by removing from the old versions of them any nodes which are not connected to the node just added to COMPSUB. This is necessary in order to remain consistent with the definition that all members of NOT and CANDIDATES are connected to all members of COMPSUB.

(3) Recursively invoke the extension function, so that it operates upon the versions of COMPSUB, CANDIDATES, and NOT just formed.

(4) Upon return, the candidate just added to COMPSUB is removed, and placed in the old version of the set NOT (i.e. the version of NOT at this recursion level).

If at any point in this search process, the set NOT contains a node which is connected to all nodes in the set CANDIDATES, then we know that this node will never be removed from NOT (in step 2) by further selection of CANDIDATES, and therefore this particular branch of the search tree can never lead to a clique, and need be followed no further. It is this pruning step (bound condition) that is responsible for much of the efficiency of this algorithm.

The algorithm is implemented by keeping the set COMPSUB as a global array, while the sets NOT and CANDIDATES are combined in a local one-dimensional array that is passed to each call of the extension function. This array, with its associated index values is shown in Figure 6.18. The values of ne and ce are adjusted as necessary. The simplest form of the algorithm would always choose the element at ne+1 as the CANDIDATE selected for addition to COMPSUB (i.e., the first element of CANDIDATES). Though this works, it not most efficient. It is in fact preferable to more carefully choose

the element of CANDIDATES to be added. The goal is to minimize the total number of recursive calls necessary to search any given branch of the tree. Since the most likely result of any such search is that the bound condition mentioned above will be encountered, it is desirable to cause this to occur as early as possible - i.e. to find ourselves in the situation where there exists a node in NOT which is connected to each node in CANDIDATES.

Let us imagine that for each node in the set NOT we have a counter, which keeps track of the number of members of CANDIDATES to which the node is not connected. After return from the extension operator, when a node is added to the set NOT, the values of the counters are decremented by one for each member of NOT which is not connected to the new node. No counter is ever decremented by more than one, and when any counter reaches zero we have a node that is connected to all of the members of CANDIDATES, implying we have reached the bound condition and can terminate this branch of the search tree.

In order to minimize the time required for this situation to occur, we would like to arrange our choice of new members of COMPSUB (and therefore the new members of NOT when the recursion is complete) so that we decrease the counter associated with one particular member of NOT each time. This situation, combined with an initial choice of this so-called "fixed point" so that it has the smallest values of all of the counters, guarantees that we will reach the bound condition as fast as possible. The time advantage of this optimal selection strategy is apparent upon examination of Figure 6.19, which compares the just described algorithm (in both its simple and optimal forms) with the Bierstone algorithm for randomly generated graphs with number of nodes varying from 10 to 50

[Bron 73]. Plotted is computing time per clique in milliseconds, versus number of nodes in the graphs. The important observation is that though the time per clique increases with increasing number of nodes for both Bierstone and the simple algorithm as explained above, when the choice of the node to be used to extend COMPSUB is made in the more optimal way, the computing time per clique remains almost constant. The storage required for this algorithm is on the order of M^2, where M is the number of nodes in the largest connected component of the input graph. The Bierstone approach on the other hand requires an amount of storage that varies as the number of cliques found, which can be quite large.

```
                 |not     |candidates
index values:    |1....ne|.....ce...
```

Figure 6.18: Clique Algorithm Storage

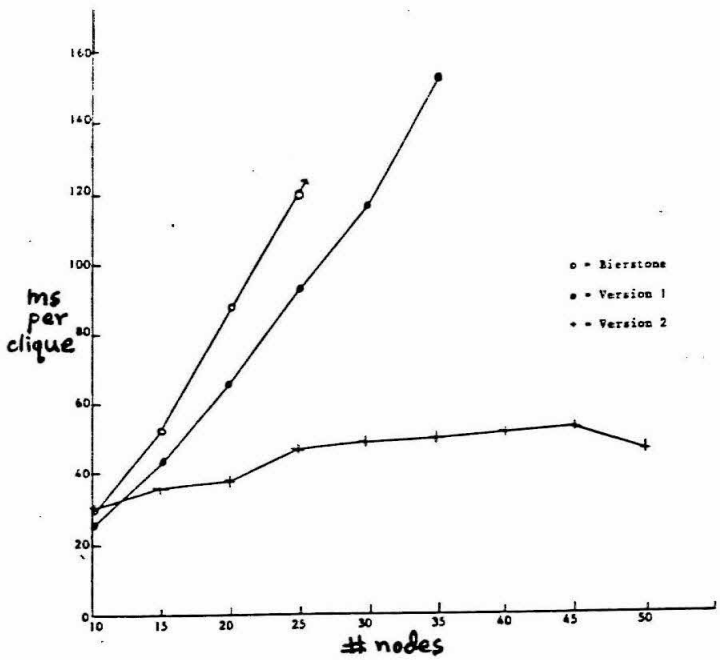


Figure 6.19: Clique Algorithm Time Comparison (Bron73)

In the current application, it is only the large cliques that are of interest - there will always be a myriad of small cliques found, but given two input graphs, it is the sizes of the larger maximal common subgraphs that are of interest. Thus we further optimize the algorithm towards finding large cliques. The optimal form of the algorithm tends to produce the largest cliques first, but still spends considerable time pursuing branches of the search tree that though they may eventually lead to a clique, can never lead to a "large" one. We define a parameter K, which is the size of the smallest clique that is of interest. It is observed that if at any time size(CANDIDATES) + size(COMPSUB) < K, than the current branch of the search tree can never lead to a clique of size greater than or equal to K, and should be abandoned. While maintaining size(CANDIDATES) + size(COMPSUB) $\geq$ K is certainly a necessary condition for generation of a clique containing K or more nodes, it is not sufficient, and thus some smaller cliques will be found (and ignored). The number of small cliques found is, however, dramatically reduced.

It will not in general be apparent at the outset of the running of the clique-finding algorithm what the appropriate value for K is. Thus the algorithm begins with a K-value of 0, allowing all cliques to be output. As soon as a clique is found, the value of K is set to the number of nodes in that clique, so that we establish a lower bound on the size of all future cliques. New cliques will now have size greater than or equal to the size of the last one found. If a new clique is found which is larger than the current value of K, the value of K is increased to the size of the new clique. As a result, with the possible exception of the initial "ramp-up" time for K, we produce only the very largest of the existent cliques. In fact, since the largest cliques tend to be produced first in any case, we rarely get small

cliques at all.

6.7. Finding Maximal Common Subgraphs of Minutia Graphs

The minutia graphs used in this work are labeled on both the nodes and the edges, in order to make the finding of maximal common subgraphs more efficient. Each edge is effectively labeled with the geometric distance between the nodes it is linking. As mentioned in Chapter 5, rather than computing the squares and square root necessary for use of actual distance, we simply use the Δx and Δy values between the nodes, which can be computed by simple subtraction. These inter-node distances need only be approximations, and are used to provide a rough guide as to similarity of node pairs (i.e., to enable the setting of some cells in the Compatibility Table to zero). In addition, the use of the Δx and Δy values has the added advantage of retaining the information regarding the relative direction of the nodes with respect to each other, which would otherwise be lost. Since links exist only between nodes in the same "local region" - i.e., nodes that are close together - these inter-node distances are measured on a local basis only, thus maintaining the desired distortion immunity of the encoding.

Each node is itself labeled with its approximate coordinates (x and y) within the fingerprint image. These are used in the early stages of the comparison process to enable some cells in the Node Correspondence Table to be zeroed. As was demonstrated above, the presence of these zeroes has a dramatic effect on the size of the resulting C-graph, and the time required to find cliques. Of course such use of absolute feature coordinates does

restrict the possible relative orientations of the two fingerprint images being compared, as well as increasing the sensitivity of the comparison process to problems such as image distortion. To minimize these problems, such labels are used only in a very approximate manner, and primarily to avoid the needless waste of processing time which would occur if the algorithm attempted to match features at opposite edges of the input prints. The strictness enforced in the matching of labels on both nodes and edges is determined by the tradeoff between acceptable execution time and the flexibility required in the fingerprint comparison process. Since throughout this work it has been assumed that the orientation of the two prints being compared is roughly the same (as would be the case in say a fingerprint controlled area access system), considerable latitude is available in selection of these parameters. Details of the actual values used will be presented below.

6.8. An Example

In order to demonstrate the operation of the algorithms described for finding maximal common subgraphs of two input graphs, the two moderate size graphs shown in Figure 6.20 were produced, based upon two sets of randomly generated feature locations within the 20 x 20 grid indicated. Once the locations of the features had been determined, the usual minutia graph creation rules were applied, with the parameters specified below, to produce the graphs G_1 and G_2, which contain 20 and 25 nodes (features) respectively.

(1) These two graphs, along with the coordinates of the nodes, were used as input to the three relevant stages in the processing - creation of the minutia graphs, creation of the C-graph, and the finding of maximal cliques within the C-graph. The input parameter settings used for each of these stages are now described:

(2) Creation of the minutia graphs (feature extraction). The parameter that must be specified is the size of the region (D_{LR}) surrounding each feature that is considered "local" to it, and therefore the set of other features to which it will be connected in the minutia graphs. A value of 5 pixels was used as the "radius" for the local region in this example.

(3) Creation of the C-graph. The concern here is with the criteria used to place entries in the Node Correspondence and Compatibility tables. The first parameter is D_{FC}, the distance to within which feature coordinates must match in order to allow mapping between them as a possibility, and thus put a 1 in the corresponding cell of the Node Correspondence Table. A value of D_{FC} of 5 pixels was used. The second parameter is D_{IF}, which is the criterion used for matching the Δx and Δy distances between features. Specifically, in order for a pair of nodes n_1^1 and n_2^1 in G_1 to be compatibly mapped onto a pair of nodes n_1^2 and n_2^2 in G_2 we must have:

$$\left|n_{1x}^1 - n_{2x}^1\right| \leq D_{IF}$$

$$\left|n_{1y}^1 - n_{2y}^1\right| \leq D_{IF}$$

$$\left|n_{1x}^2 - n_{2x}^2\right| \leq D_{IF}$$

$$\left|n_{1y}^2 - n_{2y}^2\right| \leq D_{IF}$$

A value for D_{IF} of 1 pixel was used in this example.

(4) Clique-finding. The value used here (K) determines the size of the smallest clique which is of interest to us. For purposes of this example, a value of 1 was used for K, in order that all cliques might be examined.

With the parameter values as just described, a total of 65,161 cliques was found, ranging in size up to 10 nodes per clique. Eight cliques of this size were found. A ten-node maximal common subgraph found when comparing two graphs having only on the order of twice this many nodes each, would at first seem to indicate that the original two graphs were quite similar. Yet this is not the case, for as can be seen, the randomly generated graphs being used are quite different from one another. The explanation lies in the only aspect of the graphs matching procedure not yet considered - the connectivity of the resulting maximal common subgraphs. Figure 6.21 shows the same graphs G_1 and G_2, but with the nodes contained in one of the larger maximal common subgraphs indicated. Two observations may be made. First, it is seen that the algorithm did an excellent job of finding a set of nodes which map from one graph to the other, and maintain not only inter-node connectivity, but also the coordinates of the nodes, and the distances and orientations of the links. Second, it is obvious that the MCS found is made up of not one, but five sets of connected nodes.

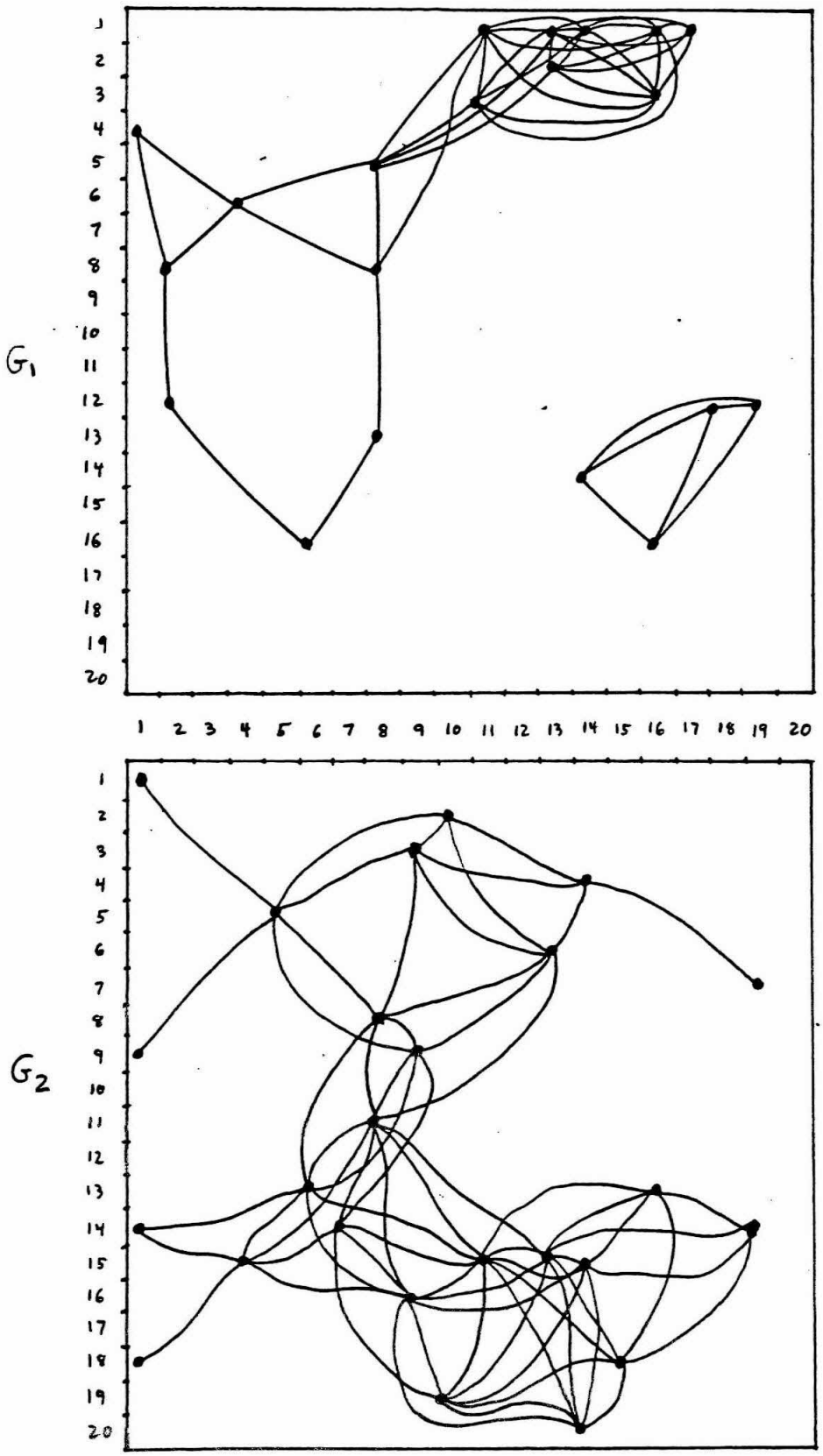


Figure 6.20: Minutia Graphs Based on Random Features

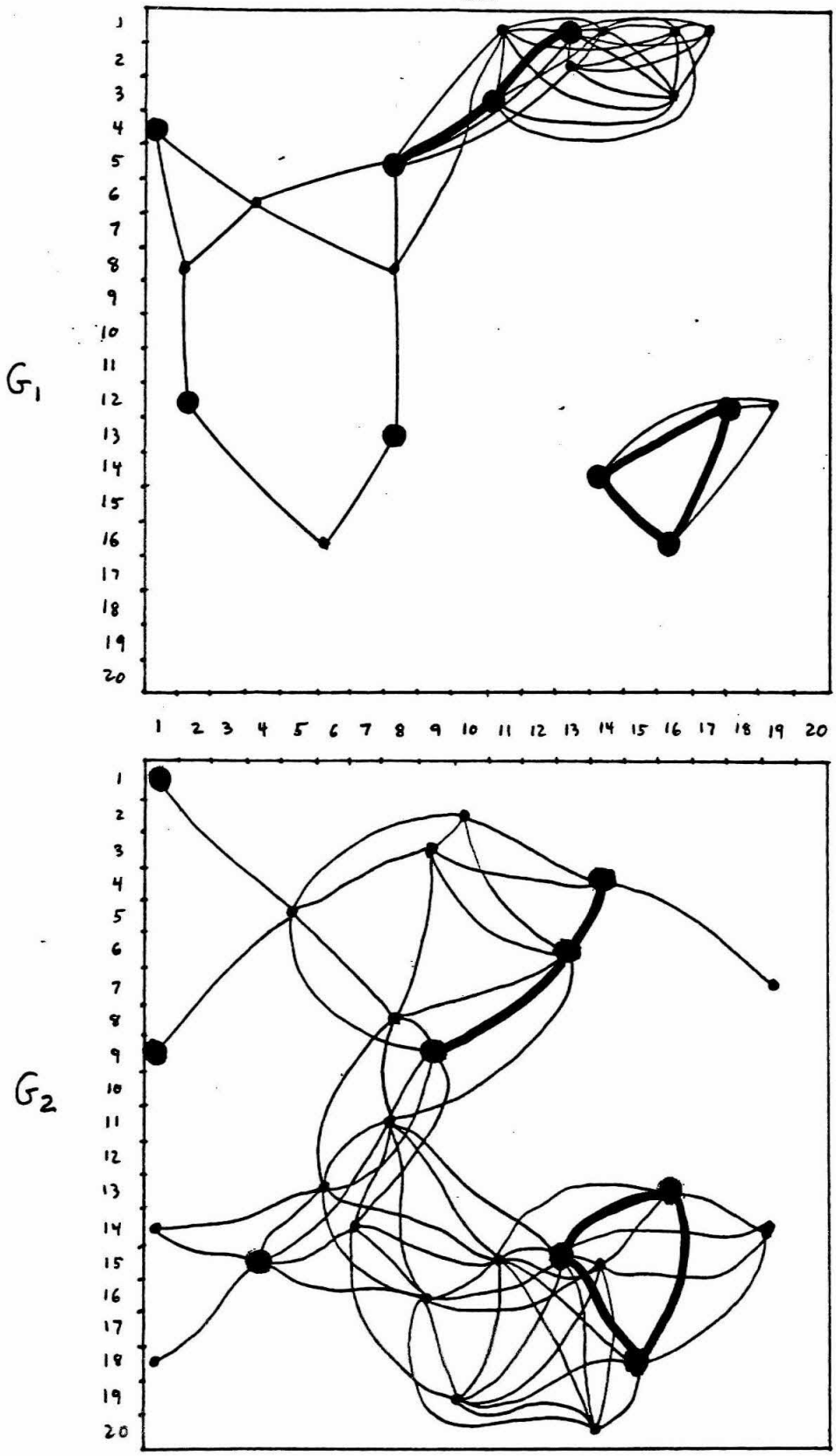


Figure 6.21: Resultant Maximal Common Subgraph

The result of interest in the above example is then not truly the sizes of the largest cliques (maximal common subgraphs) found, but rather the size of their largest connected components. For it is always possible to find trivial maximal common subgraphs of the two input graphs that consists primarily of disconnected nodes. Thus we now have the final step in the comparison process - we must process the maximal common subgraphs found, extracting information as to the sizes of their largest connected components.

6.9. The Overall Algorithm Summarized

The function S mentioned earlier as the membership function used in the definition of fuzzy isomorphism can now be quantified. The two input graphs are processed as described, producing the corresponding C-graph. The largest cliques present in this graph are derived, using the K-stepping modification to the clique algorithm. These cliques are then analyzed for the sizes of their connected components, the final result being a single number (S'), which is "the number of nodes in the largest connected component of the maximal maximum common subgraph" of the two input graphs. For the example above, the value found for S' was three.

While it is clear that finding an S' value (maximum connected component size) of three when the input graphs have twenty or more nodes each is indicative of a lack of similarity in the input graphs, we must further analyze the connected components found, in order to take into account those of trivial size. To provide some insight into this question, over a dozen random graphs of various edge densities were generated and

compared with one another. Both the settings of the parameters used and the sizes of the graphs involved were typical of those encountered in the comparison of two fingerprint minutia graphs. In no case was a connected component in the maximal common subgraph of size larger than five nodes encountered. By far the most common value for S' was 3, with a few values of 4 and only one of 5. Thus we see that if there is no real similarity between two graphs, it is still likely that small connected components will be found. These results with random graphs indicate that an appropriate definition for "trivial" (and therefore ignored) small connected components is that a connected component is trivial if it contains less than five nodes.

(1) We can now finalize the definition of the similarity measure S, as follows: S is "the total number of nodes (features) that are contained in non-trivial components of the maximal maximum common subgraph of the two input graphs". For the random graph comparison just mentioned, we would therefore have found an S value of zero in all cases but the one that produced a connected component of size five, in which case S would be 5. This result is an indication that the algorithm chosen has the desirable property of not finding a strong correlation between unrelated graphs. The other necessary feature, namely that different prints from the same finger will be identified as such, is demonstrated by the results of the next section, where S values in the 20, 30 and above range are regularly encountered when comparing prints from the same finger.

6.10. Results of Processing Actual Fingerprint Data

In order to produce a realistic assessment of the performance of the fingerprint processing and comparison algorithms that make up the majority of this work, it is of course necessary to make use of actual fingerprint data, taken from different individuals. Over the course of this research, more than thirty such images have been digitized and processed. Presented here is the result of doing all of the cross-comparisons for a set of five fingerprint images, shown in Figures 6.23 through 6.27. The prints are from the right forefinger of three individuals, here identified as MYL, MCN, and STV. For each of MYL and MCN the print was taken from the finger on two separate occasions, resulting in the complete set of prints: MYL1, MYL2, MCN1, MCN2, and STV.

(1) The five fingerprints were first processed through the entire sequence of steps previously described, resulting in a list of the coordinates of the significant features in each. The ten inter-print comparisons necessary were then done by the program "NEWSUB", which derives the minutia graphs for each, and then finds the maximal common subgraphs of those minutia graphs. The process of deriving the minutia graphs and finding their maximal common subgraphs of course requires specifying the values for the three parameters D_{LR} (local region size), D_{FC} (feature coordinate match), and D_{IF} (inter-feature distance match). The values used (D_{LR}=30, D_{FC}=25, and D_{IF}=5) were chosen based upon experience with a wide range of print comparisons, and are not overly critical. The values chosen are a compromise between the conflicting goals of producing an algorithm that is both tolerant of distortions and rotations in the input images, yet is also

reasonably time efficient.

(2) The maximal common subgraphs found are then analyzed for the size of their connected components, with components of trivial size (less than five nodes) being ignored. The result is a value for S for each comparison which represents the degree of similarity of the input fingerprints. The matrix of these S values for the five fingerprints under consideration is shown in Figure 6.22. The S values presented are the averages over the first twenty maximal common subgraphs found for each comparison, the averaging being done after elimination of trivial components.

(3) In examining the result of Figure 6.22, one should note the quite small values for S obtained when prints from different fingers are compared (e.g. the entry for MYL1 vs. MCN1). That such small values are consistently obtained is necessary, in order to guarantee that the matching procedure used will not falsely indicate that a pair of prints are from the same finger when in fact they are not. On the other side, when prints from the same finger are being compared we observe that quite large values for S are obtained (e.g. MYL1 vs. MYL2). This must be true if we are to be able to reliably determine that two different fingerprint images were taken from the same finger. The comparisons of STV with MYL2 and MCN1 each resulted in an S value of 11, which indicates some match between those prints. Though this situation was in fact the worst case encountered amongst many such matches, it does indicate that the parameter values used are probably not ideal, and could be adjusted so as to prevent such correlations. Examination of the matching sets of features showed that the match was the result

of random coincidence in the feature locations - an event extremely unlikely to occur for groups of features of size equal to or greater than the suggested threshold range of 15 to 20.

(4) The data in Figure 6.22 make no direct claim as to whether any given pair of prints being compared are from the same finger. Rather it presents the computed value of the similarity function S for each pair, which considered from a Fuzzy Set Theory point of view is simply the degree of membership of the graph representation of one print in the set of graphs "isomorphic" to the graph representation of the second print. As was mentioned above, in order to produce a decision as to whether two prints are indeed from the same finger, we must choose a threshold value T, such that if $S > T$ then we say the prints match, and if $S < T$ we say that they do not match. Fortunately, choice of a threshold is quite easy, and values anywhere in the range of 15 to 20 would work quite well, and would correctly determine that MYL1 and MYL2 are from the same finger, as are MCN1 and MCN2.

