THEORY AND METHODS FOR CHARACTER RECOGNITION OF GURMUKHI SCRIPT

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT FOR THE AWARD OF THE DEGREE OF MASTER OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING

TO

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED UNIVERSITY)

PATIALA--147001

BY

RENU DHIR

DEPARTMENTOF COMPUTER SCIENCE&ENGINEERING

T.T.T.I. SECTOR-26 CHANDIGARH--160019

JULY 1996

CERTIFICATE

This is to certify that the work reported in this dissertation by MS. RENU DHIR of the Computer Science Department of TTTI, Chandigarh has been carried out under the guidance of MR. TARUN MARHWAL, Lecturer in Computer Science as the internal guide and SH. A.L. SANGAL (Head of the Department of Computer Science & Engineer, R.E.C., Jalandhar) as the external guide.

(DR. S.K. BHATTACHARYA) 31/10/96
Head,
Department of Computer Science,
TTTI, Chandigarh.

CERTIFICATE

Certified that the dissertation entitled "THEORY AND METHODS FOR CHARACTER RECOGNITION OF GURUMUKHI SCRIPT" submitted in partial fulfilment of the requirements for the degree of Master of Engineering is a record of bonafide work carried out by MS. RENU DHIR, Student of M.E.(Computer Sc.& Engg.) course of the Technical Teacher's Training Institute, Chandigarh under my superivision. The results presented here are the outcome of the candidate's own efforts.

Tarun

Supervisor

(TARUN MARHWAL)
Lecturer
Department of Computer Science
& Engineering.
TTTI, Sector 26, Chandigarh.

CERTIFICATE

This is to certify that the dissertation entitled *"Theory And Methods For Character Recognition Of Gurmukhi Script"* submitted by Ms.RENU DHIR, for the award of the degree of Master of Technology in Computer Science and Engineering is a record of bonafide work.

This dissertation or any part thereof has not been submitted to any other university/institution for the award of any degree or diploma.

HEAD,

Department of Computer Science & Engg.,

R.E.C. JALANDHAR.

ACKNOWLEDGEMENT

I express my sincere thanks and gratitude to Sh.A.L.Sangal (Head of the Dept. of Computer Science & Engg.,R.E.C. Jalandar) for his invaluable guidance and constant encouragement during the course of my work on this project.

I am again thankful to Mr. Tarun Marhwal (Lecturer, Technical Teacher's Training Institute Chandigarh) for his guidance and support for working on this project.

I would like to thank each and every member of my family especially my husband Anil & my loving daughter Ashma for providing me a constant support and understanding without which I would not have been able to complete my work.

I am very thankful to Mr. Gurpreet Lehal & Ritu Lehal for their kind help while preparing the dissertation.

RENU DHIR

CONTENTS

LIST OF FIGURES

ABSTRACT

Gurmukhi script is a moderately complex pattern of primitive characters their half forms, upper and lower vowel-modifier symbols and diacritical marks. The present work describes the design and implementation of a scheme for the off-line recognition of the isolated primitive characters of the Gurmukhi script. A hierarchical control strategy which exploits the character stroke features of varying degree of complexity to achieve efficiency in the recognition, is used.

CHAPTER 1

INTRODUCTION TO PATTERN RECOGNITION

1.1 Introduction

The problem of pattern recognition usually denotes a discrimination or classification of a set of processes or events. The set of processes or events to be classified could be a set of physical objects or a set of mental states. The number of pattern classes is often determined by the particular application in mind. In some problems, the exact number of classes may not be known intially, and it may have to be determined from the observations of many representative patterns. In this case, we would like to detect the possibility of having new classes of patterns as we observe more and more patterns. Human beings perform the task of pattern recognition in almost every instant of their working lives.Recently, scientists and engineers started to use machines for pattern recognition.

Character recognition is an area of pattern recognition that has been subjected to a lot of research work during the last twenty years and migrated from research laboratories to the Industrial world and even to personal computing. This is due to the fact that recognition and processing of (Hand-) printed texts is basic function that should be automated in order to improve man-machine communication besides having many practical use of immense benefits. Speech recognition provides another such man-machine interface.

Presently the state of art in character recognition has advanced from the use of primitive schemes for the recognition of machine printed numerals to the application of sophisticated techniques for the recognition of a wide variety of hand printed characters and symbols. Many commercial character readers are also available, which employ these techniques for the recognition of the constrained characters with high accuracy. These readers use a variety of computer resources ranging from simple microcomputers to main frames. Current research work is generally oriented towards a system having better recognition rate, easy implementation and less computation. A lot of work has gone to devising techniques for the recognition of Latin [8], Thai[4], Chines [7], and Japanese [16] scripts. This thesis is concerned with a recognition scheme for the Gurmukhi script using a hierarchical control strategy.

The history of optical character reader got its start with the memorable patent on machine reading of printed numerals by Gustav Tauschek in 1928. The first prototypes of printed numeral OCRs were developed in the 1950s in America. Then as early as 1960s, OCRs for printed alphanumerics and symbols came into practical use and research began on machine recognition of hand written alphanumerics with the aim of recognizing free formatted handwriting.

In Japan between 1965 and 1968, research on developing automatic postal code reading and sorting machines that recognize 3-digit hand written numerals was carried out under the direction of the Ministry of posts and Telecommunications. This was followed by a national programme to research and develop a *pattern Information processing system,* which was implemented under the direction of the Ministry of International Trade and Industry in 1971. This large scale project to develop a high, yet practical, level of OCR technology for recognizing printed Kanji and handwritten Kana (Japanese phonetic characters) continued until 1980.

Two major paradigms can be seen in such past work:

1. **Emphasis on the shape of plane Regions** Invariably the input image is assumed to have one bit per pixel and the recognition effort focuses on the description of the shape of the 'black' areas of the input image. A considerable literature deals with thresholding techniques, namely methods for converting a digitized image with more than one bit per pixel into a *binary image*. Shape has been described either by structural or by incidental features. Examples of structural features are strokes, concavities of the contour of a region etc. Examples of incidental features are bit marks, Fourier coefficients etc.

2. **Emphasis on learning by Example** Regardless of the type of features used, the emphasis has been on inferring the parameters of the classifier from examples (the training set) using various statistical techniques. A set of features is selected and the parameters of maximum likelihood estimators are found from the given samples. Most of the estimation techniques are nonparametric in other words no explicit assumptions are made about the functional form of the statistical distribution of the feature vector.

   The reliance on 'learning by example' has worked against syntactic recognition techniques because of the intractability of the Problem of grammatical inference. The recent emphasis on neural nets

has amounted to an increased emphasis on the 'learning' procedure while de-emphasizing the design and selection of features.

1 .2 Performance criteria and causes of errors in OCR systems

The prevailing practice in the academic literature on OCR is to present the percentage of characters correctly recognized, first on the 'training set' and then on the 'test set'. There is a difference between different types of errors. At the very least, any percentages of errors should be separated into rejection rates and substitution rates. Such a distinction is uncommon not only in the research literature but it was also missing from a recent evaluation of commercial systems published in *Byte Magazine* [22].

The distinction is very important from a practical viewpoint. One would like zero substitution rate while the rejection rate might be as high as a few percentage points. For example, the bar code industry strives for substitution error rates ('misdecodes') of 10^{-6} or lower while a 10^{-2} rejection rate appears acceptable. (Both rates are per label, consisting of about ten characters). The reason is that misdecoded item at the check counter will charge the customer the wrong price while a rejected item will require rescanning or, at worst, manual entry. In the postal environment a substitution error means sending an item to the wrong destination while rejection error means submitting it to manual sorting.

One reason that designers of bar code readers and other communication means expect such low substitution error rates is the use of error detection and error correction techniques that take advantage of redundancies. Unfortunately in OCR we do not have the luxury of specifying how the information will be encoded and thus we must attempt to use existing redundancy. This could be achieved, for example, by using two different recognition algorithms and classifying only characters where both algorithms agree. If the two algorithms represent significant different methodologies (for example thinning versus contour tracing) it is likely that the number of substitution errors will decrease.

In order to improve performance, as well as in order to obtain accurate estimates of such performance it is necessary to classify the causes of errors encountered in OCR. The following is one such possible classification.

1. *Confusion because of shape similarity.* This is caused by mapping, for example, a 'C' onto a 'c'. This is a relatively minor transgression because human readers distinguish such pairs by context. Therefore in evaluating an OCR system weights should be given to different errors depending on how well humans can discriminate the difference. A weight might reflect the fraction of time human observers can tell the difference without context. For example the weight for the confusion between 'C' onto a 'c' might be zero, the weight for the pair 'T' and 'O' might be one, while that for the pair 'G' and 'C' might be 0.9.

2. *Confusion caused by printing.* Ink spread might cause the closing of gaps, faint reproduction might cause gaps. It is tempting to evaluate OCR systems only on high quality input but that is utterly unrealistic. When using poor quality input, errors should be weighted according to the performance of human observers.

3. *Confusion caused by digitization.* Various types of digitization distortion are there. While such errors may not be blamed on the recognition logic they represent a defect of the overall system and should be counted fully. On the other hand the realization of the cause should direct efforts towards better digitization and different type of features.Our aim should be to reduce digitization errors.

4. *Confusion because of error in feature selection.* Even if a character is printed and scanned perfectly it may be misread if the features are not computed correctly. For example, suppose the presence of a vertical stroke in the left side of the character is a feature. It should be present in such characters as 'L', 'P', 'R', etc. However, long serifs may cause an usually broad bounding box and the vertical may be detected as occurring in the center of the character[9].

5. *Confusion because of classifier design.* This will happen if the classifier has not been trained in all possible forms of characters.

It should be noted that most of the character segmentation errors are caused by the second and third factors listed and therefore segmentation is not listed separately as a cause of errors.

In view of the above, stating that an OCR system has recognition rate of 99% is a meaningless figure because it has no predictive value about the performance of the system in future tests, when, for example new fonts or different printing media might be seen. In the past, when OCR systems had to move from, say, 95% to 99% recognition rates the lack of such an analysis might have been forgivable because there was plenty of room for improvement. Now when we attempt to eliminate, if possible, all errors the categorization is essential. Ideally an OCR system should make no errors of the last three kinds, or at least achieve a rejection rate of the order of 10^{-4} or less.

Such a request for near perfection is not academic. A printed document page contains about 3,000 characters, therefore an error rate of 1% implies about 30 errors per page. A typist with such a performance is likely to be fired but some times such error rates are stated with pride in the academic literature. On the other hand a 10^{-4} rate implies one error every three pages, a better performance than that of most human typists. It might also be necessary to reach such performance levels before OCR equipment finds widespread use in office automation.

Turning to postal applications we note that an address contains about 50 characters, excluding the recipient's name. Therefore a character error rate of 1% implies an error in every other piece of mail, while a character error rate of 10^{-4} implies an error every few hundred items.

1.3 Effects of the past paradigms on the performance of OCR systems

While an ideal text image is bilevel, the image seen by the sensors has a wide range of gray scale for the following reasons (in order of importance):

a) the point spread function of the scanner;

b) nonuniform illumination;

c) multiplicative noise (nonuniform paper reflection);

d) additive noise (due to electronics).

In addition, there are cases where the background itself may not be uniform. When such a gray scale image is binarized there is a significant information loss: typically areas which are narrow compared to the support of the point spread function disappear resulting in broken characters (if the narrow area was dark) or touching characters (if the narrow area was white).

The insistence on immediate binarization introduces significant distortions of the input and the performance of and OCR system may degrade because the *shape* of the character has been severely distorted by digitization. Since the averaging of the input by the point spread function of the scanner is the major source of trouble, there has been a trend towards higher scanner resolution so that 300 samples per inch is common today and in some case 400 or more samples per inch are used. If gray scale is used then much lower resolutions might be allowed.

The heavy reliance on learning from examples causes OCR systems to fail to read text printed in a typeface different than those present in the training set even though the text is easily readable by a human. Consider for example the two forms of the letter 'P' shown in Fig. 1.1. The one on the left has a closed loop while the one on the right does not., (This is similar to the form of the letter 'P' in Palatino typefaces.) If samples of 'P' with the open loop have not been included in the training set the it is likely that the second character will be misclassified. This may happen either with structural or with incidental features. In the former a closed loop might be a required feature, in the latter there might be a mask with a form of the pixels in the lower junction of the loop. As a matter of fact, the likely distortion is to fill the narrow gap in the 'P' with the open loop so that even if such samples were included in the training set the configuration might be missed. Because of training on an incomplete set classifiers may reject characters that have a perfectly good shape.

Figure 1.1 : Example of two characters that look quite similar but they are topologically different

1.4 Direct recoginition from gray scale.

Instead of attempting to reconstruct the original bilevel image and then extract features for recognition it might be preferable to attempt to extract features directly from the gray scale image. Let us examine the case of a single vertical stroke which can be approximated by a rectangle with vertical sides at x = $\pm W$ and horizontal sides at y= $\pm H$. We assume that white has value 0 and black has value 1. Let $h(x,y)$ be the point spread function which is assumed to be bell shaped, i.e., having central symmetry, positive, and achieving its maximum at the origin. Then the gray scale image will be given by

$$g(x,y)\int_{-H}^{H}\int_{-W}^{W} h(x-u,y-v)dudv \qquad (1)$$

Because of the assumption about $h(x,y)$, it can be easily verified that $g(x,y)$ attains a maximum at the origin. Furthermore, if either x or y is fixed, the function $g(x,y)$ achieves a maximum with respect to the other variable at zero. Thus the gray scale image, viewed as a surface, exhibits two ridges along the two axes.

If one looks only for ridges where the curvature at the direction perpendicular to the ridge is high, then the result for this example is a vertical line along the middle of the rectangle. This suggests a way for feature extraction, similar to a thinning algorithm. Ridges correspond to spines of strokes, saddle points correspond to narrow gaps (either black or white) and flat areas correspond to junctions of strokes. Very wide strokes may give rise to a pair of ridges. Initially all saddle points are merged with the ridges so that a character printed by a dot matrix printer yields continuous strokes. However, if a character is not recognized, then the saddle points are treated as gaps and a new recognition is attempted for the resulting two or more charaters.

In summary, there are at least three major advantages in using topographical features.

1. One does not have to worry about the selection of dark/white threshold but only about a curvature threshold. The latter is entirely independent of the average level of brightness and largely independent of contrast.
2. By keeping track of saddle points, one has flexibitlity in handling narrow white or dark gaps.
3. Lower sampling rates than what is thought now necessary might be possible. As a result not only the scanner cost will be lower than now but also processing might be faster because the resulting image arrays will be smaller.

Since connected regions of ridge and saddle point pixels usually correspond to individual characters, any recognition technique can be applied to those regions in the same way that is applied to thresholded character images.

1.5 Analytical specification of the classifier

In contrast to cursive script, machine printed characters have formal specifications which are available as font tables. Therefore it makes sense to ask for a formal definition of the letter 'A' for example. Understanding of the sources of distortions and noise could be used to produce specifications of characters that could be used effectively for recognition. The following is an illustration of how these concepts can be applied.

Example. In many type faces the letter 'A' has two thin strokes and one thick as shown in Fig. 1.2. Therefore the right diagonal stroke is much more likely to be detected than the other two.

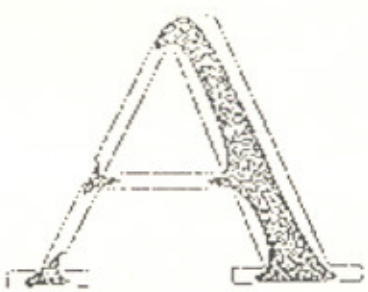


Figure 1.2 : Example of character 'A' drawn with one stroke and two dots

The example illustrates that a naive definition, such as "two slanted lines meeting at the top with a horizontal line in the middle" is not likely to be appropriate. The ideal prototypes of characters should be convolved with an estimated point spread function and characteristic features should be derived by examining the topographical features of the resulting surface. The following is a list of factors that must be taken into account.

1. Connectivity and other topological features are not appropriate. They can be affected either by distortions and noise or by the typeface specifications as shown in Fig. 1.1.
2. As a consequence of (1), loops should be specified as a sequence of arcs without insisting on complete closure.
3. Even when topology holds, angles and corners may not always be reliable dividers and alternate descriptions of strokes and arcs should be considered.
4. Proximity should be used instead of connectivity or contact.
5. Junctions tend to persist through convolution as the sample of Fig. 1.2 illustrates and they are good candidates for features.
6. Besides junctions, other 'anomalies' might constitute good features since recognition is based on the *unexpected* and not on the expected.

