DESIGN OF AN ON-LINE HANDWRITING RECOGNITION SYSTEM

by

Po-Yang Lu

Memorandum No. UCB/ERL M87/22

15 April 1987

DESIGN OF AN ON-LINE HANDWRITING RECOGNITION SYSTEM

by

Po-Yang Lu

Memorandum No. UCB/ERL M87/22

27 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DESIGN OF AN ON-LINE HANDWRITING RECOGNITION SYSTEM

by

Po-Yang Lu

Memorandum No. UCB/ERL M87/22

27 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Design of an On-Line Handwriting Recognition System

Ph.D. Po-Yang Lu E.E.C.S.

Abstract

Many computer applications are awkward to use because of keyboard limitations. A powerful on-line handwriting recognition system can solve many of these problems and thus significantly improve the user interface of a personal computer.

New algorithms for both discrete symbol recognition and cursive script recognition have been successfully developed. After given some training samples, the discrete symbol recognition algorithm can recognize virtually any symbol. The recognition rate can be nearly 100% if additional adaptive training is well done.

Based on the training of 26 letters, the cursive script recognition algorithm can recognize scripts of virtually any word. Its recognition rate can also be near 100% with appropriate adaptive training.

These algorithms, together with interfacing utilities, have been implemented on IBM PC for application exploration. A keyboard/mouse emulator has also been developed to allow a user to directly write and draw with existing software.

These algorithms have also been implemented on a dedicated processor based speech recognition system. This implementation not only allows large dictionaries in real time, but also reduces cost since one piece of hardware can be used for both speech and handwriting recognition.

Robert W. Brodersen

Committee Chairman

Table of Contents

Acknowledgements

I would like to express my sincere appreciation to my research adviser Professor Robert Brodersen for giving me this interesting project and his constant guidance and support. I would also like to acknowledge the invaluable help from my colleagues, especially Hy Murveit, Meni Lowy, David Mintz, and Robert Kavaler. Their outstanding achievement in building the speech recognition system has essential influence to this project. Working in this group has been the most rewarding experience.

Many thanks should also go to my colleagues in Communication Intelligence Corporation. I will never forget the time we fought together for the Handwriter.

Peter Ruetz has made numerous corrections and suggestions to the writing of the thesis. My debt to him is gratefully acknowledged. Finally, this thesis is dedicated to my family, especially to my wife Erh-Ning, for the love, care, and encouragement they have given me.

CHAPTER 1

INTRODUCTION

1. Motivation

Personal computers (PC) are now becoming an indispensable tool for many people in modern society. Because of their powerful processing and storage capability, the PC can take care of much tedious work and significantly increase our productivity. With the rapid breakthrough of hardware and software technology, in the foreseeable future, PC's will be in our briefcases and help us with more advanced and sophisticated tasks.

In order to make the PC a tool we can use intensively, its user interface must be natural and easy. Recently, a lot of effort has been put in to make a computer more "friendly". However, its friendliness seems severely impaired by the limitations of input devices.

The most serious limitation of current input devices is that the man-computer communication symbols are confined by the keyboard. If a symbol is not on the keyboard, the most popular input method is to assign a special keystroke string to it. For example, in text formatting software *TROFF*, the user has to type "\(*S" for Σ, "\(*a" for α, "\(mu" for $\times$, "\(*b" for β, "\(->" for $\rightarrow$, and "\(if" for ∞. Because these strings are generally hard to remember, this method obviously makes the user interface of many applications very awkward. For example, to obtain

$$\Sigma(\alpha \times \beta) \rightarrow \infty,$$

the user has to type

*\(*S(\(*a\(mu\(*b) \(-> \(if.*

This problem has an even more severe impact for people who use ideographical languages. One good example is Chinese. In Chinese, there are more than 6000 characters used daily. To input a character into a PC, the user has to decompose it into parts and type the assigned key of each part. Fig. 1.1 shows a flow chart of a decomposition and entry method. From it, it can be imagined how much training and practice are needed. Since data entry itself is so troublesome,