S	MYL1	MYL2	MCN1	MCN2	STV
MYL1		23	1	1	3
MYL2			7	7	11
MCN1				40	11
MCN2					0
STV					

Figure 6.22: Resultant S Values

Figure 6.23: MYL1

Figure 6.24: MYL2

Figure 6.25: MCN1

Figure 6.26: MCN2

Figure 6.27: STV

Chapter 7

Summary and Conclusions

This thesis deals with two different problem domains. The first is concerned with performing various useful transformations on digitized images, in a manner that lends itself to real-time implementation on special purpose VLSI processors. Chapter 3 described both the architecture proposed for such processors and the methods used to implement a wide range of image processing operations within that architecture.

The VLSI image processing modules described and simulated are a blend of several approaches to high speed computation. The image is assumed to be available in serialized form, and the operations make use of this format by accessing only rather small "windows" into the image at any one time. The ability to randomly access the image information is never required. The serial approach used allows pipelining of successive computational stages, resulting in a major increase in system throughput.

Simply making use of a pipeline of serial image processing engines would not provide sufficient computational ability for the operations needed, and thus in each stage of the pipeline an array of communicating processors is used, each mapped onto one pixel of the window in use. This is quite a powerful structure, and allows implementation of communication intensive operations, such as the adaptive thresholding technique described.

The image processing operations described in Chapter 3 transform the input images to a form such that we can extract certain key information from them, in order to reduce the initial million or so bits of information down to a more manageable size. In the case of the fingerprint images studied here, the final result of the transformation and processing steps is the thinned fingerprint image. Two different approaches were presented for proceeding beyond the thinned image.

In Chapter 4 a window processor algorithm is described which produces as an output representation of the fingerprint what we have called an "adjacency graph". This is a graph structure where each node in the graph corresponds to a ridge of the input fingerprint. Two nodes in the graph are linked if and only if they are somewhere "adjacent" in the fingerprint image. More than one definition was discussed for adjacency, but they all roughly correspond to the intuitive notion that two ridges are adjacent if at some point they are next to each other in the print. This encoding of the structure of the fingerprint is certainly compact, and was demonstrated to have excellent immunity to geometric distortions of the fingerprint image.

Chapter 5 presented another window processor algorithm and resultant graph structure representation for a fingerprint image. This representation, called a "minutia graph", is based not upon the ridges themselves, but upon the so-called "minutiae" or features of the fingerprint - the ridge ends and ridge forks that are present in great quantity in all fingerprints. Here each node in the graph corresponds to one such feature, and links between nodes are determined based upon rough measures of the distances between the features. Specifically, a "local region" is defined surrounding each feature, and two features are linked if and only if they are

in the same such region. The link between two nodes is labeled with the inter-node distance, while the nodes themselves are labeled with their xy coordinates within the fingerprint image.

The minutia graphs contain somewhat more information about the structure of the original fingerprint than do the adjacency graphs, and thus they were chosen as the representation to be pursued in the second section of this work, the comparison of non-identical but similar graphs. This is the second problem domain mentioned above. Though the algorithm used to do that comparison was optimized and specialized for fingerprint minutia graphs, the general method should find application in any of the many areas where graphs are used to represent the structure of an object or problem.

Much research attention has been given to algorithms to determine whether or not two given graphs are isomorphic. Unfortunately, the definition of isomorphism is such that none of these algorithms is suitable for producing a metric related to the similarity of two graphs which, though perhaps almost identical, are not isomorphic. Chapter 6 described in detail a method for the determination of such a metric.

The general approach used is to find the maximal common subgraphs of the two minutia graphs being compared. The size of the largest connected component of the largest such subgraph found provides the basis for a good measure of the similarity of the original two graphs. Derivation of the maximal common subgraphs must be done by use of a carefully designed algorithm, in order to minimize the time necessary for the computation.

The method used makes use of the label information present in the minutia graphs to effectively prune the search tree, and reduce the amount

of computation. Nevertheless, it is still important that the underlying algorithm be efficient. Since the maximal common subgraph derivation is based upon finding cliques in a new, higher level graph that is constructed in the early stages of the comparison process, it is essential that the cliques can be found quickly and efficiently. A specially constructed recursive clique derivation algorithm was presented, and used as part of the comparison process.

Though the research presented in this thesis was primarily in the areas of image processing and graph algorithms, the application being considered throughout was that of the comparison of images taken from fingerprint ridge patterns. This is a problem that has been the subject of a massive amount of attention both historically and recently. Though most work in the field has been oriented toward comparison by humans, there have been some less than successful attempts at automating the process, as a system able to reliably compare single-finger prints in real time would be of considerable use in a wide variety of applications, ranging from criminal identification to sophisticated access control systems. Chapter 6 presented some results gained from application of the algorithms described in this thesis to actual fingerprint data. Though the system currently exists only as a simulation, production of a hardware implementation should be straightforward.

At this point it is appropriate to discuss exactly what could be improved about the system as described, in order to optimize its performance and speed. Of course the most important improvement will come about from the actual construction of VLSI versions of the window processors. As designed they make extensive use of parallel computation,

due to the presence of multiple pipeline stages, and because of the many processors included in each stage. The simulation on the DECSYSTEM-20 makes use of only one processor, and is as a result extremely slow.

At the other end of the processing sequence, the various parameters described in Chapter 6 for the graph comparison process have a significant effect both on the time taken for results to be obtained, and on the sensitivity of the system to factors such as relative rotation of the input prints and distortions of the image. Though the values used for this work produced very good results, more attention would have to be given to their selection in a production situation. It may in fact be that an adaptive approach would be appropriate, with the exact parameter values being adjusted based upon properties of the input data.

While the approach used to implement the image processing operations makes massive use of parallel computation, the graph comparison algorithms were designed for execution on a single machine. This is acceptable, as the amount of computation necessary is much less. It may become true however that the image processing pipeline can be made fast enough that the graph comparison process will become the slowest step. The solution to this is to also make use of computational parallelism in the graph algorithms, by the construction of special purpose units to do the maximal common subgraph determination. These could be based upon a computational model such as that in [Browning80], where a simple clique-finding algorithm is implemented on a tree-structured processor array.

The advent of VLSI technology has made possible novel approaches to a wide range of computational problems. The algorithms and structures presented in this thesis would likely never have been considered if it were

not practical to construct large arrays of small, low-cost communicating processors. The application of this technology to the important problem of fingerprint recognition and processing will likely have major impact both in areas such as criminology and security, and in others yet to be considered.

Appendix 1

Input Hardware Configuration

The hardware used to input and digitize the fingerprint images used throughout this work consists of three functional sections which will be described individually:

1) INPUT SECTION - The input section of the hardware consists of a special-purpose optical assembly, plus a black and white television camera. The finger from which the print is to be taken is placed upon the diagonal face of a right-angle prism, as is shown in Figure A.1. A light source is directed into one of the short faces of the prism, while the lens of the television camera looks into the other short face. The principle involved in obtaining a high-contrast image of the fingerprint ridge structure is that of "frustrated internal reflection". Specifically, as can be seen in Figure A.1, the light from the source shown would, if no finger were present, be internally reflected off of the diagonal face of the prism directly into the camera lens. This reflection takes place due to the difference in index in refraction between the glass of the prism and the surrounding air. At the points where the raised and usually oily ridges of the finger contact the prism surface however, this great difference in refraction indices does not exist, and the light is not reflected. As a result, the image seen by the TV camera is that of a uniformly illuminated (white) field of view, with dark black ridges forming the fingerprint structure. This system works

extremely well, and indeed makes visible even the small pores contained within the ridges.

The system just described is unfortunately idealized in one important aspect. As can be seen in Figure A.1, the 45 degree angle of the face of the prism on which the finger is placed results in the TV camera system being called upon to bring into focus an image that is contained in a plane that is not parallel to the front surface of the lens. Conventional television camera close-up lenses are far from having adequate depth-of-field for such an application. As a result, a custom lens mount was constructed for the television camera, which holds the lens at an appropriate angle with respect to the front surface of the camera's internal vidicon tube. The correct choice of this angle results in the image being brought properly into focus, and is determined as per the equations shown in Figure A.1.

2) DIGITIZATION AND INTERFACE- The output of the television camera is an NTSC standard, 1 volt peak-to-peak video signal. The specially constructed digitization and interface hardware consists of approximately 40 integrated circuit packages, and is responsible for several functions. First, the synchronization signals used to control the beam scanning in the television camera must be generated. These signals are generated here rather than in the camera itself, in order to guarantee synchronization between the beam scanning and the digitization processes. A second section of the hardware consists of the counter and comparator circuitry necessary to allow digitization to occur in an adjustable "window" in the center of the television frame. This is necessary to make best use of the available resolution, as the lens system is designed so that even the largest of fingerprint images will not quite completely fill the image frame. The

desired section of the image (determined by DIP switch settings) is digitized to a resolution of 400x400 pixels, with 8 bits of grey level information per pixel. The actual analog-to-digital conversion is performed by a TRW LSI 8-bit "flash converter". Though the converter is fast enough that the entire digitization process could occur within one one-thirtieth of a second TV frame, the resulting stream of sample bytes would appear at a rate far greater than could be handled by the Hewlett-Packard 9845B computer used to temporarily store the results. As a result, the final section is a full handshaking interface to a parallel I/O port on the 9845B system. As sample bytes are passed to the 9845B, hardware counters in the interface keep track of the location on the screen of the next pixel to be sampled. As implemented, digitization and transfer to the 9845B of a full 400x400 image (160,000 bytes of data) occurs in approximately 4 seconds. The subject finger must be kept motionless on the surface of the prism for this period of time, but this has proven not to be a problem, as the adhesion of the skin surface to the prism eliminates any effects of small finger motions.

3) STORAGE AND FILE TRANSFER - As the bytes are transferred from the digitization and interface hardware to the Hewlett Packard 9845B desktop computer, they are stored in main memory. When the input operation is complete, the image data are transferred to floppy disk for short term storage. Since the 9845B is not the system on which any of the processing operations described in this work were done, the next step is to upload the image information contained on a floppy disk to the DECSYSTEM-20 over a serial terminal line. The information then resides on the DECSYSTEM-20's mass storage devices, and the processing operations can proceed.

In fact each time a fingerprint image is input, a corresponding "background" image is also taken, with no finger on the prism. This additional image serves as a reference, and is used to compensate for any variations in illumination or vidicon sensitivity across the image field. This is done as the first processing step on the DECSYSTEM-20, and is implemented by dividing (on a pixel by pixel basis) the grey level values in the fingerprint image by the corresponding values in the background image.