Ideally, we would like common definitions both for machine-printed and for hand-printed characters. We should try to include any handwriting that humans can recognize without resorting heavily to context.

1.6 Classification based on Recognition Approaches

Like other areas in pattern recognition, there exist two main approaches, namely *Decision Theoretic* or *Statistical approach* and *Syntactic approach*(or *linguistic approach*). In the *decision-theoretic approach*, a set of characteristic measurements, called features are extracted from the patterns: the recognition of each pattern (assignment to a pattern class) is usually made by partitioning the feature space. Most of the developments in pattern recognition research during the past decade deal with the

decision-theoretic approach. Application include character recognition ,crop classification, medical diagnosis, classification of electrocardiograms etc.

In the *decision-theoretic* approach, instead of simply matching the input pattern with the templates, the classification is based on a set of selected measurements, extracted from the input pattern. These selected measurements called "feature" are supposed to be invariant or less sensitive with respect to the commonly encountered variations and distortions and also containing less redundancies. Under this proposition, pattern recognition can be considered as consisting of two subproblems.

The first subproblem is what measurements should be taken from the input patterns. Usually the decision of what to measure is rather subjective and also dependent on the practical situations (for example, the availability of measurements, the cost of measurements. etc.).At present, there is very little general theory for the selection of feature measurements. However, there are some investigations concerned with the selection of a subset and the ordering of features in a given set of measurements. The criterion of feature selection or ordering is often based on either the importance of the features in characterizing the patterns or the contribution of the features to the performance of recognition (i.e., the accuracy of recognition).

The second subproblem in pattern recognition is the problem of classification (or making a decision on the class assignment to the input patterns) based on the measurements taken from the selected features. The device or machine which extracts the feature measurements from input patterns is called a feature extractor. The device or machine which performs the function of classification measurements is called a classifier. A simplified block diagram of a pattern recognition system is shown in figure 1.3. Thus in general terms the template-matching approach may be interpreted as a special case of the second approach---"feature-extraction" approach where the templates are stored in terms of feature measurements and a special classification criterion (matching) is used for the classifier.

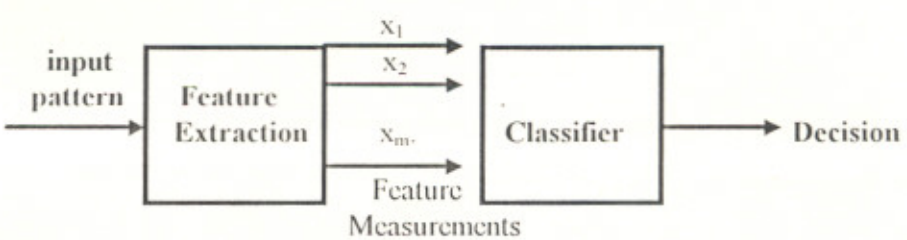


Figure 1.3 : A pattern recognition system

In some pattern recognition problems the structural information which describes each pattern is important, and the recognition process includes not only the capability of assigning the pattern to a particular class (to classify it), but also capacity to describe aspects of the pattern which make it ineligible for assignment to another class. A typical example of this class of recognition problem is picture recognition or more generally speaking, scene analysis. In this class of recognition problems, the patterns under consideration are usually quite complex and the number of features required is often very large which makes the idea of describing a complex pattern in terms of a (hierarchical) composition of simpler subpatterns very attractive. Also, when the patterns are complex and the number of possible descriptions is very large, it is impractical to regard each description as defining a class (for example, in fingerprint and face identification problems, recognition of continuous speech, Chinese characters etc.). Consequently, the requirement of recognition can only be satisfied by a description for each pattern rather than the simple task of classification. In order to represent the hierarchical (tree-like) structural information of each pattern, that is a pattern described in terms of simpler subpatterns and each simpler subpattern again be described in terms of even simpler subpatterns etc. the *syntactic* or *structural approach* has been proposed.

The *Syntactic approach* is based on an attempt to exploit the obvious structural properties inherent in many patterns, and use formal grammars to characterize and ultimately identify the characters. In *Deterministic* or *Statistical approach*, the primitive operations are all executed and the resulting features are stored in a table. Identification is achieved by a simple table lookup. Both strategies have their advantages and limitations. In *syntactic approach*, while description of the characters. is achieved very elegantly, identification leads to highly unwieldy and complex grammars. Further approach. whereas in *deterministic approach*, incorporation of learning involves simple augmenting of the table. However, if the number of features is large there could be an explosion of the data base. Now, it is generally agreed that *syntactic approach* yields better results for character recognition. In our approach, both the approaches are used selectively at different levels of classification.

1.7 Thesis Outline

Component processes of a character recognition system have been discussed. A brief overview of the Gurmukhi script and various techniques adopted in the literature is also presented.

Chapter 2 describes the details of various techniques in character recognition.

Chapter 3 discusses the feature set, feature extraction process and character classification in our approach.

Chapter 4 discusses implementation of the character recognition schemes discussed in chapter 3.

Chapter 5 concludes the present work with the remarks about the implemented system and reports the scope for future work.

CHAPTER 2

METHODOLOGIES IN CHARACTER RECOGNITION

2.1 Introduction

Characters can be recognized in as many different way as they are written and numerous techniques have been developed over the years by the various researchers and applied to recognize characters automatically.

2.2 Different techniques for character recognition

Table 1 shows conventional techniques for hand written alphanumerics recognition. The most significant characteristic in the composition of alphanumerics is that most of them are one-component characters, e.g., Arabic numerals and uppercase Roman letters. Thus, a description is not needed of the positional or structural relation between components but only of the one-component shape of each character. As indicated in the table, fundamental approaches have already been proposed for one-component shape description. However, matching methods robust enough to be distortion-tolerant under realistic conditions have not been reported yet.

Table 1

Recognition techniques for handwritten alphanumerics

Pattern matching	• primitive sequence matching by cellular automation
	• contour shape matching by dynamic programming
	• local extrema of contour matching by decision tree
	• convex hull and convex deficiency matching based on metric properties
Feature matching	• approximation by moments
	• Expansion of the intrinsic function

- crossing count and projection
- feature point enumeration based on geometric/topological properties

2.2.1 New and promising approaches

Table 2 shows the new and promising approaches for robust handwritten character recognition in the light of both pattern matching and feature matching strategies.

Table 2

New and promising approaches

Pattern matching	• distortion tolerant shape matching with structural constraints
	• similarity measures unification for shape and structure
	• combinatorial correspondence search by efficient tree pruning
	• stochastic correspondence search by energy minimization analogy
Feature matching	• supervised/unsupervised learning of optimal description of feature space by neural networks or clustering technology
	• sophistication of feature measures based on statistical validation
Common approach	• enhanced and adaptive pre-processing by gray scale morphology
	• nonlinear shape normalization
	• deformation prediction by deterministic modeling or personal characteristics
	• evaluation method of shape quality as a standardization tool

These techniques can be classified in three ways based on

1. nature of the application
2. approaches used and
3. the features used.

These classification are discussed in subsequent sections.

2.3 Classification based on Nature of Applications

On the basis of the nature of applications, the work in character recognition can be grouped into two main schemes, namely Off-line character recognition and On-line character recognition.

A common observation for both on-line and off-line handwriting recognition system is that to read cursive script, system assumes well framed words--the task of segmenting and recognizing several words from a block of cursive writing remains mostly unaddressed. To present different systems in the field, we first analyse handwriting properties and recognition problems. There are different philosophies in tackling the problems of cursive handwriting recognition.

a) Handwriting properties and recognition problems

Off-line handwriting recognition is performed after the writing is complete. An optical scanner converts the image of the writing into raw data. Scanners have x and y resolutions of typically 300 dpi.. Although most OCR work has been on machine printed characters, there has been considerable effort on handwriting.

Any handwriting interpretation system must face the high variability of character shapes. One can easily think of examples where even the topology of the character changes from one instance to another. An algorithm for unconstrained hand-written word recognition must be able to successfully recognize the image of any word which can be formed by discrete characters, broken cursive(group of characters written with a single continuous motion), or a combination of them, or can be fully cursive.

There are many pattern recognition problems for handwriting. The main source of difficulty lies in the segmentation of the words into isolated characters. Segmentation is ambiguous, especially locally. Thus, different interpretations are possible at the letter level which make the recognition process of the whole word ambiguous, and even very costly.

b) State of the difficulty of the problem of off-line cursive recognition

Owing to the difficulty of the problem of off-line recognition, there are two main philosophical approaches: approaches with reliance on lexical knowledge, and the other introduced recently, without reliance on lexical knowledge.

First class of approaches which depend heavily upon lexical knowledge. Usually this class can be divided into two subclasses: analytic approaches and global approaches.

Analytic approaches attempt to identify letters or parts of letters (pertinent segments) which constitute the whole word. They have the advantage of being generalized to a large vocabulary with limited training.

Global approaches attempts to recognize the whole word or fragments of the word, where letters are concatenated. These methods suffer from the limitation of the vocabulary.

The second class without reliance on lexical knowledge is based mainly on a model of the process of handwriting. An effort is made to attain human-like performance by using a method based on pictorial alignment and on a model of the process of handwriting. The alignment approach permits recognition of character instances that appear embedded in fully cursive words. The alignment is affine transformation based in order to remove most of the variability of handwritten shapes.

In On-line character recognition characters are recognized at the scanning time. Consequently, more information regarding the order of input strokes is available. An example of this application is automatic letter sorting. In on-line recognition, characters are represented by line drawings. Therefore there is no need for skeletonization or contour extraction. But recognition time is a crucial factor in the design of on-line character recognition system. Whereas in off-line system, recognition is not done at the time of scanning the input. Therefore, the input image typically passes through the preprocessing steps like smoothing, thinning, segmenting etc. before character recognition. Thus, off-line recognition is a rather difficult process. However, an off-line system can use the knowledge about the script for the more accurate interpretation of the characters using contextual post processing of characters. Our system is intended to be an off-line system.

2.4 System Overview

The task of character recognition system can be divided into three phases (Figure 2.1)

1. Data acquisition
2. Preprocessing
3. Recognition

The input test is scanned and digitized on an optical scanner to produce a digital image consisting of a matrix of grey level points(pixels).

Due to the limitations of the scanner devices, different kinds of noise are present in the image. Therefore, this digitized image is put through preprocessing routines that smooth the image and eliminate the noise. Preprocessing of handwriting data is done prior to the application of shape recognition algorithms. The main interest in the preprocessing is to localize the words in the handwritten text. On the other hand some particular parameters such as 'base-line', 'ascenders' and 'descenders' position can be estimated more reliably if they are computed on the whole text rather than on individual words.

Word localisation is the task of isolating various written units, such as words or parts of words, prior to their recognition. Early spatial segmentation used only the X-coordinate information to separate the written units by their projection on the X-axis. Recent spatial segmentation techniques check for a two-dimensional separation of the written units.

A method for interpreting hand-written address estimates text lines in three steps: first, blocks are created from the input text which then are linked to constitute topological graphs. Some heuristics are used to adjust the obtained graph in order to make lines apparent. .Finally, text lines are consequently, words or parts of words are separated by adding knowledge to the system.

The method most often used for determining the baseline(or the main body) of handwritten text is that of a histogram of horizontal densities. The goal of this method is to find three zones in a line of writing: the upper zone of ascenders, the middle zone of small letters, and the lower zone of descenders. The definition of zones of uncertainty between the middle zone and respectively the upper zone and the

lower zone is obviously realistic as is the quantification of ascenders and descenders proportionate to their extension from the main body of the word.

Further preprocessing segments the image into textlines followed by the isolation of the words, then the isolation of the characters. Finally character is thinned to obtain the skeleton of the character in a standard form.

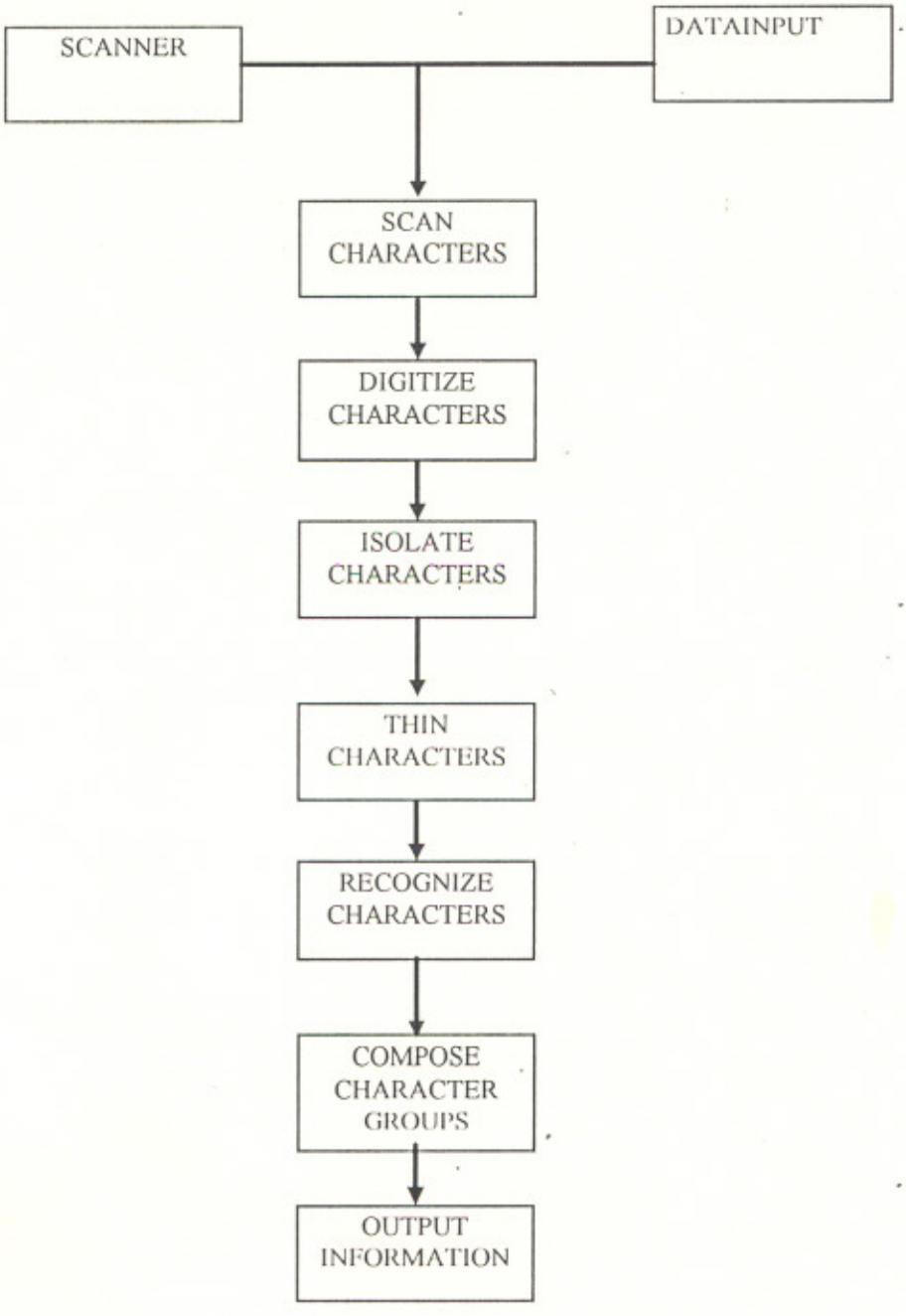


Figure 2.1 : Character Recognition

In the recognition phase, different features are extracted from the noise free and thinned character and the character is classified on their basis. These features can be classified into two general types:

Locally defined features such as discontinuities (end points), joins of two strokes (tee points) and crossings of two strokes (intersection points)

Globally defined features within a given character such as, character aspect ratio and gross descriptions of the shape or curvature of a stroke or sequence of character strokes.

Simple features are trivially extracted from the given skeleton of the character. However, the considerable information content of the features used allows for a rapid reduction of the set of possible character matches. The detection of high level (and more computationally expensive) features is postponed until after the low level features are identified and used to reduce the candidate set. Extraction of high level features occurs on a limited scale and is facilitated and directed by the small set of remaining candidate character match. Thus, if any ambiguity is there after reclassification, then it is resolved in the final classification and the decision about the identity of the character is made.

2.5 Application Areas

Initially character readers were developed to provide an aid to the visually Handicapped. Later on, after the advent of digital computers in mid 1940s, the potential of optical character readers in processing of large volume of data in form of bank cheques, commercial forms and government records etc., was realised. The following are the major application areas for which optical character readers have been used or suggested by the researchers:

Data Entry: Data entry can be reduced by the use of optical character readers in applications like customer billing in telephone exchange system and processing of cheques in Banking environment.