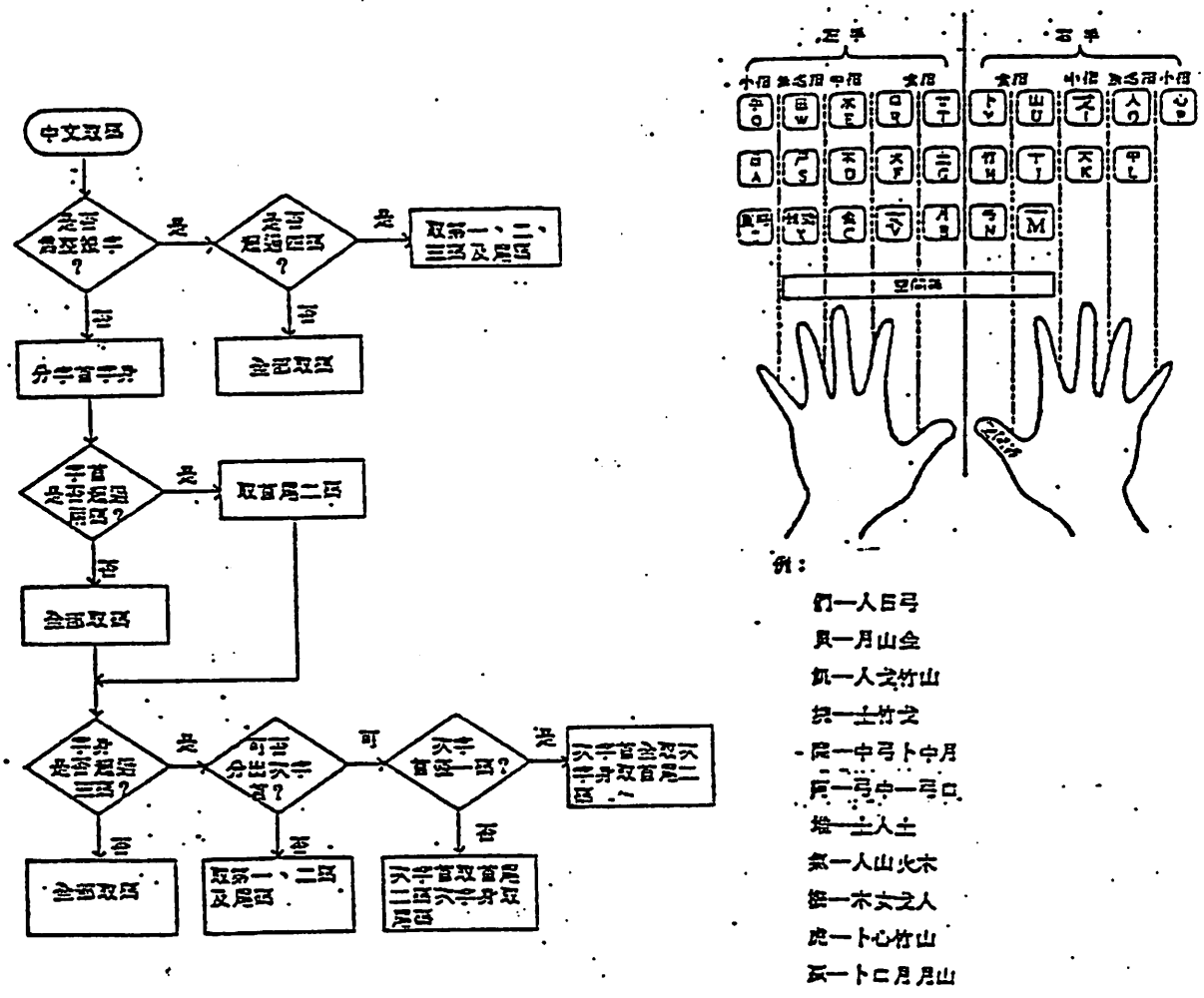


Figure 1.1 A chinese data entry method

how can the PC be used for creative tasks ?

Besides the keyboard symbol limitation, the mouse also has some problems. First, the user has to push, pull, and even lift the mouse to move the curser to a specific point on screen. Second, the mouse can't be used for freehand drawing.

To overcome these limitations, it seems that the future PC should be designed as shown in Fig. 1.2. The user can interface with it in a similar way as he is using a *notebook*. To allow the user to be able to control the PC through a *stylus* (pen), three new subsystems are needed. The first is a display which can be laid flat. The second is a tablet which can sense the stylus movement. The third is an *on-line handwriting recognition (OHR) system* which can translate the stylus movement to either object selection or data entry.

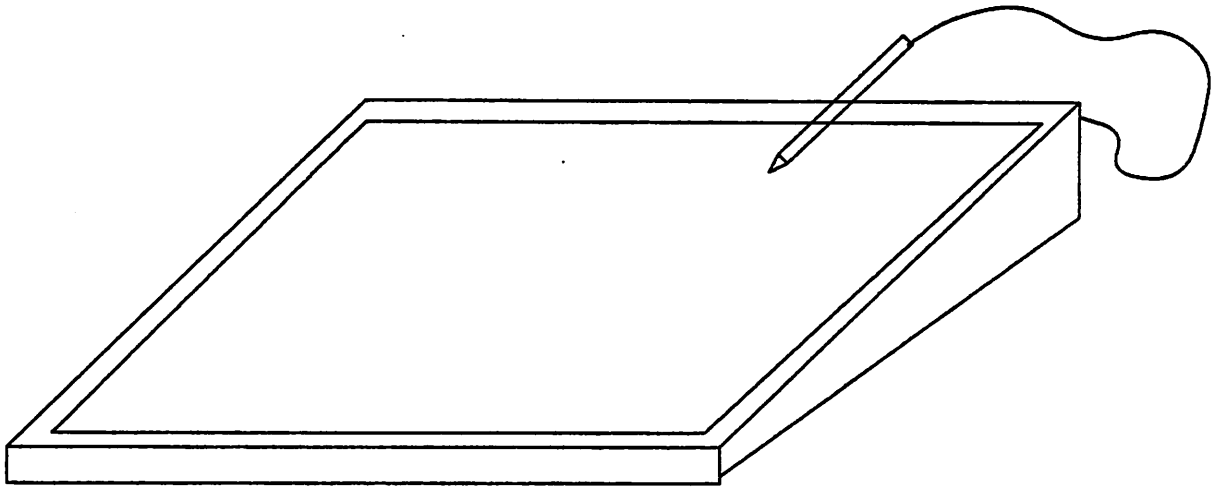


Figure 1.2 Notebook PC

The notebook PC has several essential advantages over the current PC. The most important one is, if the *OHR* is powerful enough, the man-computer communication symbols are no longer limited by the keyboard. The user should be able to create Fig. 1.3(b) by simply writing as shown in Fig. 1.3(a). As this illustration shows, the *OHR* can bring the PC applications to an era which has never been explored before.

Besides symbol entry, the notebook PC has two other significant features. First, the user can easily access any point on the screen by "one touch". Second, the stylus is the ultimate dev-

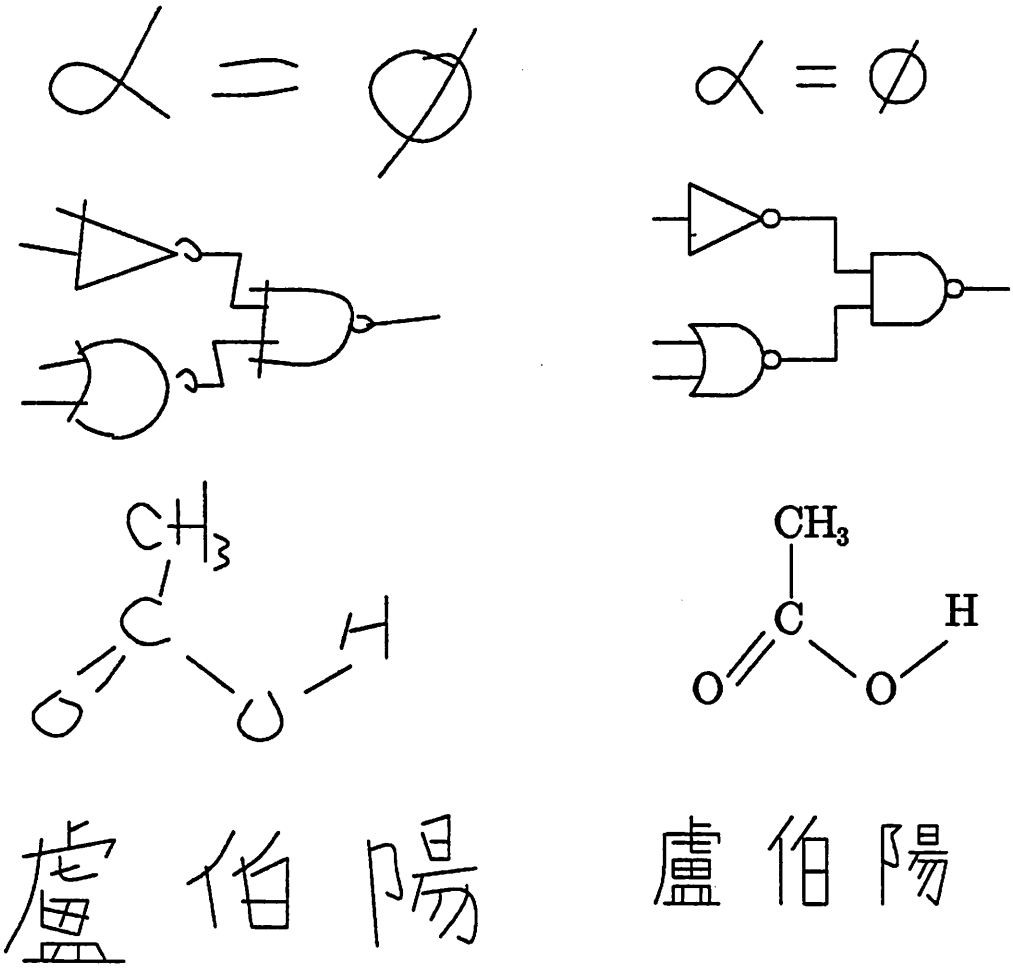


Figure 1.3 Symbols input by handwriting

ice for freehand drawing.