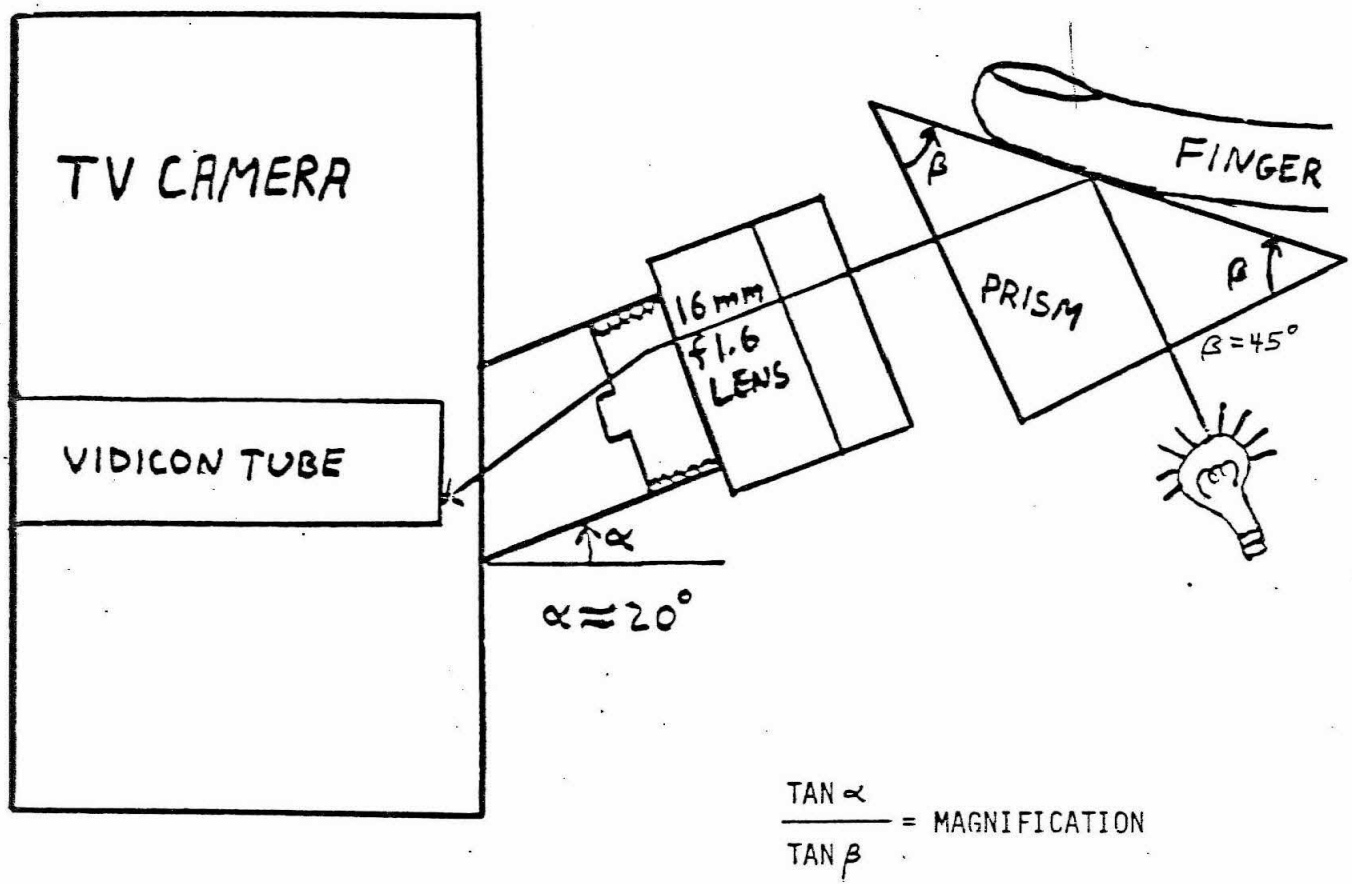


Figure A.1: Input Optical System

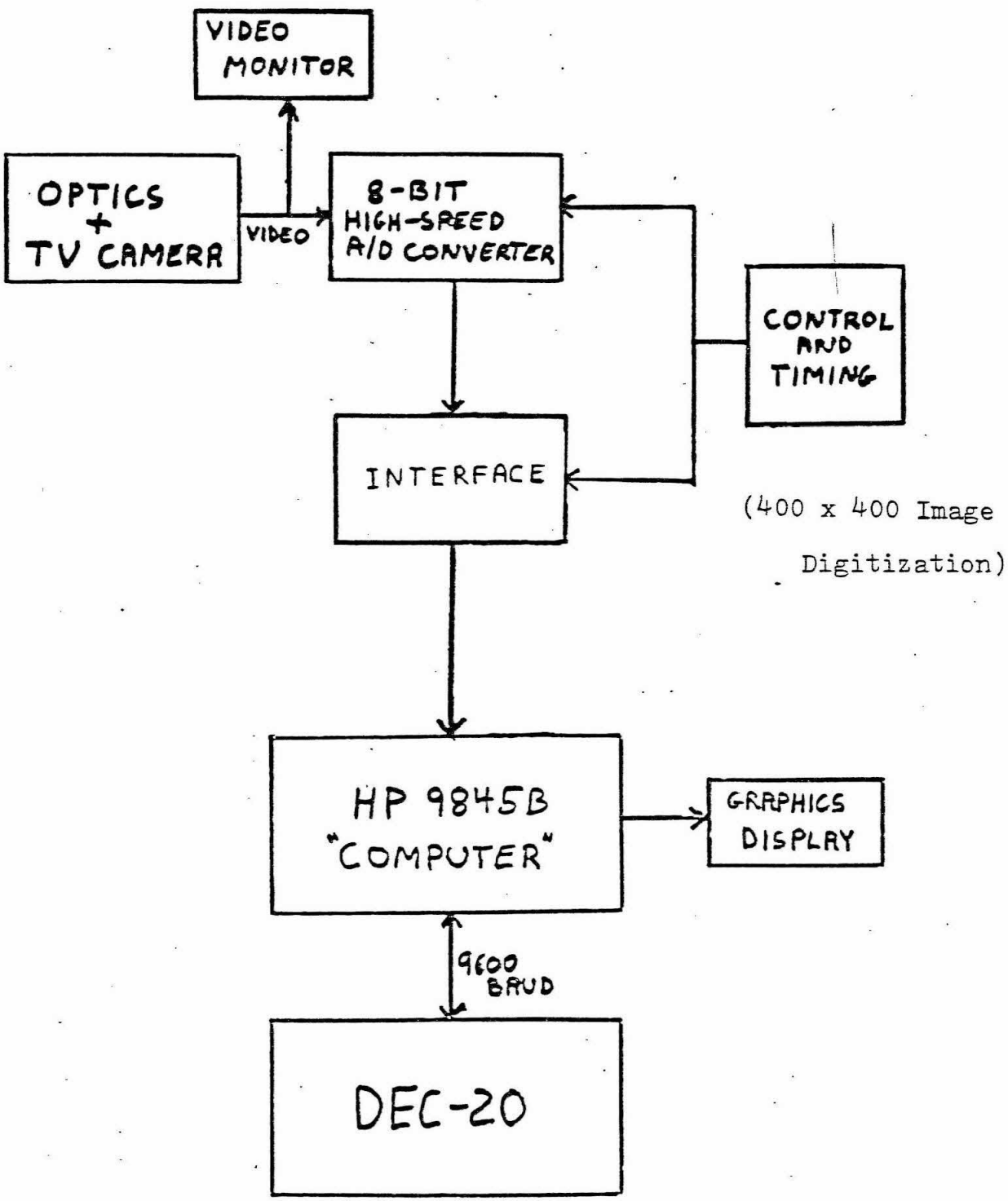


Figure A.2: Data Collection System

Appendix 2

Selected SIMULA Listings

Following are listings of some of the SIMULA programs used in this work, included in order to present details of algorithms discussed in the text.

```
!"DEPORE";
!This program removes "pores" from input 1-bit per pixel non-thinned image;
!A pore is defined as a region of white (background) pixels that (within
!the chosen window) are surrounded completely by black (ridge) pixels.
!For purposes of this program, ridge pixels are considered connected
!if they are 8-connected (diagonal ok), while for background we use
!4-connectedness.  The method used to determine which pixels are "pore"
!pixels is as follows: We recursively call a routine which examines
!the four non-diagonal neighbors of a pixel. If the pixel is black,
!nothing is done.  If it is white, it is marked as connected to the
!center pixel of the window, and we recurse.  If it is one of the window
!edge pixels (and white), then all the connected pixels are not in a pore
!(at least for the current window position). If we never reach a white
!edge-of-window pixel, then we are on a pore pixel, and we turn all of the
!"connected" pixels black (fill in the pore);

BEGIN
    EXTERNAL CLASS io20,pict1,pict8;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;

    REF(pict1) picin,picout,picconn;
    TEXT line,infilename,outfilename;
    INTEGER leftedge, rightedge, topedge, bottomedge,rr,cc,row,col;
    INTEGER windowsize,halfwindow,uplim;

    BOOLEAN PROCEDURE porepixel(r,c); INTEGER r,c;
       !This procedure checks if the pixel at (r,c) is a porepixel,
       !as described above;
    BEGIN
         IF picin.pixel(r,c)=1 or picconn.pixel(r,c)=1
         THEN porepixel:=TRUE ELSE
            !if we've run into a ridge pixel then we dont yet know that we
            !didnt start at a porepixel. Same is true if run into an
            !alredy traversed pixel;
         BEGIN
              IF c=leftedge OR c=rightedge OR r=topedge OR r=bottomedge
                 !if we are at the edge of the window;
              THEN porepixel:=FALSE
                 !if we hit a window edge then all
                 !the pixels traversed so far are not in a pore;
              ELSE !we are on a background pixel, not at window edge;
              BEGIN
                   picconn.putpixel(r,c,1); !set the bit saying this pixel
                                            !is connected to center pixel;
                   IF porepixel(r-1,c) THEN
                   BEGIN
                       IF porepixel(r+1,c) THEN
                       BEGIN
                           IF porepixel(r,c-1) THEN
                           BEGIN
                               IF porepixel(r,c+1) THEN porepixel:=TRUE;
                           END;
                       END;
                   END;
                      !if all the neighbors are porepixels then so are we;
              END;
         END;
```

```
  END of porepixel;


  line:-sysin.image;          !input buffer for TTY;
outtext("Name (XXX-YYY) of file (XXX-YYY.BIM) from which to remove pores: ");
  breakoutimage;
  inimage;
  infilename:-copy(line.strip);      !get rid of trailing blanks;

  outtext("Name (WWW-XXX) of file (WWW-XXX.BIM) for output image: ");
  breakoutimage;
  inimage;
  outfilename:-copy(line.strip);       !get rid of trailing blanks;


  picin:-NEW pict1;
  picconn:-NEW pict1;

  picin.load(infilename);    !load the input data;

  windowsize:=15;     !must be odd;
  halfwindow:=(windowsize-1)/2;
  uplim:=480-halfwindow;

  FOR row:=1+halfwindow STEP 1 UNTIL uplim DO
    BEGIN
      outtext("."); breakoutimage;
      FOR col:=1+halfwindow STEP 1 UNTIL uplim DO
          !move the window over the whole image, without running the window
          !off the edges;
      BEGIN
           IF picin.pixel(row,col)=0 THEN !if center pixel is background;
           BEGIN
                     leftedge:=col-halfwindow;
                     rightedge:=col+halfwindow;
                     topedge:=row-halfwindow;
                     bottomedge:=row+halfwindow;  !compute the
                       !coords of window edges;

                IF porepixel(row,col) THEN
                BEGIN
                     FOR rr:=topedge STEP 1 UNTIL bottomedge DO
                         FOR cc:=leftedge STEP 1 UNTIL rightedge DO
                         IF picconn.pixel(rr,cc)=1 THEN
                            picin.putpixel(rr,cc,1);
                             !set all pore pixels to black (fill pore);
                 END;
                 FOR rr:=topedge STEP 1 UNTIL bottomedge DO
                     FOR cc:=leftedge STEP 1 UNTIL rightedge DO
                     picconn.putpixel(rr,cc,0);  !clear flag bits;
           END;
      END;
    END;

  picin.store(outfilename);  !write out the image;


END of depore;
```

```
!"DESPUR";
!This program removes "spurs" from a thinned fingerprint image.
!A "spur" is defined as a sequence of pixels bounded on one end by
!a "ridge-end" pixel (one neighbor), and on the other end by a "fork" pixel
!(def'n complicated, see "FORK.SIM"), of less than a pre-defined length.
!They are removed as follows: We search for a ridge-end pixel. From there, we
!traverse the string of 2-neighbor pixels, until re run into one with
!more than two neighbors, or we have gone too far. If the >2 nbr pixel is
!not a fork pixel, it is marked for deletion along with the rest of the
!spur.  If it isnt a fork pixel, it is kept. The actual deletion is done in
!a seperate pass, so that no side-effects occur (both branches of a short
!Y at the end of a ridge will be removed;
!Then a pass of the thinner is invoked to clean up any extra pixels;

BEGIN
    EXTERNAL CLASS io20,pict1,pict8;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;

    REF(pict1) picin,encountered,delete,picout;
    REF(pict8) pictemp;
    TEXT line,infilename,outfilename;
    INTEGER maxspur, minridge, row, col, pixels, newrow, newcol, i;
    INTEGER ARRAY rowcoords[1:100], colcoords[1:100];

    PROCEDURE processpixel(r,c); INTEGER r,c;
      !This procedure, which is invoked recursively, does all the work;
    BEGIN
       BOOLEAN foundnbr;

       PROCEDURE markfordelete(howmany); INTEGER howmany;
         !marks the first "howmany" of the so far encountered pixels
         !for deletion;
       BEGIN
            INTEGER i;
            FOR i:=1 STEP 1 UNTIL howmany DO
              delete.putpixel(rowcoords[i],colcoords[i],1);
                 !mark pixels for deletion;
       END of markfordelete;

       PROCEDURE cleanup;
       !resets all the "encountered" bits, and the pointer into the
       !stack of encountered pixels;
       BEGIN
            INTEGER i;

            FOR i:=1 STEP 1 UNTIL pixels DO
                encountered.putpixel(rowcoords[i],colcoords[i],0);
                     !zero the "encountered" flags;

            pixels:=0;   !reset the stack pointer;
       END of cleanup;

       BOOLEAN PROCEDURE forkpixel(rr,cc); INTEGER rr,cc;
       !returns true if (rr,cc) is a fork pixel;
       BEGIN
            INTEGER nbrs,changes;
```

```
        INTEGER PROCEDURE nnbrs(picture,row,col);
        INTEGER row,col;
        REF(pict1)picture;
        !This procedure returns as its value the number (0-8) of neighbors
        !of the current pixel which are "on" (value=1);
        BEGIN
        INTEGER num,i,nvalue;
        num:=0;       !init number of numbered neighbors;
        FOR i:=1 STEP 1 UNTIL 8 DO       !for each neighbor;
        BEGIN
            nvalue:=picture.pixnbr(row,col,i);  !value of neighbor pixel;
            IF nvalue>0 THEN
                BEGIN
                num:=num+1;   !increment neighbor count;
                END of IF;
        END of FOR;
        nnbrs:=num;
    END of nnbrs;

    INTEGER PROCEDURE nbrchanges(picture,row,col);
    INTEGER row,col;
    REF(pict1)picture;
     !This procedure returns as its value the number of 'changes' from
     !on to off or vice versa in the neighborhood of the specified pixel;

     BEGIN
     INTEGER num,i;
     num:=0;       !init number of changes;
     FOR i:=1 STEP 1 UNTIL 8 DO       !for each neighbor;
         BEGIN
              IF picture.pixnbr(row,col,i)\=picture.pixnbr(row,col,i-1)
              THEN num:=num+1;        !count changes;
              !note that nbr 0 is same as nbr 8;
         END of FOR;
         nbrchanges:=num;
     END of nbrchanges;


         nbrs:=nnbrs(picin,rr,cc);   !number of neighbors of this pixel;
         changes:=nbrchanges(picin,rr,cc);  !# changes in nbrhd;
         IF nbrs>=3 AND nbrs <=5 AND changes>=6 THEN
            forkpixel:=TRUE ELSE forkpixel:=FALSE;
    END of forkpixel;

INTEGER PROCEDURE getrow(r,c,nbr); INTEGER r,c,nbr;
!This procedure returns as its value the row coordinate of the neighbor
!pixel of (r,c) specified by 'nbr';
BEGIN
INTEGER newrow;
        newrow:=r;          !if nbr=4 or nbr=8;
        IF nbr=1 OR nbr=2 OR nbr=3 THEN newrow:=r-1;
        IF nbr=7 OR nbr=6 OR nbr=5 THEN newrow:=r+1;    !the row values;
getrow:=newrow;
END of getrow;


INTEGER PROCEDURE getcol(r,c,nbr); INTEGER r,c,nbr;
!This procedure returns as its value the column coordinate of the
!neighbor pixel of (r,c) specified ny 'nbr';
BEGIN
INTEGER newcol;
        newcol:=c;          !if nbr=2 or nbr=6;
        IF nbr=1 OR nbr=8 OR nbr=0 OR nbr=7 THEN newcol:=c-1;
```

```
        IF nbr=3 OR nbr=4 OR nbr=5 THEN newcol:=c+1;
getcol:=newcol;
END of getcol;

   encountered.putpixel(r,c,1);      !mark pixel as encountered;
   pixels:=pixels+1;                 !increment count of encountered pixels;
   rowcoords[pixels]:=r;             !and save row and col info;
   colcoords[pixels]:=c;
   IF picin.pixnbrnum(r,c)=0  THEN    !are on an isolated pixel;
   BEGIN
        markfordelete(pixels);       !delete it;
        cleanup;
   END ELSE

   IF picin.pixnbrnum(r,c)=1 AND pixels > 1 THEN
      !if we hit another ridge end (not the original one);
   BEGIN
      IF pixels < minridge THEN      !if ridge is too short;
      BEGIN
        markfordelete(pixels);       !mark ridge pixels for deletion and;
        cleanup                      !cleanup;
      END ELSE
      cleanup;                       !else just clean-up (no deletions);
   END ELSE

   IF picin.pixnbrnum(r,c)=2 OR (picin.pixnbrnum(r,c)=1 AND
      pixels=1) THEN          !we are still going along a ridge;
                              !or are at first pixel of ridge;
   BEGIN
      foundnbr:=FALSE;               !havent found next pixel to go to;
      IF pixels >= maxspur THEN      !"spur" is too long;
      cleanup ELSE                   !clean-up (no deletions);
      BEGIN                          !find new neighbor and recurse;
         FOR i:= 1 STEP 1 UNTIL 8 DO  !for each neighbor;
         BEGIN
            IF picin.pixnbr(r,c,i)=1 AND
               encountered.pixel(getrow(r,c,i),getcol(r,c,i))=0 THEN
               !if neighbor is "on", and hasn't been used;
            BEGIN
                 newrow:=getrow(r,c,i);
                 newcol:=getcol(r,c,i);  !row and column for neighbor;
                 foundnbr:=TRUE;         !found where to go;
            END;
         END of FOR;

      IF foundnbr THEN
      processpixel(newrow,newcol) ELSE
      BEGIN
      outtext("Horrible error...didnt find a neighbor, we should have");
      outimage;
         !recurse, pointing at our un-encountered neighbor;
      END;
     END;
   END ELSE

   IF picin.pixnbrnum(r,c) > 2 THEN  !almost done;
   BEGIN
      IF forkpixel(r,c) THEN         !if on a fork;
      BEGIN
         markfordelete(pixels-1);    !delete all but the fork pixel;
         cleanup;
      END ELSE                       !not a fork pixel;
      BEGIN
```

```
        markfordelete(pixels);      !delete all;
        cleanup;
      END of IF;
   END of IF;

END of processpixel;

PROCEDURE DoAPass;
BEGIN
pixels:=0;                  !init count of "encountered" pixels;

FOR row:= 1 STEP 1 UNTIL 400 DO
BEGIN
    IF mod(row,5)=0 THEN BEGIN outtext("."); breakoutimage; END;
    FOR col:= 1 STEP 1 UNTIL 400 DO   !for each pixel in input image;
    IF  picin.pixel(row,col)=1 THEN   ! if pixel is "on";
    BEGIN
      IF picin.pixnbrnum(row,col)=1 OR picin.pixnbrnum(row,col)=0 THEN
       !if is a ridge end (or isolated pixel);
        processpixel(row,col);      !all the work happens here;
    END of FOR;
END of FOR;

FOR row:= 1 STEP 1 UNTIL 400 DO
    FOR col:= 1 STEP 1 UNTIL 400 DO    !for each pixel;
    IF delete.pixel(row,col)=1 THEN picin.putpixel(row,col,0);
       !if pixel is marked for deletion, do it;

END of DoAPass;

PROCEDURE copypict1to8(p1,p8); REF(pict1) p1; REF(pict8) p8;
!This procedure copies a single-bit (binary) picture into an 8-bit;
!(gray level) picture. Off (0) bits become pixels with value 0, while;
!on (1) bits become pixels with value 1;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p1.pixel(row,col)=1 THEN p8.putpixel(row,col,1)
       ELSE p8.putpixel(row,col,0);  !transfer the values;
END of copypict1to8;

PROCEDURE copypict8to1(p8,p1); REF(pict8) p8; REF(pict1) p1;
!This procedure copies an 8-bit (grey-level) picture into a 1-bit;
!(binary) image. Pixels with value 0 become off(0) pixels, and;
!pixels with value >=1 become on(1) pixels.;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p8.pixel(row,col)=0 THEN p1.putpixel(row,col,0)
       ELSE p1.putpixel(row,col,1);   !do the transfer;
END of copypict8to1;

PROCEDURE usetemplate(pict,t1,t2,t3,t4,t5,t6,t7,t8,t9,newval,rots);
INTEGER newval,rots,t1,t2,t3,t4,t5,t6,t7,t8,t9; REF(pict8) pict;
!This procedure operates on an 8-bit (grey-level) picture ('pict').;
!The template array is used as the pattern which the neighbor cells;
!of each cell in the picture (and the cell itself) must match in order;
!for the value of that cell to be changed to 'newval';
!The cell numbering scheme is:
!                                  1 2 3
!                                  8 9 4
```

```
!                                7 6 5
!
!The possible values for the entries in the template array are 0-255
!(which correspond to the possible values of a pixel), and -1, which
!implies that that spot in the template is a 'dont care' (i.e. it matches
!any pixel, no matter what its value).
!The parameter 'rots' is used to specify if rotations of the template are
!to be used for matching as well. Rot=0 ==> use no rotations.
!Rot=4 ==> use all 4 4-rotations. Rot=8 ==> use all 8-rotations.;
BEGIN
  INTEGER row,col,i,nbrptr;
  INTEGER ARRAY nhood[1:9],t[1:9];
  outtext(":"); breakoutimage;
  t[1]:=t1; t[2]:=t2; t[3]:=t3; t[4]:=t4; t[5]:=t5; t[6]:=t6;
  t[7]:=t7; t[8]:=t8; t[9]:=t9; !move the template into its array;
  FOR row:=1 STEP 1 UNTIL 480 DO
    FOR col:=1 STEP 1 UNTIL 480 DO BEGIN  !for each pixel in the image;
      IF ((t[9]=-1) OR (t[9]=pict.pixel(row,col))) THEN
      BEGIN     !if center cell of template is a dont care, or it matches
                !the current cell in the picture, then go on;
       FOR i:=1 STEP 1 UNTIL 9 DO
         nhood[i]:=pict.pixnbr(row,col,i);   !get the neighborhood data;

        IF rots=0 THEN BEGIN
IF templatematch(t,nhood,1) THEN BEGIN
            pict.putpixel(row,col,newval);
            END;
          !check for template match...do not rotate template (start at 1);
          END
        ELSE IF rots=4 THEN  BEGIN
        FOR nbrptr:=1 STEP 2 UNTIL 7 DO  !do all 4 4-rotations;
         IF templatematch(t,nhood,nbrptr) THEN BEGIN
                              pict.putpixel(row,col,newval);
            !and check for a template match;
                              END; END
        ELSE IF rots=8 THEN BEGIN
        FOR nbrptr:=1 STEP 1 UNTIL 8 DO  !do all 8 8-rotations;
        IF templatematch(t,nhood,nbrptr) THEN BEGIN
                    pict.putpixel(row,col,newval);
            !and check for template matches;
                    END; END
          ELSE BEGIN
         outtext(" Illegal rotation spec. in call to 'usetemplate'");
             outimage; halt; END;
      END of IF;
      END of FOR;
END of usetemplate;



BOOLEAN  PROCEDURE templatematch(t,nhood,nbrptr);
INTEGER ARRAY t,nhood;INTEGER nbrptr;
 !This routine checks to see if the template in array 't' matches
 !the pixels in the array 'nhood'. The template is rotated as specified
 !by 'nbrptr' (nbrptr is the template cell which ends up in the '1'
 !cell position after template rotation;
 BEGIN
   BOOLEAN okay;
   INTEGER cell,modcell;
   okay:=TRUE;   !initialize our flag;
   IF ((t[9]\=-1) AND (t[9]\=nhood[9]))  THEN
      templatematch:=FALSE  !if we definitely dont have a match;
      ELSE BEGIN FOR cell:=1 STEP 1 UNTIL 8 DO
```

```
            !for each cell in the neighborhood;
            BEGIN
            modcell:=MOD(nbrptr+cell-2,8)+1; !rotate our template;
            IF (t[modcell]\=-1) THEN BEGIN  !if this spot in template not
                                            !a dont care;
            IF t[modcell]\=nhood[cell] THEN okay:=FALSE; !didnt match;
            END of if;
            END of FOR;
            templatematch:=okay;
        END of IF;
    END of templatematch;

    line:-sysin.image;
    outtext('Name (XXX) of thinned image (XXX.BIM) to despur: ');
    breakoutimage;
    inimage;
    infilename:-copy(line.strip);       !get rid of trailing blanks;

    outtext('Name (YYY) of file (YYY.BIM) for output: ');
    breakoutimage;
    inimage;
    outfilename:-copy(line.strip);      !the output filename;


    picin:-NEW pict1;           !input image;
    picout:-NEW pict1;          !output image;
    encountered:-NEW pict1;     !bits saying we traversed this pixel;
    delete:-NEW pict1;          !bits saying which pixels must go;
    pictemp:-NEW pict8;         !temp buffer;


    maxspur:=10;                !anything longer (in pixels) stays;
    minridge:=7;                !any shorter complete ridge goes;

    picin.load(infilename);     !load the numbered image;

    DoAPass;                    !run through three times;
    DoAPass;
    DoAPass;


    picin.store(outfilename);           !save the image;
    delete.store(copy("DESPUR-DELETIONS"));  !see what was removed;


END of DESPUR;
```

```
!"FEATUR":
!This program processes an .BIM file which is a thinned
!fingerprint image. It finds all forks and ridge-ends. A fork is
!defined as any pixel that has 3,4 or 5 neighbors, and 6 or more
!"changes" in the pixel's neighborhood (a change is a switch from on to
!off or vice versa as we go around the pixels in order (say 1 to 8);
!Any pixel with 6, 7 or 8 neighbors represents an error (we are assuming
!the input image was thinned to one pixel). The output is a list
!(in a file) of for each feature: 1) its sequence number,
!and 2) the row and column of the critical pixel;
!Also output is an .BIM file with pixels on only where each feature is;
!Also, features are deleted if they are at (or very near to) the edge
!of the actual area of the print (see notebook for details). Basically
!the method used is to look for empty (no pixels) areas in a window
!surrounding the feature under consideration;

BEGIN
    EXTERNAL CLASS io20,pict1,pict8;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;

    REF(pict1) picin;
    REF(pict1) picout;
    REF(io20) listfile;
    TEXT line,infilename,outfilename,listfilename;
    INTEGER row,col,nbrs,changes,features,StripL,StripW;
    INTEGER ARRAY ulr[1:8], ulc[1:8], lrr[1:8], lrc[1:8];

    BOOLEAN PROCEDURE NotAtEdge(r,c); INTEGER r,c;
    !Returns TRUE if pixel (r,c) is not too close to image edge;
    BEGIN
        INTEGER strip, emptystrips;

        BOOLEAN PROCEDURE StripEmpty(ulrow,ulcol,lrrow,lrcol);
                        INTEGER ulrow,ulcol,lrrow,lrcol;
        !Searches the strip whose upper-left and lower-right bounds
        !are defined by the parameters. Returns TRUE if there are no "on"
        !pixels in the strip;
        BEGIN
            INTEGER rr,cc;

            StripEmpty:=TRUE;   !default result;

            FOR rr:=ulrow STEP 1 UNTIL lrrow DO
                FOR cc:=ulcol STEP 1 UNTIL lrcol DO
                IF picin.pixel(rr,cc)=1 THEN StripEmpty:=FALSE;
                   !we got one;

        END of StripEmpty;


        emptystrips:=0;       !init counter;

        FOR strip:=1 STEP 1 UNTIL 8 DO    !for all 8 strips;
        BEGIN
            IF StripEmpty(r+ulr[strip],c+ulc[strip],r+lrr[strip],c+lrc[strip])
               !the parameters are the coords of the upperleft and lowerright
               !corners of the strip;
```

```
        THEN emptystrips:=emptystrips+1;  !increment counter;
    END of FOR;

    IF emptystrips >= 2 THEN NotAtEdge:=FALSE ELSE NotAtEdge:=TRUE;
       !if get at least 2 empty strips we are at edge;

END of NotAtEdge;

PROCEDURE InitStrips;
!initializes the offsets from the center pixel of the upper-left
!and lower right corner of each of the eight strips (see notebook pg. 71);
BEGIN
    ulr[1]:=ulr[2]:=ulr[3]:=ulr[8]:=-1*StripL;
    ulr[4]:=ulr[7]:=1;
    ulr[5]:=ulr[6]:=StripL-StripW+1;

    ulc[1]:=ulc[6]:=ulc[7]:=ulc[8]:=-1*StripL;
    ulc[2]:=ulc[5]:=1;
    ulc[3]:=ulc[4]:=StripL-StripW+1;

    lrr[1]:=lrr[2]:=-1*StripL+StripW-1;
    lrr[3]:=lrr[8]:=-1;
    lrr[4]:=lrr[5]:=lrr[6]:=lrr[7]:=StripL;

    lrc[1]:=lrc[6]:=-1;
    lrc[2]:=lrc[3]:=lrc[4]:=lrc[5]:=StripL;
    lrc[7]:=lrc[8]:=-1*StripL+StripW-1;

END of InitStrips;

INTEGER PROCEDURE nnbrs(picture,row,col);
   INTEGER row,col;
   REF(pict1)picture;
    !This procedure returns as its value the number (0-8) of neighbors
    !of the current pixel which are on;
BEGIN
INTEGER num,i,nvalue;
    num:=0;      !init number of numbered neighbors;
    FOR i:=1 STEP 1 UNTIL 8 DO      !for each neighbor;
        BEGIN
            nvalue:=picture.pixnbr(row,col,i);  !value of neighbor pixel;
            IF nvalue>0 THEN
                BEGIN
                num:=num+1;   !increment neighbor count;
                END of IF;
        END of FOR;
    nnbrs:=num;
END of nnbrs;

INTEGER PROCEDURE nbrchanges(picture,row,col);
   INTEGER row,col;
   REF(pict1)picture;
    !This procedure returns as its value the number of 'changes' from
    !on to off or vice versa in the neighborhood of the specified pixel;

BEGIN
INTEGER num,i;
    num:=0;      !init number of changes;
    FOR i:=1 STEP 1 UNTIL 8 DO      !for each neighbor;
        BEGIN
             IF picture.pixnbr(row,col,i)\=picture.pixnbr(row,col,i-1)
             THEN num:=num+1;       !count changes;
             !note that nbr 0 is same as nbr 8;
```

```
        END of FOR;
    nbrchanges:=num;
END of nbrchanges;

line:-sysin.image;
outtext("Name (XXX) of thinned image (XXX.BIM) to find features in: ");
breakoutimage;
inimage;
infilename:-copy(line.strip);        !get rid of trailing blanks;

outtext("Name (YYY) of file (YYY.BIM) for feature dot output: ");
breakoutimage;
inimage;
outfilename:-copy(line.strip);       !the output filename;

outtext("Name (ZZZ) of file (ZZZ.DAT) for feature coordinate output: ");
breakoutimage;
inimage;
listfilename:-copy(line.strip);      !get rid of trailing blanks;
listfile:-NEW io20(conc(listfilename,".DAT"));
listfile.write.ascii.open(blanks(80));

StripL:=16;          !width of search strip for removing features near edge;
StripW:=10;          !same, but width;

InitStrips;          !init the corner offsets of the strips;

picin:-NEW pict1;
picout:-NEW pict1;

picin.load(infilename);      !load the numbered image;

features:=0;                         !initially we've found none;

FOR row:=1 STEP 1 UNTIL 400 DO
  BEGIN outtext("."); breakoutimage;
    FOR col:=1 STEP 1 UNTIL 400 DO
       IF picin.pixel(row,col)=1 THEN        !if we are on a pixel;
       BEGIN
        nbrs:=nnbrs(picin,row,col);          !number of nbrs of this pixel;
        changes:=nbrchanges(picin,row,col); !how many changes in nbrhd;
        IF (nbrs>=3 AND nbrs <=5 AND picin.pixel(row,col)>0 AND changes>=6)
           OR (nbrs=1)
        THEN                                 !we have a feature pixel;
        BEGIN IF NotAtEdge(row,col) THEN
          BEGIN
             features:=features+1;  !increment features count;
             picout.putpixel(row,col,1);     !write out a pixel here;
    listfile.outint(features,5);             !sequence number of feature;
    listfile.outint(row,5);
    listfile.outint(col,5);                  !and row and col to list file;
    listfile.outimage;
          END ELSE picout.putpixel(row,col,0); !write 0 pixel;
       END ELSE picout.putpixel(row,col,0); !write a zero pixel;

        IF nbrs>=6 THEN
        BEGIN
             outtext("pixel at row=");
             outint(row,3); outtext(", col="); outint(col,3);
             outtext(" has"); outint(nbrs,2); outtext(" neighbors.");
             outimage;          !this is an error (improper thinning);
        END;
      END of IF;
```

```
        END of FOR;
    outimage;
    outint(features,5); outtext(" features encountered");outimage;
    picout.store(outfilename);          !save the overlay file;
    listfile.close;                     !close the listfile;
    listfile.release;                   !and release the jfn;
END of FEATUR;
```

```
!"FILTER"
!This program uses as input a picture digitized to 8-bits,
!and outputs a filtered version of the same picture.
!The filter function used  is 6db down
!at about .25 cycles/pixel-distance;
BEGIN
EXTERNAL CLASS io20,pict1,pict8;
EXTERNAL PROCEDURE enterdebug;
EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
EXTERNAL INTEGER PROCEDURE lshift,lxor;
EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
EXTERNAL CHARACTER PROCEDURE findtrigger;
EXTERNAL PROCEDURE halt;

REF(pict8) picin,picout;
TEXT line,infilename,outfilename;

PROCEDURE filterit;
BEGIN
 INTEGER r,c;
 REAL newpixval;
 FOR r:=1 STEP 1 UNTIL 400 DO       !for each row;
   FOR c:=1 STEP 1 UNTIL 400 DO     !for each column;
   BEGIN
   !npixnbr used rather than pixnbr so pixels off the edge come back
   !with the value of the "center" pixel, rather than 255, which would
   !screw up the width of the histogram;

newpixval:=picin.npixnbr(r,c,1)+2*picin.npixnbr(r,c,2)+picin.npixnbr(r,c,3);
newpixval:=newpixval+2*picin.npixnbr(r,c,4)+picin.npixnbr(r,c,5);
newpixval:=newpixval+2*picin.npixnbr(r,c,6)+picin.npixnbr(r,c,7);
newpixval:=newpixval+2*picin.npixnbr(r,c,8)+4*picin.pixel(r,c);
picout.putpixel(r,c,ENTIER(0.5+newpixval/16));  !the new value;
    END;
  END of filterit;


  line:-sysin.image;
  outtext("Name (XXX) of file (XXX.IMG) to filter: ");
  breakoutimage;
  inimage;
  infilename:-copy(line.strip);   !file name for input w/o trailing blanks;

  outtext("Name (YYY) of file (YYY.IMG) for output file: ");
  breakoutimage;
  inimage;
  outfilename:-copy(line.strip);   !filename for output file;

  picin:-NEW pict8;      !the input picture;
  picout:-NEW pict8;     !the output picture;

  picin.load(infilename);  !load the input image;
  filterit;                !like the name says...;
  picout.store(outfilename);  !and write out the results;

END of filte