Text Entry: Document processing can be made faster in today's automated offices with the use of character readers.

Process Automation: In USA, character readers have been used to read typewritten addresses from letter mail and automatic sorting of the mail to its destinations.

Mapping: In recent years, character readers have been of great help in transitioning collected real world data into digital format maps and charts, i.e. to assist in Automated Cartography.

Blind readers: Character readers combined with speech synthesis system can help blind to understand written documents and literature.

Hand writing analyser: Character readers are now being used for automatic writer recognition and signature verification.

In Educational Institutions: Character readers have been used to speed up assessment of examinations, evaluation of attendance records and processing of application is to use character reader for teaching script on computer.

2.6 Classification based on Features

These recognition techniques vary widely according to the features chosen the way these features are extracted, and the classification scheme used. Based on these characteristic, techniques may be classified into three main categories

- Global features based
- Based on features derived from the distribution of the points
- Geometrical and topological feature based

These techniques have been discussed in following paragraphs.

2.6.1 Global Features based

This category comprises of techniques (described below) which extract features from every point which lies within a rectangle or a given defined area circumscribing the character (frame). Such features do not reflect any local, geometric or topological properties of the drawing itself.

1. **Template Matching and correlations** This technique takes as features the state (black or white) of all those points which lie with the frame. it simply measures the similarity between the input character and the stored references by matching and correlating points or groups of points in the frame.
2. **Transformations and Series Expansions** These techniques try to reduce the high dimensionally of feature vector resulting from the template matching and attempt to extract features invariant to some global deformation (e.g. global translation or rotation) by transformations and series expansions.

The methods mentioned above are characterized by their low sensitivity to noise. But these methods are dependent on position alignment and highly sensitive to distortion or style variations, which are typical sources of errors for hand print recognition. Therefore these methods are better suited for machine typed fixed fonts scripts.

2.6.2 Features derived from the Distribution of Points

The techniques under this category differ on the derivation of features.

Zoning: The frame containing the character is divided into several overlapping and non overlapping zones and the densities of points in these different regions form the features. This method is commonly used for on-line recognition.

Moments: The moments of black points about a chosen center e.g. the center of gravity of the character or a chosen coordinate system are used as features.

Characteristic loci For every white point in the background of the character, vertical and horizontal vectors are generated. The number of times the line segments intersected by these vectors are used as features.

Crossing and distances: Features are measured from the number of times line segments are traversed by vectors in specific directions or the distances of elements or line segments from a given boundary such as the frame which contains the character.

All the above techniques attempt, in one way or other, to relax, in a certain sense the rigidity of the techniques described in section 2.6.1. Tolerance to distortions and small stylistic variations are achieved because some features do take into account some sort of topological information.

2.6.3 Geometrical and Topological Features

This technique is based on the extraction of feature which describe the interesting geometry or topology of the drawing. These features may represent global and properties of the character. This is by far the most popular technique investigated by the researchers and some examples of the features extracted are:

1. Strokes and bays in different directions

2. End points, Intersections of line segments and loops

The main advantage of Geometrical and Topological features is their high tolerance to distortion and style variations compared to other techniques.

However, feature extraction is quite complex in these techniques. But once they are implemented, they can process characters at high speed independently. The present thesis work uses these features for the recognition of Gurmukhi characters.

CHAPTER 3

A SCHEME FOR GURMUKHI CHARACTER RECOGNITION

3.1 Introduction

Recognition of Gurmukhi script is a complex process, since it has a large character set containing vowels, Matras, consonants and their half forms. The large number of composite characters that are formed with them present a major problem in the computer aided recognition system. This chapter discusses a scheme for the recognition of primitive characters (excluding composite characters and modifiers) of Gurmukhi script for the printed and typewritten text. The approach taken here combines elements from several of the methods described in the previous chapter.

3.2 Gurmukhi Script

Gurmukhi script is a moderately complex pattern. Unlike the simple juxtaposition in Roman script, a word in Gurmukhi is composed of composite characters joined by a horizontal line at the top. The basic alphabet of Gurmukhi consists of 11 Matras, 40 consonants and 4 diacritical marks totaling 53 basic symbols Appendix B.1. In addition, for most of the consonants there are "half forms" or other derived forms, which are used in making conjuncts. Thus the total number of symbols in Gurmukhi is 90. Composite characters (which sometimes become syllables) are formed by a valid graphical composition of consonants, their half forms, diacritical marks and Matras and their number can go upto a few thousand depending upon the quality in production and purity in script desired.

In the following section, preprocessing of input, feature extraction and classification process are discussed.

3.3 Preprocessing

There are important interactions between the preprocessing of character images and the feature extraction process. Feature extraction or shape measurement can be misled if the images have little or no preprocessing. Important preprocessing steps are smoothing (noise filtering), thinning and segmenting the input text to locate characters.

3.3.1 Segmentation

Input text is scanned and digitized on by an Optical scanner to produce a digital image consisting of grey level values. This image is segmented into lines of text. A line of Gurmukhi text may contain composite characters and "Matras", so we may get extensions above and below the extremities of primitive characters. Therefore, a line of text can be divided vertically into three regions:

1. The core strip containing the flow of main characters
2. Upper Matra strip
3. Lower Matra strip

Upper and Lower modifier symbols and diacritical marks falling in upper and lower strips are separately framed and recognized. Core strip is bounded on the top by a horizontal line formed by the roofs of the primitive characters. Thus, it is easy to extract the core strip by counting the densities of "on" pixels in horizontal pixel row of a text line (a pixel is on means that it is a part of the picture and off means a part of the background). It will indicate a sharp fall and sharp peak at the two ends of the core strip. After the extraction of core strip, it is segmented into characters. Two methods of segmentation were found in literature.

1. Segmentation based on pitch estimation or character size (pitch is the distance of separation between two consecutive characters).

2. Segmentation based on contour analysis.

In this work, these methods were not tried and it is assumed that the training set contains only isolated primitive characters.

3.3.2 Smoothing

In smoothing, discontinuities at the connected pixels is removed by filling the gaps, isolated dots are removed and irregularities in character contour are removed. A new technique for smoothing suitable for character recognition was developed by Brown, Fay and Walker , which is as follows. A window of size (wk+1)x(2k+1) is moved over the input image of the character. If more than half of the pixels in the window centered on the pixel in question are black, then the centered pixel is marked black; otherwise it is marked as a white point. Choice of parameter k depends upon the size of the input character. It was found that value k = 1 is suitable for character size less than 50*50 pixels and k = 2 is suited for characters of greater size.

Sometimes, a linear artifact is present in the character image. This may result in the filling of a tree hole when above method is applied. To prevent this, the linear artifact must be removed. This is done by moving a 1x5 and a 5x1 matrix over the image in a single pass. A black point center pixel is changed to white if the two white points. The linear artifact is removed point by point by this method.

3.3.3 Thinning

Thinning is a process to transform the input image of thick character into an image composed of linearly connected points, i.e. a "Stick figure". It is also called skeletonisation. Use of skeletons has the advantage that a thin line representation of characters would be closer to the human conception of these patterns, and so they allow for a simpler structural analysis and more intuitive design of recognition algorithms. In addition, the reduction of an image to its essentials can eliminate some

contour distortions while retaining significant topological and geometric properties. In more concerts terms, the skeletons would be more amenable to extraction of critical features such as end points, junction points, and connections among the components. Naturally, for a skelonization algorithms to be really effective, it should ideally compress data, retain significant features of the pattern, eliminate local noise without introducing distortions of its own. To accomplish all that using the simplest and fastest algorithm is the challenge involved.

The immense magnitude of the challenge to overcome in designing a skeletonization package satisfying all the above demands can be seen from the large number and variety of algorithms that have been proposed and implemented [23]. For a comprehensive survey of thinning algorithms, it is widely stipulated that a skeletonization algorithm should not create noisy branches, and at the same time it should not cause excessive erosion of the pattern.

Figure 3.1 shows the results of thinning a pattern by two parallel algorithms. As can be seen, the second algorithm generally produces skeletons with more noise spurs; how-ever, it also preserves the short horizontal stroke (on the right side of the pattern '4') which is sometimes crucial for distinguishing patterns of this character from those of '9'. For such reasons, it is extremely difficult, if not impossible, to achieve an 'ideal' general purpose skeletonization algorithm, and the choice of algorithm can be application-dependent.

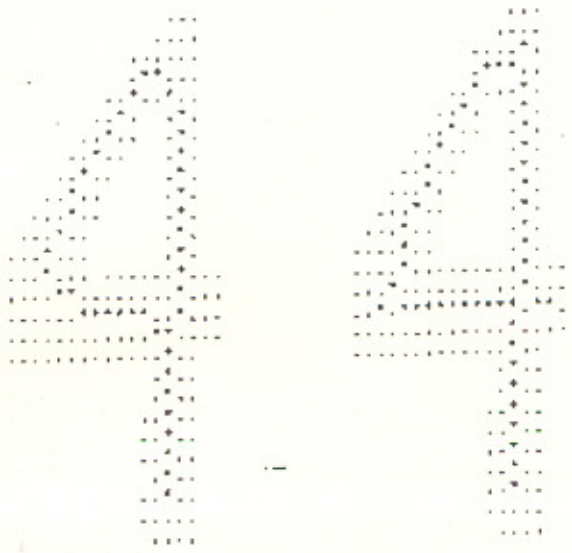


Figure 3.1 : Thinning of a pattern by two parallel algorithms

The above example also illustrates one of the inevitable problems encountered in skeletonization. When pixel removal conditions defined in terms of local configurations are applied uniformly throughout the pattern (as in the case with parallel algorithms), the skeleton obtained may not be always desirable. This is a natural consequence of the fact that local windows cannot convey global information about the structure of the pattern; for this reason, these algorithms cannot distinguish between different larger contexts, nor can they take into consideration the structural components that are significant for recognition.

As a way of resolving this problem, some recent research has been focused on improving the quality of skeletons. This often involves a two-step process: a skeleton is first obtained by some algorithm, after which some global characteristics of the pattern (especially the characteristics that are important for the recognition task) are determined and used to improve the skeleton in these areas. However, this would appear to be a rather indirect approach---feature extraction is applied to improve skeletonization rather than the other way round. If significant features can be extracted from the pattern, it would be more logical to use these features directly for recognition without having to apply them first to improve the skeleton, then extracting the same features again from the resulting skeleton.

If the line of reasoning were to be extended, it would seem logical for the thinning operation to be an integral part of a feature extraction process in which some of the features should be derived from the skeleton while others would be extracted from the original pattern. Some vectorization algorithms do produce skeletons and extract features from them at the same time as the entire feature extraction process. Here we propose more complicated approach which makes use of the advantages of skeletons while avoiding their shortcomings.

A successful implementation of this strategy would depend on a number of components. To begin with, it would require an insightful determination of the features that are important for the recognition task to be performed. After all, it can be, and has been argued that the choice of features is more important than the classification method in handwriting recognition. These features should be selected with the incorporation of human expertise and through extensive experimentation on their significance and rates of occurrence in large training sets. With respect to the features selected, it would be useful to perform an objective evaluation of skeletonization algorithms from the perspective of shape analysis in order to determine the features that can best be preserved in the process. Such features (including endpoints, for example) should then be extracted from the skeleton, while other features can be extracted from the

original pattern, from the contours or otherwise. This more 'intelligent' approach to feature extraction would result in a reliable set of features that is fundamental to handwriting recognition.

Our system uses the algorithm proposed by Zhang and Suen[21] . An example of a character before and after thinning is illustrated in figures 3.2 and 3.3.

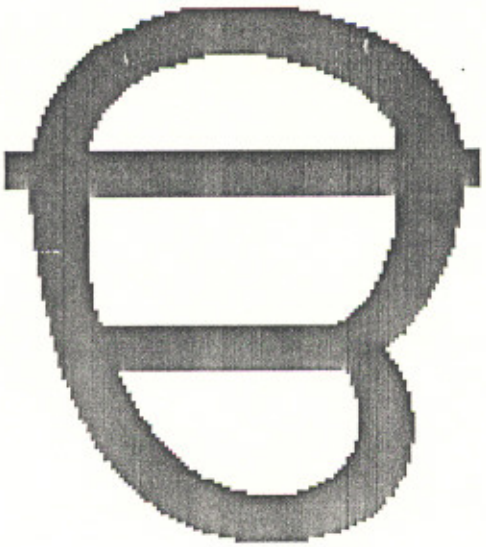


Figure 3.2 : Example of a character before thinning

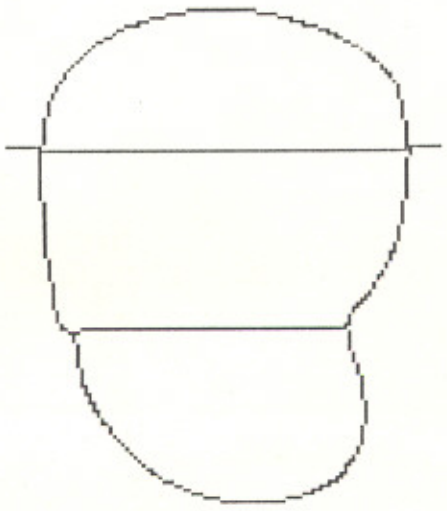


Figure 3.3 : Example of a character after thinning

3.4 Feature Definitions and Feature Extraction

An "Optimal set" of character features for a character recognition system should

1. fully capture the "essence" of the character set and
2. require a minimum of processing time for extraction.

Moreover, It should be able to provide sufficient discriminating power among the characters. The features chosen for our system achieve goal (a) by combining both local and global information into each feature. Goal (b) is achieved through both the choice of features and the use of a control structure which exploits easily obtainable information to limit and direct the extraction of higher level complex features. The features used in our system are not new; many of them have been used in various combinations in other character recognition systems. These features can be grouped into three levels of complexity or abstraction.

3.4.1 Low Level Features

The lowest level of features are very simply defined, yet provide much of the power of the proposed system due to the combination of local and global structural character information contained in them. These features are defined as follows:

1. **Terminal point (T):** A pixel on a segment which is adjacent to only one other pixel is said to be a "Terminal point". This point indicates the termination of a stroke segment.

2. **Tee point (Te):** A pixel about which there exist three distinct neighbours is called a Tee point. This point indicates the joining of two stroke segments.

3. **Intersection point (I):** The pixel about which there exist least four distinct neighbours is called an Intersection point. This point indicates crossing of two stroke segments.

The presence of these points comprise the local information, as illustrated in Figure 3.4. Once these simple features have been located, the relative location of these points with respect to the character as a whole are used to determine the subset of the character set which may coincide with the input character pattern.

3 terminal points

3 tee points

1 intersection point

Figure 3.4 : Illustration of A figure showing low level features

Skeleton of the character after thinning usually contains "imperfect features". Therefore prior to classifying the input pattern using the low level features, all points lying within a threshold distance of one another are "logically" merged, removing these points from the consideration as features (see Figure 3.5). For instance, if the two end points of a single stroke are within the threshold distance, they are merged to make a loop (see Figure 3.5). Similarly, Tee points occurring close to an Intersection point are merged in it. The threshold used for joining the points is directly related to the overall size of the input character pattern.,

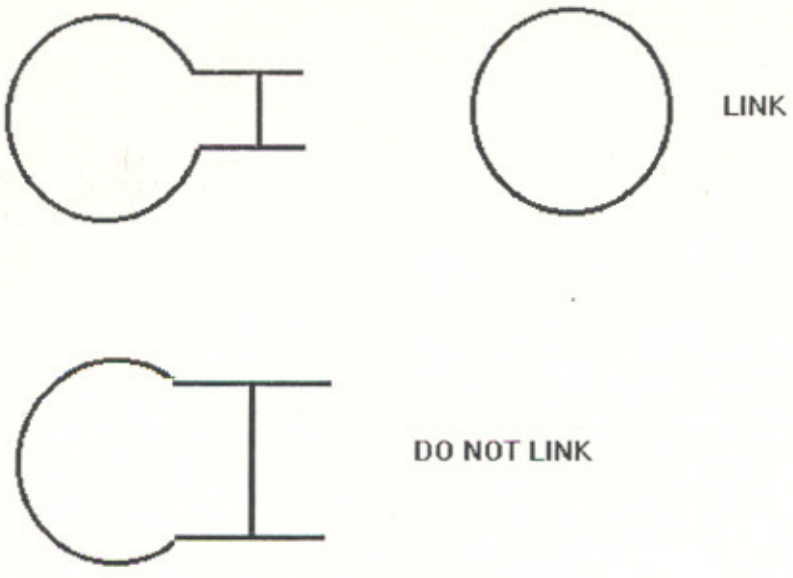


Figure 3.5 : Merging of Low Level Feature Points

The global information computed from the low level features is contained in the relative locations of feature points within the character. Each character is bounded by left, right, top and bottom extremes. The body of the character falls within this region and is divided into nine quadrants. A point can be classified depending upon the zone in which it falls. The classification of the quadrants or zones is shown in Figure 3.6. After a low level feature had been identified (as Terminal, Tee or Intersection point), its quadrant location is determined and this information is added to the feature descriptor. The low level classification process depends not only on the existence of certain features but also on the relative (quadrant) location of the existing features.