From a hardware point of view, with the advent of flat display and tablet technology, the emerging of such a PC is in fact not far away. However, from a software point of view, there seems no *OHR* system which has the desirable versatility needed in the notebook PC. (The *OHR* system should at least be able to recognize all the symbols in Fig. 1.3.) As such, the goal of this project is to study the feasibility of a PC based *OHR* system which can recognize a wide range of symbols for a variety of applications.

2. Design goals

2.1. Development system

Unfortunately, because the flat display was not available for this project, the notebook computer environment could not be established. The development system was set up as shown in Fig. 1.4. A Seiko tablet, which has 200 points per inch resolution and 100 points per second sampling rate, is connected to an IBM PC (or compatible) through a serial communication port. The IBM PC was chosen as the primary platform for two reasons. One, the hardware and software architecture of IBM PC is relatively simple and open. This makes a low-level system interface possible. Two, the IBM PC is the most widely used computer. There is abundant software available for various applications.

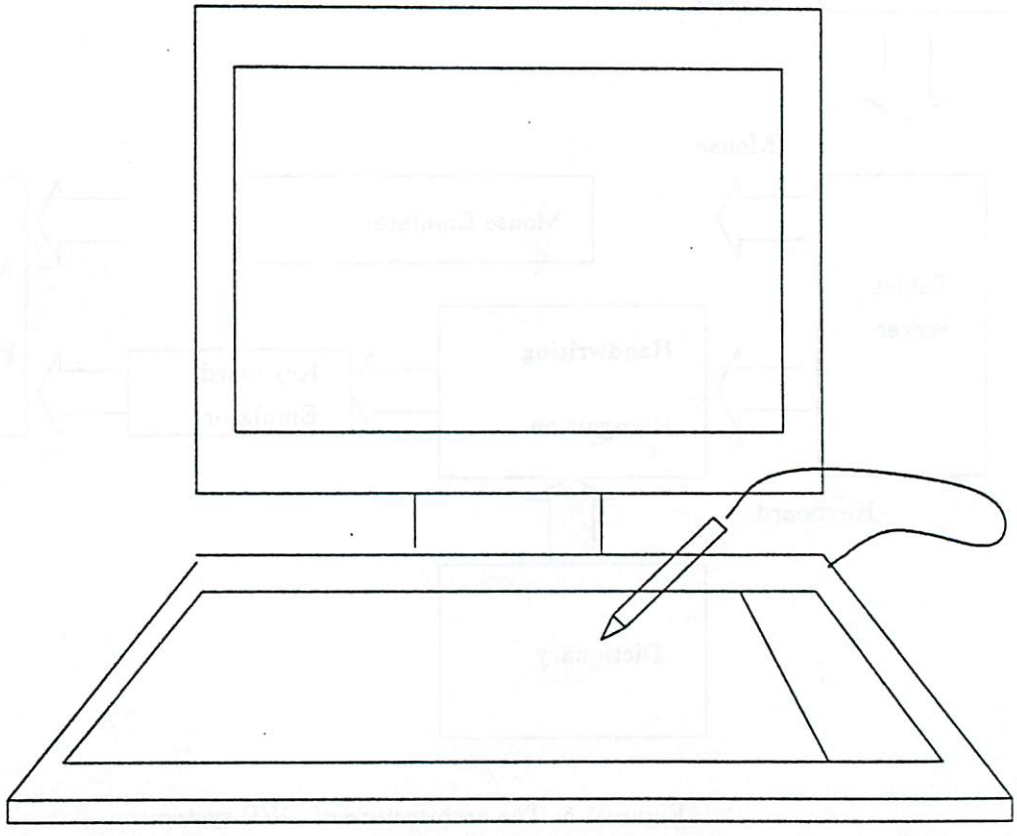


Figure 1.4 Development system

2.2. Application interface

From application interface point of view, the architecture of the *OHR* system should be as illustrated in Fig. 1.5. The tablet server handles the data acquisition from the tablet. If the

OHR is in keyboard mode, the tablet coordinates are sent to the recognition unit. The recognition unit compares these coordinates with the symbols in the dictionary. The keystrokes of the recognized symbol are sent to the ***keyboard emulator.*** The keyboard emulator then sends the keystrokes to the application programs as though they were typed in from the keyboard. If the *OHR* is in mouse mode, the tablet coordinates are passed to the ***mouse emulator.*** The mouse emulator processes the coordinates and then sends them to application programs as though they were from the mouse.