```

```
!"GRAPH";
!This program takes as input the output produced by the "RIGNUM" program,;
!which is a thinned and ridgenumbered fingerprint image (PICT8). In this;
!image, pixels with value 0 are background (off) pixels, while the value;
!of any pixels not equal to 0 is the name of the ridge to which they belong.;
!This program is responsible for producing as output a graph structure (as;
!an ascii file) representing the adjacency relationships of the ridges in;
!the input image. This is produced during 4 scans through the input image,;
!in the four possible up-down and left-right combinations.  As the input image;
!is scanned, vectors are "grown" from the endpoints of the ridges, until;
!they intersect another ridge, at which point an entry is made in the;
!adjacency graph. Four scans are necessary, since for a given scan, vectors;
!can only be grown over a 90 degree range of angles.;
!Vectors are grown in directions perpindicular to the local slope at the;
!end of the ridge in question, using a DDA algorithm;
!"Vector" pixels have the high-order bit set (value>255) to differentiate;
!them from simple "ridge" pixels;

    BEGIN
    EXTERNAL CLASS io20,pict1,pict8,pict9;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;


    pict9 CLASS vect9;
          !This is where live the routines to help in accessing the
          !various bit fields in the nine bits, used to store vector
          !parameters. The fields are: (bit 0 = LSB, bit 8 = MSB);
          !    bit 0 : Is X or Y the direction of greater movement
          !            (and therefore the direction always stepped by one
          !            pixel in the DDA algorithm)? 1=>X, 0=>Y.
          !    bit 1 : Should the variable of greater movement (i.e. the
          !            one being stepped by one) be incremented or decremented
          !            each time? 1=>incremented, 0=>decremented.
          !    bits 2-4: Slope (dy/dx, or dx/dy where the denom. variable is
          !              always the "move by 1" variable. Range is 000 to
          !              100 (a fraction between 0 and 1, in steps of .25).
          !    bit 5 : Sign of the slope (i.e., do we increment or decrement
          !            by that amount)? 0=> + (increment), 1=> - (decrement).
          !    bits 6-7 : Fractional part of the slowly (i.e., noy by 1 each
          !               time) variable. Range is 00-11 (i.e. 0 to .75 in
          !               steps of 0.25).
          !    bit 8 : Reserved for future use (sounds official, huh?).;
          ! see Newmann and Sproull (1st ed. page 43) for DDA algorithm;

    BEGIN
         REF(pict9) vpict;

         REAL PROCEDURE getslope(row,col); INTEGER row,col;
         !returns slope, including sign, as real value.;
         BEGIN
         INTEGER allbits,slopebits,signbit;
              allbits:=vpict.pixel(row,col);     !get the value;
              slopebits:=LSHIFT(LAND(allbits,28),-2);     !kill all but bits 2-4
                                                          !and justify;
              signbit:=LSHIFT(LAND(allbits,32),-5);       !same for sign bit -
                                                          !0==>+;
              IF slopebits > 4 THEN
```

```
    BEGIN
        outtext("illegal slope value in 'getslope'"); outimage;
    END;

    IF signbit=0 THEN getslope:=slopebits/4.0   !0<=slope<=1.0;
    ELSE getslope:=-1.0*slopebits/4.0;          !-1.0<=slope<=0.0;
END of getslope;


PROCEDURE setslope(row,col,slope); INTEGER row,col; REAL slope;
!set slope bits at (row,col) (including sign). -1<=slope<=1.;
BEGIN
INTEGER oldbits,newbits,slopebits,signbit;
    IF ABS(slope) > 1 THEN
    BEGIN
        outtext("illegal slope in 'setslope'"); outimage;
    END;

    oldbits:=vpict.pixel(row,col);    !get current value;
    slopebits:=LSHIFT(ENTIER(0.5+4.0*ABS(slope)),2); !the slope field,
                                                     !in place;
    signbit:=IF slope>0 THEN 0 ELSE 32;   !and the sign bit;
    newbits:=LOR(slopebits,signbit);      !both fields;
    newbits:=LOR(LAND(oldbits,256+128+64+2+1),newbits);
        !kill off old sign and slope field, and OR in new;
    vpict.putpixel(row,col,newbits);  !and write it in;
END of setslope;


BOOLEAN PROCEDURE xgreater(row,col); INTEGER row,col;
!answers the question: "Is X the direction of greater movement
!for the vector growth process?". If not, then Y is.;
BEGIN
INTEGER allbits,greaterbit;
        allbits:=vpict.pixel(row,col);  !all the fields;
        greaterbit:=LAND(allbits,1);    !the "x greater" bit;
        xgreater:=IF greaterbit=1 THEN TRUE ELSE FALSE;
                    !our returned answer;
END of xgreater;


PROCEDURE setxgreater(row,col,itis); INTEGER row,col; BOOLEAN itis;
!sets the bit saying that the change in x is greater if called
!with TRUE, else clears the bit;
BEGIN
INTEGER oldbits,newbits;
        oldbits:=vpict.pixel(row,col);
        newbits:=IF itis THEN 1 ELSE 0;   !new bit;
        newbits:=LOR(LAND(oldbits,511-1),newbits);
           !kill old low order bit, and OR in new one;
        vpict.putpixel(row,col,newbits);
           !and store it away;
END of setxgreater;

BOOLEAN PROCEDURE incgreater(row,col); INTEGER row,col;
!answers the question "do we increment (vs. decrement) the variable;
!that changes fastest (i.e., the one being changed by 1 each time)?";
BEGIN
INTEGER allbits,incbit;
        allbits:=vpict.pixel(row,col);          !all the fields;
        incbit:=LAND(allbits,2);                !the bit in question;
        incgreater:=IF incbit=2 THEN TRUE ELSE FALSE;
                    !the answer;
```

```
      END of incgreater;

      PROCEDURE setincgreater(row,col,yes); INTEGER row,col; BOOLEAN yes;
      !If called with TRUE, sets bit to indicate that the variable with;
      !greater movement is to be incremented (else sets to decremented);
      BEGIN
      INTEGER oldbits,newbits;
              oldbits:=vpict.pixel(row,col);
              newbits:=IF yes THEN 2 ELSE 0;     !the bit;
              newbits:=LOR(LAND(oldbits,511-2),newbits);
                     !kill old bit 1 and OR in new one;
              vpict.putpixel(row,col,newbits);
                     !and store result;
      END of setincgreater;

      REAL PROCEDURE getfrac(row,col); INTEGER row,col;
      !returns the fractional part of the slowly varying variable;
      BEGIN
      INTEGER allbits,fracbits;
              allbits:=vpict.pixel(row,col);
              fracbits:=LSHIFT(LAND(allbits,192),-6);
                     !kill all but bits 6 and 7, and justify;
              getfrac:=fracbits/4.0;     !the fractional part;
      END of getfrac;


      PROCEDURE setfrac(row,col,fracpart); INTEGER row,col; REAL fracpart;
      !sets the fractional part of the slowly varying variable;
      !input value must be 0<=fracpart<=.75;
      BEGIN
      INTEGER oldbits,newbits,fracbits;
              IF fracpart<0 OR fracpart>.75 THEN
              BEGIN
                   outtext("illegal fractional part in call to 'setfrac'");
                   outimage;
              END;

              oldbits:=vpict.pixel(row,col);
              fracbits:=LSHIFT(ENTIER(0.5+4.0*ABS(fracpart)),6);
                     !move the field into place;
              newbits:=LOR(LAND(oldbits,256+32+16+8+4+2+1),fracbits);
                     !clear proper bits, and OR in new ones;
              vpict.putpixel(row,col,newbits);  !and write it in;
      END of setfrac;


    !Initialization code;
    vpict:-NEW pict9;        !create the pict9 to hold things;

   END of CLASS vect9;


SIMSET
BEGIN
   head CLASS graph;
   BEGIN
        BOOLEAN looking;
        REF(node) PROCEDURE getnode(n); INTEGER n;
        !This procedure runs down the chain of nodes until one whose
        !rignum equals n is found. Returns NONE if node doesnt exist;
        BEGIN
             REF(node) x; INTEGER i;
          IF NOT THIS graph.empty THEN !if there is at least one node already;
```

```
        BEGIN
          x:-first QUA node;                !first node in the graph;
          looking:=TRUE;                    !havent found right node yet;
          WHILE looking AND x =/= NONE DO   !until we find right one;
          BEGIN
               IF x.rignum = n THEN looking:=FALSE ELSE  !found it;
               x:-x.suc;                    !next node;
          END;
          getnode:-x;
        END ELSE getnode:-NONE;      !if graph empty return NONE;
    END of getnode;                         !returns NONE if no such node;

END of CLASS graph;

 link CLASS node(rignum); INTEGER rignum;
 BEGIN
      REF(path_list) pathlst;     !list of paths (links) to other nodes;
      pathlst:-NEW path_list;     !init the list;
 END of CLASS node;

 head CLASS path_list;
 BEGIN
     !list of paths to other nodes;
     BOOLEAN looking;
     REF(path) PROCEDURE getpath(n); INTEGER n;
     !This procedure runs down the chain of paths until one whose
     !nodenum equals n is found. Returns NONE if path doesnt exist;
     BEGIN
          REF(path) x; INTEGER i;
       IF NOT THIS path_list.empty THEN
          !if there is at least one node already;
        BEGIN
          x:-first QUA path;                !first path from this node;
          looking:=TRUE;                    !havent found right path yet;
          WHILE looking AND x =/= NONE DO   !until we find right one;
          BEGIN
               IF x.nodenum = n THEN looking:=FALSE ELSE  !found it;
               x:-x.suc;                    !next path;
          END;
          getpath:-x;
        END ELSE getpath:-NONE;      !if node had no paths, return NONE;
     END of getpath;                        !returns NONE if no such path;
 END of CLASS path_list;

 link CLASS path(nodenum); INTEGER nodenum;
 BEGIN
 END of CLASS path;


REF(pict8) picin;
REF(pict9) picin9;
REF(vect9) vectim;
REF(graph) adjacencygraph;
REF(io20) outfile;
TEXT line,infilename,outfilename;
INTEGER pass,endpixels,row,col,rowdot;
REAL deltax,deltay;


PROCEDURE processend;
!This procedure is called when we have found ourselves a ridge-end. It is
!responsible for deciding whether on the current pass we can begin growing
!a vector from this end (answer should be yes for 2 of the 4 passes), based
```

```
!upon the slope of the end of the ridge (i.e., upon the previously
!calculated deltax and deltay values).  If we can grow one, we place the
!first vector point in the proper neighbor pixel of the ridge end. Note
!that this placement involves operations in two data structures:
!1) naming the pixel (what ridge it belongs to) in picin9, and
!2) setting up its growth data in vectim.;
!note that npixnbr is used in test for free neighbors, rather than
!pixnbr, so that pixels off the screen show as available. Later on
!we test for being off the screen, so we dont try and put any vector
!pixels there;
BEGIN
     INTEGER newrow,newcol,neighbor,nr,nc;
     BOOLEAN xg;

     newrow:=row;
     newcol:=col;   !will become the address of the first pixel of vector;

     IF (SIGN(deltax)=SIGN(deltay)>=0 AND (pass=2 OR pass=3)) OR
        (SIGN(deltax)=SIGN(deltay)<=0 AND (pass=1 OR pass=4)) THEN
          !note that SIGN(x)=-1 if x negative, +1 if x positive,
          !and 0 if x=0.  So, we do the rest of the code if the sign
          !of deltax and deltay are the same (for passes 2 and 3),
          !or if they are different (for passes 1 and 4).;
     BEGIN
          !time to decide where to put the vector point;
          IF pass=1 THEN neighbor:=5;  !which neighbor of the endpoint;
          IF pass=2 THEN neighbor:=7;  !pixel is the best first guess for;
          IF pass=3 THEN neighbor:=3;  !our first vector pixel;
          IF pass=4 THEN neighbor:=1;

          IF picin9.npixnbr(row,col,neighbor) <> 0 THEN
             !if the first guess isnt empty;
          BEGIN
               IF picin9.npixnbr(row,col,MOD(neighbor+1,8)) <> 0 THEN
                  !if the next pixel around the neighborhood is full too;
               BEGIN
                    IF picin9.npixnbr(row,col,MOD(neighbor-1,8)) <> 0 THEN
                       !and if the next one the other way is full;
                    BEGIN
                         outtext("no place for start of vector!!!");
                         outimage;
                         outtext("row= "); breakoutimage; outint(row,4);
                         outimage;
                         outtext("col= "); breakoutimage; outint(col,4);
                         outimage;
                         halt;
                    END ELSE neighbor:=MOD(neighbor-1,8);
                        !else this is the one to use;
               END ELSE neighbor:=MOD(neighbor+1,8);
                   !or this is the correct one;
          END;
          !at this point we can assume that 'neighbor' points to the
          !correct place to put the first vector pixel;

          newrow:=getrow(row,col,neighbor);      !set up the row and col;
          newcol:=getcol(row,col,neighbor);      !for the vector point;

IF newrow>=1 AND newrow<=488 AND newcol>=1 AND newcol<=488 THEN
   !if we are not off the screen;
        BEGIN
          picin9.putpixel(newrow,newcol,256+picin9.pixel(row,col));
             !label the vector point with the name of the ridge it came
             !from, as well as turn on the hhigh bit to indicate that
```

```
          !it is a vector point, not a pixel;

        !now time to set up the data in "vectim" to go with the
        !vector point;

        IF ABS(deltax) > ABS(deltay) THEN
          BEGIN
           xg:=FALSE;
           vectim.setxgreater(newrow,newcol,FALSE) END ELSE
          BEGIN
           xg:=TRUE;
           vectim.setxgreater(newrow,newcol,TRUE);
          END;
              !based upon the x and y deltas for the ridgeend, decide
              !whether x or y is to be the direction of greater movement
              !during vector growth, and therefore the variable which
              !is always incremented or decremented by 1.  Note that
              !the apparent reversal in the above statement is due
              !to the fact that the direction of growth is perpindicular
              !to the direction of the ridgeend;

              nr:=newrow;
              nc:=newcol;          !just to get shorter names;

          IF pass=1 THEN BEGIN
          IF xg THEN vectim.setincgreater(nr,nc,TRUE)
          ELSE vectim.setincgreater(nr,nc,TRUE);
          IF xg THEN vectim.setslope(nr,nc,ABS(deltax/deltay)) ELSE
                     vectim.setslope(nr,nc,ABS(deltay/deltax));
                         END;

          IF pass=2 THEN BEGIN
          IF xg THEN vectim.setincgreater(nr,nc,FALSE)
          ELSE vectim.setincgreater(nr,nc,TRUE);
          IF xg THEN vectim.setslope(nr,nc,ABS(deltax/deltay)) ELSE
                     vectim.setslope(nr,nc,-1*ABS(deltay/deltax));
                         END;

          IF pass=3 THEN BEGIN
          IF xg THEN vectim.setincgreater(nr,nc,TRUE)
          ELSE vectim.setincgreater(nr,nc,FALSE);
          IF xg THEN vectim.setslope(nr,nc,-1*ABS(deltax/deltay)) ELSE
                     vectim.setslope(nr,nc,ABS(deltay/deltax));
                         END;

          IF pass=4 THEN BEGIN
          IF xg THEN vectim.setincgreater(nr,nc,FALSE)
          ELSE vectim.setincgreater(nr,nc,FALSE);
          IF xg THEN vectim.setslope(nr,nc,-1*ABS(deltax/deltay)) ELSE
                     vectim.setslope(nr,nc,-1*ABS(deltay/deltax));
                         END;

          !weve now set up whether to increment or decrement the faster
          !changing variable, and also the slope (complete with sign);

          vectim.setfrac(newrow,newcol,0);
          !init the fractional part of the slowly changing variable to 0;

       END of IF;
      END of IF;
END of processend;

PROCEDURE processvector;
```

```
!This procedure is called when we encounter a pixel which is a vector pixel
!We then must calculate where the next pixel in the vector is to be, and
!if that new location is free, move the vector pixel with its data there,
!and erase the old vector pixel.  However, if the new location already has
!a vector pixel, then we have a conflict (may have code to deal with this
!later). For now though, we try using the two closest neighbors instead,;
!before giving up. If the new location is a ridge pixel
!(or if we cross a ridge
!(possible due to the allowed diagonal connectivity of ridges)) then we
!have found a ridge adjacency, and make the appropriate entry in our graph;
!note that row and y are synonomous, as are col and x;
BEGIN
     INTEGER greaterinc,newrow,newcol,nbr,orignewrow,orignewcol,orignbr;
     REAL sum,newfrac;
     BOOLEAN xg;

     BOOLEAN PROCEDURE empty(nr,nc); INTEGER nr,nc;
        !This procedure checks whether the pixel at (nr,nc) is
        !'on the screen', and if so, is its value 0?;
     BEGIN
          empty:=TRUE;  !a good start;
          IF nr<1 OR nr>400 OR nc<1 OR nc>400 THEN empty:=FALSE ELSE
             !if off screen, say its not empty;
          IF picin9.pixel(nr,nc) \= 0 THEN empty:=FALSE;
             !not equal 0 ==> not empty;
     END of empty;

     BOOLEAN PROCEDURE movevector(nr,nc,nnbr); INTEGER nr,nc,nnbr;
        !This procedure actually does the moving of the vector
        !pixel to its new (and usually empty) home;
        !nr and nc are the row and column to move the pixel to,
        !and nnbr is the neighbornumber of that pixel relative to
        !the old vector pixel (row,col);
     BEGIN
          IF (nnbr=1 OR nnbr=3 OR nnbr=5 OR nnbr=7) AND
             picin9.pixnbr(row,col,MOD(nnbr-1,8)) > 0 AND
             picin9.pixnbr(row,col,MOD(nnbr-1,8)) < 256 AND
             picin9.pixnbr(row,col,MOD(nnbr+1,8)) > 0 AND
             picin9.pixnbr(row,col,MOD(nnbr+1,8)) < 256 THEN
                !though the new pixel location is free, it has
                !a ridge pixel on both sides of it, so we would
                !actually be crossing a ridge if we went on ==>
                !we found an adjacency;

          BEGIN
             setadjacency(picin9.pixel(row,col)-256,
                picin9.pixnbr(row,col,MOD(nnbr+1,8)));
                    !set up the adjacency...use of nnbr+1 is
                    !arbitrary, since it is unlikely that the
                    !neighbors at nnbr-1 and nnbr+1 belong to
                    !different ridges;
             picin9.putpixel(row,col,0);
             vectim.putpixel(row,col,0);   !clean up the vector point;
          END ELSE

               !else the new location is empty, and we havent gone
               !through a ridge, so we move the vector point to its new
               !location, and delete the old one;
             BEGIN
                  picin9.putpixel(nr,nc,picin9.pixel(row,col));
                    !move the vector name, and vector flag bit;

                  IF xg THEN vectim.setxgreater(nr,nc,TRUE) ELSE
```

```
                    vectim.setxgreater(nr,nc,FALSE);
              !move the xgreater flag;

            IF vectim.incgreater(row,col) THEN
               vectim.setincgreater(nr,nc,TRUE)
               ELSE vectim.setincgreater(nr,nc,FALSE);
               !move the incgreater flag;

            vectim.setslope(nr,nc,vectim.getslope(row,col));
              !move the slope info;

            vectim.setfrac(nr,nc,newfrac);
              !and the new fractional part;

            picin9.putpixel(row,col,0);
            vectim.putpixel(row,col,0);   !clean up old vector point;
        END;
END of movevector;

BOOLEAN PROCEDURE ridgepixel(nr,nc); INTEGER nr,nc;
   !This procedure check whether the pixel at (nr,nc) has
   !value between 1 and 255 (i.e. is a ridge pixel);
BEGIN
     ridgepixel:=FALSE;
     IF nr>=1 AND nr<=400 AND nc>=1 AND nc<=400 THEN
        BEGIN
         IF picin9.pixel(nr,nc)>=1 AND picin9.pixel(nr,nc)<=255 THEN
         ridgepixel:=TRUE;  !return FALSE if (nr,nc) is off the
                            !screen;
        END;
END of ridgepixel;

PROCEDURE gotadjacency;
   !This procedure is called when we run into a ridge pixel;
   !Its sets up the data in the adjacency graph, and clears
   !the vector pixel that did the 'running-in-to';
BEGIN
     setadjacency(MOD(picin9.pixel(row,col),256),
                  MOD(picin9.pixel(newrow,newcol),256));
        !the ridge our vector belonged to is adjacent to
        !the ridge we just encountered, so update the graph to
        !indicate so;

     picin9.putpixel(row,col,0);
     vectim.putpixel(row,col,0);  !clean up vector;
END of gotadjacency;


xg:=vectim.xgreater(row,col);  !set flag telling us if x (col) is
                               !the variable of greater change;

greaterinc:=IF vectim.incgreater(row,col) THEN 1 ELSE -1;
     !set up the amount to be added to the greater variable;

IF xg THEN   !if x (col) is the faster variable;
BEGIN
     newcol:=col+greaterinc;   !col is faster variable;
     sum:=(row+vectim.getfrac(row,col))+vectim.getslope(row,col);
         !the exact new value for row;
     newrow:=ENTIER(sum);      !the new integer row value;
     newfrac:=sum-newrow;      !the new fractional part of row;
END ELSE     !if y (row) is the faster variable;
BEGIN
```

```
     newrow:=row+greaterinc;   !do the faster one;
     sum:=(col+vectim.getfrac(row,col))+vectim.getslope(row,col);
         !exact new value for col;
     newcol:=ENTIER(sum);      !the new integer col value;
     newfrac:=sum-newcol;      !the new fractional part of col;
END;

orignewrow:=newrow;   orignewcol:=newcol;
                      !save the optimum new location for vector pixel;

nbr:=calcneighbor(row,col,newrow,newcol);
       !returns the neighbor number of (mewrow,newcol) with
       !respect to (row,col);
orignbr:=nbr;   !save this too;

IF empty(newrow,newcol) THEN  !if (newrow,newcol) is on-screen and its
                              !value is zero;
   movevector(newrow,newcol,nbr) ELSE !grow our vector there, else;

   IF ridgepixel(newrow,newcol) THEN !if its a ridgepixel;

   gotadjacency ELSE !set adjacency in graph,clear vector pixel,
                     !else its a vector pixel, and we dont want to
                     !overwrite it unless necessary, so try other
                     !neighbors of (row,col) that are in more or less
                     !the same direction as (newrow,newcol);
 BEGIN
      newrow:=getrow(row,col,MOD(nbr-1,8));
      newcol:=getcol(row,col,MOD(nbr-1,8));
      newfrac:=0; !no fractional part;
          !the previous neighbor around;

      IF empty(newrow,newcol) THEN !if not off-screen and value=0;
         movevector(newrow,newcol,nbr-1) ELSE
         IF ridgepixel(newrow,newcol) THEN
            gotadjacency ELSE  !else that too was a vector pixel,
                               !and we must try the next neighbor
                               !around the other way;
      BEGIN
           newrow:=getrow(row,col,MOD(nbr+1,8));
           newcol:=getcol(row,col,MOD(nbr+1,8));
           newfrac:=0;
               !next neighbor back the other way;

           IF empty(newrow,newcol) THEN
              movevector(newrow,newcol,nbr+1) ELSE
              IF ridgepixel(newrow,newcol) THEN
                 gotadjacency ELSE !else neither of the alternate
                                   !possible places to grow was free,
                                   !so we will grow into the original
                                   !choice, overwriting what was there
                                   !and yelling;
                 BEGIN
                   IF orignewrow>=1 AND orignewrow<=400 AND
                      orignewcol>=1 AND orignewcol<=400 THEN
                          !if new location is on screen;
                          !overwrite what was there;
                          !and clear vector pixel;
                    BEGIN
                     outtext("vector conflict at: ROW= ");
                     outint(orignewrow,4);
                     outtext("   COL= ");
                     outint(orignewcol,4); outimage;
```

```
outtext(" vector from ridge ");
outint(picin9.pixel(orignewrow,orignewcol)-256,4);
outtext(" being overwritten by vector from ridge ");
outint(picin9.pixel(row,col)-256,4); outimage; outimage;
                        movevector(orignewrow,orignewcol,orignbr);
                       END ELSE
                          BEGIN
                             picin9.putpixel(row,col,0);
                             vectim.putpixel(row,col,0);
                             !clear the vector, since we hit the screen
                             !edge;
                          END;
                        END;
           END;
      END;
END of processvector;


PROCEDURE checkpixel;
    !this procedure is invoked 400x400xfour times, for each pixel
    !for each of the passes
    !thru the image. The value (1-4) of 'pass' is used to keep track
    !of which pass we are in  (and therefore what the direction of
    !scan is).;
BEGIN
     IF itsanend THEN                        !if the current pixel is the
                                             !endpixel for a ridge. Returns
                                             !the mean dx and dy for the
                                             !last 'endpixels' other pixels;

     processend                              !check if we can grow vector
                                             !on this pass, and if so,
                                             !start it.;

     ELSE   !If not a ridge end pixel;
     BEGIN
          IF picin9.pixel(row,col) >= 256 THEN
                                  !If current pixel is a point of a;
                                  !vector, then let us grow the vector,
                                  !and see if it has hit a ridge, etc.;
          processvector;
     END;


END of checkpixel;



BOOLEAN procedure itsanend;
    !This procedure returns TRUE if the current pixel (i.e.,
    !the one at (row,col)) is the end pixel of a ridge, and
    !FALSE if not. If TRUE case, the mean of the delta x and
    !delta y displacements of pixels near the end pixel are
    !returned as deltax and deltay. In addition to the end pixel,
    !'endpixels' next to it are used for this mean calculation.;
   BEGIN
          INTEGER numnbrs,i,nbrval;
          itsanend:=FALSE;
          IF picin9.pixel(row,col) < 256 AND picin9.pixel(row,col) > 0 THEN
               !If the current pixel is NOT a vector point, and it IS a
               !ridge point.;
               BEGIN
                numnbrs:=0;
```

```
            FOR i:=1 STEP 1 UNTIL 8 DO
            BEGIN
            nbrval:=picin9.pixnbr(row,col,i);     !the value of the nbr;
            IF nbrval <256 AND nbrval > 0 THEN numnbrs:=numnbrs+1;
              !If nbr is a ridge pixel, increment the count;
            END of FOR;
            IF numnbrs=1 THEN   !1 nbr.=> ridge endpoint;
            BEGIN
                 itsanend:=TRUE;        !we do have an end;
                 calcdeltas;  !calculate deltax and deltay;
            END;
           END;
  END of itsanend;


PROCEDURE calcdeltas;