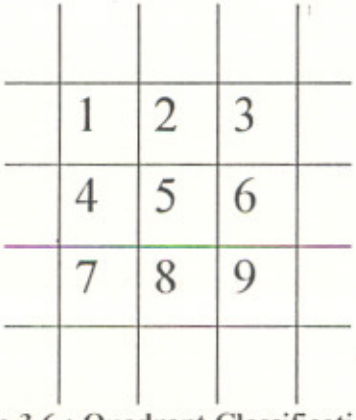
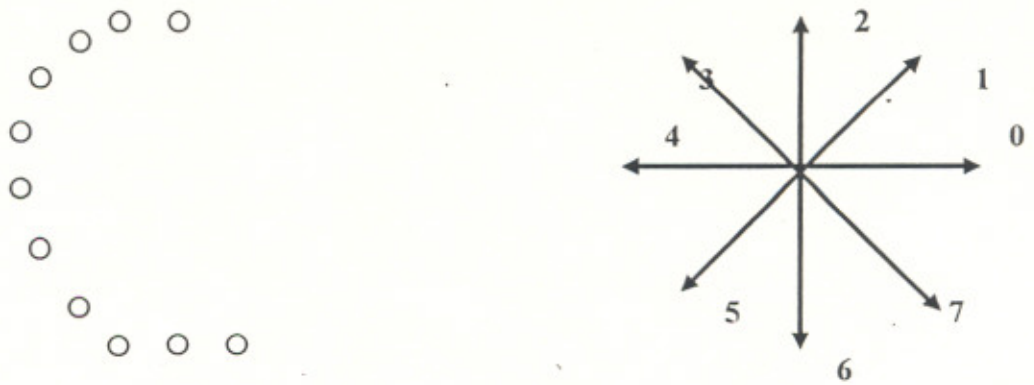


Figure 3.6 : Quadrant Classification

Computationally speaking, the feature extraction process is trivial. Although the features themselves are primitive, they serve remarkably well in the classification scheme. Given a set of possible input characters, even allowing substantial variation, using only Terminal, tee and Intersection points, the scheme inevitably results in a relatively small number of possible matches. These points capture some essential properties of characters which vary little between different renditions. For example, regardless of how a character 'ਣ' is printed or drawn, three Terminal points and one Tee point must invariably be present, the Tee point being constrained to the third quadrant.

3.4.2 High Level Stroke Properties

The most complex type of feature or stroke property used in our recognition system is stroke shape. A stroke segment is defined as a set of pixels bounded at both the ends by a Terminal point, Tee point or an Intersection point. A segment whose start point and end point are same is called a loop or hole. Two major stroke properties are used; orientation and shape (curvature). Orientation is used only for strokes whose shape is straight (straight line strokes). Possible orientations are 90, 45, -45 from the x axis with a lee-way of ± 12.5. In Gurmukhi, most of the straight strokes are either vertical or horizontal. A simple approach for describing the approximate shape of curved stroke is used. Since stroke shape and orientation may describe the entire input character in some cases (e.g. character 'ਤ' is a single stroke bounded by the Terminal points), this feature is mainly global, however, it should be noted that since the strokes are bounded by low level features which contain local information, there is some implicit local information contained in the stroke shape definition.



CHAIN CODE 4 5 5 5 6 7 7 7 0 0

Figure 3.7: Chain Code Representation and Directions used for 8-directional Chain Code.

Shape description is the most computationally expensive stroke feature or property in our system. The basis for the computation of the orientation or shape is the chain code representation of the stroke. Since in our system, strokes are initially represented as a set of (x,y) pairs, they are converted to the sequences of chain code values bounded by low level feature points (see Figure 3.7). Orientation of a stroke is trivial to compute from the chain code representation, given that the stroke is straight. If a curved stroke is expected, the algorithm verifies whether the stroke has the appropriate shape. To accomplish this, a very simple, easily computed and effective *shape descriptor* is defined, as follows, each (curved) stroke is divided into four segments of equal length. The chain code for each segment is determined and the average chain code value of each is computed. Thus, a four digit number is obtained which roughly describes the shape of stroke. The four digit number is then compared to a similar number associated with the idealised curve shape. If any of the four values differ by more than one from the corresponding value in the ideal code, the stroke shape is regarded as not having the expected shape. Since the computation of stroke shape is expensive, it is done on a need basis. That is, stroke shape is computed only if a candidate character match contains a stroke of a specific shape.

Every character in the current set can be classified using the stroke feature properties described thus far. Moreover, these features also allow for the recognition of slightly slanted renditions of the character.

3.5 Classification and Control Structure

The classification of characters in the system is accomplished in three stages. Firstly, low level features are used to reduce the number of possible character matches to a small candidate set using a table lookup procedure. Secondly, the candidate character matches remaining are used to parse the input character pattern into a set of stroke segments. Finally, for those characters for which the parsing stage is successful, individual stroke properties are compared with the input character pattern.

3.5.1 Initial Classification Stage

The goal of this stage of classification (stage I classification) is simple: reduce the number of candidate character matches to a small set. Character descriptors are stored in a table. A table look up is performed to get a set of characters matching the input character. The full description of the table is given in the Appendix B. Classification using low level features is straight forward. The presence or absence of a feature is used to decide the membership for the set of matching characters.

One important advantage of the table data structure is that alternate character definitions need only be included by augmenting the table. Thus, severely slanted or skewed character rendition can also be detected by including an appropriate character definition in the table.

3.5.2 Character Parsing

The completion of stage I results in a small set of possible character matches. Final classification involves comparing the properties of individual stroke in the input character to the properties expected in each for these candidate matches. Character parsing is a mean of organizing the input character into strokes in order to make these comparisons. Before parsing, input character is broken into segments and information regarding stroke segments is stored in a record of stroke vectors for the input

character. This information is comprised of number of strokes, type of each stroke, their expected shape and the location and type of their end points.

In parsing, we try to find the correspondence between each candidate character stroke definitions and stroke definitions of the input character. The stroke definitions for all the characters are stored in a table. The parsing algorithm uses the features to identify and label strokes according to a possible character match. Comparison of the input character pattern to each member of (small) candidate character set results in a different parsing of the input pattern. If the input pattern can be parsed according to the candidate character, a possible character match is generated, otherwise that candidate character cannot be a match and is removed from the candidate character set. Thus, parsing not only organizes the input character pattern for the final classification based on stroke shape, but also it results in further reduction of the candidate character set.

3.5.3 Stroke and Character Verification

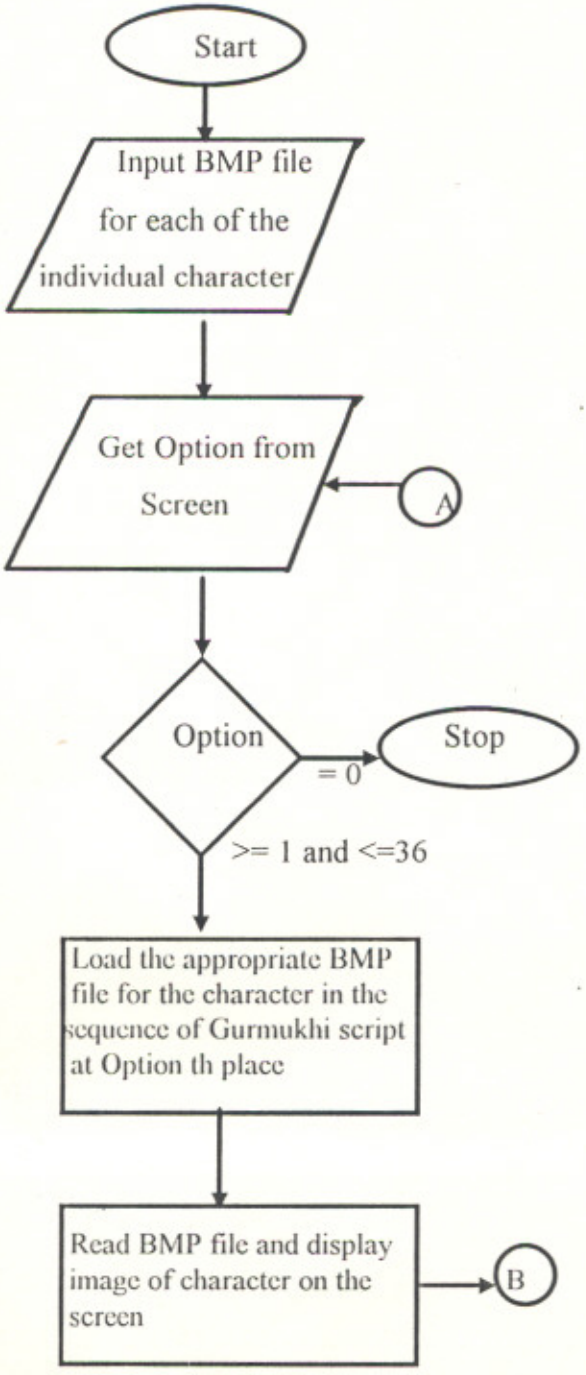
In final classification stage, each character in the candidate character set is compared with input character on a stroke by stroke basis, to determine which of the candidates, if any, sufficiently approximates the input pattern. Thus, final classification resolves any remaining ambiguity in the final identification of the input character pattern based on stroke shape properties.

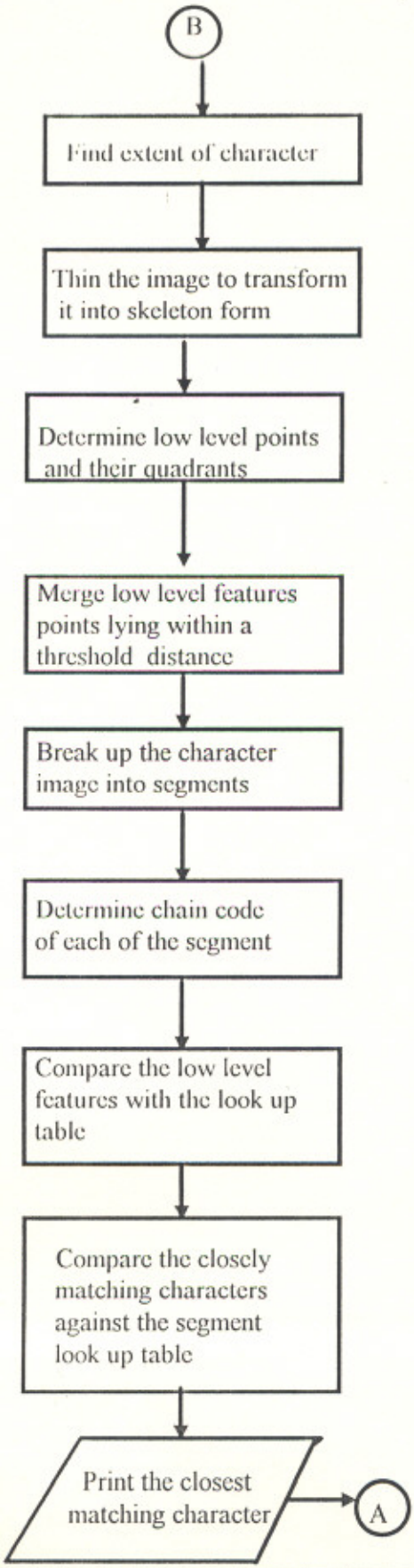
Final classification uses the correspondence between the input character and each character in the candidate set, which is generated by the character parsing stage. Properties of each stroke segment in the input character pattern is compared with properties of the corresponding segment of each character in the candidate character set. If all stroke segments properties match with the expected properties, then the character is regarded as a match to the input character. If any stroke segment fails to match the corresponding stroke in the input character, that character is removed from the candidate character set and a new candidate character is tried. This process is repeated until the character set is exhausted. Finally, the character having highest degree of match is identified as a possible match to the input character.

CHAPTER 4

PROGRAM IMPLEMENTATION

4.1 System Flowchart




B
Find extent of character
Thin the image to transform it into skeleton form
Determine low level points and their quadrants
Merge low level features points lying within a threshold distance
Break up the character image into segments
Determine chain code of each of the segment
Compare the low level features with the look up table
Compare the closely matching characters against the segment look up table
Print the closest matching character
A

4.2 Data Structures used

Structure	Description
struct segment { int npoints; struct coordinates coordinates[MAXPTS]; }structsegment	Structure for segment consists of information about the number .of 2D-points that make up the segment and also the coordinate of those points are stored. One character may be made up of number of such segments.
struct lowlevelpoints { int npoints; struct coordinates coordinates[9]; int quadrant[9]; }lpoint	Structure for low-level-points consist of information about the number of low level points. They can be either Terminal,Tee or Intersection points and also coordinate and quadrant of those points are stored.
struct neighbour_coordinates { int npoints; struct coordinates coordinates[8]; }nc	Structure of neighbourhood points consist of number of non-zero pixels in the neighbourhood of a particular pixel.
struct lowleveltable { int nterminal; int qterminal[9]; int ntee; int qtee[9]; int ninter; int qinter[9]; }table1;	This structure stores the expected low level features of each character. The number of terminal, tee and intersection points as well as their quadrants are stored in this structure
struct segmenttable { int nsegment; int chain_code[15]; }table2;	This structure stores the for each character the expected shape of the segments that make up a character. The shape of the segment is stored in the form of a 4 digit chain code.

4.3 Implementation of Algorithms

4.3.1 Thinning

Thinning is carried out by the following steps:

1. Unpack the whole raster image into a two-dimensional array. A black point pixel is designated by a 1. A white point pixel is designated by a 0.
2. Scan the whole image using a 3 by 3 window (see Figure 4.1). A border black point pixel is flagged for deleting (marked "D") if it satisfies following conditions:

   (a) $1 < N(P1) < 7$

   (b) $S(P1) = 1$

   (c) $P2 \times P4 \times P6 = 0$

   (d) $P4 \times P6 \times P8 = 0$

3. Delete all the flagged points (i.e. changed to 0).
4. Scan the whole image using a 3 by 3 window (see Figure 4.1). A contour black point pixel is flagged for deletion (marked "D") if it satisfies following conditions:

   (a) $1 < N(P1) < 7$

   (b) $S(P1) = 1$

   (c) $P2 \times P4 \times P8 = 0$

   (d) $P2 \times P6 \times P8 = 0$

5. Delete all the flagged points (i.e. changed to 0).

6. If no further point can be deleted, stop. Otherwise recycle back to step 1.

P9	P2	P3
P8	P1	P4
P7	P6	P5

Figure 4.1 : Neighbourhood arrangement used by the thinning algorithm

P1,P2,P3,....P9 are the labels given to pixel points in the 3x3 window (see Figure 4.1). N(P1) is the total number of black point (non zero) pixels in the 3x3 Neighbourhood of the pixel P1 (center pixel). S(P1) is the total number of 0 to 1 transitions in the ordered sequence of P2,P3,P4,...,P8,p9. For example, n(P1) = 4 and S(P1) = 3 in Figure 4.2. Note that a point flagged for deletion is not deleted until one pass is completed, since this flagged point is also used for determining the status of the neighboring points in the same pass.

0	0	1
1	P1	0
1	0	1

Figure 4.2 : Illustration of A Particular Case

4.3.2 Detection of LowLevel Points

The detection of low level points is carried out in the following steps:

1. Unpack the whole raster image into a two-dimensional array.

2. Scan the whole image using a 3 by 3 window (see Figure 4.1).

3. If $N(P_1) = 1$ and pixel P_1 is on, then P_1 is a Terminal point. Determine the quadrant in which P_1 falls by breaking the character extent into a 3 by 3 window as already illustrated in figure 3.5. Store the coordinates and quadrant of the terminal point.

4. Set NP to zero.

   If Pi is even and Pi is on (non-zero), then increment NP.