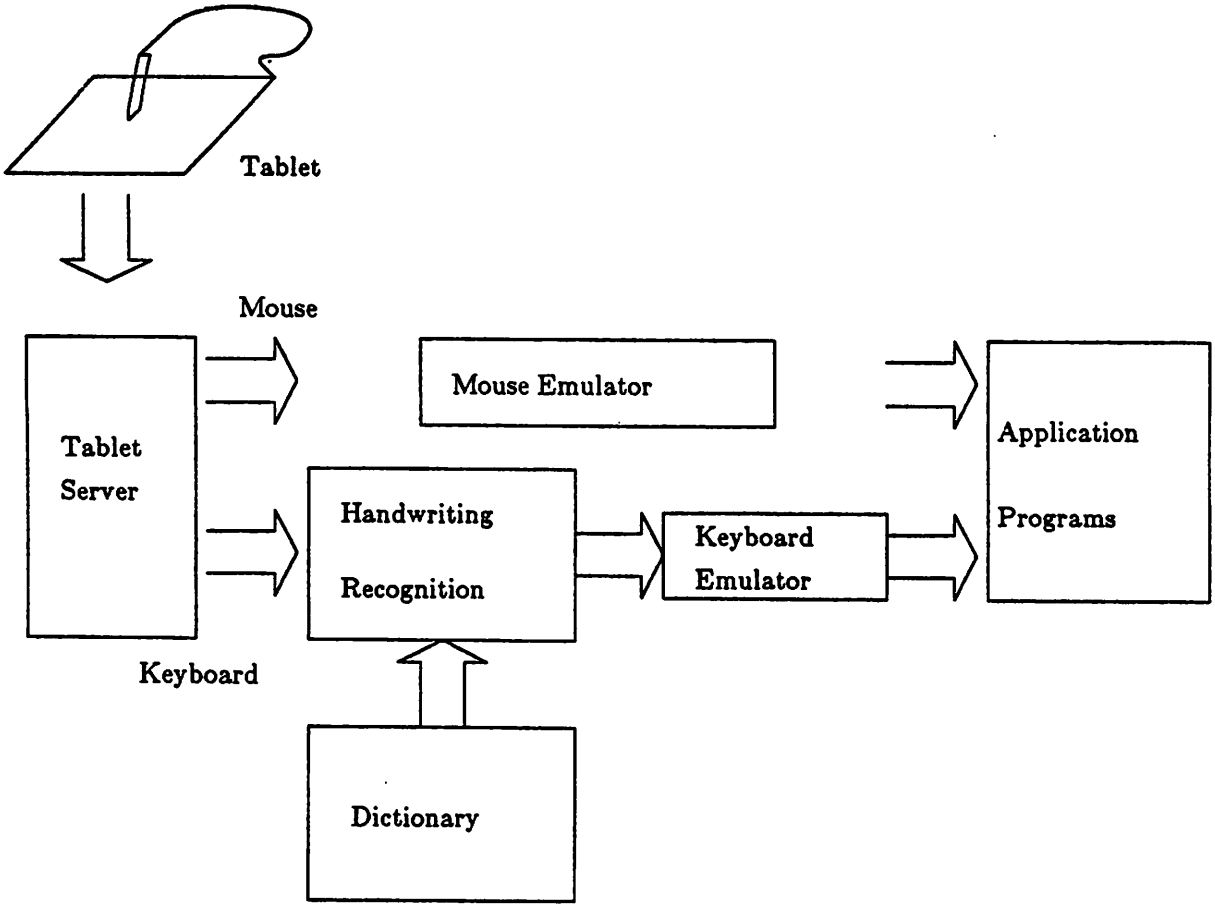


Figure 1.5 The architecture of *OHR* system

2.3. Tasks

This project was divided into the following five major tasks.

(1) Develop a recognition algorithm which can accommodate symbols written with different sizes, different speeds, and reasonable variations.

(2) Develop a training algorithm such that good recognition can be achieved with minimal training.

(3) Make sure the recognition can be performed in real-time.

(4) Develop a tablet server to provide all tablet related services. Develop a keyboard emulator and a mouse emulator to allow a user to be able to write and draw to existing application software.

(5) Develop an algorithm which can recognize *cursive scripts* with only the training of letters.

2.4. Recognition strategy

The *OHR* system was intended to accurately recognize any symbol input by any user. The ideal way to use it is like this: if a user finds that some of his writing are not correctly recognized, or he wants to add some symbols, he can modify the system dictionary by simply giving a few additional training samples. To achieve this, the system must have a strong learning capability.

One valuable piece of information available to the *OHR* is the writing sequence of a symbol. However, a symbol has always been treated as an "image" instead of a "coordinate sequence" in the past. People seldom pay attention to what is the right sequence in which to write a symbol. This creates an enormous number of sequence variations. It has been found that there are more than two million ways to write a simple "four stroke" symbol 'E'.[1]

If the writing sequence is ignored and the written symbol is treated as an "image", the *OHR* problem is essentially the same as the freehand *optical character recognition (OCR)* problem, which is very hard and beyond the scope of the project. Therefore, decision was made that the recognition would use the sequence information heavily. The user is advised to be cautious and make his writing sequence consistent.

3. Outline of the thesis

In chapter 2, the pros and cons of major algorithms which have been proposed for *OHR* are

reviewed. The reason why the template matching approach is chosen is explained.

In chapter 3, the preprocessing, feature extraction, and distance measurement algorithms are first described. Various template matching algorithms are then experimented with and compared. The strength and weakness of the *DTW* matching algorithm are discussed.

Covered in chapter 4 is how to automatically generate disambiguation rules to differentiate symbols which are not accurately distinguished by template matching.

Chapter 5 addresses how to optimize the developed algorithm for speed.

In Chapter 6, the implementations of the keyboard emulator, the mouse emulator, and the tablet server are described. The utility programs and how to use them to achieve a high recognition rate without training overhead are also presented.

In Chapter 7, previous work on the cursive script recognition is first reviewed. Then, how to extend the discrete symbol recognition algorithms for cursive scripts is explained. The techniques of using template extraction, profile check, and syntax check to enhance the recognition rate are also covered.

Discussed in chapter 8 is how to implement the *OHR* system on a speech recognition system which has a dedicated processor for *DTW* matching.

Chapter 9 summaries the key conclusions of the project.

References

1. T. T. Kuklinski, "Components of Handprint Style Variability," *Proc. of 7th International Conference on Pattern Recognition*, pp. 924-926, 1984.

CHAPTER 2

REVIEW

The work of on-line handwriting recognition started twenty years ago when the graphic tablet was invented. Many algorithms have been proposed since then and most of them claim to achieve a very high recognition rate. However, because few of them have been seriously tested, there is no consensus as to which one is the best. In this chapter, several major proposed algorithms will be evaluated from a very practical point of view.

As in any pattern recognition problem, there are two major approaches to attack the problem: syntactic and mathematical. In following sections, variations of each approach will be presented.

1. Syntactic methods

In syntactic methods, the basic philosophy is to extract a class of easily identified *primitives* from a complicated symbol and then *parse* these primitives through *grammars*. Following are three variations of this approach.

Algorithm 1:[1]

In this algorithm, the rectangle which surrounds the symbol is divided into 4×4 regions as shown in Fig. 2.1. The major primitives that are used include: (1) the number of segments, (2) the slope sequence, (3) the position of the start point, (4) the position of the end point, (5) the position of the corners, (6) the height/width ratio.