!This procedure calculates the means of the x and y deltas from the
!ridge endpoint pixel to each of the next 'endpixels' pixels along
!the ridge in question.  These deltas (deltax, deltay) will be later used
!to determine the proper direction for vector growth.  Note that
!the full number ('endpixels') of adjacent pixels are used in the
!calculation only so long as no forks are encountered.  If a fork
!(or ridge end) is encountered, the calculation of the dx and dy
!is terminated (and does not include the fork or new end pixel).
!Also note that this routine is the only part of the entire adjacency-
!determination algorithm which requires greater than a 3x3 window
!into the image. In fact, a window of (2*endpixels) x (2*endpixels)
!is nominally required;
BEGIN
     INTEGER xsum,ysum,pixelsused,r,c,nbr,newrow,newcol,oldrow,oldcol;
     INTEGER nbrvalue;
     BOOLEAN stilllooking;
     xsum:=ysum:=0;                    !init. sum of deltas;
     pixelsused:=0;                    !init. # nbr. pixels used;
     r:=row; c:=col;                   !we start at the end: (row,col);
     oldrow:=row; oldcol:=col;         !as good values as any;

     IF ridgenbrs(r,c) \= 1 THEN       !error if we dont have exactly 1 nbr;
     BEGIN
          outtext("'calcdeltas' called when pixel not a ridgeend");
          outimage;
     END ELSE WHILE pixelsused < endpixels  AND (pixelsused =0 OR
                      ridgenbrs(r,c)=2) DO
                      !keep going as long as we have used less than
                      !'endpixels', and as long as the current pixel
                      !has exactly two neighbors (unless its the original
                      !ridgeend pixel). These conditions cause the
                      !averaging process to stop if another end, or
                      !a fork is encountered;
     BEGIN
       stilllooking:=TRUE;  !we havent found a valid neighbor yet;
       FOR nbr:= 1 STEP 1 UNTIL 8 DO IF stilllooking THEN
       BEGIN
            nbrvalue:=picin9.pixnbr(r,c,nbr);   !value of neighbor;
            IF nbrvalue < 256 AND nbrvalue > 0 THEN
               !If the nbr. is a ridge point;
            BEGIN
                 newrow:=getrow(r,c,nbr);   !returns the row value for nbr;
                 newcol:=getcol(r,c,nbr);   !same for column value;
                 IF newrow\=oldrow OR newcol\=oldcol THEN
                    !if we arent going back the wrong way along ridge;
                 BEGIN
```

```
                oldrow:=r;  oldcol:=c;          !update stuff;
                r:=newrow;  c:=newcol;          !update r and c;
                stilllooking:=FALSE;            !we found it;

            END of IF;
        END of IF;
    END of FOR;
    xsum:=xsum+c-col;
    ysum:=ysum+r-row;    !update xsum and ysum;
    pixelsused:=pixelsused+1;    !update pixel used count;

END of WHILE;
deltax:=xsum/pixelsused;
deltay:=ysum/pixelsused;        !finally, the results;

END of calcdeltas;


PROCEDURE setadjacency(ridge1,ridge2); INTEGER ridge1,ridge2;
!This procedure is called when we have found an two ridges to be adjacent;
!It checks to make sure that each of those ridges have nodes in our
!adjacency graph, and if not, it creates the nodes. It then puts in
!a bi-directional link between the nodes, indicating the adjacency
!relationship -- but only if that link doesnt already exist;
BEGIN

    IF adjacencygraph.getnode(ridge1) == NONE THEN
       NEW node(ridge1).into(adjacencygraph);
       !if no node for ridge1, make one;

    IF adjacencygraph.getnode(ridge2) == NONE THEN
       NEW node(ridge2).into(adjacencygraph);
       !if no node for ridge2, make one;


    IF adjacencygraph.getnode(ridge2).pathlst.getpath(ridge1) == NONE THEN
    NEW path(ridge1).into(adjacencygraph.getnode(ridge2).pathlst);

    IF adjacencygraph.getnode(ridge1).pathlst.getpath(ridge2) == NONE THEN
    NEW path(ridge2).into(adjacencygraph.getnode(ridge1).pathlst);
        !put in the links (in both directions) if not already there;

END of setadjacency;


INTEGER PROCEDURE ridgenbrs(r,c); INTEGER r,c;
!This procedure returns as its value the number of neighbor pixels;
!of (r,c) which are ridge pixels;
BEGIN
INTEGER num,i;
        num:=0;     !init number of neighbors;
        FOR i:=1 STEP 1 UNTIL 8 DO  !for each neighbor;
            IF picin9.pixnbr(r,c,i) < 256 AND picin9.pixnbr(r,c,i) > 0
               THEN num:=num+1;
        ridgenbrs:=num;
END of ridgennbrs;

INTEGER PROCEDURE getrow(r,c,nbr); INTEGER r,c,nbr;
!This procedure returns as its value the row coordinate of the neighbor
!pixel of (r,c) specified by 'nbr';
BEGIN
INTEGER newrow;
        newrow:=r;          !if nbr=4 or nbr=8;
```

```
        IF nbr=1 OR nbr=2 OR nbr=3 THEN newrow:=r-1;
        IF nbr=7 OR nbr=6 OR nbr=5 THEN newrow:=r+1;     !the row values;
getrow:=newrow;
END of getrow;


INTEGER PROCEDURE getcol(r,c,nbr); INTEGER r,c,nbr;
!This procedure returns as its value the column coordinate of the
!neighbor pixel of (r,c) specified ny 'nbr';
BEGIN
INTEGER newcol;
        newcol:=c;          !if nbr=2 or nbr=6;
        IF nbr=1 OR nbr=8 OR nbr=0 OR nbr=7 THEN newcol:=c-1;
        IF nbr=3 OR nbr=4 OR nbr=5 THEN newcol:=c+1;
getcol:=newcol;
END of getcol;

INTEGER PROCEDURE calcneighbor(row,col,newrow,newcol);
                INTEGER row,col,newrow,newcol;
!This procedure returns an integer which is the "neighbor number"
!of the pixel (newrow,newcol) as viewed from pixel at (row,col),
!using the standard neighbor numbering scheme;
!
!                    1 2 3
!                    8 X 4
!                    7 6 5
!                                                    ;
BEGIN
     INTEGER nbr;
     BOOLEAN foundit;
     foundit:=FALSE;    !flag to test if was a valid neighbor;
     FOR nbr:=1 STEP 1 UNTIL 8 DO
     BEGIN
          IF getrow(row,col,nbr)=newrow AND getcol(row,col,nbr)=newcol THEN
             !test if this is the right neighbor;
          BEGIN
               calcneighbor:=nbr;  !heres the value to return;
               foundit:=TRUE;      !found the right neighbor;
          END;
     END;

          IF NOT foundit THEN
          BEGIN
               outtext("calcneighbor called with non-neighbor");
               outimage;   !if (newrow,newcol) is not a neighbor;
               outtext("row,col="); outint(row,5); outint(col,5); outimage;
               outtext("newrow,newcol="); outint(newrow,5);
               outint(newcol,5); outimage;
               halt;
          END;

END of calcneighbor;


PROCEDURE writeoutgraph;
!This procedure processes the adjacencygraph generated by the rest of
!the program, and writes the result out in ascii;
!Note that it is assumed that the ridge numbers used are contiguous, and
!begin with 2;
BEGIN
     INTEGER n;
     REF(path) pth;
     n:=2;   !first possible ridgenumber;
```

```
    WHILE adjacencygraph.getnode(n) =/= NONE DO   !for each ridge;
    BEGIN
    outint(n,4); outtext(":  "); breakoutimage;
    outfile.outint(n,4); outfile.outtext(":  "); outfile.breakoutimage;
         pth:-adjacencygraph.getnode(n).pathist.first QUA path;
             !the first link to this node;
         WHILE pth =/= NONE DO   !for each link from this node;
         BEGIN
              outint(pth.nodenum,4);   !write out ridge path goes to;
              outfile.outint(pth.nodenum,4);  !write to file the ridge
                                              !this path goes to;
              breakoutimage;
              outfile.breakoutimage;

              pth:-pth.suc;        !get the next path;
         END;
         outimage;
         outfile.outimage;
         n:=n+1;   !next node;
     END;
  END of writeoutgraph;

PROCEDURE clearvectors;
!This procedure is invoked between the four passes through the image;
!to make sure that no vector pixels remain, as would happen for example;
!when a vector encounters the edge of the image rather than a ridge;
BEGIN
INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
        FOR col:=1 STEP 1 UNTIL 400 DO     !for the entire image;
        BEGIN
         IF picin9.pixel(row,col) >= 256 THEN picin9.putpixel(row,col,0);
            !If pixel is a vector (non-ridge) pixel, zero it;
         vectim.putpixel(row,col,0);
            !and clear all the slope etc. information;
        END;
END of clearvectors;


PROCEDURE copypict8to9(p8,p9); REF(pict8) p8; REF(pict9) p9;
!This procedure copies an 8-bit (gray-level) image into a 9-bit one;
!Pixels are copied with their value intact, and therefore the high-order;
!bit is always set to 0;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
        FOR col:=1 STEP 1 UNTIL 400 DO     !for each pixel in the image;
        p9.putpixel(row,col,p8.pixel(row,col));
           !copy values as is;
END of copypict8to9;



line:-sysin.image;
outtext("Name (XXX) of file (XXX.IMG) containing ridge-numbered image: ");
breakoutimage;   inimage;
infilename:-copy(line.strip);    !get rid of trailing blanks;

outtext("Name (YYY) for output graph file (YYY.DAT): ");
breakoutimage;  inimage;
outfilename:-copy(line.strip);   !filename for output;
```

```
outfile:-NEW io20(conc(outfilename,".DAT"));  !the output file;
outfile.write.ascii.open(blanks(80));

picin:-NEW pict8;                  !ridge-numbered input image;
picin9:-NEW pict9;                 !9-bit copy goes here;

picin.load(infilename);            !get the input image;

copypict8to9(picin,picin9);        !make a 9-bit copy of the input image,;
                                   !with the high bit always zero;
                                   !high-bit will be used to indicate points;
                                   !which are on vectors(rather than ridges);
picin:-NONE;                       !kill off the 8-bit version to save space;

endpixels:=5;                      !how many pixels to use at the end of a
                                   !ridge in slope calculations (not
                                   !including the actual end pixel);

rowdot:=10;                        !output a dot every this many rows
                                   !processed;

vectim:-NEW vect9;                 !where we store the vector growth stuff;

adjacencygraph:-NEW graph;         !our results go here;

pass:=1;                           !top-to-bottom, left-to-right pass;
outtext("PASS 1"); outimage;
FOR row:=1 STEP 1 UNTIL 400 DO
    BEGIN
    IF MOD(row,rowdot)=0 THEN BEGIN outtext("."); breakoutimage; END;
    FOR col:=1 STEP 1 UNTIL 400 DO
    checkpixel;              !find all adjacencies, and enter them in graph;
    END;

pass:=2;                           !top-to-bottom, right-to-left pass;
outimage;
outtext("PASS 2"); outimage;
clearvectors;                      !remove any remaining vector pixels;
FOR row:=1 STEP 1 UNTIL 400 DO
    BEGIN
    IF MOD(row,rowdot)=0 THEN BEGIN outtext("."); breakoutimage; END;
    FOR col:=400 STEP -1 UNTIL 1 DO
    checkpixel;
    END;

pass:=3;                           !bottom-to-top, left-to-right pass;
outimage;
outtext("PASS 3"); outimage;
clearvectors;
FOR row:=400 STEP -1 UNTIL 1 DO
    BEGIN
    IF MOD(row,rowdot)=0 THEN BEGIN outtext("."); breakoutimage; END;
    FOR col:=1 STEP 1 UNTIL 400 DO
    checkpixel;
    END;

pass:=4;                           !bottom-to-top, right-to-left pass;
outimage;
outtext("PASS 4"); outimage;
clearvectors;
FOR row:=400 STEP -1 UNTIL 1 DO
    BEGIN
    IF MOD(row,rowdot)=0 THEN BEGIN outtext("."); breakoutimage; END;
```

```
        FOR col:=488 STEP -1 UNTIL 1 DO
        checkpixel;
        END;

    vectim:-NONE;
    picin9:-NONE;          !free up some core;

    !and now all thats left is to write out the graph structure into the
    !output file;

    outimage;
    writeoutgraph;         !write graph to file (and maybe terminal);

    outfile.close;
    outfile.release;

  END of SIMSET block;
END of graph;
```

```
!"NEWSUB";
!This program, given two ascii files which are the lists of the 'features' in
!two fingerprint images (with pixel coordinates for each feature), output by
!FEATUR.SIM, and computes the relevant distance graphs G1 and G2, and
!prints all of the maximal common subgraphs of G1 and G2.  The method used
!is to (conceptually at least) use the Boolean matrix representations of the
!input graphs to produce a large "compatibility matrix", which represents
!all possible mappings of pairs of nodes in G1 onto pairs of nodes in G2.
!The entries in this matrix specify whether the mapping in question is
!valid. The graphs G1 and G2 have the property that two nodes are connected
!if and only if the distance between them is less than a fixed amount.
!Also, the connection (link) is labeled with the distance. A mapping between
!a pair of nodes in G1 and a pair in G2 is valid if 1)node pair not linked
!in G1 and not linked in G2 or 2)linked in G1 and linked in G2 and
!distance labels are "close enough".
!Given this matrix, one need only find the cliques (maximal complete
!subgraphs) of the graph that this matrix represents, and we have our
!maximal common subgraphs of G1 and G2.
!Also note that optionally, the actual x,y pixel coords of the
!features can be used (coarsely) to prune the search tree.
!In fact, this matrix is never created (due to its size), but rather
!its entries are calculated as needed.
!The clique finding algorithm used is a modified form of that due to Bron
!and Kerbosch (ACM algorithm 457).;

OPTIONS (/A);  !-A would turn off array bounds checking, but doesn't help much;
   BEGIN
   EXTERNAL CLASS io20,pict1,pict8,pict9;
   EXTERNAL PROCEDURE enterdebug;
   EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
   EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
   EXTERNAL INTEGER PROCEDURE lshift,lxor;
   EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
   EXTERNAL CHARACTER PROCEDURE findtrigger;
   EXTERNAL PROCEDURE halt,exit;
   EXTERNAL REAL PROCEDURE random;

 SIMSET
 BEGIN
    head CLASS graph;
    BEGIN
         BOOLEAN looking;
         REF(node) PROCEDURE getnode(n); INTEGER n;
         !This procedure runs down the chain of nodes until one whose
         !featurenum equals n is found. Returns NONE if node doesnt exist;
         BEGIN
              REF(node) x; INTEGER i;
           IF NOT THIS graph.empty THEN !if there is at least one node already;
            BEGIN
              x:-first QUA node;              !first node in the graph;
              looking:=TRUE;                  !havent found right node yet;
              WHILE looking AND x =/= NONE DO  !until we find right one;
              BEGIN
                   IF x.featurenum = n THEN looking:=FALSE ELSE  !found it;
                   x:-x.suc;                  !next node;
              END;
              getnode:-x;
            END ELSE getnode:-NONE;      !if graph empty return NONE;
         END of getnode;                       !returns NONE if no such node;

    END of CLASS graph;
```

```
  link CLASS node(featurenum); INTEGER featurenum;
  BEGIN
       INTEGER cnum;
        !the number of the cluster to which this node belongs;
       BOOLEAN InClique;
        !whether this node is in current clique;
       REF(path_list) pathlst;     !list of paths (links) to other nodes;
       cnum:=0;  !initially belong to none;
       InClique:=FALSE;
       pathlst:-NEW path_list;     !init the list;
  END of CLASS node;

  head CLASS path_list;
  BEGIN
       !list of paths to other nodes;
       REF(path) x;

       REF(path) PROCEDURE first_path;
       !returns first path from this node;
       BEGIN
            IF NOT THIS path_list.empty THEN !if there is a path out;
            x:-first QUA path
            ELSE x:-NONE;  !else return NONE;

            first_path:-x;
       END of first_path;

       REF(path) PROCEDURE next_path;
       !returns next path from this node, or NONE;
       BEGIN
            x:-x.suc;
            next_path:-x;
       END of next_path;

  END of CLASS path_list;

  link CLASS path(nodenum); INTEGER nodenum;
  BEGIN
  END of CLASS path;


REF(io20) g1file, g2file,resultfile;
REF(graph) g1graph;
INTEGER g1nodes, g2nodes, g1g2nodes,total,maxc;
INTEGER local_region_size,smallest_subg;
INTEGER pruning_margin,match_margin,CallsToExtend;
BOOLEAN pruning_by_coords,AutoMode,GotFirst;
TEXT line, g1filename, g2filename;
REF(pict1) outpict;

INTEGER PROCEDURE digits(int);  INTEGER int;
!returns number of digits in decimal expression of the integer param;
BEGIN
     digits:=1+ ENTIER(LN(int)/LN(10));
END of digits;

PROCEDURE scanfiles;
!This procedure asks the user for the input and output file names,
!and then scans each input file to find the size of the input graphs.
!Variables g1nodes, g2nodes, and g1g2nodes are set based on these sizes;
!The input files are then closed and released;
BEGIN
```

```
        TEXT answer;
        line:-sysin.image;

        outtext("Name (XXX) of file (XXX.DAT) containing G1 features: ");
        breakoutimage; inimage;
        g1filename:-copy(line.strip);  !strip trailing blanks;
        g1file:-NEW io28(conc(g1filename,".DAT"));
        g1file.read.ascii.open(blanks(80));  !open it;

        outtext("Name (YYY) of file (YYY.DAT) containing G2 features: ");
        breakoutimage; inimage;
        g2filename:-copy(line.strip);  !strip trailing blanks;
        g2file:-NEW io28(conc(g2filename,".DAT"));
        g2file.read.ascii.open(blanks(80));  !open it;

        g1file.inimage;  !first line;
        WHILE NOT g1file.endfile DO
        BEGIN
             g1nodes:=g1file.inint;      !count features;
             g1file.inimage;   !read next line;
        END;
        g1file.close;  !counted features, done with file for now;
        g1file.release;

        g2file.inimage;  !first line;
        WHILE NOT g2file.endfile DO
        BEGIN
             g2nodes:=g2file.inint;      !count features;
             g2file.inimage;
        END;
        g2file.close;   !counted features, done with file for now;
        g2file.release;


outtext("Should I use feature coordinates to determine node compatibility? ");
        breakoutimage; inimage;
        answer:-copy(line.strip);  !strip trailing blanks;
        IF (answer="y" OR answer="yes" OR answer="Y" OR answer="YES") THEN
        BEGIN   !we should do pruning based on coords;
            pruning_by_coords:=TRUE;  !do pruning;

outtext("To within how many pixels must feature coordinates match : ");
            breakoutimage; inimage;
            pruning_margin:=inint;  !match limit;
        END ELSE pruning_by_coords:=FALSE;  !no pruning;

        outtext("Region (+- number of pixels) defined as local: ");
        breakoutimage; inimage;
        local_region_size:=inint;  !defn of how close together the X and Y
                                   !coords of two features must be to be local;

 outtext("To within how many pixels must inter-feature distances match: ");
        breakoutimage; inimage;
        match_margin:=inint;

outtext("Should I automatically ignore small maximal common subgraphs? ");
        breakoutimage; inimage;
        answer:-copy(line.strip);  !strip trailing blanks;
        IF (answer="y" OR answer="yes" OR answer="Y" OR answer="YES") THEN
        BEGIN
             Automode:=TRUE;
             smallest_subg:=8;   !have to start somewhere;
        END ELSE
```

```
        BEGIN
         Automode:=FALSE;
outtext("How many nodes in the smallest maximal common subgraph to find: ");
        breakoutimage; inimage;
        smallest_subg:=inint;
        END;

    END of scanfiles;

    PROCEDURE doit;
    !This procedure contains everything but the declarations of the array
    !sizes, and only exists so that the size of the arrays can be declared
    !only as large as needed;
    BEGIN
         INTEGER ARRAY g1x[1:g1nodes],g1y[1:g1nodes];  !the (x,y) pixel coords;
         INTEGER ARRAY g2x[1:g2nodes],g2y[1:g2nodes];  !of the features;
         INTEGER counter,ii,jj,xval,yval,PerLine,nsummary;

         PROCEDURE getgraphs;
         !This procedure reads in the data from the input files, and sets
         !up the feature coordinate arrays. Also sets up g1graph;
         BEGIN
              INTEGER featurenum,g1dx,g1dy;
              TEXT t;

              t:-blanks(80);   !init the parse buffer;

              g1file:-NEW io28(conc(g1filename,".DAT"));
              g1file.read.ascii.open(blanks(80));  !open g1;

              g2file:-NEW io28(conc(g2filename,".DAT"));
              g2file.read.ascii.open(blanks(80)); !open g2;

              g1file.inimage;       !read in first line;
              WHILE NOT g1file.endfile DO
              BEGIN
                   t:=g1file.image;          !fill parse buffer;
                   featurenum:=t.getint;     !first column is feature number;
                   t:=rest(t);               !whats left in the image;
                   g1x[featurenum]:=t.getint;      !x coord;
                   t:=rest(t);                     !again whats left;
                   g1y[featurenum]:=t.getint;      !y coord;
                   g1file.inimage;                 !next line;
              END;
              g1file.close;
              g1file.release;

              g2file.inimage;       !read in first line;
              WHILE NOT g2file.endfile DO
              BEGIN
                   t:=g2file.image;             !fill parse buffer;
                   featurenum:=t.getint;    !first column is feature number;
                   t:=rest(t);              !whats left;
                   g2x[featurenum]:=t.getint;      !x coord;
                   t:=rest(t);                     !again whats left;
                   g2y[featurenum]:=t.getint;      !y coord;
                   g2file.inimage;                 !next line;
              END;
              g2file.close;
              g2file.release;

              !and now time to find the translation table size needed;
              counter:=0;   !initially no slots in translation table are filled;
```

```
    FOR ii:=1 STEP 1 UNTIL g1nodes DO
        FOR jj:=1 STEP 1 UNTIL g2nodes DO !for each possible (i,j);
        IF NOT pruning_by_coords THEN !if not using feature coords;
        BEGIN
            counter:=counter+1;   !keep count of used slots;
        END ELSE
        BEGIN  !do want to check node compatibility using coords;
          xval:=abs(g1x[ii]-g2x[jj]);  !error (in pixels);
          yval:=abs(g1y[ii]-g2y[jj]);

            IF (xval <= pruning_margin) AND
               (yval <= pruning_margin) THEN  !close enough;
               BEGIN
                    counter:=counter+1;
               END;
        END of IF;

        g1g2nodes:=counter;   !how big our table need be;

        g1graph:-NEW graph;   !create the graph;

        FOR featurenum:=1 STEP 1 UNTIL g1nodes DO !for each G1 node;
        BEGIN
             NEW node(featurenum).into(g1graph);
             !create a node in the graph;

             FOR jj:=1 STEP 1 UNTIL g1nodes DO
             !compare above node against all but itself;
             BEGIN
                  IF jj =/= featurenum THEN !dont compare against self;
                  BEGIN
                       g1dx:=g1x(featurenum)-g1x(jj);
                       g1dy:=g1y(featurenum)-g1y(jj); !offsets;

                       IF (abs(g1dx)<=local_region_size) AND
                          (abs(g1dy)<=local_region_size) THEN
               NEW path(jj).into(g1graph.getnode(featurenum).pathlst);
               !if they are local to each other, put link in graph;
                          END of IF;
                  END of FOR;
        END of FOR;

END of getgraphs;

PROCEDURE maxcomsubgs;
!This procedure derives the maximal common subgraphs of g1 and g2,
!and writes the result to a file (prints summary);
BEGIN
     INTEGER ARRAY all[1:g1g2nodes], compsub[1:g1g2nodes];
     INTEGER c,tablesize,i;
     INTEGER ARRAY g1coeff[1:g1g2nodes], g2coeff[1:g1g2nodes];
       !These arrays are used to hold the translation table between
       !the number of a row or column in the compatibility matrix,
       !and its correct g1-g2 pair label;
     INTEGER ARRAY min[1:PerLine],avg[1:PerLine],max[1:PerLine];
       !the cumulative values for various thresholds;

     PROCEDURE extend(old,nee,ce); VALUE nee,ce; INTEGER nee,ce;
                                   INTEGER ARRAY old;
     BEGIN
          INTEGER ARRAY neww[1:ce];
          INTEGER nod,fixp,newne,newce,i,j,count,pos,p,s,sel,minnod;
          INTEGER ai,aj,bi,bj,row,col,g1dx,g2dx,g1dy,g2dy,loc;
```

```
BOOLEAN linkedin1, linkedin2;

BOOLEAN PROCEDURE connected(row,col);
        INTEGER row,col;
        !really should be called "compatible". Anyway,
        !it computes "on the fly" whether the entry in
        !the compatibility matrix at (row,col) is TRUE
        !or FALSE;

BEGIN

    IF row=col THEN connected:=TRUE ELSE
       !diagonal is all ones of course;
    BEGIN
         ai:=g1coeff[row];
         aj:=g1coeff[col];
         bi:=g2coeff[row];
         bj:=g2coeff[col];
             !go from row and column number in compatibility
             !matrix back to the two (g1,g2) node mappings
             !(ai,bi) and (aj,bj);
         IF ai=aj OR bi=bj THEN connected:=FALSE ELSE
            !if we have a nonn-unique mapping between nodes
            !in the graph (e.g. g1 node 1 mapped to two
            !nodes in g2;
         BEGIN
            g1dx:=g1x(ai)-g1x(aj);    !compute the x and y;
            g1dy:=g1y(ai)-g1y(aj);    !offsets for feature;
            g2dx:=g2x(bi)-g2x(bj);    !pairs in g1 and g2;
            g2dy:=g2y(bi)-g2y(bj);

            linkedin1:=IF (abs(g1dx)<=local_region_size) AND
                          (abs(g1dy)<=local_region_size)
                       THEN TRUE ELSE FALSE;

            linkedin2:=IF (abs(g2dx)<=local_region_size) AND
                          (abs(g2dy)<=local_region_size)
                       THEN TRUE ELSE FALSE;
              !booleans indicating whether features in g1
              !(g2) are close enough that they have a link
              !between them in the graph;

            IF ((linkedin1 AND NOT linkedin2) OR
                (NOT linkedin1 AND linkedin2)) THEN
                connected:=FALSE ELSE

                BEGIN
                IF (NOT linkedin1 AND NOT linkedin2) THEN
                   connected:=TRUE ELSE

                   BEGIN
                    IF ((abs(g1dx-g2dx)<=match_margin) AND
                        (abs(g1dy-g2dy)<=match_margin)) THEN
                        connected:=TRUE ELSE !linked in both;
                                             !and lengths match;

                        connected:=FALSE;
                   END;
                END;
         END;
    END;
END of connected;
```

```
PROCEDURE ProcessClique;
!This procedure is called each time a clique is found.
!It is responsible for deciding the size of each of the
!connected components of the clique, and outputting
!to 'resultfile' a summary of the number of features that
!would remain in the clique for various values of a
!"minimum connected component" threshold (i.e., ignore
!any feature not in a connected cluster of at least certain
!size.  After 'nsummary' cliques summary of the max, avg, and