   If Pi is odd and Pi is on (non-zero) and P_{i-1} and P_{i+1} are off, then increment NP.

   If NP = 3 and pixel P is on, then P is a Tee point. Store the coordinates and quadrant of the P.

5. Set P_4 and P_8 equal to 1 only if both P_4 and P_8 are on and pixels adjacent to P_4 and P_8 on the line P_4P_8 are also on, else set P_4 and P_8 to zero.

   Set P_2 and P_6 equal to 1 only if both P_2 and P_6 are on and pixels adjacent to P_2 and P_6 on the line P_2P_6 are also on, else set P_2 and P_6 to zero.

   Set P_3 and P_7 equal to 1 only if both P_3 and P_7 are on and pixels adjacent to P_3 and P_7 on the line P_3P_7 are also on, else set P_3 and P_7 to zero.

   Set P_5 and P_9 equal to 1 only if both P_5 and P_9 are on and pixels adjacent to P_5 and P_9 on the line P_5P_9 are also on, else set P_5 and P_9 to zero.

   If $N(P_1) > 3$ and pixel P_1 is on, then P_1 is an Intersection point. Store the coordinates and quadrant of the Intersection point.

4.3.3 Breaking the character into segments

1. Initialise a 2 dimensional array visited to zero. This array keeps track of the relative coordinates in character image that have been visited.
2. For each of the low level points, determine the number of pixels on in their neighbourhood.
3. Visit the first point in the neighbourhood of first low level point. Mark it as visited. Store its coordinates in the structure of first segment.
4. Determine all the pixels on in the neighbourhood of current point which have not been visited. If there is more than one unvisited point in the neighbourhood, first visit the point which is at a multiple of 90 degree of the current point. Store the coordinates of neighbourhood points in the structure of the segment.
5. Check for the presence of a low level point in the neighbourhood of current point. If it is true then mark the end of segment and repeat steps 3 and 4 for next neighbourhood point , till all the neighbourhood and low level points are exhausted.

4.3.4 Calculating the chain code of the segments

1. Starting with first segment and the first point in the segment determine the direction of the next point. Store the value of the direction (figure 3.6) in the direction array. Set the next point as the current point. Determine the direction of the next point and store its value in the direction array. Repeat the process till all the points in the first segment are exhausted.
2. Break the direction array in to four equal parts. Find the average value of each part and combine them to form the chain code of the segment.
3. Repeat step 1 and 2 for each of the segment.

4.3.5 Recognizing the character

1. Compare the low level features of the character to be recognized against the stored low level features of all characters and store the closely matching characters in array match[]
2. Compare the high level features of the character to be recognized against the stored high level features of the characters stored in match[].
3. Display the character which matches closest with the character to be recognized in terms of high and low level features.

4.4 Coding

The program for visualization, *PATTERN.C*, has been written and compiled in Borland C (ver. 3.1). The program has about 1500 lines of code. Structured programming style has been used in coding by following the given below approach:

- GOTO statements have not been used anywhere to avoid spaghetti coding.
- The implementation of the algorithms has been properly designed in a top-down fashion and the path of execution flows in a straight line from top to bottom.
- For each module there is only one entry and one exit except in case of error.

Structured programming is also supported by C as discussed later on. Structured programming has led to the development of program and subprograms which are quite easy to understand and debug. They are also much easier to modify should the need arise.

Top-down modular approach has been used in program design. This approach consists of identifying the main function and then decomposing it into subroutines. Each of this subroutine is further broken down and this process continues until the broken down elements are in easy comprehensible form and straight forward to work with.

Another implementation detail that has made the program more meaningful and easier to understand is the choice of appropriate variable and constant names.

Modern style of function coding involving the use of function prototypes for function declarations and parameter list for function definitions, has been used. This has increased in error checking for bugs and ensured that the declarations and definitions agree.

The code is written in Borland C because of the following reasons :

- C is a general purpose programming language, which features economy of expression, modern control and data structures, and a rich set of operators.
- C supports structured programming by providing the fundamental flow-control constructions required for well structured programs; decision making (if); looping with the termination test at the top (while, for), or at the bottom (do); and selecting one of a set of possible cases (switch).
- C has provision for dynamic allocation of memory.
- Borland C provides a powerful graphics library of over 70 graphics functions ranging from high-level calls (like setviewport, bar3d and drawpoly) to bit oriented functions (like getimage and putimage). The graphics library supports numerous fill and line styles, and provides several text fonts.

4.5 Functions

As already stated, the program has been broken up into more than 25 functions. A brief writeup on some the main functions is as follow:

Name : find_ch_extent

Prototype : void find_ch_extent(int *x1, int *y1, int *x2, int *y2)

Description : Finds the extent of the rectangular body enclosing the character and stores the corner points of the rectangle at (x1,y1) and (x2,y2).

Name : getquadrant

Prototype : int getquadrant(int x1, int y1, int xextent, int yextent, int i, int j)

Description : Finds the quadrant of the low level point whose coordinates are(i,j) where (x1,y1) are the top most coordinates of the rectangle enclosing the character and xextent and yextent are the length and height of the rectangle.

Name : create_segments

Prototype : create_segments(struct lowlevelpoints tp, struct lowlevelpoints te,

struct lowlevelpoints inter, struct segment segment[],

int x1, int y1, int x2, int y2)

Description : Breaks the character into segments bounded by low level points. Tp,Te,Inter are the structers storing information about terminal points ,tee points and intersection points respectively.The information about the segments is stored in the array of segment structure. (x1,y1) and (x2, y2) are the coordinates of the corner points of the rectangle enclosing the character.

Name : visit

Prototype : int visit(struct lowlevelpoints tp, struct lowlevelpoints te,

struct lowlevelpoints inter, struct coordinates coordinates,

char visited[150][150], struct coordinates last_low_level_point,

struct segment *segment, int x1, int y1);

Description : Visits all the points of the segment and stores their coordinates in the structure segment. The two dimensional array visited keeps track of the points visited. Tp,Te,Inter are the structures storing information about terminal points ,tee points and intersection points respectively. (x1,y1) are the top most coordinates of the rectangle enclosing the character.

Name : get_neighbour

Prototype : int get_neighbour(struct lowlevelpoints tp, struct lowlevelpoints te,

struct lowlevelpoints inter, struct coordinates coordinates,

struct neighbour_coordinates *neighbour_coordinates,

char visited[150][150], int x1, int y1);

Description : Determines the unvisited non zero pixels in the neighbourhood of the point whose coordinates are stored in the structure coordinates. The information about such points is stored in the structure neighbour_coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

Name : get_neighbour_end_of_segment

Prototype : int get_neighbour_end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
struct lowlevelpoints inter, struct coordinates coordinates,
struct neighbour_coordinates *neighbour_coordinates);

Description : Determines the low level points in the neighbourhood of the point whose coordinates are stored in the structure coordinates. The information about such points is stored in the structure neighbour_cooRdinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

Name : end_of_segment

Prototype : int end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
struct lowlevelpoints inter, struct coordinates coordinates);

Description : Determines if the end of the segment is reached which is marked by the presence of a low level point in the neighbourhood of the point whose coordinates are stored in the structure coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

Name : create_chain_code_array

Prototype : int create_chain_code_array(struct segment segment, int chain_code[]);

Description : Assigns chain_code values to all the points of the segment and stores it in array chain_code.

Name : calculate_chain_code

Prototype : int calculate_chain_code(int chain_code[], int n);

Description : Calculates the four digit chain_code for the segment from the chain_code values assigned to all the points of the segment. N is the total number of points in the segment.

Name : load_bmp_file

Prototype : int load_bmp_file(char * fname,int xstart,int ystart,int xwidth, int yheight);

Description : Reads the image stored in the file fname. The image is stored in the BMP format and is displayed on the viewport with starting coordinates (xstart,ystart)and width xwidth and height yheight.

Name : draw_image

Prototype : int draw_image(int image_num);

Description : Draws the full size screen images of the various characters of the Gurmukhi script stored in the BMP format. Image number gives the number in the Gurmukhi script of the character to be displayed.

Name : thin_image

Prototype : int thin_image(int x1, int y1, int x2, int y2);

Description : Transforms the image of the character in to skeleton form. (x1,y1) and (x2, y2) are the coordinates of the corner points of the rectangle enclosing the character.

Name : recognise_character

Prototype : int recognise_character(int x1, int y1, int x2, int y2);

Description : Recognizes the image of the character enclosed in the rectangle with corner coordinates (x1,y1) and (x2,y2).

Name : arrange_neighbour_points

Prototype : void arrange_neighbour_points(struct coordinates current_point ,

struct neighbour_coordinates *neighbour_coordinates);

Description : Arranges the coordinates in the neighbourhood of the point specified by the structure current_point in such a manner so that the non diagonal neighbourhood coordinates come before the diagonal ones.

Name : remove_redundant_endpoints

Prototype : void remove_redundant_endpoints(struct lowlevelpoints *te,

struct lowlevelpoints *inter);

Description : Smoothens the skeleton form image of the character by removing the redundant low level points

---

Name : read_tables

Prototype : int read_tables();

Description : Reads the tables storing the information about low level and high level features of all the characters of the Gurmukhi script.

Name : compare_low_level_points

Prototype : int compare_low_level_points(struct lowlevelpoints tp, struct lowlevelpoints te,

struct lowlevelpoints inter, int *nmatch,

int match[]);

Description : Compares the low level features of the character against the low level features of all the characters stored in the table and closely matching characters are stored in the match array.Nmatch keeps record of the number of the matching characters.

Name : compare_segments

Prototype : int compare_segments(int match, int nseg, int chain_code_array[]);

Description : Compares the shape of the segments of the character by testing their chain code values against the chain code values of all the segments of all the characters and closest matching character is stored in the variable match. Nseg stores the number of the segments of the character.

Name : display_image

Prototype : int display_image(int image_num);

Description : Displays the image of the closest matching character on the screen.

CHAPTER 5

CONCLUSION

5.1 Remarks

The effectiveness of a new approach combining syntactic and deterministic approaches was observed. A hierarchical scheme for feature extraction and character classification has been designed and implemented. More than half of the characters in the Gurmukhi character set can be recognized with the help of low level and intermediate level features only. High level features are required only for the classification of a few similar characters. Moreover, high level features are computed on a need basis only when a very few character matches remain after the initial classification using the low level and intermediate level features. Features defined in this scheme were found sufficient to describe the entire Gurmukhi set. Though recognition of Gurmukhi numerals was not incorporated, it can be dealt in a similar way using above mentioned features. Importance of the preprocessing steps, especially, thinning, is also realised for the correct recognition of the characters. The scheme presented here can work even under modest computing resources.

5.2 Future Scope

The technique can be easily extended to cover hand printed Gurmukhi characters with different styles. This can be achieved by

1. increasing the size of the character set to allow for a large variety of stylized or ill-formed characters
2. incorporating contextual information into the classification based on cooperative examinations of surrounding characters.

The work can also be extended to cover Gurmukhi word parsing. Another area for future work is the development of suitable preprocessing techniques for the segmentation of words into characters and thinning of characters for the Gurmukhi text.

APPENDIX A

GURMUKHI CHARACTER SET

ੳ	ਅ	ੲ	ਸ	ਹ
ਕ	ਖ	ਗ	ਘ	ਙ
ਚ	ਛ	ਜ	ਝ	ਞ
ਟ	ਠ	ਡ	ਢ	ਣ
ਤ	ਥ	ਦ	ਧ	ਨ
ਪ	ਫ	ਬ	ਭ	ਮ
ਯ	ਰ	ਲ	ਵ	ੜ

Figure A.1: Gurmukhi Character Set

APPENDIX B

LOW LEVEL POINTS TABLE DESCRIPTION

Description of look up table used in the recognition process for low level features. The expected locations of stroke terminal points, tee points and intersection points used in defining the table are given for each character recognized by the algorithm.

Character	Terminal Points	Tee Points	Intersection Points
ੳ	1, 3	4, 6	1,3
ਅ	1, 3, 7, 8, 9	6, 5, 4	-
ੲ	1, 3, 9	5, 3, 1	-
ਸ	1, 3, 7, 9	6, 4, 1, 3	-
ਹ	1, 3, 5	3	-
ਕ	1, 3, 9	6, 6, 3	-
ਖ	1, 3, 9	9, 6, 4	-
ਗ	1, 3, 9	5, 3, 2	-
ਘ	1, 3, 2, 9	9, 8	-
ਙ	1, 3, 9	9, 9, 1	-
ਚ	1, 3, 4, 4	6, 4, 3	4
ਛ	1, 3	8, 4, 5, 3	-

ਜ	1, 3, 4, 4, 9, 7	6, 4, 3	4
ਝ	1, 3, 4, 7, 9	8, 6, 4, 3, 1	-
ਞ	1, 3, 6, 9	5, 1, 3, 1	-
ਟ	1, 3, 9	3	-
ਠ	1, 3	5, 2	-
ਡ	1, 3, 4	9, 6, 3	-
ਢ	1, 3, 4	8, 3	4
ਣ	1, 3, 6, 9	3, 2	-
ਤ	1, 3, 4, 7	6, 3	-
ਥ	1, 3, 9	9, 6, 4, 3, 1	-
ਦ	1, 3, 4, 4, 9	4, 3	4
ਧ	1, 3, 9	9, 3, 1	-
ਨ	1, 3, 8, 8	5, 2	-
ਪ	1, 3, 9	9	-
ਫ	1, 3,	8, 3	-
ਬ	1, 3, 9	9, 5, 6, 3, 1	-
ਭ	1, 3, 7	6, 6, 3	-
ਮ	1, 3, 7, 9	6	4

ਯ	1, 3, 9	6, 3, 1	-
ਰ	1, 3	6, 3	-
ਲ	1, 3, 8, 8	6, 4, 3, 1	-
ਵ	1, 3, 6, 9	5, 3	-
ਸ਼	1, 3, 4, 7, 7, 9	9, 7, 6, 3	-

APPENDIX C

SEGMENT TABLE DESCRIPTION

Description of look up table used in the recognition process for high level features. The four digit stroke numbers for all the segments of each of the character is shown next to the character.

Character	Number of segments	Segment Chain Code
ੳ	8	3222 5312 0000 1122 4444 3445 1000 4444
ਅ	7	4444 2221 6666 2345 6555 3234 6666
ੲ	7	5411 0012 3332 4444 1100 3444 2334
ਸ	8	4444 2222 6666 2222 6665 4444 0000 0000
ਹ	3	6531 1000 4444
ਕ	6	4313 2222 3566 1222 4444 0000
ਖ	6	4432 2222 6666 4444 2220 2234
ਗ	7	4512 2222 6666 1000 4444 4444 0344
ਘ	5	2221 4443 6666 4323 2222
ਙ	6	4312 2222 4776 1442 4444 0000
ਚ	8	4444 2222 6532 2332 4445 2112 4444 0000
ਛ	7	4432 2222 0134 0000 2002 1000 4444

ਜ	9	4444 2222 6666 2233 6666 5455 2022 1000 4444
ਝ	10	4444 7423 1123 4444 1122 3332 5444 0100 4444 4444
ਞ	9	4321 0000 6410 2222 1002 4444 1000 4444 0344
ਟ	3	4444 5441 0100
ਠ	4	5313 2222 0000 4444
ਡ	6	4412 1223 4444 2222 1000 4444
ਢ	6	3222 0353 2444 0012 4444 1000
ਣ	5	3322 4521 1342 0000 4444
ਤ	5	5543 4444 1222 4444 0000
ਥ	10	4432 2222 6666 4444 2222 2222 1000 4444 4444 2344
ਦ	7	2233 6411 0012 4455 2212 1100 4444
ਧ	7	4422 2222 6666 1000 4444 4444 2344
ਨ	5	5553 2222 3564 0000 4444
ਪ	3	4423 2221 6666
ਫ	4	3102 6666 4444 0100
ਬ	10	4431 2222 6666 3332 0000 2222 1000 4444 4444 2343
ਭ	6	5312 2222 5543 1222 4444 0000
ਮ	5	4444 2221 6666 2224 6665

ਯ	7	4532 2222 6666 4444 1000 4444 2344
ਰ	4	4522 2222 4444 0000
ਲ	8	5543 2222 4554 5642 2322 4444 0000 4444
ਵ	5	3102 6420 0000 4444 0000
ਸ	9	4444 1223 6444 4243 6665 4444 1122 4444 0000