The definition of a slope code is determined by two consecutive points as illustrated in Fig. 2.1. A corner is defined as a point at which the slope angle changes more than 90°. Using these definitions, the '8' in Fig. 2.1 is described as

(1) number of segments: 6,

(2) direction sequence: {2,3,0,3,2,1},

(3) start point: region 0,

(4) end point: region 0,

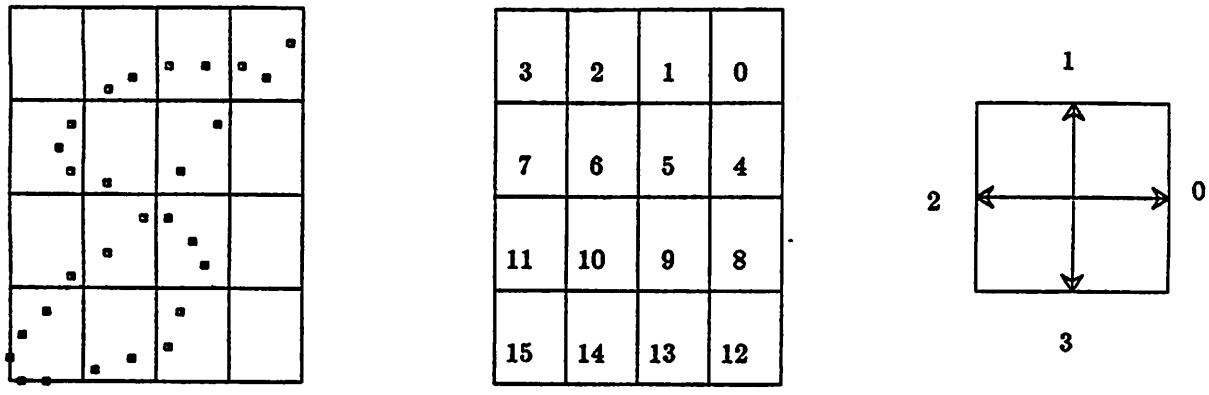


Figure 2.1 Character coding in algorithm 1

(5) corners: region 7 and 15,

(6) aspect ratio: 1.36 .

In the training phase, the primitives of all symbols are organized as a *decision tree* as shown in Fig. 2.2. In the recognition phase, the primitives of the unknown symbol are sent to the decision tree to check whether they satisfy the requirements of each symbol. For example, the '8' in Fig. 2.1 has 6 segments. Its first 4 slope codes are (2,3,0,3). Its 5th and 6th slope codes are (2,1). Its endpoint is in the upper half of the rectangle. The decision tree classifies it to be '8'.

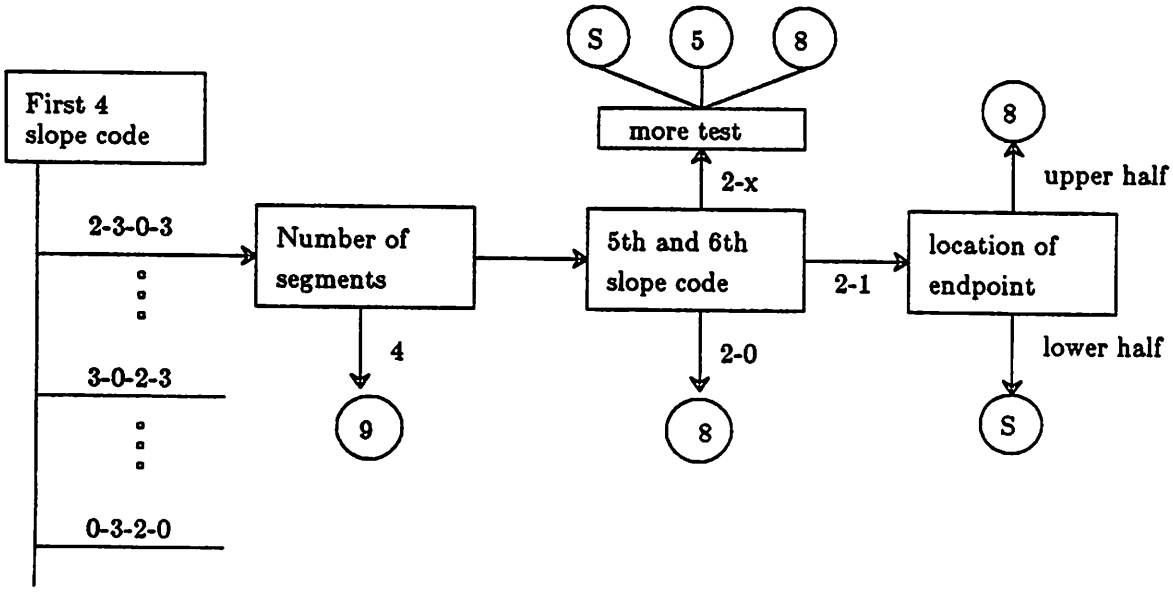


Figure 2.2 Decision tree in algorithm 1

Algorithm 2:[2]

In this algorithm, the target symbols are uppercase characters. The constituent strokes of these symbols are divided into two categories: straight-line (SL) strokes and curvilinear (CL) strokes. Fig. 2.3 illustrates standard strokes of each category.

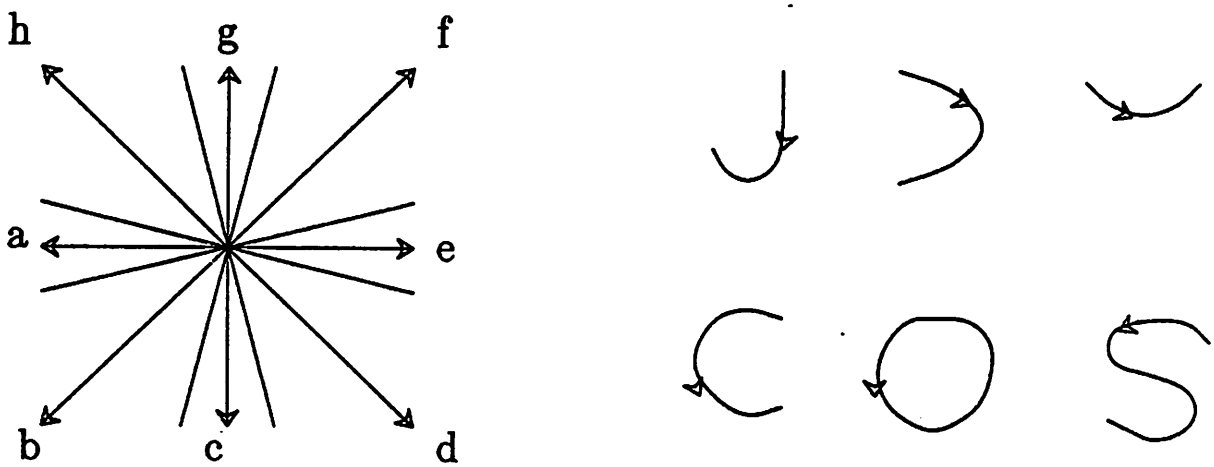


Figure 2.3 SL and CL in algorithm 2

The sequence of slope, which is defined by the relative position of two consecutive points as shown in Fig. 2.3, is the only recognition feature used. From it, an SL is recognized if all slopes of a stroke are the same. The recognition of an CL stroke is not as straight forward.