!min number of features for each possible threshold value
!is output, and the program terminates.
!(cumulative to beginning of run);
BEGIN
     INTEGER numclusters;
     INTEGER ARRAY clustersize[1:300];
       !save the size of each cluster here..should never
       !get more than 300 clusters (or even close);
     INTEGER ARRAY remaining[1:PerLine];
      !how many features remain with varying threshold values;


     PROCEDURE LabelGraph;
     !For each node in the graph, we clear the "label" field;
     !Then, for each unlabeled node which is in the current
     !clique, we label it with the
     !number of its cluster (cluster is  set of connected
     !nodes), update the size count for that cluster, and
     !recursively call ProcessNode on all unlabeled nodes
     !connected to the current one which are in the clique;
     BEGIN
          INTEGER i;

          PROCEDURE ProcessNode(i); INTEGER i;
          !This procedure when called with the number of a
          !node as argument does: 1)IF node already has a
          !value indicating membership in a cluster, or
          !node is not in current clique, do
          !nothing. 2) Else mark current node as being
          !in cluster "numclusters".
          !Increment clustersize[clusternum]
          !by 1. Now recursively call ourselves on all nodes
          !connected to the current one which are in current
          !clique;
          BEGIN
               REF(path) x;

               IF glgraph.getnode(i).cnum=0 AND
                  glgraph.getnode(i).InClique THEN
                  !if not in cluster yet, but is in clique;
               BEGIN
                    glgraph.getnode(i).cnum:=numclusters;
                    !mark node as in current cluster;

                    clustersize[numclusters]:=
                      clustersize[numclusters]+1;
                      !keep count of how big this cluster is;


                   x:-glgraph.getnode(i).pathlst.first_path;
                   !first path out of this node. NONE if arent
                   !any;

                   WHILE x =/= NONE DO
```

```
            BEGIN !recurse on all connected nodes;
            IF g1graph.getnode(x.nodenum).InClique THEN
             ProcessNode(x.nodenum);
             !IF path points to a node in the current
             !clique, then recurse;
             x:-g1graph.getnode(i).pathlst.next_path;
            END of WHILE;
        END of IF;
    END of ProcessNode;

    !code for LabelGraph;

    FOR i:=1 STEP 1 UNTIL g1nodes DO
        g1graph.getnode(i).cnum:=0;
        !clear the cluster number data in the graph;

    FOR i:=1 STEP 1 UNTIL g1nodes DO
        g1graph.getnode(i).InClique:=FALSE;
        !turn off all clique membership bits;

    FOR loc:=1 STEP 1 UNTIL c DO
    !for each node in the current clique;
  g1graph.getnode(g1coeff[compsub[loc]]).InClique:=TRUE;
    !mark node as in current clique;

    FOR loc:=1 STEP 1 UNTIL c DO
    !for each node in the current clique;
        IF g1graph.getnode(g1coeff[compsub[loc]]).cnum=0
        !If node is unlabeled in G1 graph (i.e., not
        !marked as in a cluster yet;
        THEN
        BEGIN
          numclusters:=numclusters+1;
          ProcessNode(g1coeff[compsub[loc]]);
           !label node and all connected to it
           !(which are in the current clique) as in
           !same cluster;
        END of IF;

END of LabelGraph;

PROCEDURE ProcessClusters;
!For each cluster (entry in clustersize) based on its
!size fill in the entries in "remain" array. Output
!results ("remain" array to file). Also update Min,
!Max and Avg arrays;
BEGIN
     INTEGER cluster,thresh;
     FOR cluster:= 1 STEP 1 UNTIL numclusters DO
         FOR thresh:=1 STEP 1 UNTIL PerLine DO
         IF clustersize[cluster] >= thresh THEN
        remaining[thresh]:=remaining[thresh]+
          clustersize[cluster];
            !if clustersize is above threshold, then
            !add in its size to proper thresh niche;

     !now output data for this clique;
     resultfile.outtext("     ");
     FOR thresh:=1 STEP 1 UNTIL PerLine DO
         resultfile.outint(remaining[thresh],4);
     resultfile.outimage;

     !now update the cumulative statistics;
```

```
            FOR thresh:=1 STEP 1 UNTIL PerLine DO
            BEGIN
                 IF remaining[thresh] > max[thresh] THEN
                    max[thresh]:=remaining[thresh];
                    !update maximum;

                 IF remaining[thresh] < min[thresh] THEN
                    min[thresh]:=remaining[thresh];
                    !update minimum;

                 avg[thresh]:=avg[thresh]+remaining[thresh];
                 !total up for later average calculation;
            END of FOR;

        END of ProcessClusters;


        PROCEDURE OutputSummary;
        !Output summary to file, followed by blank line;
        BEGIN
          INTEGER i;
          resultfile.outimage;

          resultfile.outtext("MIN: ");
          FOR i:=1 STEP 1 UNTIL PerLine DO
              resultfile.outint(min[i],4);
          resultfile.outimage;

          resultfile.outtext("AVG: ");
          FOR i:=1 STEP 1 UNTIL PerLine DO
              resultfile.outint(avg[i]/total,4);
          resultfile.outimage;

          resultfile.outtext("MAX: ");
          FOR i:=1 STEP 1 UNTIL PerLine DO
              resultfile.outint(max[i],4);
          resultfile.outimage;

          resultfile.outimage;
          resultfile.outimage;

          resultfile.close;
          resultfile.release;

          exit(1);          !want the time message, etc.;
        END of OutputSummary;


        !and now for the code for ProcessClique;

IF NOT GotFirst THEN
BEGIN
     FOR loc:=1 STEP 1 UNTIL c DO
     outpict.putpixel(glx(glcoeff(compsub(loc))),gly(glcoeff(compsub(loc))),1);
     outpict.store(copy("NEWSUB"));
     GotFirst:=TRUE;
END;

        numclusters:=0;  !none found yet;

        FOR i:=1 STEP 1 UNTIL 300 DO
            clustersize[i]:=0;  !no clusters yet;
```

```
        FOR i:=1 STEP 1 UNTIL PerLine DO
            remaining[i]:=0;  !init things;

        LabelGraph;
        ProcessClusters;

        IF total=nsummary THEN OutputSummary;
           !output summary after this many cliques found
           !(and then stop);

    END of ProcessClique;



    !and now the code for "extend";

     CallsToExtend:=CallsToExtend+1;

   IF c+(ce-nee) >= smallest_subg THEN
   BEGIN  !if size of current clique (c) plus size of set
          !of potential candidates for addition to the clique
          !(ce-nee) is large enough,
          !then there is still a chance of finding clique of
          !at least size smallest sub_g;
     minnod:=ce;
     i:=0;  nod:=0;

     ! determine each counter value and look for minimum;
     FOR i:=i+1 WHILE i<=ce AND minnod <> 0 DO
     BEGIN
      p:=old[i]; count:=0; j:=nee;

      ! count disconnections;
      FOR j:=j+1 WHILE j<=ce AND count<minnod DO
      BEGIN

         IF NOT connected(p,old[j]) THEN
            BEGIN
                count:=count+1;
                ! save position of potential candidiate;
                pos:=j;
            END of if;
      END of FOR;

      ! test new minimum;
      IF count < minnod THEN
          BEGIN
              fixp:=p;   minnod:=count;
              IF i <= nee THEN s:= pos
              ELSE
              BEGIN
                   s:=i; nod:=1; ! pre-increment;
              END;
          END of IF;
     END of FOR;
! if fixed point initially chosen from 'candidates' then;
! number of disconnections will be preincreased by one;

! BACKTRACECYCLE;
FOR nod := minnod+nod STEP -1 UNTIL 1 DO
    BEGIN
        ! interchange;
        p:=old[s];  old[s]:=old[nee+1];
```

```
sel:=p;  old[nee+1]:=p;

! fill new set 'not';
newne:=0;  i:=0;
FOR i:=i+1 WHILE i<= nee DO
BEGIN

        IF connected(sel,old[i]) THEN
            BEGIN
                newne:=newne+1;
                neww[newne]:=old[i];
            END of IF;
END of FOR;

        ! fill new set 'cand';
        newce:=newne;  i:=nee+1;
        FOR i:=i+1 WHILE i<= ce DO
        BEGIN

            IF connected(sel,old[i]) THEN
               BEGIN
                    newce:=newce+1;
                    neww[newce]:=old[i];
               END of IF;
        END of FOR;

        ! add to 'compsub';
        c:= c + 1;  compsub[c]:=sel;
        IF newce=0 THEN
            BEGIN
            IF c >= smallest_subg THEN !no small cliques;
            BEGIN

            total:=total+1;  !count them;

            IF c>maxc THEN
            BEGIN
                 outint(c,6);
                 maxc:=c;
              IF AutoMode THEN smallest_subg:=c;
                 !neednt find smaller;
            END ELSE        !collect size of largest;
            BEGIN
             IF c=maxc THEN
             BEGIN
                 outtext("!");
                 breakoutimage;
             END;  !we got another the size of current max;
            END;

            ProcessClique;  !analyze for connected
                            !components, and output
                            !data to summary file;

              END of IF;
            END ELSE
            IF newne<newce THEN extend(neww,newne,newce);

        ! remove from 'compsub';
        c:=c-1;

        ! add to 'not';
        nee:=nee+1;
```

```
                    IF nod>1 THEN
                        BEGIN
                            ! select a candidate disconnected;
                            ! to the fixed point;
                            s:=nee;

                            ! look for candidate;
                            look:
                            s:= s + 1;

                            IF connected(fixp,old[s]) THEN GOTO look;
                        END of IF;
          END of FOR;
         END of IF;
        END of extend;

        !and now time to fill in the translation table;
        counter:=0;   !nothing filled in yet;
        FOR ii:=1 STEP 1 UNTIL g1nodes DO
            FOR jj:=1 STEP 1 UNTIL g2nodes DO !for each possible (i,j);
            IF NOT pruning_by_coords THEN !if not using feature coords;
            BEGIN
                counter:=counter+1;
                g1coeff[counter]:=ii; !no skipped entries since all nodes;
                g2coeff[counter]:=jj; !are mutually compatible;
            END ELSE
            BEGIN  !do want to check node compatibility using coords;
              xval:=abs(g1x[ii]-g2x[jj]);  !error (in pixels);
              yval:=abs(g1y[ii]-g2y[jj]);

                IF (xval <= pruning_margin) AND
                   (yval <= pruning_margin) THEN  !close enough;
                   BEGIN
                        counter:=counter+1;
                        g1coeff[counter]:=ii;  !this mapping OK;
                        g2coeff[counter]:=jj;
                   END;
            END of IF;


        FOR i:=1 STEP 1 UNTIL PerLine DO
        BEGIN
             min[i]:=100000;   !need large values to start;
             max[i]:=0;
             avg[i]:=0;
        END of FOR;

        !and now for the code...;
        FOR c:=1 STEP 1 UNTIL g1g2nodes DO all[c]:=c;
        c:=0;
        extend(all,0,g1g2nodes);

    END of maxcomsubgs;


  !code for "doit";

PerLine:=16;   !let threshold range from 1 to 16;
nsummary:=20;  !output summary data after this many cliques, then stop;

    getgraphs;      !read in data;
```

```
        resultfile:-NEW io20("newsub.dat");
        resultfile.write.ascii.open(blanks(80));

        resultfile.outtext("Size of '"); resultfile.outtext(g1filename);
        resultfile.outtext("' is"); resultfile.outint(g1nodes,4);
        resultfile.outtext(" features."); resultfile.outimage;

        resultfile.outtext("Size of '"); resultfile.outtext(g2filename);
        resultfile.outtext("' is"); resultfile.outint(g2nodes,4);
        resultfile.outtext(" features."); resultfile.outimage;

        IF pruning_by_coords THEN
        BEGIN
        resultfile.outtext(
                "Using feature coordinates to determine node compatibility.");
        resultfile.outimage;
        resultfile.outtext("Feature coordinates must match to within ");
        resultfile.outint(pruning_margin,digits(pruning_margin));
        resultfile.outtext(" pixels."); resultfile.outimage;
        END ELSE
        BEGIN
          resultfile.outtext(
             "Not using feature coordinates to determine node compatibility.");
          resultfile.outimage;
        END of IF;

        resultfile.outtext("Size of local region is +- ");
        resultfile.outint(local_region_size,digits(local_region_size));
        resultfile.outtext(" pixels.");  resultfile.outimage;

        resultfile.outtext("Inter-feature distances must match to within ");
        resultfile.outint(match_margin,digits(match_margin));
        resultfile.outtext(" pixels.");  resultfile.outimage;

        IF Automode THEN
           resultfile.outtext(
           "Automatically ignoring small maximal common subgraphs.")
ELSE
    BEGIN
 resultfile.outtext("Smallest maximal common subgraph to find contains ");
        resultfile.outint(smallest_subg,digits(smallest_subg));
        resultfile.outtext(" features.");
        resultfile.outimage;
    END;

        resultfile.outimage;
        resultfile.outimage;

        resultfile.outtext("Minimum"); resultfile.outimage;
        resultfile.outtext("allowed"); resultfile.outimage;
        resultfile.outtext("cluster"); resultfile.outimage;
        resultfile.outtext("size:  ");
        resultfile.outtext(
        " 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16");
        resultfile.outimage;
        resultfile.outtext(
"_______________________________________________________________________");
        resultfile.outimage;

        total:=0;    !number of max com. subg's found;


    maxcomsubgs;            !find and print maximal common subgraphs;
```

```
    outimage;
    outtext("total max. com. subg's found: "); outint(total,5); outimage;

  END of doit;


 !main program;
 scanfiles;  !user dialog and find array sizes;

GotFirst:=FALSE;
outpict:-NEW pict1;  !for testing;

 CallsToExtend:=0;
 doit;       !all the rest;
 outint(CallsToExtend,8); outtext(" calls to extend"); outimage;

 END of SIMSET block;
END of newsub;
```

```
!'NTHRSH" ;
!This program accepts as input a filename (e.g. NAMEIN)
!which is an 8-bit image.  For each pixel in the input image, the
!following takes place: A NxN neighborhood around the pixel in question
!(but not extending outside the 400x400 image) is histogrammed. If the
!value distribution in the histogram is unimodal (one peak), then the pixel
!in question is a background pixel, and is set to 0  (assuming the window
!is large enough that it cannot be completely filled by a ridge).  If the
!distribution is bimodal, the threshhold is set at half the total
!distribution width, and then this threshold
!is used to seperate the image pixels from the background pixels.
!The image (ridge) pixels are set to 1, while the background pixels are
!set to 0.
!The histogram width criterion for deciding if a distribution is unimodal
!or bi-modal is a parameter input from the terminal. Values from 15 to
!25 are typical.  The upper right corner of the image is always forced to
!background, as that area is large and dark due to some masking in the
!optics, and therefore violates the "if its unimodal its light (backgrnd)"
!rule.
!The output 1-bit image is called NAMEOUT.BIM;

BEGIN
    EXTERNAL CLASS io20,pict1,pict8;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;

    REF(pict8) picin;
    REF(pict1) picthresh;       !the threshold image;
    TEXT line,infilename,threshfilename;
    INTEGER row,col,nhoodsize,halfnhood,point,minval,maxval,oldrow,oldcol;
    INTEGER width,minwidth,maxwidth,width_threshold;
    INTEGER ARRAY hist[0:255],width_hist[0:255];


    PROCEDURE histogram(row,col); INTEGER row,col;
    !This procedure fills the array hist[0:255] with the histogram of the
    !pixel values in the region of size nhoodsize x nhoodsize surrounding
    !(row,col).  Note that pixels outside the 400 x 400 image are not
    !used.  Also note that if we have simply moved to the right one column
    !since the last call to this routine, the histogram is updated, rather
    !than being computed from scratch.  In all other cases, it is computed
    !from scratch;
    BEGIN
        INTEGER pixelvalue,nrow,ncol,maxrowlim,maxcollim,minrowlim,mincollim;

    PROCEDURE addright(row,col); INTEGER row,col;
    !This procedure adds the pixel-values in the right-most column of the
    !window into the histogram;
    BEGIN
        INTEGER rowptr,pixval;

        FOR rowptr:=minrowlim STEP 1 UNTIL maxrowlim DO  !each row of window;
            BEGIN
             pixval:=picin.pixel(rowptr,maxcollim);  !get pixel value;
             hist[pixval]:=hist[pixval]+1;  !add it in to histogram;
            END;
    END of addright;
```

```
PROCEDURE delleft(row,col); INTEGER row,col;
!This procedure removes the pixel values in the left-most column of the
!previous window (i.e., it removes the pixels in the column just to the
!left of the left-most column in the current window;
BEGIN
     INTEGER rowptr,pixval;

     FOR rowptr:=minrowlim STEP 1 UNTIL maxrowlim DO  !each row;
         BEGIN
          pixval:=picin.pixel(rowptr,mincollim-1);  !get pixel value;
                 !from column that is one to the left of the left-most
                 !in the current window;
          hist[pixval]:=hist[pixval]-1;   !remove from the histogram;
         END;
END of delleft;


      maxrowlim:= IF row+halfnhood <= 400 THEN row+halfnhood ELSE 400;
      maxcollim:= IF col+halfnhood <= 400 THEN col+halfnhood ELSE 400;
      minrowlim:= IF row-halfnhood >=1 THEN row-halfnhood ELSE 1;
      mincollim:= IF col-halfnhood >=1 THEN col-halfnhood ELSE 1;
             !clip window to stay within image;

     IF row=oldrow AND col=oldcol+1 THEN   !if we are in the same row;
         !as the last histogram we did, and have only moved over one col.;
     BEGIN
       !update the histogram;
       IF col <= halfnhood+1 THEN
                     !left edge of window at least contacts left edge
                     !of image ==> only add right side column into
                     !histogram...no deletions at left needed;
                               BEGIN
                               addright(row,col);
                               END

       ELSE
       BEGIN
            IF col >= 400-halfnhood+1 THEN
                     !right edge of window at least contacts right edge
                     !of image ==> only delete left-side column from
                     !histogram...no additions at right;
                                        BEGIN
                                        delleft(row,col);
                                        END

            ELSE    !window is fully inside the image;
            BEGIN
                addright(row,col);
                delleft(row,col);  !then need to do both;
            END;
       END of IF;

     END ELSE  !if we reached the end of a row, or are doing random histos;
               !then do the histogram from scratch;
     BEGIN
      FOR pixelvalue:=0 STEP 1 UNTIL 255 DO
         hist[pixelvalue]:=0;  !init the histogram;


      FOR nrow:=minrowlim STEP 1 UNTIL maxrowlim DO
         FOR ncol:=mincollim STEP 1 UNTIL maxcollim DO
             !for each pixel in the potential neighborhood;
             BEGIN
```

```
                pixelvalue:=picin.pixel(nrow,ncol);
                    !the value of the pixel at (nrow,ncol);
                hist[pixelvalue]:=hist[pixelvalue]+1;
                    !we have one more pixel with that value;
            END;
    END of IF;

  oldrow:=row;  oldcol:=col;   !for next time around;
END of histogram;


line:-sysin.image;          !input buffer for TTY;
outtext("Name (XXX-YYY) of file (XXX-YYY.IMG) to threshold: ");
breakoutimage;
inimage;
infilename:-copy(line.strip);      !get rid of trailing blanks;

outtext("Name (UUU-VVV) of file (UUU-VVV.BIM) for thresheld image: ");
breakoutimage;
inimage;
threshfilename:-copy(line.strip);      !the output filename;


picin:-NEW pict8;
picthresh:-NEW pict1;

picin.load(infilename);   !load the input data;

outtext("Histogram width threshold (0 to 255): ");
breakoutimage;              ! prompt for threshold ;
inimage;
width_threshold:=inint;     ! get the threshold value ;

nhoodsize:=15;   !the size (nhoodsize**2) of the neighborhood surrounding
                 !each pixel to use for thresholding. Should be odd;
halfnhood:=(nhoodsize-1)/2;  !the amount to go + and - on each side of
                             !(row,col);
oldrow:=99999;  oldcol:=99999;       !init vars used to do histogram
    !differentialy. These values guarantee a "from scratch" computation
    !the first time (unless we ask for histogram(100000,99999);


FOR row:=1 STEP 1 UNTIL 400 DO
  BEGIN
    outtext("."); breakoutimage;
    FOR col:=1 STEP 1 UNTIL 400 DO
    BEGIN
        histogram(row,col);   !array hist[0:255] gets histogram of
                              !the nhoodsize x nhoodsize region
                              !surrounding (row,col);

        point:=0;
        WHILE hist[point]=0 AND point <=255 DO
              !search through all the 0 entries;
              point:=point+1;  !next entry;
              !when done, point is pointing at first non-zero entry,
              !or is equal to 255 (if all entire zero);
        minval:=point;   !save left edge of useful histogram area;

        point:=255;
        WHILE hist[point]=0 AND point >= 0 DO
              point:=point-1;
        maxval:=point;   !save right edge;
```

```
            width:=maxval-minval+1;  !the width of the histogram;

            IF minval>maxval THEN
               !if histogram was empty;
               BEGIN
                    outtext("histogram was empty");
                    width:=8;
               END;

            IF picin.pixel(row,col) < (minval+maxval)/2 THEN
               picthresh.putpixel(row,col,1)
            ELSE picthresh.putpixel(row,col,8);   !use center of histogram;
                                                  !as threshold;

            IF width <= width_threshold THEN picthresh.putpixel(row,col,8);
               !if histogram is unimodal, must be a background pixel;

            IF 352*col-355*row >= 100000 THEN picthresh.putpixel(row,col,8);
               !if we are in upper right corner, force it to background;
               !(115 pixels in from corner in row and col direction;
               !overall shape is right triangular;

   END of FOR;
       END of FOR;

   picthresh.store(threshfilename);     !write out the result file;

END of NTHRSH;
```

```
OPTIONS(/e);
EXTERNAL CLASS io28;
EXTERNAL PROCEDURE enterdebug;
EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
EXTERNAL INTEGER PROCEDURE lshift,lxor;
EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
EXTERNAL CHARACTER PROCEDURE findtrigger;
EXTERNAL PROCEDURE halt;

    !This class is used to deal with 488 x 488 8 intensity bits per pixel
    !digitized images, packed 4 8-bit bytes per word (left justified).
    !Image requires 168888 bytes = 48888 words of storage.;


CLASS pict8;
BEGIN
    INTEGER i;
    INTEGER ARRAY dataarray[1:48888];
    INTEGER ARRAY bytemask[8:3],bytemaskn[8:3];

    PROCEDURE load(filename); TEXT filename;
        !This procedure loads the image data from the disk file
        !to the dataarray in memory (still packed).
        !It then closes and release the disk file.;
    BEGIN
        INTEGER ARRAY ac[1:4];
        REF(io28) file;
        file:-NEW io28(conc(filename,".IMG"));  !create instance of file;
        file.read.imagemode.Open(Blanks(88));   !open it;

        ac[1]:=file.jfn;
        ac[2]:=xwd(8R884488,aaddr(dataarray)-1);  !load data into
            !memory as 36-bit bytes, beginning at arrayname[1];
        ac[3]:=-48888;                          !number of 36-bit words;
        IF NOT jsys(8R52,ac) THEN           !SIN;
        error("SIN in 'load' failed");

        file.Close;                  !close the file;
        file.release;                !release it;

    END of load;


    PROCEDURE store(filename); TEXT filename;
        !This procedure stores the image data in the dataarray
        !back into a disk file,then closes and release the file.;
    BEGIN
        INTEGER ARRAY ac[1:4];
        REF(io28) file;
        file:-NEW io28(conc(filename,".IMG")); !create instance of file;
        file.write.imagemode.Open(Blanks(88)); !open it;

        ac[1]:=file.jfn;
        ac[2]:=xwd(8R884488,aaddr(dataarray)-1);   !write data out
                !as 36-bit words;
        ac[3]:=-48888;           !number of words;
        IF NOT jsys(8R53,ac) THEN  !SOUT;
        error("SOUT in 'store' failed");

        file.Close;                  !close the file;
        file.release;                !release it;
```

```
END of store;

INTEGER PROCEDURE pixel(row,col); INTEGER row,col;
!This procedure returns the value of the specified pixel.
!More exactly, the [(row-1)*400+col]th pixel value is returned.
!Note that this will only be the value of the pixel at (row,col) in
!the image if the image has been corrected for the fact that the
!raw images as scanned in from the hardware have seperated interlace
!fields, and are scanned column-wise rather than row-wise.  The
!program 'CONVRT' takes care of this.
!The value returned for the pixel will be 0 to 255;
!If row and col are not both in range from 1 to 400, value
!returned is 255;
BEGIN
    INTEGER seqnum,word,bytenum;
  IF row>0 AND row<401 AND col>0 AND col<401 THEN
    BEGIN
    seqnum:=400*(row-1)+col;      !want the seqnum'th pixel;
    word:=dataarray[1+(seqnum-1)//4];  !the word the byte is in;
    bytenum:=MOD(seqnum-1,4);  !the byte we want(#0 at left);
    pixel:=LAND(LSHIFT(word,8*bytenum-28),255);
                  !get the pixel (8-bits);
    END ELSE pixel:=255;  !if off the image edges;
END of pixel;

PROCEDURE putpixel(row,col,val); INTEGER row,col,val;
!This procedure sets the value of the specified pixel.
!More exactly, the [(row-1)*400+col]th pixel value is set.
!Note that this will only be the pixel at (row,col) in
!the image if the image has been corrected for the fact that the
!raw images as scanned in from the hardware have seperated interlace
!fields, and are scanned column-wise rather than row-wise.  The
!program 'CONVRT' takes care of this.
!The value for the pixel should be 0 to 255;
BEGIN
    INTEGER seqnum,word,bytenum,pointer,mask;
    seqnum:=400*(row-1)+col;      !want the seqnum'th pixel;
    pointer:=1+(seqnum-1)//4;  !point to the byte;
    bytenum:=MOD(seqnum-1,4);  !the byte we want(#0 at left);
    dataarray[pointer]:=LAND(dataarray[pointer],bytemaskn(bytenum));
              !zero the byte;
    dataarray[pointer]:=LOR(dataarray[pointer],LSHIFT(val,
                28-8*bytenum));  !OR in the proper val;
 END of putpixel;

 INTEGER PROCEDURE pixnbr(row,col,n); INTEGER row,col,n;
 BEGIN
 INTEGER ncol,nrow;
     ncol:=col;       !init value;
     IF n=1 OR n=8 OR n=0 OR n=7 THEN ncol:=col-1;