SOURCE CODE

```
/*
*****************************************************************************
*                                                                           *
*       DEVELOPED BY   :   RENU DHIR                                        *
*        FILE NAME     :   PATTERN.C                                        *
*                                                                           *
*****************************************************************************
*/
#include <graphics.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define MAXPTS 250
struct coordinates
{
  int x;
  int y;
}coord;

struct lowlevelpoints
{
  int npoints;
  struct coordinates coordinates[9];
  int quadrant[9];
}lpoint;

struct lowleveltable
{
  int nterminal;
  int qterminal[9];
  int ntee;
  int qtee[9];
  int ninter;
  int qinter[9];
}table1;

struct segmenttable
{
  int nsegment;
  int chain_code[15];
}table2;

struct neighbour_coordinates
{
  int npoints;
  struct coordinates coordinates[8];
}nc;

struct segment
{
  int npoints;
  struct coordinates coordinates[MAXPTS];
}structsegment;

void find_ch_extent(int *x1, int *y1, int *x2, int *y2);
int getquadrant(int x1, int y1, int xextent, int yextent, int i, int j);
```

```
create_segments(struct lowlevelpoints tp, struct lowlevelpoints te,
              struct lowlevelpoints inter, struct segment segment[],
              int x1, int y1, int x2);
int visit(struct lowlevelpoints tp, struct lowlevelpoints te,
         struct lowlevelpoints inter, struct coordinates coordinates,
         char visited[150][150], struct coordinates last_low_level_point,
         struct segment *segment, int x1, int y1);
int get_neighbour(struct lowlevelpoints tp, struct lowlevelpoints te,
                struct lowlevelpoints inter, struct coordinates coordinates,
                struct neighbour_coordinates *neighbour_coordinates,
                char visited[150][150], int x1, int y1);
int get_neighbour_end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints
te,
                struct lowlevelpoints inter, struct coordinates coordinates,
                struct neighbour_coordinates *neighbour_coordinates);
int end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
                 struct lowlevelpoints inter, struct coordinates coordinates);
int check_for_end_points(struct coordinates last_low_level_point,
                       struct neighbour_coordinates *neighbour_coordinates,
                       struct segment *segment);
int add_end_point_to_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
         struct lowlevelpoints inter, struct coordinates coordinates,
         struct segment *segment);
void initialise(char visited[150][150]);
display_segments(struct segment segment[], int nseg, int x1, int x2);
int create_chain_code_array(struct segment segment, int chain_code[]);
int calculate_chain_code(int chain_code[], int n);
int load_bmp_file(char * fname,int XSTART,int YSTART,int XWIDTH,int YHEIGHT);
int draw_image(int image_num);
int thin_image(int x1, int y1, int x2, int y2);
int recognise_character(int x1, int y1, int x2, int y2);
void arrange_neighbour_points(struct coordinates coordinates ,
                            struct neighbour_coordinates *neighbour_coordinates);
void remove_redundant_endpoints(struct lowlevelpoints *te,
                             struct lowlevelpoints *inter);
int read_tables();
int compare_low_level_points(struct lowlevelpoints tp, struct lowlevelpoints te,
                       struct lowlevelpoints inter, int *nmatch,
                       int match[]);
int compare_segments(int match, int nseg, int chain_code_array[]);
int display_image(int image_num);

int nseg=0, freq;
struct lowleveltable lowleveltable[40];
struct segmenttable segmenttable[40];

int main(void)
{
   /* request auto detection */
   int errorcode, i, j, array[10],
       mp,sp,product1,product2,k, size, x1, x2, y1, y2, xextent, yextent,
       chain_code[100], chain_code_array[10], choice;
   struct lowlevelpoints tp, te, inter;
   struct segment segment[10];
   void *buffer;

   read_tables();
   draw_image(-2);
   gotoxy(19, 24);
   scanf("%d",&choice);
   while(choice)
   {
     if(choice < 0 || choice >36)choice = -1;
     else
     {
       draw_image(choice);
```

```
        x1=y1=x2=y1=0;
        find_ch_extent(&x1, &y1, &x2, &y2);
        nseg=0;
        recognise_character(x1, y1, x2, y2);
      }
      draw_image(-1);
      gotoxy(19, 24);
      scanf("%d",&choice);
    }
}

/*****************************************************************************************************************
* Finds the extent of the rectangular body enclosing the character and stores the corner points
* of the rectangle *at (x1,y1) and (x2,y2).
*
*****************************************************************************************************************/
void find_ch_extent(int *x1, int *y1, int *x2, int *y2)
{
  int i, j;

  for(*x1=getmaxx(),*y1=getmaxy(),*x2=*y2=0,i=0;i<getmaxx(); i++)
    for(j=0; j<getmaxy(); j++)
      if(getpixel(i, j))
      {
       if(i < *x1) *x1=i;
       if(j < *y1) *y1=j;
       if(i > *x2) *x2=i;
       if(j > *y2) *y2=j;
      }
}


/*****************************************************************************************************************
* Finds the quadrant of the low level point whose coordinates are(i,j) where (x1,y1) are the top most
*coordinates of the rectangle enclosing the character and xextent and yextent are the length and
*height  of the rectangle.
*
*****************************************************************************************************************/
int getquadrant(int x1, int y1, int xextent, int yextent, int i, int j)
{
  int quad;

  if((i-x1)<=xextent/3)quad=1;
  else
  {
    if((i-x1)<=(2*xextent/3))quad=2;
      else
    quad=3;
  }

  if((j-y1)<=yextent/3);
  else
  {
    if((j-y1)<=(2*yextent/3))quad+=3;
      else
    quad+=6;
  }
  return quad;
}

/*****************************************************************************************************************
```

Breaks the character into segments bounded by low level points. Tp,Te,Inter are the structers storing information about terminal points ,tee points and intersection points respectively.The information about the segments is stored in the array of segment structure. (x1,y1) and (x2, y2) are the coordinates of the corner points of the rectangle enclosing the character.

```
**********************************************************************************************************************/
create_segments(struct lowlevelpoints tp, struct lowlevelpoints te,
                struct lowlevelpoints inter, struct segment segment[],
                int x1, int y1, int x2)
{
   char visited[150][150];
   struct neighbour_coordinates neighbour_coordinates;
   int i, j, k;

   initialise(visited);
   for(i=0; i<inter.npoints; i++)
   {
     j=get_neighbour(tp, te, inter, inter.coordinates[i], &neighbour_coordinates,
visited, x1, y1);
     arrange_neighbour_points(inter.coordinates[i], &neighbour_coordinates);
     for(k=0; k<j; k++)
     {
       if(!visited[neighbour_coordinates.coordinates[k].x-
x1][neighbour_coordinates.coordinates[k].y-y1])
       visit(tp, te, inter, neighbour_coordinates.coordinates[k], visited,
            inter.coordinates[i], &segment[nseg++], x1, y1);
     }
   }
   for(i=0; i<te.npoints; i++)
   {
     j=get_neighbour(tp, te, inter, te.coordinates[i], &neighbour_coordinates,
visited, x1, y1);
     arrange_neighbour_points(te.coordinates[i], &neighbour_coordinates);
     for(k=0; k<j; k++)
     {
       if(!visited[neighbour_coordinates.coordinates[k].x-
x1][neighbour_coordinates.coordinates[k].y-y1])
       visit(tp, te, inter, neighbour_coordinates.coordinates[k], visited,
            te.coordinates[i], &segment[nseg++], x1, y1);
     }
   }
   for(i=0; i<tp.npoints; i++)
   {
     j=get_neighbour(tp, te, inter, tp.coordinates[i], &neighbour_coordinates,
visited, x1, y1);
     for(k=0; k<j; k++)
       visit(tp, te, inter, neighbour_coordinates.coordinates[k], visited,
            tp.coordinates[i], &segment[nseg++], x1, y1);
   }
   display_segments(segment, nseg, x1, x2);
}
```

```
/**********************************************************************************************************************
```

Visits all the points of the segment and stores their coordinates in the structure segment. The two dimensional array visited keeps track of the points visited. Tp,Te,Inter are the structures storing information about terminal points ,tee points and intersection points respectively. (x1,y1) are the top most coordinates of the rectangle enclosing the character.

```
**********************************************************************************************************************/
int visit(struct lowlevelpoints tp, struct lowlevelpoints te,
        struct lowlevelpoints inter, struct coordinates coordinates,
```

```
        char visited[150][150], struct coordinates last_low_level_point,
        struct segment *segment, int x1, int y1)
{
  struct neighbour_coordinates neighbour_coordinates;
  struct coordinates old_low_level_point=last_low_level_point;
  int i, j, k, next, a1, a2;

  (segment->npoints) = 1;
  segment->coordinates[0] = last_low_level_point;
  visited[coordinates.x-x1][coordinates.y-y1] = 1;
  putpixel(last_low_level_point.x, last_low_level_point.y, 3);
  delay(1520);
  freq=1000;
  (segment->npoints)++;
  segment->coordinates[segment->npoints-1] = coordinates;
  i=get_neighbour_end_of_segment(tp, te, inter, coordinates,
&neighbour_coordinates);
  k=0;
  if(i)
    k=check_for_end_points(old_low_level_point, &neighbour_coordinates, segment);
  j=get_neighbour(tp, te, inter, coordinates, &neighbour_coordinates, visited,
x1, y1);
  if(!j || k)
  {
    add_end_point_to_segment(tp, te, inter, coordinates, segment);
    return;
  }

  for(next=0,k=0; k<j; k++)
  {
    if(!(coordinates.y!=neighbour_coordinates.coordinates[k].y &&
        coordinates.x!=neighbour_coordinates.coordinates[k].x))

        {
        next=k;break;
        }
  }
  last_low_level_point = coordinates;
  coordinates = neighbour_coordinates.coordinates[next];
  visited[coordinates.x-x1][coordinates.y-y1] = 1;
  putpixel(coordinates.x, coordinates.y, 2);
  sound(freq);
  delay(100);
  nosound();
  freq+=20;
  putpixel(coordinates.x, coordinates.y, 1);
  segment->coordinates[segment->npoints] = coordinates;
  (segment->npoints)++;
  while(!end_of_segment(tp, te, inter, coordinates))
  {
    i=get_neighbour_end_of_segment(tp, te, inter, coordinates,
&neighbour_coordinates);
    k=0;
    if(i)
      k=check_for_end_points(old_low_level_point, &neighbour_coordinates,
segment);
    j=get_neighbour(tp, te, inter, coordinates, &neighbour_coordinates, visited,
x1, y1);
    if(!j || k)
    {
      add_end_point_to_segment(tp, te, inter, coordinates, segment);
      return;
    }
    next=0;
    if(j==2)
    {
```

```
      if(!(coordinates.y!=neighbour_coordinates.coordinates[1].y &&
        coordinates.x!=neighbour_coordinates.coordinates[1].x))

          next=1;
    }
    if(j==2 && next==0)
    {
      a1 = abs(neighbour_coordinates.coordinates[1].y-last_low_level_point.y);
      a2 = abs(neighbour_coordinates.coordinates[1].x-last_low_level_point.x);
      a1+=a2;
      if((coordinates.y!=neighbour_coordinates.coordinates[next].y) &&
        (coordinates.x!=neighbour_coordinates.coordinates[next].x) &&
        (a1 > 2))next=1;
    }
    if(j>2)continue;
    last_low_level_point = coordinates;
    coordinates = neighbour_coordinates.coordinates[next];
    visited[coordinates.x-x1][coordinates.y-y1] = 1;
  putpixel(coordinates.x, coordinates.y, 2);
  delay(20);
  sound(freq);
  delay(freq/25);
  nosound();
  freq+=20;
  putpixel(coordinates.x, coordinates.y, 1);
    segment->coordinates[segment->npoints] = coordinates;
    (segment->npoints)++;
  }
}
```

/**********

Determines the unvisited non zero pixels in the neighbourhood of the point whose coordinates are stored in the structure coordinates. The information about such points is stored in the structure neighbour_coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

**********/

```
int get_neighbour(struct lowlevelpoints tp, struct lowlevelpoints te,
                struct lowlevelpoints inter, struct coordinates coordinates,
                struct neighbour_coordinates *neighbour_coordinates,
                char visited[150][150], int x1, int y1)
{
  int i, j, xc = coordinates.x, yc = coordinates.y;

  neighbour_coordinates->npoints=0;
  for(i=-1; i<2; i++)
    for(j=-1; j<2; j++)
    {
      coordinates.x=xc+i;
      coordinates.y=yc+j;
      if((!(i==0 && j == 0)) && getpixel(xc+i,yc+j) &&
         !visited[coordinates.x-x1][coordinates.y-y1] &&
         !end_of_segment(tp, te, inter, coordinates))
      {
       (neighbour_coordinates->npoints)++;
       neighbour_coordinates->coordinates[neighbour_coordinates->npoints-1].x =
xc+i;
       neighbour_coordinates->coordinates[neighbour_coordinates->npoints-1].y =
yc+j;
      }
    }
  coordinates.x = xc; coordinates.y = yc;
  putpixel(xc, yc, 2);
```

```
  delay(20);

  putpixel(coordinates.x, coordinates.y, 1);
  return neighbour_coordinates->npoints;
}
```

/**********

Determines the low level points in the neighbourhood of the point whose coordinates are stored in the structure coordinates. The information about such points is stored in the structure neighbour_coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

****

```
int get_neighbour_end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints
te,
                struct lowlevelpoints inter, struct coordinates coordinates,
                struct neighbour_coordinates *neighbour_coordinates)
{
  int i, j, xc = coordinates.x, yc = coordinates.y;

  neighbour_coordinates->npoints=0;
  for(i=-1; i<2; i++)
    for(j=-1; j<2; j++)
    {
      coordinates.x=xc+i;
      coordinates.y=yc+j;
      if((!(i==0 && j == 0)) && getpixel(xc+i,yc+j) &&
         end_of_segment(tp, te, inter, coordinates))
      {
       (neighbour_coordinates->npoints)++;
       neighbour_coordinates->coordinates[neighbour_coordinates->npoints-1].x =
xc+i;
       neighbour_coordinates->coordinates[neighbour_coordinates->npoints-1].y =
yc+j;
      }
    }
  return neighbour_coordinates->npoints;
}
```

/**********

Determines if the end of the segment is reached which is marked by the presence of a low level point in the neighbourhood of the point whose coordinates are stored in the structure coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

**********/

```
int end_of_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
                struct lowlevelpoints inter, struct coordinates coordinates)
{
   int i;

   for(i=0; i<tp.npoints; i++)
     if(tp.coordinates[i].x==coordinates.x &&
tp.coordinates[i].y==coordinates.y)return 1;
   for(i=0; i<te.npoints; i++)
     if(te.coordinates[i].x==coordinates.x &&
te.coordinates[i].y==coordinates.y)return 1;
   for(i=0; i<inter.npoints; i++)
     if(inter.coordinates[i].x==coordinates.x &&
inter.coordinates[i].y==coordinates.y)return 1;
   return 0;
}
```

/**********

Determines if the end of the segment is reached which is marked by the presence of a low level point in the neighbourhood of the point whose coordinates are stored in the structure coordinates. Tp,Te,Inter and visited carry the same meaning as in above mentioned functions.