For each CL stroke, two grammars are defined in the training phase. They are the *transformational grammar* and the *pattern grammar*. The purpose of the transformational grammar is to smooth sample points and to reject sequences which can not be further transformed. The pattern grammar is constructed bottom-up to describe basic variations of that stroke. Fig. 2.4 lists the two grammars of the stroke '⊃'.

In the recognition phase, the slope sequence of an unknown stroke is parsed by the *pattern grammar* of each standard stroke to see whether it is that stroke. If not, the *transformation grammar* is applied to see whether the input sequence can be transformed. If it can, the transformed sequence is parsed again. This procedure continues until either the sequence can not be further transformed or it is verified to be that stroke.

Transformational Grammar

T = terminal symbols
= (a, b, c, d, e, f, g, h)

(1) fd –> fed
(2) fc –> fedc
(3) ec –> edc
(4) eb –> edcb
(5) db –> dcb
(6) da –> dcba
(7) ca –> cba
(8) ch –> cbah
(9) bh –> bah

Pattern Grammar

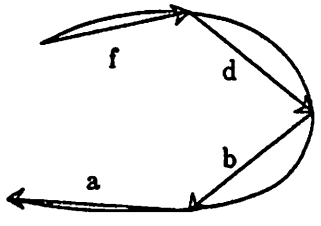
G = (N, T, P, S)
S = Sentential Symbol
N = non-terminal symbol
= (A, B, C, D, E, S)

(1) S –> A ^ B ^ C ^ D ^ E
(2) A –> dcb
(3) B –> Aa
(4) C –> Bh
(5) D –> eA ^ eB ^ eC
(6) E –> fD

(^ means 'or')

Figure 2.4 Transformational grammar and pattern grammar

Fig. 2.5 illustrates an example. The slope sequence of that stroke is *fdba*. This sequence is transformed to *fedba* by transformational grammar rule 1 and then to *fedcba* by rule 5. The *fedcba* is then parsed by the pattern grammar and is recognized as '⊃'.



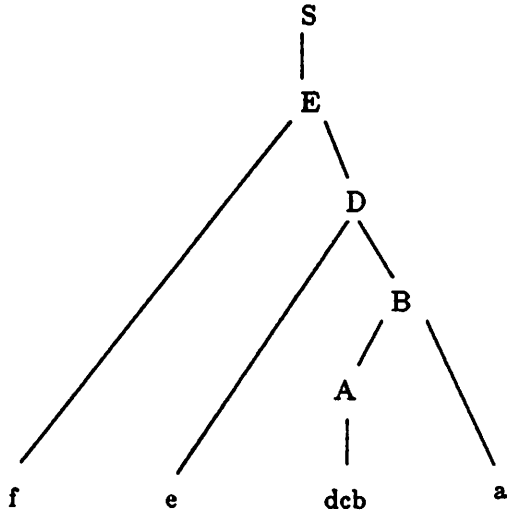


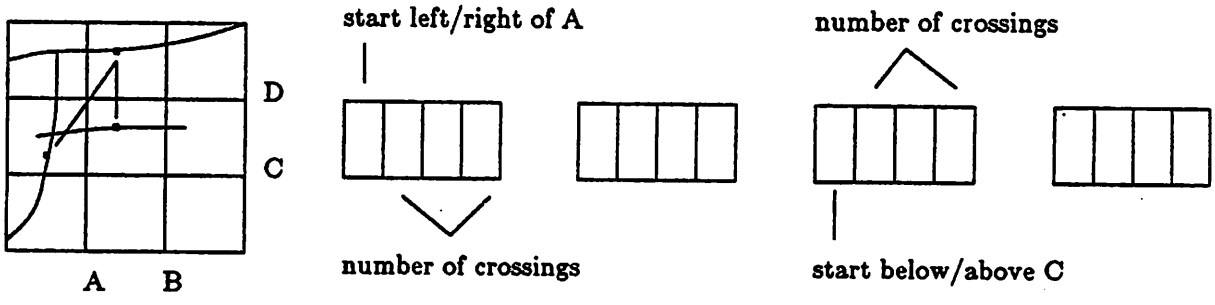
Figure 2.5 Parsing in algorithm 2

Algorithm 3:[3]

In this algorithm, the surrounding rectangle of a symbol is divided into 3×3 regions by four dividing lines as shown in Fig. 2.6. The recognition features extracted from a stroke are its

starting position and the number of crossings of each dividing line. These features are encoded into 4 nibbles. Each nibble (4 bits) represents the relation between the stroke and one dividing line. In it, the first bit indicates on which side the stroke starts. It is set to 1 if the stroke starts to the left of (or below) a dividing line. The remaining 3 bits record the number of crossings.

For each symbol, a pseudo stroke is constructed by joining the center points of all strokes. The code of this pseudo stroke is also recorded. As such, the codes of a three stroke symbol *'F'* in Fig. 2.6 is as illustrated.



0000-1000-0001-0001 Vertical stroke
1001-1001-0000-1000 Upper horizontal stroke
1001-1001-0000-1000 Lower horizontal stroke
1001-1000-0000-1010 Additional stroke

Figure 2.6 Character coding in algorithm 3

In the training phase, the codes of each training samples are extracted, compared, and stored in dictionary. In the recognition phase, the same encoding scheme is applied to the unknown symbol. It is recognized as the symbol which has exactly the same codes.

2. Mathematical methods

In mathematical methods, instead of parsing features through grammars, a *distance function* is applied to measure the similarities between features of the unknown and the prototype features of each symbol. The unknown is recognized as the one which yields the minimum distance.

2.1. Statistical algorithms

Let U denote the feature vector of the unknown symbol and T^k denote the feature vector of the *kth* symbol in dictionary. If the *a priori* distribution functions $P(T^k)$ and $P(U/T^k)$ are known, theoretically the optimal distance function is the *posteriori* probability function $P(T^k/U)$. Algorithms based on this principal are called *statistical algorithms*. Following are two examples.

Algorithm 4:[4]

In this algorithm, a stroke is first segmented by its start point, break points and end point. A break point is defined as a sample point whose slope is either 0 or ∞.Then, for each segment, the maximum discrepancy between sample points and the line segment is compared to a threshold.If the discrepancy is greater than the threshold, the segment is split into two segments at the point which renders the maximum discrepancy. This procedure continues until all points are less than the threshold of its segmented version.

Fig. 2.7 shows the results of segmentation.