     IF n=3 OR n=4 OR n=5 THEN ncol:=col+1;
     nrow:=row;
     IF n=1 OR n=2 OR n=3 THEN nrow:=row-1;
     IF n=7 OR n=6 OR n=5 THEN nrow:=row+1;
     !set up the row and col values for the neighbors 1 thru 8 as:
     !
     !                1 2 3
     !                8 P 4
     !                7 6 5
     !
     !(note that neighbor 0 will be equivalent to neighbor 8.)
     !where P is the pixel under consideration;
     IF nrow<1 THEN pixnbr:=255 ELSE
```

```
    IF nrow>400 THEN pixnbr:=255 ELSE
    IF ncol<1 THEN pixnbr:=255 ELSE
    IF ncol>400 THEN pixnbr:=255
    ELSE  !cant' have values off edge of image;
    pixnbr:=pixel(nrow,ncol);  !get the value;
END of pixnbr;

INTEGER PROCEDURE npixnbr(row,col,n); INTEGER row,col,n;
!This routine returns a value for a neighbor that is off the screen;
!equal to the value of the "center" pixel, rather than returning 255;
!Needed for new thresholding scheme (looking at histogram widths);
BEGIN
INTEGER ncol,nrow;
    ncol:=col;      !init value;
    IF n=1 OR n=8 OR n=0 OR n=7 THEN ncol:=col-1;
    IF n=3 OR n=4 OR n=5 THEN ncol:=col+1;
    nrow:=row;
    IF n=1 OR n=2 OR n=3 THEN nrow:=row-1;
    IF n=7 OR n=6 OR n=5 THEN nrow:=row+1;
    !set up the row and col values for the neighbors 1 thru 8 as:
    !
    !               1 2 3
    !               8 P 4
    !               7 6 5
    !
    !(note that neighbor 0 will be equivalent to neighbor 8.)
    !where P is the pixel under consideration;
    IF nrow<1 THEN npixnbr:=pixel(row,col) ELSE
    IF nrow>400 THEN npixnbr:=pixel(row,col) ELSE
    IF ncol<1 THEN npixnbr:=pixel(row,col) ELSE
    IF ncol>400 THEN npixnbr:=pixel(row,col)
    ELSE  !cant' have values off edge of image;
    npixnbr:=pixel(nrow,ncol);  !get the value;
END of npixnbr;


   PROCEDURE error(message);VALUE message; TEXT message;
   BEGIN
       INTEGER ARRAY ac[1:4];
       Outimage;
       Outtext(message);
       Outimage;
       ac[1]:=xwd(0,8R400000);          !process handle...current process;      ;
       jsys(8R12,ac);                   !GETER...get most recent error;
       ac[1]:=xwd(0,8R777777);          !destination designator...TTY;
       !ac[2] was setup by GETER;
       jsys(8R11,ac);                   !type the error string on the TTY;      ;
       jsys(8R170,ac);                  ! HALTF - Just give up;
   END of error;

   !Initialization code;
   FOR i:=0 STEP 1 UNTIL 3 DO BEGIN
        bytemask[i]:=LSHIFT(255,28-8*i);   !make the byte mask;
        bytemaskn[i]:=LNOT(bytemask[i]);   !and its complement;
   END;



END of CLASS pict8;
```

```
!"RIGNUM";
!This program operates on a binary, thinned image, and produces as output;
!an 8-bit image, where the value of each pixel is either 8 for background;
!(off in the input image) pixels, or is the "number" of the ridge the;
!pixel is contained in. The pixels are numbered as they are encountered,;
!in raster scan order, based upon their adjacency to already numbered pixels.;
!Conflicts that arise are resolved by a renumbering, done after the raster;
!scan is complete.;


    BEGIN
    EXTERNAL CLASS io28,pict1,pict8,pict9;
    EXTERNAL PROCEDURE enterdebug;
    EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
    EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
    EXTERNAL INTEGER PROCEDURE lshift,lxor;
    EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
    EXTERNAL CHARACTER PROCEDURE findtrigger;
    EXTERNAL PROCEDURE halt;

    SIMSET
    BEGIN
    head CLASS conflict_list;;
         !a list of pairs of numbers, which are the conflicts that have
         !been encountered during the ridge numbering process;

    link CLASS conflict_pair(ridge1,ridge2); INTEGER ridge1,ridge2;
         !ridge1 should be < ridge2 (though order doesnt really matter;
         !they should NEVER be equal (else a conflict didnt exist);
    BEGIN
         IF ridge1>ridge2 THEN
         BEGIN
          outtext("Conflict-pair created with ridges in wrong order");outimage;
         END;
         IF ridge1=ridge2 THEN
         BEGIN
          outtext("Conflict-pair created with ridge numbers equal");outimage;
         END;
    END of CLASS conflict-pair;


    head CLASS graph;
    BEGIN
         REF(node) PROCEDURE getnode(n); INTEGER n;
         BEGIN
              REF(node) x; INTEGER i;
              x:-first QUA node;                !first node in the graph;
              FOR i:= 2 STEP 1 UNTIL n DO x:-x.suc;
              getnode:-x;                       !found the n'th node;
              IF x.rignum =/= n THEN
              BEGIN
                   outtext ("incorrect node chose in 'getnode'");
              END;
         END of getnode;
    END of CLASS graph;

     link CLASS node(rignum); INTEGER rignum;
     BEGIN
          REF(path_list) pathlst;     !list of paths (links) to other nodes;
          BOOLEAN resolved;
          INTEGER transval;

          transval:=rignum;           !if node is never touched, want it to;
```

```
                              !translate to its original value;
    resolved:=FALSE;          !initially none of the nodes are resolved;
    pathlst:-NEW path_list;   !init the list;
END of CLASS node;

head CLASS path_list;;
    !list of paths to other nodes;

link CLASS path(nodenum); INTEGER nodenum;
BEGIN
    IF nodenum>usedrignums THEN
    BEGIN
        outtext("attempt to create path to illegal node");
        outimage;
    END;
END of CLASS path;
```

```
REF(pict9) pictemp;
REF(pict1) picin;
REF(pict8) picout;
REF(conflict_list) clist;          !the list of numbering conflicts;
TEXT line,infilename,outfilename;
INTEGER minval,usedrignums,arraysize;
INTEGER ARRAY nbrvals[1:8];
BOOLEAN eql;


PROCEDURE numberthem(picture); REF(pict9) picture;
    !This procedure does the initial numbering of the ridges, as;
    !the pixels are encountered in raster scan order;
    !It also creates the list of numbering conflicts which is later;
    !used to assign the ridges their final numeric values;
BEGIN
    INTEGER row,col,nextnum,numberednbrs,i;
    nextnum:=1;                    !first ridge number to use minus one;
    FOR row:=1 STEP 1 UNTIL 400 DO
        FOR col:=1 STEP 1 UNTIL 400 DO   !for each pixel, raster-scan wise;
            BEGIN
            IF picture.pixel(row,col)=1 THEN  !if we are on an 'on' pixel;
            BEGIN
            numberednbrs:=nnbrs(picture,row,col);
              !find number of numbered neighbors;
              !and return boolean eql, TRUE if all neighbors equal;
            IF numberednbrs=0 THEN  !if it has no numbered neighbors, it
                                    !gets its own, new number;
                BEGIN
                nextnum:=nextnum+1; !next avail ridgenumber;
                picture.putpixel(row,col,nextnum); !give it next ridgenum;
                END
            ELSE IF eql THEN picture.putpixel(row,col,nbrvals[1])
                !if all nbrs have same value (1 or more nbrs) then give
                !current pixel the same value;
                 ELSE
                   BEGIN
                    IF numberednbrs>4 THEN
                    BEGIN
                    outtext(">4 neighbors in 'numberthem'");
                    outtext("  row="); outint(row,3);
                    outtext("  col="); outint(col,3);
                    outimage;
                    END;
                       picture.putpixel(row,col,minval); !give pixel the
                            !same value as the lowest of the neighbors;
                       FOR i:=1 STEP 1 UNTIL numberednbrs DO
                         BEGIN
                         IF minval<>nbrvals[i] THEN
                         BEGIN
                         NEW conflict_pair(minval,nbrvals[i]).into(clist);
                             !add to our list of conflict pairs;
                         END of IF;
                         END of FOR;
                   END of IF;
            END of IF;
            END of FOR ;
       usedrignums:=nextnum;       !value of highest ridge number used;
END of numberthem;
```

```
INTEGER PROCEDURE nnbrs(picture,row,col);
   INTEGER row,col;
   REF(pict9)picture;
    !This procedure returns as its value the number (0-8) of neighbors
    !of the current pixel which have been 'numbered' (i.e. whose value
    !is greater than 1. It also returns an array (nbrvals) containing
    !the values of all >1 pixels. I changes a global variable (minval)
    !which is the minimum value of all the neighbors.;
    !Also returned is a boolean "eql", which is true if all the neighbor
    !values are equal;
BEGIN
INTEGER num,i,nvalue;
    minval:=511;   !init minimum neighbor value;
    num:=0;      !init number of numbered neighbors;
    eql:=TRUE;    !init flag for neighbor values equal;
    FOR i:=1 STEP 1 UNTIL 8 DO      !for each neighbor;
        BEGIN
            nvalue:=picture.npixnbr(row,col,i);  !value of neighbor pixel;
                                                 !returns 0 if off screen;
            IF nvalue>1 THEN
                BEGIN
                num:=num+1;   !increment neighbor count;
                nbrvals[num]:=nvalue; !enter neighbor value in array;
                IF nvalue<minval THEN minval:=nvalue; !update minimum;
                IF num>1 THEN
                  BEGIN
                  IF nvalue\=nbrvals[num-1] THEN eql:=FALSE;
                  !if this neighbor has value different from previous
                  !neighbor, then turn off the neighbors equal flag;
                  END of IF;
                END of IF;
        END of FOR;
    nnbrs:=num;
END of nnbrs;
```

```
PROCEDURE revisenumbers(picture); REF(pict9) picture;
          !This procedure uses the conflict list previously generated
          !to create a graph of node intersections, and from that
          !graph calculates the proper renumbering of the pixels.
          !the final step insures that only a contiguous set of ridge
          !numbers are used.;
          !Note that a path must be made both ways between two nodes.;
BEGIN
     INTEGER n,minridge,row,col,address,nvalue,availoutnum,rnum,i;
     REF(graph) cgraph;      !conflict graph;
     REF(conflict_pair) cp;
     INTEGER ARRAY translatetbl[0:arraysize];
     BOOLEAN present;

     PROCEDURE tracepaths(n); INTEGER n;
     !This procedure is called with the number of a graph node as its;
     !parameter. Presumably, the node is unresolved. This routine then;
     !set the node as "resolved", and sets the translation value to;
     !be the current smallest value (i.e., the smallest ridge number on;
     !this connected sub-graph). We then loop over all nodes connected;
     !to this one via a "path", and if any of those are unresolved,;
     !we recursively call this routine on that node.;
     BEGIN
          REF(path) pth;
          IF cgraph.getnode(n).resolved THEN  !if called on resolved node;
          BEGIN
            outtext("TRACEPATHS called on resolved node #");
            outint(n,4);outimage;
          END ELSE
            cgraph.getnode(n).resolved:=TRUE; !else set it resolved;
          cgraph.getnode(n).transval:=minridge; !set the translation val;

          pth:-cgraph.getnode(n).pathlst.first QUA path;
              !get the first path (if any) out of this node;

          WHILE pth =/= NONE DO
          BEGIN
               IF NOT cgraph.getnode(pth.nodenum).resolved THEN
                 tracepaths(pth.nodenum);
                    !if the node this path points to is unresolved,;
                    !trace it;
               pth:-pth.suc;  !get the next path from this node (if any);
          END of WHILE;
     END of tracepaths;


     cgraph:-NEW graph;      !create our conflict graph;

     FOR n:=1 STEP 1 UNTIL usedrignums DO !for each of the rignums we used;
          NEW node(n).into(cgraph);      !create a node in the graph;

     cp:-clist.first QUA conflict_pair;  !first conflict pair;
     WHILE cp =/= NONE DO
     BEGIN
          NEW path(cp.ridge1).into(cgraph.getnode(cp.ridge2).pathlst);
              !insert the path (link) into the graph from node for;
              !ridge 1 to node for ridge2;
          NEW path(cp.ridge2).into(cgraph.getnode(cp.ridge1).pathlst);
              !insert the path in the opposite direction.;
          cp:-cp.suc;      !get the next conflict pair;
     END of WHILE...graph is now complete;
```

```
FOR n:=1 STEP 1 UNTIL usedrignums DO   !for each node in the graph;
    !though there should never be any connections to node 1;
BEGIN
     IF (cgraph.getnode(n).pathlst.empty) THEN
       cgraph.getnode(n).resolved:=TRUE;  !if this node has no paths;
              !to it, it is automatically resolved;

     IF NOT cgraph.getnode(n).resolved THEN !if this node nt resolved;
     BEGIN
          minridge:=n;              !the smallest value on the current;
                                    !connected sub-graph;
          tracepaths(n)             !recursively trace all the paths;
                                    !connected to node n;
     END of IF;

     IF cgraph.getnode(n).resolved THEN
       translatetbl[n]:=cgraph.getnode(n).transval ELSE
              !put the translation value in the table;
       BEGIN
       outtext("node number");outint(n,4);outtext(" never got "
                    "resolved");outimage;
       END;
END of FOR;
       !all nodes have hopefully now been resolved, and the correct;
       !translation values are in the array;

translatetbl[0]:=0;          !translate off pixels to off pixels;
translatetbl[1]:=1;          !same for 1 pixels (should be none);

availoutnum:=2;          !first output ridge number;
FOR rnum:=2 STEP 1 UNTIL usedrignums DO  !for each possible current r#;
BEGIN
     present:=FALSE;     !init to that ridge # being unused;
     FOR i:=2 STEP 1 UNTIL usedrignums DO !search for presence of rnum;
       IF translatetbl[i]=rnum THEN present:=TRUE; !found it;

     IF present THEN             !if rnum is currently used;
     BEGIN
          FOR i:=2 STEP 1 UNTIL usedrignums DO
            IF translatetbl[i]=rnum THEN translatetbl[i]:=availoutnum;
               !for each instance of a ridge to be replaced with rnum;
               !instead replace it with "availoutnum";

          availoutnum:=availoutnum+1;     !increment it;
     END of IF;
 END of FOR;
 !we have now "packed" things, so that no ridge numbers are wasted;

outint(availoutnum-1,4);outtext(" is the highest output ridge number");
outimage;


IF availoutnum>256 THEN
BEGIN
     outtext(">255 ridges are present...too bad.");
     outimage;
END;

FOR row:=1 STEP 1 UNTIL 400 DO
    FOR col:=1 STEP 1 UNTIL 400 DO
        BEGIN
             address:=picture.pixel(row,col);   !current pixel value;
```

```
            nvalue:=translatetbl[address];    !new value;
            picture.putpixel(row,col,nvalue);  !install new value;
        END of FOR;
        !***this is a hack for now, as values will be >255***;

END of revisenumbers;
```

```
PROCEDURE error(message);VALUE message; TEXT message;
BEGIN
    INTEGER ARRAY ac[1:4];
    outimage;
    outtext(message);
    outimage;
    ac[1]:=xwd(0,8R400000);          !process handle - current process;
    jsys(8R12,ac);                   !GETER...get most recent error;
    ac[1]:=xwd(0,8R777777);          !destination designator...TTY;
    !ac[2] was setup by GETER;
    jsys(8R11,ac);                   !type the error string on the TTY;
    jsys(8R170,ac);                  ! HALTF - Just give up;
END of error;

PROCEDURE copypict1to8(p1,p8); REF(pict1) p1; REF(pict8) p8;
!This procedure copies a single-bit (binary) picture into an 8-bit;
!(gray level) picture. Off (0) bits become pixels with value 0, while;
!on (1) bits become pixels with value 1;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p1.pixel(row,col)=1 THEN p8.putpixel(row,col,1)
       ELSE p8.putpixel(row,col,0);  !transfer the values;
END of copypict1to8;

PROCEDURE copypict8to1(p8,p1); REF(pict8) p8; REF(pict1) p1;
!This procedure copies an 8-bit (grey-level) picture into a 1-bit;
!(binary) image. Pixels with value 0 become off(0) pixels, and;
!pixels with value >=1 become on(1) pixels.;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p8.pixel(row,col)=0 THEN p1.putpixel(row,col,0)
       ELSE p1.putpixel(row,col,1);   !do the transfer;
END of copypict8to1;

PROCEDURE copypict1to9(p1,p9); REF(pict1) p1; REF(pict9) p9;
!This procedure copies a single-bit (binary) picture into an 9-bit;
!(gray-level) picture. Off (0) bits become pixels with value 0,;
!while on (1) bits become pixels with value 1;
BEGIN
     INTEGER row,col;
     FOR row:=1 STEP 1 UNTIL 400 DO
         FOR col:=1 STEP 1 UNTIL 400 DO     !for each pixel in the image;
         IF p1.pixel(row,col)=1 THEN p9.putpixel(row,col,1)
         ELSE p9.putpixel(row,col,0);       !transfer the values;
END of copypict1to9;

PROCEDURE copypict9to8(p9,p8); REF(pict9) p9; REF(pict8) p8;
!This procedure copies an 9-bit (gray-level) picture into an 8-bit;
!(gray-level) image. Pixels are copied with their value intact,;
!except any pixels with values >255 are copied as 255;
BEGIN
     INTEGER row,col;
     FOR row:=1 STEP 1 UNTIL 400 DO
         FOR col:=1 STEP 1 UNTIL 400 DO     !for each pixel in the image;
         IF p9.pixel(row,col)>255 THEN p8.putpixel(row,col,255)
         ELSE p8.putpixel(row,col,p9.pixel(row,col));
              !copy as is, unless >255, then put in a 255;
END of copypict9to8;
```

```
    line:-sysin.image;
    outtext("Name (XXX) of file (XXX.BIM) of thinned image file: ");
    breakoutimage;  inimage;
    infilename:-copy(line.strip);  !get rid of trailing blanks;

    outtext("Name (YYY) for output file (YYY.IMG): ");
    breakoutimage;  inimage;
    outfilename:-copy(line.strip);  !filename for output;



    picin:-NEW pict1;        !thinned input image;
    picout:-NEW pict8;       !numbered output image;
    pictemp:-NEW pict9;     !temporary intermediate image;
    picin.load(infilename); !get the input image;

    copypict1to9(picin,pictemp);   !make pictemp an 9-bit copy of the input
                                        !picture. 0-->0, 1-->1;
    picin:-NONE;                        !get rid of the pict1;
    arraysize:=2000;                !maximum number of initia! ridge numbers;

    clist:-NEW conflict_list;       !init our list of conflicts;

    numberthem(pictemp);             !assign preliminary numbers to each ridge;
                                    !also create list of numbering conflicts;

    outint(usedrignums,4); outtext(" is highest ridge number used");outimage;
    outint(clist.cardinal,4); outtext(" conflicts encountered");outimage;
        !tell how many conflicts we got;
    IF usedrignums>arraysize THEN
    BEGIN
         outtext("too many ridge numbers needed...increase array size");
         outimage;
    END of IF;

    IF usedrignums>511 THEN
    BEGIN
         outtext("more than 511 ridge numbers used...too bad");
         outimage;
    END of IF;

    revisenumbers(pictemp);              !renumber the ridges based upon the;
                                         !conflicts found;

    copypict9to8(pictemp,picout);      !copy from 9 bit image to 8 bit one;

    picout.store(outfilename);           !put image away;
  END of SIMSET block;
END of rignum;
```

```
!"THIN"
!This program operates on a binary (thresholded) image and
!produces one with all regions thinned to single pixel skeletons;
!The algoritm used looks only on a 3 x 3 region surrounding and
!including the pixel in question.;
BEGIN
EXTERNAL CLASS io20,pict1,pict8;
EXTERNAL PROCEDURE enterdebug;
EXTERNAL BOOLEAN PROCEDURE jsys,skipin;
EXTERNAL INTEGER PROCEDURE xwd,right,left,land,lor,lnot,taddr,aaddr;
EXTERNAL INTEGER PROCEDURE lshift,lxor;
EXTERNAL TEXT PROCEDURE conc,checkextension,rest,frontstrip;
EXTERNAL CHARACTER PROCEDURE findtrigger;
EXTERNAL PROCEDURE halt;

REF(pict1) picin,picout;
REF(pict8) pictemp;
TEXT line,infilename,outfilename;
INTEGER row,col;
BOOLEAN gotsomething;

PROCEDURE copypict1to8(p1,p8); REF(pict1) p1; REF(pict8) p8;
!This procedure copies a single-bit (binary) picture into an 8-bit;
!(gray level) picture. Off (0) bits become pixels with value 0, while;
!on (1) bits become pixels with value 1;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p1.pixel(row,col)=1 THEN p8.putpixel(row,col,1)
       ELSE p8.putpixel(row,col,0);  !transfer the values;
END of copypict1to8;

PROCEDURE copypict8to1(p8,p1); REF(pict8) p8; REF(pict1) p1;
!This procedure copies an 8-bit (grey-level) picture into a 1-bit;
!(binary) image. Pixels with value 0 become off(0) pixels, and;
!pixels with value >=1 become on(1) pixels.;
BEGIN
    INTEGER row,col;
    FOR row:=1 STEP 1 UNTIL 400 DO
       FOR col:=1 STEP 1 UNTIL 400 DO    !for each pixel in the image;
       IF p8.pixel(row,col)=0 THEN p1.putpixel(row,col,0)
       ELSE p1.putpixel(row,col,1);   !do the transfer;
END of copypict8to1;

PROCEDURE usetemplate(pict,t1,t2,t3,t4,t5,t6,t7,t8,t9,newval,rots);
INTEGER newval,rots,t1,t2,t3,t4,t5,t6,t7,t8,t9; REF(pict8) pict;
!This procedure operates on an 8-bit (grey-level) picture ('pict').;
!The template array is used as the pattern which the neighbor cells;
!of each cell in the picture (and the cell itself) must match in order;
!for the value of that cell to be changed to 'newval';
!The cell numbering scheme is:
!                                  1 2 3
!                                  8 9 4
!                                  7 6 5
!
!The possible values for the entries in the template array are 0-255
!(which correspond to the possible values of a pixel), and -1, which
!implies that that spot in the template is a 'dont care' (i.e. it matches
!any pixel, no matter what its value).
!The parameter 'rots' is used to specify if rotations of the template are
!to be used for matching as well. Rot=0 ==> use no rotations.
!Rot=4 ==> use all 4 4-rotations. Rot=8 ==> use all 8-rotations.;
```

```
BEGIN
  INTEGER row,col,i,nbrptr;
  INTEGER ARRAY nhood[1:9],t[1:9];
  outtext(":"); breakoutimage;
  t[1]:=t1; t[2]:=t2; t[3]:=t3; t[4]:=t4; t[5]:=t5; t[6]:=t6;
  t[7]:=t7; t[8]:=t8; t[9]:=t9; !move the template into its array;
  FOR row:=1 STEP 1 UNTIL 400 DO
    FOR col:=1 STEP 1 UNTIL 400 DO BEGIN  !for each pixel in the image;
      IF ((t[9]=-1) OR (t[9]=pict.pixel(row,col))) THEN
      BEGIN     !if center cell of template is a dont care, or it matches
                !the current cell in the picture, then go on;
       FOR i:=1 STEP 1 UNTIL 9 DO
         nhood[i]:=pict.pixnbr(row,col,i);   !get the neighborhood data;

        IF rots=0 THEN BEGIN
IF templatematch(t,nhood,1) THEN BEGIN
            pict.putpixel(row,col,newval);
            gotsomething:=TRUE;      !flag sent upstairs...we got one;
            END;
          !check for template match...do not rotate template (start at 1);
          END
        ELSE IF rots=4 THEN  BEGIN
        FOR nbrptr:=1 STEP 2 UNTIL 7 DO  !do all 4 4-rotations;
         IF templatematch(t,nhood,nbrptr) THEN BEGIN
                              pict.putpixel(row,col,newval);
                              gotsomething:=TRUE;
            !and check for a template match;
                              END; END
        ELSE IF rots=8 THEN BEGIN
        FOR nbrptr:=1 STEP 1 UNTIL 8 DO  !do all 8 8-rotations;
        IF templatematch(t,nhood,nbrptr) THEN BEGIN
                    pict.putpixel(row,col,newval);
                    gotsomething:=TRUE;
            !and check for template matches;
                    END; END
           ELSE BEGIN
         outtext(" Illegal rotation spec. in call to 'usetemplate'");
             outimage; halt; END;
      END of IF;
      END of FOR;
END of usetemplate;



BOOLEAN  PROCEDURE templatematch(t,nhood,nbrptr);
INTEGER ARRAY t,nhood;INTEGER nbrptr;
 !This routine checks to see if the template in array 't' matches
 !the pixels in the array 'nhood'. The template is rotated as specified
 !by 'nbrptr' (nbrptr is the template cell which ends up in the '1'
 !cell position after template rotation;
 BEGIN
   BOOLEAN okay;
   INTEGER cell,modcell;
   okay:=TRUE;   !initialize our flag;
   IF ((t[9]\=-1) AND (t[9]\=nhood[9]))  THEN
      templatematch:=FALSE  !if we definitely dont have a match;
      ELSE BEGIN FOR cell:=1 STEP 1 UNTIL 8 DO
          !for each cell in the neighborhood;
        BEGIN
        modcell:=MOD(nbrptr+cell-2,8)+1; !rotate our template;
        IF (t[modcell]\=-1) THEN BEGIN  !if this spot in template not
                                        !a dont care;
        IF t[modcell]\=nhood[cell] THEN okay:=FALSE; !didnt match;
```

```
        END of if;
        END of FOR;
        templatematch:=okay;
     END of IF;
END of templatematch;



line:-sysin.image;
outtext('Name (XXX-YYY) of file (XXX-YYY.BIM) to thin: ');
breakoutimage;
inimage;
infilename:-copy(line.strip);   !get rid of trailing blanks;

outtext('Name (UUU-VVV) of file (UUU-VVV.BIM) for result: ');
breakoutimage;
inimage;
outfilename:-copy(line.strip);   !filename for output;

picin:-NEW pict1;         !our image;
pictemp:-NEW pict8;       !temporary 8-bit picture;
picout:-NEW pict1;        !output image;
picin.load(infilename);  !get the image;

copypict1to8(picin,pictemp);      !make pictemp an 8-bit copy of picin;
                                  !0-->0, 1-->1;

gotsomething:=TRUE;   !so we go thru the loop at least once;
WHILE gotsomething EQV TRUE DO  !loop until nothing is changing;
BEGIN
gotsomething:=FALSE;        !nothing has changed yet;
   usetemplate(pictemp,0,-1,1,1,1,-1,0,0,1,2,8);  !use the template and
                                                   !all of its 8-rotations;
   usetemplate(pictemp,0,-1,-1,1,-1,-1,0,0,2,0,8); !again..new template;
  END of WHILE;

usetemplate(pictemp,-1,-1,-1,-1,-1,-1,-1,-1,2,1,0); !change 2's to 1's;

                                                  !thin 2-pixel skeleton;
usetemplate(pictemp, 0, 0,-1, 1,-1, 1,-1, 0, 1, 0, 0);     !to 1-pixel one;
usetemplate(pictemp, 0, 0, 0,-1,-1, 1, 1,-1, 1, 0, 0);      !C2;
usetemplate(pictemp,-1, 0, 0, 0,-1, 1,-1, 1, 1, 0, 0);      !C3;
usetemplate(pictemp, 1,-1, 0, 0, 0,-1,-1, 1, 1, 0, 0);      !C4;
usetemplate(pictemp,-1, 1,-1, 0, 0, 0,-1, 1, 1, 0, 0);      !C5;
usetemplate(pictemp,-1, 1, 1,-1, 0, 0, 0,-1, 1, 0, 0);      !C6;
usetemplate(pictemp,-1, 1,-1, 1,-1, 0, 0, 0, 1, 0, 0);      !C7;
usetemplate(pictemp, 0,-1,-1, 1, 1,-1, 0, 0, 1, 0, 0);      !C8;

!now for the reflections;
usetemplate(pictemp,-1, 0, 0, 0,-1, 1,-1, 1, 1, 0, 0);      !C1;
usetemplate(pictemp, 0, 0, 0,-1, 1, 1,-1,-1, 1, 0, 0);      !C2;
usetemplate(pictemp, 0, 0,-1, 1,-1, 1,-1, 0, 1, 0, 0);      !C3;
usetemplate(pictemp, 0,-1, 1, 1,-1,-1, 0, 0, 1, 0, 0);      !C4;
usetemplate(pictemp,-1, 1,-1, 1,-1, 0, 0, 0, 1, 0, 0);      !C5;
usetemplate(pictemp, 1, 1,-1,-1, 0, 0, 0,-1, 1, 0, 0);      !C6;
usetemplate(pictemp,-1, 1,-1, 0, 0, 0,-1, 1, 1, 0, 0);      !C7;
usetemplate(pictemp,-1,-1, 0, 0, 0,-1, 1, 1, 1, 0, 0);      !C8;


copypict8to1(pictemp,picout);     !make picout a 1-bit version of pictemp;
                                  !0-->0 (>=1)-->1;
picout.store(outfilename);   !put image away;

END of thin;
```