**********/

```
int check_for_end_points(struct coordinates last_low_level_point,
                         struct neighbour_coordinates *neighbour_coordinates,
                         struct segment *segment)
{
  int i;
  struct coordinates coordinates;

  if(neighbour_coordinates->npoints==1)
  {
    if((neighbour_coordinates->coordinates[0].x == last_low_level_point.x) &&
       (neighbour_coordinates->coordinates[0].y == last_low_level_point.y))
    {
       if(segment->npoints > 3)return 1;
       else return 0;
     }
     else
     {
       if(segment->npoints > 1)return 1;
       else return 0;
     }
  }
  else
  if(neighbour_coordinates->npoints==2)
  {
    if((neighbour_coordinates->coordinates[0].x == last_low_level_point.x) &&
       (neighbour_coordinates->coordinates[0].y == last_low_level_point.y))
    {
      coordinates = neighbour_coordinates->coordinates[0];
      neighbour_coordinates->coordinates[0] = neighbour_coordinates-
>coordinates[1];
      neighbour_coordinates->coordinates[1] = coordinates;
    }
    return 1;
  }
}


int add_end_point_to_segment(struct lowlevelpoints tp, struct lowlevelpoints te,
        struct lowlevelpoints inter, struct coordinates coordinates,
        struct segment *segment)
{
  int i, j, k, present;
  struct neighbour_coordinates neighbour_coordinates;

  j=get_neighbour_end_of_segment(tp, te, inter, coordinates,
&neighbour_coordinates);
  for(i=0; i<j; i++)
  {
    for(present=0,k=0; k<segment->npoints; k++)
      if(segment->coordinates[k].x==neighbour_coordinates.coordinates[i].x &&
       segment->coordinates[k].y==neighbour_coordinates.coordinates[i].y)
      {
       present=1;
       break;
      }
    if(!present)
    {
```

```
      segment->coordinates[segment->npoints] =
neighbour_coordinates.coordinates[i];
      (segment->npoints)++;
      break;
    }
  }
}


void initialise(char visited[150][150])
{
  int i, j;
  for(i=0; i<150; i++)
    for(j=0; j<150; j++)
      visited[i][j]=0;
}

display_segments(struct segment segment[], int nseg, int x1, int x2)
{
  int i, j;

  for(i=0; i<nseg; delay(50), i++)
    for(j=0; j<segment[i].npoints; j++)
      putpixel(x2+6-x1+segment[i].coordinates[j].x,segment[i].coordinates[j].y,
              (i%3)+1);
}

/*****************************************************************************************************************
Assigns chain_code values to all the points of the segment and stores it in array chain_code.
*****************************************************************************************************************/
create_chain_code_array(struct segment segment, int chain_code[])
{
  int i;


  for(i=0; i<segment.npoints-1; i++)
  {
    if(i==1 && (segment.npoints < 5))
    {
      chain_code[3] = chain_code[2] = chain_code[1] = chain_code[0];
      return;
    }
    if((segment.coordinates[i+1].x-segment.coordinates[i].x)==1)
    {
      if((segment.coordinates[i+1].y-
segment.coordinates[i].y)==1)chain_code[i]=7;
      if((segment.coordinates[i+1].y-
segment.coordinates[i].y)==0)chain_code[i]=0;
      if((segment.coordinates[i+1].y-segment.coordinates[i].y)==-
1)chain_code[i]=1;
    }
    if((segment.coordinates[i+1].x-segment.coordinates[i].x)==0)
    {
      if((segment.coordinates[i+1].y-
segment.coordinates[i].y)==1)chain_code[i]=6;
      if((segment.coordinates[i+1].y-segment.coordinates[i].y)==-
1)chain_code[i]=2;
    }
    if((segment.coordinates[i+1].x-segment.coordinates[i].x)==-1)
    {
      if((segment.coordinates[i+1].y-
segment.coordinates[i].y)==1)chain_code[i]=5;
      if((segment.coordinates[i+1].y-
segment.coordinates[i].y)==0)chain_code[i]=4;
      if((segment.coordinates[i+1].y-segment.coordinates[i].y)==-
1)chain_code[i]=3;
```

```
    }
  }
}
```

/**********

Calculates the four digit chain_code for the segment from the chain_code values assigned to all the points of the segment. N is the total number of points in the segment.

**********/

```
int calculate_chain_code(int chain_code[], int n)
{
  float div=n/4.0, ofrac=(n/4.0)-(n/4), s;
  float frac=ofrac;
  int i, j, c, k, is;

  for(j=c=s=0, i=k=1; i<=n; i++,j++)
  {
    if(i>=k*div)
    {
      if(!frac)s+=chain_code[j];
      else s+=chain_code[j]*frac;
      s/=div;
      is=s;
      if((s-is)>=.5)is++;
      c+=is*pow(10.0, 4-k);
      if(!frac)s=0.0;
      else s=chain_code[j]-chain_code[j]*frac;
      frac+=ofrac;
      if(frac>=1.0)frac=frac-1.0;
      k++;
    }
    else
    s += chain_code[j];
  }
  return c;
}
```

/**********

Reads the image stored in the file fname. The image is stored in the BMP format and is displayed on the viewport with starting coordinates (xstart,ystart)and width xwidth and height yheight.

**********/

```
load_bmp_file(char * fname,int XSTART,int YSTART,int XWIDTH,int YHEIGHT)
{

int fvalue,nbitmap;
int width,height;
register int i,j;
double scalex=0.0,scaley=0.0;

char ch;
FILE *f1;
int value1,value2;

      if((f1 = fopen(fname,"rb")) == NULL) {
            printf("file open error");
            getch();
            exit( 1 );
      }
```

```
for (i = 1; i <= 6; i++)
        fvalue = getw(f1);

nbitmap = fvalue;

for (i = 1; i <= 4; i++) fvalue = getw(f1);

width = (double)(fvalue);

fvalue = getw(f1);

fvalue = getw(f1);

height = (double)(fvalue);

for (i = 1; i <= 2; i++)   fvalue = getw(f1);

rewind(f1);

i = nbitmap;

while(i != 0) {
        fvalue = fgetc(f1);
        i--;
}

if(width > XWIDTH+1)
       scalex = (double)XWIDTH/(double)width;
if(height > YHEIGHT+1)
       scaley = (double)YHEIGHT/(double)height;

if( (scalex != 0.0) || (scaley != 0.0) ) {
       for(i = height; i > 0;i--){
              for(j = 0;j < width;j+=2) {
                     fvalue = fgetc(f1);
                     value1 = fvalue & 0x0F;
                     if(value1==0)value1=1;
                     if(value1==15)value1=0;

                     if(value1) putpixel((int)(j*scalex)+XSTART,
                            YSTART+(int)(i-height)*scaley,value1);
                     value2 = (fvalue & 0xF0) >> 4 ;
                     if(value2==0)value2=1;
                     if(value2==15)value2=0;
                     if (value2) putpixel((int)((j+1)*scalex)+XSTART,
                            YSTART+(int)(i-height)*scaley,value2);
              };
              value1 = value2;
       }
}
else {
       for(i = height; i > 0;i--){
              for(j = 0;j < width;j+=2) {
                     fvalue = fgetc(f1);
                     value1 = fvalue & 0x0F;
                     if(value1==0)value1=1;
                     if(value1==15)value1=0;

                     if (value1) putpixel(j+XSTART,
                            YSTART+i-height,value1);

                     value2 = (fvalue & 0xF0) >> 4 ;
                     if(value2==0)value2=1;
                     if(value2==15)value2=0;
                     if (value2) putpixel(j+1+XSTART,
```

```
                                        YSTART+i-height,value2);
                        );
                        value1 = value2;
                }
        }
        fclose(fl);

}
```

/**********

Draws the full size screen images of the various characters of the Gurmukhi script stored in the BMP format. Image number gives the number in the Gurmukhi script of the character to be displayed.

**********/

```
draw_image(int image_num)
{
char fname[40];
int XSTART,YSTART,XWIDTH,YHEIGHT;
int gd=CGA;
int gm=CGAC0,err;

        switch(image_num)
        {
          case -2:
            strcpy(fname,"menu.bmp");
            initgraph(&gd,&gm,"c:\\tc");
            setcolor(1);
            err = graphresult();
            if (err!=0)
            {
                  printf("%d,%s",err,grapherrormsg(err));
                  getch();
            }
            setbkcolor(BLACK);
            break;
          case -1:
            strcpy(fname,"menu.bmp");
            break;
          case 0: closegraph();return;
          case 1:
            strcpy(fname,"u.bmp");
            break;

          case 2:
            strcpy(fname,"a.bmp");
            break;
          case 3:
            strcpy(fname,"e.bmp");
            break;
          case 4:
            strcpy(fname,"s.bmp");
            break;
          case 5:
            strcpy(fname,"h.bmp");
            break;
          case 6:
            strcpy(fname,"c.bmp");
            break;
          case 7:
            strcpy(fname,"k.bmp");
            break;
          case 8:
            strcpy(fname,"g.bmp");
```

```
    break;

  case 9:
    strcpy(fname,"sg.bmp");
    break;
  case 10:
    strcpy(fname,"sl.bmp");
    break;
  case 11:
    strcpy(fname,"sc.bmp");
    break;
  case 12:
    strcpy(fname,"x.bmp");
    break;
  case 13:
    strcpy(fname,"j.bmp");
    break;
  case 14:
    strcpy(fname,"sj.bmp");
    break;
  case 15:
    strcpy(fname,"sm.bmp");
    break;
  case 16:
    strcpy(fname,"t.bmp");
    break;
  case 17:
    strcpy(fname,"st.bmp");
    break;
  case 18:
    strcpy(fname,"sd.bmp");
    break;
  case 19:
    strcpy(fname,"sq.bmp");
    break;
  case 20:
    strcpy(fname,"sn.bmp");
    break;
  case 21:
    strcpy(fname,"sv.bmp");
    break;
  case 22:
    strcpy(fname,"sw.bmp");
    break;
  case 23:
    strcpy(fname,"d.bmp");
    break;
  case 24:
    strcpy(fname,"sy.bmp");
    break;
  case 25:
    strcpy(fname,"n.bmp");
    break;
  case 26:
    strcpy(fname,"p.bmp");
    break;
  case 27:
    strcpy(fname,"f.bmp");
    break;
  case 28:
    strcpy(fname,"b.bmp");
    break;
  case 29:
    strcpy(fname,"sb.bmp");
    break;
  case 30:
```

```
            strcpy(fname,"m.bmp");
            break;
          case 31:
            strcpy(fname,"y.bmp");
            break;
          case 32:
            strcpy(fname,"r.bmp");
            break;
          case 33:
            strcpy(fname,"l.bmp");
            break;
          case 34:
            strcpy(fname,"v.bmp");
            break;
          case 35:
            strcpy(fname,"sr.bmp");
            break;
        }
        clearviewport();
        XWIDTH = getmaxx()-4;
        YHEIGHT = getmaxy()-3*14-8;
        YHEIGHT = getmaxy()-3*14-8;
        YHEIGHT = getmaxy()-2;
        YSTART = 2;
        XSTART = getmaxx()/2 - 80;
        XSTART = getmaxx()/2 - 80;

        load_bmp_file(fname,XSTART,YSTART,XWIDTH,YHEIGHT);
        if(image_num < 0)
          outtextxy(10, 170, "Enter your choice : Press 0 to quit");
}
```

Transforms the image of the character in to skeleton form. (x1,y1) and (x2, y2) are the coordinates of the corner points of the rectangle enclosing the character.

```
*****************************************************************************************************************/
int thin_image(int x1, int y1, int x2, int y2)
{
   /* request auto detection */
   int gdriver = CGA, gmode=CGAC0, errorcode, i, j, array[10],
       np,sp,product1,product2, product3, k, size;
   char visited[150][150], flag;
   void *buffer;

   setcolor(1);
   setfillstyle(1, 1);

   flag = 1;
   while(flag)
   {
     flag=0;
     initialise(visited);
     for(j=y1; j<=y2; j++)
       for(i=x1; i<=x2; i++)
       {
        array[1]=getpixel(i,j);
        array[2]=getpixel(i,j-1);
        array[3]=getpixel(i+1,j-1);
        array[4]=getpixel(i+1,j);
        array[5]=getpixel(i+1,j+1);
        array[6]=getpixel(i,j+1);
        array[7]=getpixel(i-1,j+1);
        array[8]=getpixel(i-1,j);
```

```
        array[9]=getpixel(i-1,j-1);

        for(sp=np=0,k=2; k<10; k++)
        {
          np+=array[k];
          if(k<9 && !array[k] && array[k+1])sp++;
          if(k==9 && !array[k] && array[2])sp++;
        }
        product1=array[2]*array[4]*array[6];
        product2=array[4]*array[6]*array[8];
        if(array[1] && np>1 && np<7 && sp==1 && !product1 && !product2)
          visited[i-x1][j-y1] = 1;
      }
     for(j=y1; j<=y2; j++)
       for(i=x1; i<=x2; i++)
        if(visited[i-x1][j-y1])
        {  putpixel(i, j, 0); delay(1); flag=1;}
     for(j=y1; j<=y2; j++)
       for(i=x1; i<=x2; i++)
       {
        array[1]=getpixel(i,j);
        array[2]=getpixel(i,j-1);
        array[3]=getpixel(i+1,j-1);
        array[4]=getpixel(i+1,j);
        array[5]=getpixel(i+1,j+1);
        array[6]=getpixel(i,j+1);
        array[7]=getpixel(i-1,j+1);
        array[8]=getpixel(i-1,j);
        array[9]=getpixel(i-1,j-1);
        for(sp=np=0,k=2; k<10; k++)
        {
          np+=array[k];
          if(k<9 && array[k]==0 && array[k+1]==1)sp++;
          if(k==9 && !array[k] && array[2])sp++;
        }
        product1=array[2]*array[4]*array[8];
        product2=array[2]*array[6]*array[8];
        if(array[1] && np>1 && np<7 && sp==1 && !product1 && !product2)
          visited[i-x1][j-y1] = 1;
      }
     for(j=y1; j<=y2; j++)
       for(i=x1; i<=x2; i++)
        if(visited[i-x1][j-y1])
        {  putpixel(i, j, 0); delay(1); flag=1;}
    }
   getch();
return 0;
}
```

/**********

Recognizes the image of the character enclosed in the rectangle with corner coordinates (x1,y1) and (x2,y2).

**********/

```
recognise_character(int x1, int y1, int x2, int y2)
{
   /* request auto detection */
   int errorcode, i, j, array[10], np,sp,product1,product2,k, size, xextent,
       yextent, chain_code[MAXPTS], chain_code_array[10], nmatch, match[10];
   struct lowlevelpoints tp, te, inter;
   struct segment segment[10];

//   getch();
```

```
    thin_image(x1, y1, x2, y2);
//    get_character_extent(&x1, &y1, &x2, &y2);
    tp.npoints=te.npoints=inter.npoints=0;
    xextent=x2-x1;
    yextent=y2-y1;
    for(j=y1-1; j<y2+1; j++)
      for(i=x1-1; i<x2+1; i++)
      {
        array[1]=getpixel(i+1,j+1);
        array[2]=getpixel(i+1,j);
        array[3]=getpixel(i+2,j);
        array[4]=getpixel(i+2,j+1);
        array[5]=getpixel(i+2,j+2);
        array[6]=getpixel(i+1,j+2);
        array[7]=getpixel(i,j+2);
        array[8]=getpixel(i,j+1);
        array[9]=getpixel(i,j);
        for(sp=np=0,k=2; k<10; k++)
        {
        np+=array[k];
        }
        if(np==1 && array[1])
        {
        tp.coordinates[tp.npoints].x=i+1;
        tp.coordinates[tp.npoints].y=j+1;
        tp.quadrant[tp.npoints++]=getquadrant(x1, y1, xextent, yextent, i+1, j+1);
        putpixel(i+1,j+1,2);

        putpixel(i+1,j+1,1);
        }
        for(np=0,k=2; k<10; k+=2)
        {
        np+=array[k];
        }
        for(k=3; k<10; k+=2)
        {
        int next;
        if(k==9)next=2; else next=k+1;
        if(array[k] && !array[k-1] && !array[next])
          np++;
        }
        if(np==3 && array[1])
        {
        te.coordinates[te.npoints].x=i+1;
        te.coordinates[te.npoints].y=j+1;
        te.quadrant[te.npoints++]=getquadrant(x1, y1, xextent, yextent, i+1, j+1);
        putpixel(i+1,j+1,3);

        putpixel(i+1,j+1,1);
        }
        array[4]=array[8]=min(array[4]*array[8]*getpixel(i-1, j+1),
                              array[4]*array[8]*getpixel(i+3, j+1));
        array[2]=array[6]=min(array[2]*array[6]*getpixel(i+1, j-1),
                              array[2]*array[6]*getpixel(i+1, j+3));
        array[3]=array[7]=min(array[3]*array[7]*getpixel(i-1, j+3),
                              array[3]*array[7]*getpixel(i+3, j-1));
        array[5]=array[9]=min(array[5]*array[9]*getpixel(i-1, j-1),
                              array[5]*array[9]*getpixel(i+3, j+3));
        for(sp=np=0,k=2; k<10; k++)
        {
        np+=array[k];
        }
        if(np>3 && array[1])
        {
        inter.coordinates[inter.npoints].x=i+1;
        inter.coordinates[inter.npoints].y=j+1;
```

```
        inter.quadrant[inter.npoints++]=getquadrant(x1, y1, xextent, yextent, i+1,
j+1);
        putpixel(i+1,j+1,2);

        putpixel(i+1,j+1,1);
        }
       product1=array[2]*array[4]*array[8];
       product2=array[2]*array[6]*array[8];
       if(np>1 && np<7 && sp==1 && !product1 && !product2)
       {
        putpixel(i+1,j, 0);
        putpixel(i+2,j, 0);
        putpixel(i+2,j+1,0);
        putpixel(i+2,j+2,0);
        putpixel(i+1,j+2,0);
        putpixel(i,j+2,0);
        putpixel(i,j+1,0);
        putpixel(i,j,0);
      }
     }
    remove_redundant_endpoints(&te, &inter);
    create_segments(tp, te, inter, segment, x1, y1, x2);

    for(i=0; i<nseg; i++)
    {
      create_chain_code_array(segment[i],  chain_code);
      chain_code_array[i]=calculate_chain_code(chain_code, segment[i].npoints-1>4
                                        segment[i].npoints-1 : 4);

    }

    i = compare_low_level_points(tp, te, inter, &nmatch, match);
    for(k=0, j=0; j<i; j++)
    {
      if(compare_segments(match[j], nseg, chain_code_array))
      {
        display_image(match[j]);

        k=1;
        break;
      }
    }
    gotoxy(2, 1);
    setcolor(1);
    printf("The matching character is");
    setcolor(2);
    gotoxy(2, 20);
    printf("Chain code:");
    for(i=0; i<nseg; i++)
    {
      if(i==5)gotoxy(13, 21);
      if(i==nseg-1 || i==4)
      printf("%04d ",chain_code_array[i]);
      else
      printf("%04d, ",chain_code_array[i]);
    }
    gotoxy(2, 22);
    if(tp.npoints)printf("Terminal point quadrant: ");
    for(i=0; i<tp.npoints; i++)
      printf("%1d ",tp.quadrant[i]);
    gotoxy(2, 23);
    if(te.npoints)printf("Tee point quadrant: ");
    for(i=0; i<te.npoints; i++)
      printf("%1d ",te.quadrant[i]);
    gotoxy(2, 24);
    if(inter.npoints)printf("Intersection point quadrant: ");
```

```
    for(i=0; i<inter.npoints; i++)
      printf("%ld ",inter.quadrant[i]);
    if(k==0)printf("\nNo matching character found");
    getch();
    return 0;
}
```

/**********

Arranges the coordinates in the neighbourhood of the point specified by the structure current_point in such a manner so that the non diagonal neighbourhood coordinates come before the diagonal ones.