After segmentation, the *ith* segment is represented by vector (ϕ_i, d_i). (ϕ_i, d_i) are parameters of the line $x \times cos\phi + y \times sin\phi + d = 0$ which passes through the segment. The whole symbol is therefore described as

$$U = (\phi_1, d_1, \phi_2, d_2, \cdots, \phi_n, d_n) .$$

For each symbol T^k, the *priori* probability function $P(U/T^k)$ is assumed to be Multivariate Gaussian, i.e.

$$P(U/T^k) = \frac{1}{(2\pi)^{n/2} \sigma_k^{1/2}} \exp\left[\frac{1}{2}(U - M_k)^t \sigma^{-1} (U - M_k)\right].$$

M_k and σ_k are the mean vector and covariance matrix of symbol T^k.

In the training phase, M_k and σ_k are obtained from training samples $Y_1, \cdots Y_N$.

$$M_k = \frac{1}{N} \sum_{j=1}^{N} Y_j$$

$$\sigma_k = \frac{1}{N} \sum_{j=1}^{N} (Y_j - M)(Y_j - M)^t$$

In the recognition phase, the unknown symbol U is recognized as the T^k which has the maximum $P(U/T^k)$.

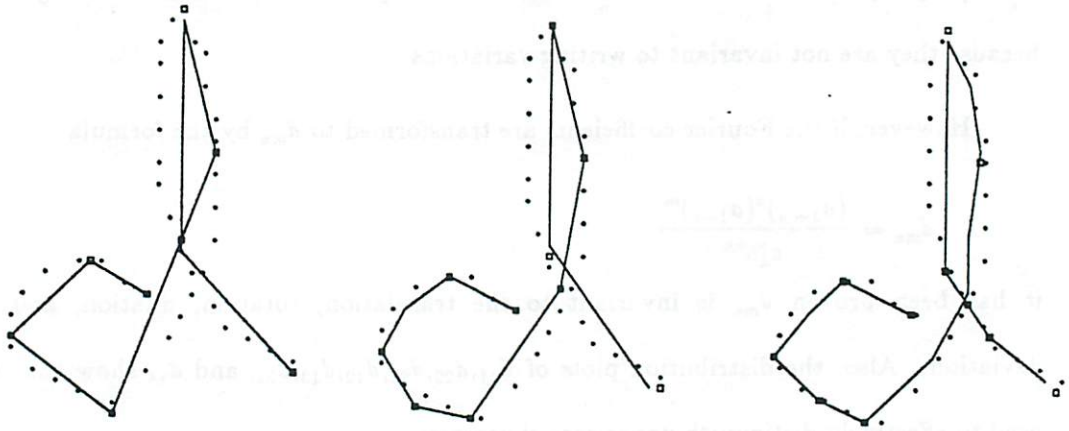


Figure 2.7 Segmentation in algorithm 4

Algorithm 5:[5]

In this algorithm, *Fourier descriptors* are used for recognition. As shown in Fig. 2.8, the trajectory of a stroke can be treated as a parametric curve $U(t)$, $U(t) = x(t) + jy(t)$, on the complex plane. The Fourier coefficient a_n of the trajectory is:

$$a_n = \frac{1}{2\pi}\int_0^{2\pi} u(t)e^{-jnt}dt \ .$$

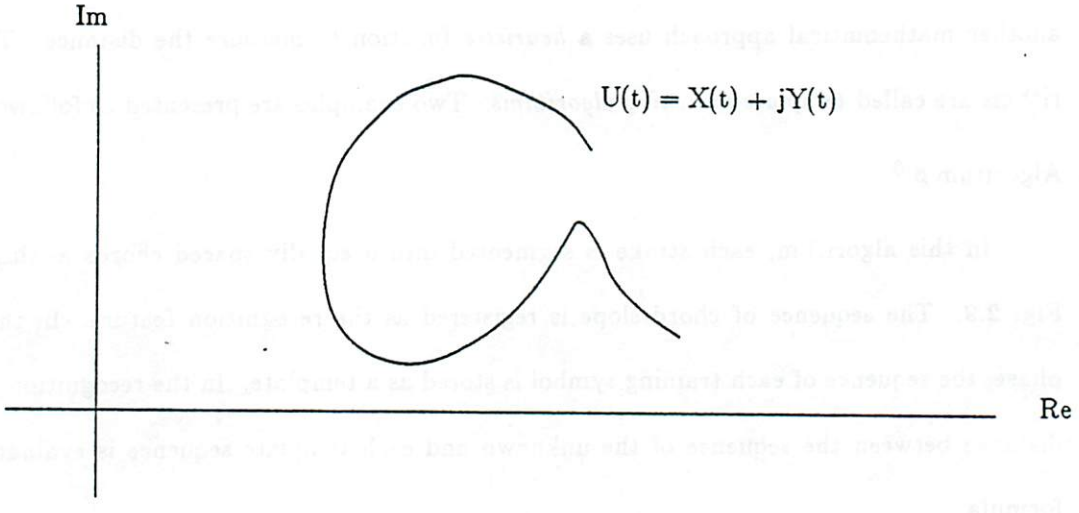


Figure 2.8 Character trajectories on complex plane

Let $a_n^{(o)}$ denote a Fourier coefficient of a certain stroke. If the stroke, as illustrated in Fig. 2.8, is shifted by a complex vector Z, rotated by an angle ϕ , dilated by a factor R, and delayed

its start point by τ , it has been derived the a_n will be

$$a_n = exp(jn\tau)\times R\times exp(j\phi)\times a_n^{(o)} \quad (+Z \text{ if } n = 0) \ .$$

This property makes the Fourier coefficients not particularly useful as recognition features because they are not invariant to writing variations.

However, if the Fourier coefficients are transformed to d_{mn} by the formula

$$d_{mn} = \frac{(a_{1+m})^n (a_{1-n})^m}{a_1^{m+n}} \ ,$$

it has been proven d_{mn} is invariant to the translation, rotation, dilation, and start point deviation. Also, the distribution plots of $d_{11}, d_{22}, d_{21}, d_{12}, d_{13}, d_{31}$, and d_{44} show that they can be used to effectively distinguish upper case characters.

As such, assume the probability distribution of the real part and imaginary part of d_{mn} is two-dimensional Gaussian. In the training phase, the mean value and covariance matrix of d_{mn} of each symbol are obtained. In the recognition phase, the unknown is recognized as the symbol whose *posteriori* probability of d_{mn} is the maximum.

2.2. Template matching algorithms

Instead of recognizing a symbol based on probability model and parameter estimation, another mathematical approach uses a *heuristic* function to measure the distance. These algorithms are called *template matching algorithms*. Two examples are presented as follows.