Bibliography

[Ambler73]
A.P. Ambler, H.G. Barrow et al
"A Versatile Computer-Controlled Assembly System"
1973 Stanford University AI Conference

[Arcelli81]
Carlo Arcelli
"Pattern Thinning by Contour Tracing"
Computer Graphics and Image Processing 17, 1981, pp. 130-144

[Augustson70]
J. Gary Augustson and Jack Minker
"An Analysis of Some Graph Theoretical Cluster Techniques"
Journal of the ACM 17, 1970, pp. 572-587

[Balaban76]
A.T. Balaban (ed.)
Chemical Applications of Graph Theory
Academic Press, 1976

[Banner75]
Conrad S. Banner and Robert M. Stock
"The FBI's Approach to Automatic
Fingerprint Identification"
FBI Law Enforcement Bulletin, January-February 1975

[Barrow76]
H.G. Barrow and R.M. Burstall
"Subgraph Isomorphism, Matching Relational Structures,
and Maximal Cliques"
Information Processing Letters 4:4, January 1976, pp. 83-84

[Benzer59]
Seymour Benzer
"On the Topology of the Genetic Fine Structure"
Proceedings of the National Academy of Sciences 45
1959, p. 1607

[Berztiss73]
A.T. Berztiss
"A Backtrack Procedure for Isomorphism of Directed Graphs"
Journal of the ACM 20:3, July 1973, pp. 365-377

[Bron73]
Coen Bron and Joep Kerbosch
"Algorithm 457: Finding All Cliques of an Undirected Graph"

Communications of the ACM 16:9, September 1973, pp. 575-577

[Browning80]
Sally Browning
The Tree Machine: A Highly Concurrent Computing Environment
PhD Thesis, Caltech, 1980

[Castleman79]
Kenneth R. Castleman
Digital Image Processing
Prentice-Hall, 1979

[Cheng81]
J.K. Cheng and T.S. Huang
"A Subgraph Isomorphism Algorithm Using Resolution"
Pattern Recognition 13:5, 1981, pp. 371-379

[Corneil70]
D.G. Corneil and C.C. Gotlieb
"An Efficient Algorithm for Graph Isomorphism"
Journal of the ACM 17:1, January 1970, pp. 51-64

[Duda73]
Richard O. Duda and Peter E. Hart
Pattern Classification and Scene Analysis
Wiley-Interscience, 1973

[Dyer81]
Charles R. Dyer and Azriel Rosenfield
"Parallel Image Processing by Memory-Augmented
Cellular Automata"
IEEE Transactions on Pattern Analysis and
Machine Intelligence PAMI-3:1, January 1981, pp. 29-41

[FBI77]
Federal Bureau of Investigation
"Fingerprint Identification"
1977

[Foote74]
Robert D. Foote
"Fingerprint Identification: A Survey of Present Technology,
Automated Applications and Potential for Future Development"
Criminal Justice Monograph V:2, 1974
Institute of Contemporary Corrections and the
Behavioral Sciences

[Fulkerson65]
D.R. Fulkerson and O.A. Gross
"Incidence Matrices and Interval Graphs"
Pacific Journal of Mathematics 15, 1965, pp. 835-855

[Garey80]
M.R. Garey et al
"The Complexity of Coloring Circular Arcs and Chords"
Siam Journal Alg. Disc. Meth. 1:2, June 1980, pp. 216-227

[Garey79]
Michael R. Garey and David S. Johnson
Computers and Intractability: A Guide to the Theory of NP-Completeness
W.H. Freeman and Co., 1979

[Gati79]
Georg Gati
"Further Annotated Bibliography on the Isomorphism Disease"
Journal of Graph Theory 3, 197, pp. 95-109

[Gilmore64]
P.C. Gilmore and A.J. Hoffman
"A Characterization of Comparability Graphs and of Interval Graphs"
Canadian Journal of Mathematics 16, 1974, pp. 539-548

[Golumbic80]
Martin Charles Golumbic
Algorithmic Graph Theory and Perfect Graphs
Academic Press, 1980

[Gonzalez77]
Rafael C. Gonzalez and Paul Wintz
Digital Image Processing
Addison-Wesley, 1977

[Gonzalez78]
Rafael C. Gonzalez and Michael G. Thomason
Syntactic Pattern Recognition - An Introduction
Addison-Wesley, 1978

[Grasselli66]
A. Grasselli
"A Note on the Derivation of Maximal Compatibility Classes"
Calcolo 3, 1966, pp. 165-176

[Gray71]
Stephen B. Gray
"Local Properties of Binary Images In Two Dimensions"
IEEE Transactions on Computers C-20:5, May 1971, pp. 551-561

[Hall80]
Patrick A.V. Hall and Geoff R. Dowling
"Approximate String Matching"
Computing Surveys 12:4, December 1980, pp. 381-402

[Hankley]
W.J. Hankley and J.T. Tou
"Automatic Fingerprint Interpretation and Classification via Contextual Analysis and Topological Coding"
Learning Machines

[Harary69]
Frank Harary
Graph Theory
Addison-Wesley, 1969

[Hopcroft72]
J. Hopcroft and R. Tarjan
"Isomorphism of Planar Graphs". In *Complexity of Computations*
Plenum Press. New York, 1972, pp. 143-150.

[Hughes67]
Hughes Research Laboratories
"Proposal for the Development, Demonstartion, and Testing of a Device for Reading Fingerprint Minutiae"
February 1967

[Johnson81]
David S. Johnson
"The NP-Completeness Column: An Ongoing Guide"
Journal of Algorithms 4, 1981, pp. 393-405

[Johnson82]
David S. Johnson
Bell Laboratories
Personal Communication

[Kandel79]
Abraham Kandel and Samuel C. Lee
Fuzzy Switching and Automata: Theory and Applications
Crane Russak, 1979

[Kruse]
Bjorn Kruse
Design and Implementation of a Picture Processor
Linkoeping University

[Ladner75]
R.E. Ladner
"On the Structure of Polynomial Time Reducibility"
J. Association Comput. Mach., 22 (1975) pp. 155-171

[Lekkerkerker62]
C.G. Lekkerkerker and C.J. Boland
"Representation of a Finite Graph by a Set of Intervals on the Real Line"
Fundamenta Mathematicae LI, 1962, pp. 45-64

[Levi72]
G. Levi
"A Note on the Derivation of Maximal Common Subgraphs of Two Directed or Undirected Graphs"
Calcolo 9-10, 1972-1973, pp. 341-352

[Levi73]
Giorgio Levi and Fabrizio Luccio
"A Technique for Graph Embedding with Constraints on Node and Arc Correspondences"
Information Sciences 5, 1-24, 1973, pp. 1-25

[Levitt72]
K.N. Levitt and W.H. Kautz
"Cellular Arrays for the Solution of Graph Problems"
Communications of the ACM 15:9, September 1972, pp. 789-801

[Lougheed80]
Robert M. Lougheed et al
"Cytocomputers: Architectures for Parallel Image Processing"
IEEE 1980

[Lubiw81]
Anna Lubiw
"Some NP-Complete Problems Similar to Graph Isomorphism"
Siam J. Comput. 10:1, February 1981, pp. 11-21

[Lueker79]
George S. Lueker and Kellogg S. Booth
"A Linear Time Algorithm for Deciding Interval Graph Isomorphism"
Journal of the ACM 26:2, April 1979, pp. 183-195

[Meetham68]
A. Roger Meetham
"Partial Isomorphisms in Graphs and Structural Similarities in Tree-Like Organic Molecules"
Information Processing 68, 1969, pp. 210-213

[Moayer76]
Bijan Moayer and King-Sun Fu
"A Tree System Approach for Fingerprint Pattern Recognition"
IEEE Transactions on Computers C-25:3, March 1976, pp. 262-274

[Moenssens71]
Andre A. Moenssens
Fingerprint Techniques
Chilton, 1971

[Moon65]
J.W. Moon and L. Moser
"On Cliques in Graphs"
Israel Journal of Mathematics 3-4, 1965-1966, pp. 23-29

[Mulligan72]
Gordon D. Mulligan and D.G. Corneil
"Corrections to Bierstone's Algorithm for Generating Cliques"
Journal of the ACM 19, 1972, pp. 244-247

[Newman73]
William M. Newman and Robert F. Sproull
Principles of Interactive Computer Graphics
McGraw-Hill, 1973

[Ohtsuki79]
Tatsuo Ohtsuki et al
"One-Dimensional Logic Gate Assignment and Interval Graphs"
IEEE Transactions on Circuits and Systems CAS-26:9
September 1979, pp. 675-684

[Pavlidis80]
Theo Pavlidis
"A Thinning Algorithm for Discrete Binary Images"
Computer Graphics and Image Processing 13, 1980, pp. 142-157

[Randic77]
Milan Randic
"On Canonical Numbering of Atoms in a Molecule and
Graph Isomorphism"
Journal of Chemical Information and Computer Sciences 17:3
1977, pp. 171-180

[Read77]
Ronald C. Read and Derek G. Corneil
"The Graph Isomorphism Disease"
Journal of Graph Theory 1, 1977, pp. 339-363

[Rosenfeld66]
Azriel Rosenfeld and John L. Pfaltz
"Sequential Operations in Digital Picture Processing"
Journal of the ACM 13:4, October 1966, pp. 471-494

[Rosenfeld70]
Azriel Rosenfeld
"Connectivity in Digital Pictures"
Journal of the ACM 17:1, January 1970, pp. 146-160

[Savage81]
Carla Savage and Joseph Jaja
"Fast, Efficient Parallel Algorithms for Some Graph Problems"
Siam Journal Comput. 10:4, November 1981, pp. 682-691

[Schmidt76]
Douglas Schmidt and Larry Druffel
"A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism
Using Distance Matrices"

Journal of the ACM, 23:3, No. 3, July 1976, pp. 433-445.

[Seno77]
Atsuro Seno, Kunio Fukunaga, and Tamotsu Kasai
"An Algorithm for Graph Isomorphism"
Unknown Journal

[Sternberg76]
S.R. Sternberg
"Automatic Image Processor"
U.S. Patent 4,167,728

[Sussenguth65]
Edward H. Sussenguth, Jr.
"A Graph-Theoretic Algorithm for Matching Chemical Structures"
J. Chemical Doc. 5:1, February 1965, pp. 36-43

[Tou74]
J.T. Tou and R.C. Gonzalez
Pattern Recognition Principles
Addison-Wesley, 1974

[Trauring61]
Mitchell Trauring
"On the Automatic Comparison of Finger Ridge
Patterns for Personal-Identity Verification"
Research Report 190
Hughes Research Laboratories, March 1961

[Trauring63]
Mitchell Trauring
"An Automatic System for Personal Identity Verification
Based on Finger Ridge Pattern Comparison"
Research Report 249
Hughes Research Laboratories, April 1963

[Tsai79]
Wen-Hsiang Tsai and King-Sun Fu
"Error-Correcting Isomorphisms of Attributed Relational
Graphs for Pattern Analysis"
IEEE Transactions on Man, Machine, and Cybernetics SMC-9:12
December 1979, pp. 757-768

[Tucker]
Alan Tucker
"Circular-Arc Graphs: New Uses and a New Algorithm"
Lecture Notes in Mathematics 642 ed. by A. Dold
Springer-Verlag

[Tucker74]
Alan Tucker
"Structure Theorems for Some Circular-Arc Graphs"

Discrete Mathematics 7-8, 1974, pp. 167-195

[Tucker81]
Alan Tucker
State University of New York at Stonybrook
Personal Communication

[Ullmann76]
J.R. Ullmann
"An Algorithm for Subgraph Isomorphism"
Journal of the ACM 23:1, January 1976, pp. 31-42

[Unger64]
Stephen H. Unger
"GIT - A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism"
Communications of the ACM 7:1, January 1964, pp. 26-34

[Wah81]
Benjamin W. Wah and Y.W. Ma
"The Architecture of MANIP - a Parallel Computer System for Solving NP-Complete Problems"
National Computer Conference 1981, pp. 149-161

[Wegstein68a]
J.H. Wegstein
"A Computer Oriented Single-Fingerprint Identification System"
NBS Technical Note 443, March 1968

[Wegstein68b]
J.H. Wegstein and J.F. Rafferty
"Matching Fingerprints by Computer"
NBS Technical Note 466, July 1968

[Wegstein69]
J.H. Wegstein
"A Semi-Automated Single Fingerprint Identification System"
NBS Technical Note 481, April 1969

[Wegstein70]
J.H. Wegstein
"Automated Fingerprint Identification"
NBS Technical Note 538, August 1970

[Wong79]
Vincent S. Wong
Computational Structures for Extracting Edge Features from Digital Images for Real-Time Control Applications
PhD Thesis, Caltech, 1979