**********/

```
void arrange_neighbour_points(struct coordinates coordinates ,
                              struct neighbour_coordinates *neighbour_coordinates)
{
  int i, j;
  struct coordinates temp_coordinates;

  for(i=0; i<neighbour_coordinates->npoints-1; i++)
    for(j=0; j<neighbour_coordinates->npoints-1; j++)
    {
      if(coordinates.y!=neighbour_coordinates->coordinates[j].y &&
       coordinates.x!=neighbour_coordinates->coordinates[j].x)
      {
       temp_coordinates = neighbour_coordinates->coordinates[j];
       neighbour_coordinates->coordinates[j] = neighbour_coordinates-
>coordinates[j+1];
        neighbour_coordinates->coordinates[j+1] = temp_coordinates;
       }
     }
}
```

/**********

Smoothens the skeleton form image of the character by removing the redundant low level points

**********/

```
void remove_redundant_endpoints(struct lowlevelpoints *te,
                                struct lowlevelpoints *inter)
{
  int i, j, k ;
  struct coordinates temp;


  for(i=0; i<te->npoints; i++)
    for(j=i+1; j<te->npoints; j++)
    {
      if(!getpixel(te->coordinates[i].x+1, te->coordinates[i].y)
        || !getpixel(te->coordinates[i].x-1, te->coordinates[i].y)
        || !getpixel(te->coordinates[i].x, te->coordinates[i].y-1)
        || !getpixel(te->coordinates[i].x, te->coordinates[i].y+1))
      {
        temp = te->coordinates[i];
        te->coordinates[i] = te->coordinates[j];
        te->coordinates[j] = temp;
        k = te->quadrant[i];
        te->quadrant[i] = te->quadrant[j];
        te->quadrant[j] = k;
      }
    }


  for(i=0; i<inter->npoints; i++)
    for(j=i+1; j<inter->npoints; j++)
```

```
    {
      if(!getpixel(inter->coordinates[i].x+1, inter->coordinates[i].y)
        || !getpixel(inter->coordinates[i].x-1, inter->coordinates[i].y)
        || !getpixel(inter->coordinates[i].x, inter->coordinates[i].y-1)
        || !getpixel(inter->coordinates[i].x, inter->coordinates[i].y+1))
      {
        temp = inter->coordinates[i];
        inter->coordinates[i] = inter->coordinates[j];
        inter->coordinates[j] = temp;
        k = inter->quadrant[i];
        inter->quadrant[i] = inter->quadrant[j];
        inter->quadrant[j] = k;
      }
    }


  for(i=0; i<te->npoints; i++)
    for(j=i+1; j<te->npoints; j++)
    {
      if((abs(te->coordinates[i].x - te->coordinates[j].x) <= 1) &&
        (abs(te->coordinates[i].y - te->coordinates[j].y) <= 1))
      {
        for(k=j; k<te->npoints-1; k++)
        {
          te->coordinates[k].x =   te->coordinates[k+1].x;
          te->coordinates[k].y =   te->coordinates[k+1].y;
          te->quadrant[k] = te->quadrant[k+1];
        }
        te->npoints--;
        j--;
       }
     }

  for(i=0; i<inter->npoints; i++)
    for(j=i+1; j<inter->npoints; j++)
    {
      if((abs(inter->coordinates[i].x - inter->coordinates[j].x) <= 1) &&
        (abs(inter->coordinates[i].y - inter->coordinates[j].y) <= 1))
      {
        for(k=j; k<inter->npoints-1; k++)
        {
          inter->coordinates[k].x =   inter->coordinates[k+1].x;
          inter->coordinates[k].y =   inter->coordinates[k+1].y;
          inter->quadrant[k] = inter->quadrant[k+1];
        }
        inter->npoints--;
        j--;
       }
     }
}

/*****************************************************************************************************************
Reads the tables storing the information about low level and high level features of all the
characters of the Gurmukhi script.
*****************************************************************************************************************/
int read_tables()
{
  FILE *stream;
  int i, j, k, np;

  if((stream=fopen("pattern.tbl","r"))==NULL)
  {
    printf("Unable to read pattern.tbl");
```

```
    exit(0);
  }

  for(i=1; i<=35; i++)
    for(j=0; j<3; j++)
    {
      fscanf(stream,"%d",&np);
      for(k=0; k<np; k++)
       switch(j)
       {
         case 0 : lowleveltable[i].nterminal = np ;
                  fscanf(stream,"%d",&lowleveltable[i].qterminal[k]);
                  break;
         case 1 : lowleveltable[i].ntee = np ;
                  fscanf(stream,"%d",&lowleveltable[i].qtee[k]);
                  break;
         case 2 : lowleveltable[i].ninter = np ;
                  fscanf(stream,"%d",&lowleveltable[i].qinter[k]);
                  break;
       }
    }
  fclose(stream);

  if((stream=fopen("pattern.tb2","r"))==NULL)
  {
    printf("Unable to read pattern.tb2");
    exit(0);
  }

  for(i=1; i<=35; i++)
    {
      fscanf(stream,"%d",&segmenttable[i].nsegment);
      for(k=0; k<segmenttable[i].nsegment; k++)
        fscanf(stream,"%d",&segmenttable[i].chain_code[k]);
    }
  fclose(stream);
}
```

/**********

Compares the low level features of the character against the low level features of all the characters stored in the table and closely matching characters are stored in the match array.Nmatch keeps record of the number of the matching characters.

**********/

```
int compare_low_level_points(struct lowlevelpoints tp, struct lowlevelpoints te,
                     struct lowlevelpoints inter, int *nmatch,
                     int match[])
{
  int i, j, k, l=0, found;
  struct lowleveltable table1;

  for(k=0; k<35; k++)
  {
    if((lowleveltable[k].nterminal != tp.npoints) ||
    (lowleveltable[k].ntee != te.npoints) || (lowleveltable[k].ninter !=
inter.npoints))
      continue;
    for(i=0; i<lowleveltable[k].nterminal; i++)
    {
      for(found=0,j=0; j<tp.npoints; j++)
      {
       if(lowleveltable[k].qterminal[i] == tp.quadrant[j])
       {
         found=1;
         break;
       }
```

```
        }
        if(!found)break;
      }
      if(!found)continue;
      for(i=0; i<lowleveltable[k].ntee; i++)
      {
        for(found=0,j=0; j<te.npoints; j++)
        {
         if(lowleveltable[k].qtee[i] == te.quadrant[j])
         {
           found=1;
           break;
         }
        }
        if(!found)break;
      }
      if(!found)continue;
      for(i=0; i<lowleveltable[k].ninter; i++)
      {
        for(found=0,j=0; j<inter.npoints; j++)
        {
         if(lowleveltable[k].qinter[i] == inter.quadrant[j])
         {
           found=1;
           break;
         }
        }
        if(!found)break;
      }
      if(!found)continue;
      match[l++] = k;
      *nmatch = l;
    }
    return l;
}
```

/**********

Compares the shape of the segments of the character by testing their chain code values against the chain code values of all the segments of all the characters and closest matching character is stored in the variable match. Nseg stores the number of the segments of the character.

**********/

```
int compare_segments(int match, int nseg, int chain_code_array[])
{
  int i, j, found;

  if(segmenttable[match].nsegment != nseg)return 0;
  for(i=0; i<segmenttable[match].nsegment; i++)
  {
    for(found=0,j=0; j<nseg; j++)
    {
      if(segmenttable[match].chain_code[i] == chain_code_array[j])
      {
       found=1;
       break;
      }
    }
    if(!found)return 0;
  }
  return 1;
}
```

```
int display_image(int image_num)
{
char fname[40];
int XSTART,YSTART,XWIDTH,YHEIGHT;
int gd=CGA;
int gm=CGAC0,err;

       switch(image_num)
       {
         case -2:
           strcpy(fname,"menu.bmp");
           initgraph(&gd,&gm,"c:\\tc");
           err = graphresult();
           if (err!=0)
           {
                 printf("%d,%s",err,grapherrormsg(err));
                 getch();
           }
           setbkcolor(BLACK);
           break;
         case -1:
           strcpy(fname,"menu.bmp");
           break;
         case 0: closegraph();return;
         case 1:
           strcpy(fname,"cu.bmp");
           break;

         case 2:
           strcpy(fname,"ca.bmp");
           break;
         case 3:
           strcpy(fname,"ce.bmp");
           break;
         case 4:
           strcpy(fname,"cs.bmp");
           break;
         case 5:
           strcpy(fname,"ch.bmp");
           break;
         case 6:
           strcpy(fname,"cc.bmp");
           break;
         case 7:
           strcpy(fname,"ck.bmp");
           break;
         case 8:
           strcpy(fname,"cg.bmp");
           break;

         case 9:
           strcpy(fname,"csg.bmp");
           break;
         case 10:
           strcpy(fname,"csl.bmp");
           break;
         case 11:
           strcpy(fname,"csc.bmp");
           break;
         case 12:
           strcpy(fname,"cx.bmp");
           break;
         case 13:
           strcpy(fname,"cj.bmp");
           break;
         case 14:
```

```
      strcpy(fname,"csj.bmp");
      break;
    case 15:
      strcpy(fname,"csm.bmp");
      break;
    case 16:
      strcpy(fname,"ct.bmp");
      break;
    case 17:
      strcpy(fname,"cst.bmp");
      break;
    case 18:
      strcpy(fname,"csd.bmp");
      break;
    case 19:
      strcpy(fname,"csq.bmp");
      break;
    case 20:
      strcpy(fname,"csn.bmp");
      break;
    case 21:
      strcpy(fname,"csv.bmp");
      break;
    case 22:
      strcpy(fname,"csw.bmp");
      break;
    case 23:
      strcpy(fname,"cd.bmp");
      break;
    case 24:
      strcpy(fname,"csy.bmp");
      break;
    case 25:
      strcpy(fname,"cn.bmp");
      break;
    case 26:
      strcpy(fname,"cp.bmp");
      break;
    case 27:
      strcpy(fname,"cf.bmp");
      break;
    case 28:
      strcpy(fname,"cb.bmp");
      break;
    case 29:
      strcpy(fname,"csb.bmp");
      break;
    case 30:
      strcpy(fname,"cm.bmp");
      break;
    case 31:
      strcpy(fname,"cy.bmp");
      break;
    case 32:
      strcpy(fname,"cr.bmp");
      break;
    case 33:
      strcpy(fname,"cl.bmp");
      break;
    case 34:
      strcpy(fname,"cv.bmp");
      break;
    case 35:
      strcpy(fname,"csr.bmp");
      break;
  }
```

```
        clearviewport();
        XWIDTH = getmaxx()-4;
        YHEIGHT = getmaxy()-3*14-8;
        YHEIGHT = getmaxy()-3*14-8;
        YHEIGHT = getmaxy()-2;
        XSTART = 2;
        YSTART = getmaxy()-20;
        YSTART = getmaxy()-2;

        load_bmp_file(fname,XSTART,YSTART,XWIDTH,YHEIGHT);
        if(image_num < 0)
          outtextxy(10, 170, "Enter your choice : Press 0 to quit");
}
```

BIBLIOGRAPHY

1. Andrews. Multi-dimensional rotations in feature selection. *IEEE Transactions on Computers*, 20:1045-1051, 1971.
2. Glenn Baptista and K.M. Kulkarni. A high accuracy algorithm for recognition of handwriten numerals. *Pattern Recognition*, 21(4):287-291, 1988.
3. Radmilo M. Bozinovic and S.N. Srihari. Off-line cursive script word recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11:68-83, 1989.
4. Itoh C. Kimpan and K. Kawanishi. The classification of printed thai character recognition using k-l expansion. *IEEE Proceedings*, pages 257-264, 1987.
5. Toru Wakahara. Toward robust handwritten character recognition. *Pattern Recognition*, 345-354, 26(4):1993.
6. Elliman and I.T.Lancaster. A review of segmentation and contextual analysis techniques for text recognition. *Pattern Recognition*, 23(3):337-346, 1990.
7. J.Huang and M. Chung. Seaparating similar complex chinese characters by walsh transform. *Pattern Recognition*, 20(4):425-428, 1987.
8. David D.Kerrick and Alan C.Bovik. Microprocessor-based recognition of handprinted characters from a tablet input. *Pattern Recognition*, 21(5):525-537, 1988.
9. Theo Pavlidis. Recognition of printed text under realistic conditions. *Pattern Recognition*, 317-326, 26(4):1993.
10. M.Michael and W.C.Linn. Experimental study of information measure and inter intra class distance ratios on feature detection and ordering. *IEEE Transactions on System, Man and Cybernetics*, SMC3:172-181, 1973.
11. M.J.Minneman. Handwritten character recognition employing topology, cross correlation and decision theory. *IEEE Transactions on System, Man and Cybernetics*, SMC2:86-96, 1966.
12. E.Persarn and F.S.Fu. Shape discrimination using fourier descriptors. *IEEE Transactions on System, Man and Cybernetics*, SMC7:170-179, 1977.
13. N.M.Herbst R.Bakis and G. Nagy. An experimental study of machine recognition of hand-printed numerals. *IEEE Transactions on System, Man and Cybernetics*, SMC4:119-132, 1968.
14. T.H.Fay R.M.Brown and C.L.Walker. Handprinted symbol recognition system. *Pattern Recognition*, 21(2):91-118, 1988.
15. G.Gagnuix, S.Wendling and G.Staman. A set of invariants within the power spectrum of unitary transformations. *IEEE Transactions on Computers*, 27:1213-1216, 1978.
16. I.Sekita and K.Toraichi. Feature extraction by handwritten japanese characters by spline functions for relaxation matching. *Pattern Recognition*, 21(1):9-17, 1988.
17. I.K.Sethi and B. Chatterjee. Machine recognition of constrained hand printed Devnagri characters. *Pattern Recognition*, 11(1):69-75, 1977.

18. Theo Pavlidis Simon Kahan and Henry S.Baird. On the recognition of printed characters of any font and size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9:274-287, 1987.

19. R.M.K.Sinha. A knowledge based script reader. *Proceeding of 7th international conference on Pattern Recognition*, ages 763-765 , 1984.

20. R.M.K.Sinha. Role of context in Devnagri script recognition. *Journal of Institution of Electronic and Telecommunication Engineers*, pages 86-91, 1987.

21. T.Y.Zhang and C.Y.Suen. A fast parallel algorithm for thinning digital patterns. *Communications of ACM*, 27(3):236-239, 1984.

22. Diehl, S. andH. Eglowstein(1991). Tame the paper tiger. *Byte magazine*, April 1991, 220-238.

23. Lam, L., S. W. Lee and C. Y. Suen (1992). Thinning methodologies: a comprehensive survey. *IEEE Trans. Pattern Anal. Machine Intell.* 14(9), 869-885