Algorithm 6:[6]

In this algorithm, each stroke is segmented into 6 equally spaced chords as illustrated in Fig. 2.9. The sequence of chord slope is registered as the recognition feature. In the training phase, the sequence of each training symbol is stored as a template. In the recognition phase, the distance between the sequence of the unknown and each template sequence is evaluated by the formula

$$f(U, T^k) = \sum_{i=1}^{6} L(U_i, T_i^k) \ .$$

The $L(U_i, T_i^k)$ in the formula is the Lee metric between the *ith* slope code of the unknown and

the *ith* slope code of template T^k. It is defined as

$$L(a, b) = MIN(\ a - b\ , 8 - \ a - b\ )\ .$$

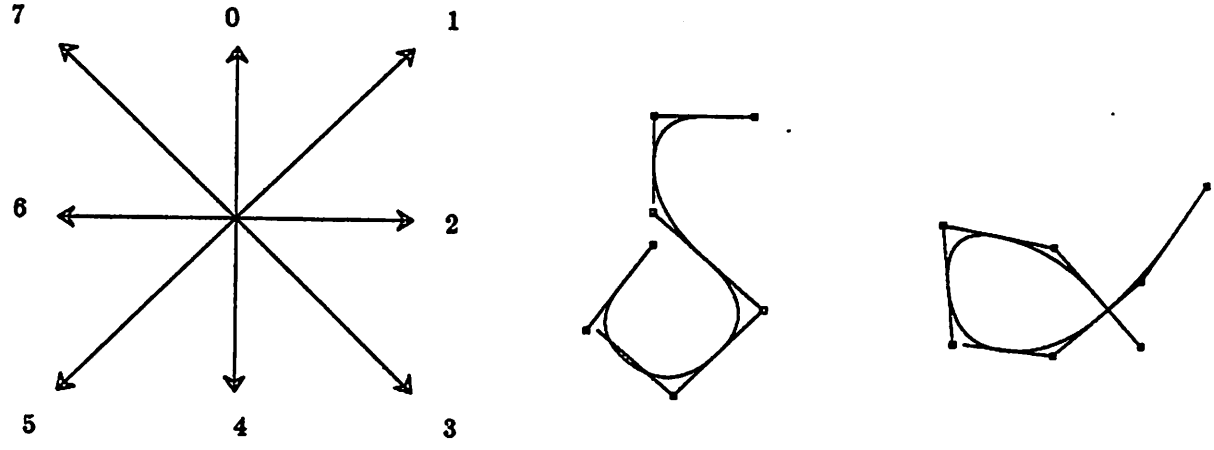


Figure 2.9 Character coding and Lee Metric $L(a,b)$

If there is only one template which yields minimum distance, it is picked out as the unknown. If there are more than one candidate, some ad hoc disambiguation tests are applied to make the final decision. These tests basically check the position of the center point of each stroke. As Fig. 2.10 shows, to distinguish between 'T' and '+', it is 'T' if the center of the first stroke is in region U and it is '+' if the center of the first stroke is in region C.

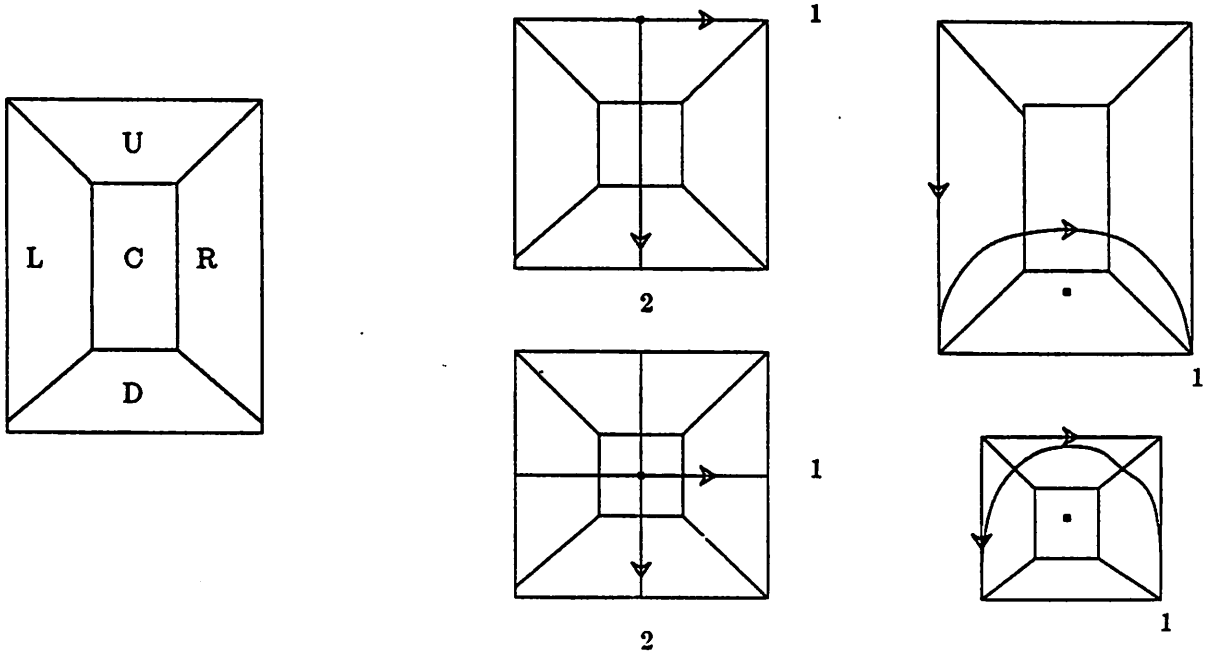


Figure 2.10 Disambiguation in algorithm 6

Algorithm 7:[7]

In this algorithm, a stroke is first segmented. The segmentation is based on the Ramer algorithm[8] which is slightly different from the one used in algorithm 4. Referring to Fig. 2.11, the segmentation starts from a segment which is created by connecting the start point and end point of a stroke. If all sample points of the stroke lie within a specified threshold of the segment, segmentation is complete. Otherwise, the segment is split into two segments at the point which is farthest away from the segment. This procedure continues until all sample points on the original stroke are within a threshold of its segmented version.

In the training phase, the segments of a symbol are stored as its template. In the recognition phase, after segmenting the unknown, all *local distances* (the distance between the *ith* segment of the unknown and the *jth* segment of the template) are first evaluated. The local distance between two segments $\vec{A}$ and $\vec{B}$ is defined as:

$$d(\vec{A},\vec{B}) = \frac{\vec{B}\times\vec{A}}{\vec{A}} + 3U(-\vec{B}\cdot\vec{A})\frac{\vec{B}\cdot\vec{A}}{\vec{A}} + \frac{\vec{C}\times\vec{A}}{\vec{A}} +$$
$$U(\frac{\vec{C}\cdot\vec{A}}{\vec{A}} - \vec{A}\,)\,\frac{\vec{C}\cdot\vec{A}}{\vec{A}} - \vec{A} + U(-\frac{\vec{C}\cdot\vec{A}}{\vec{A}})\frac{\vec{C}\cdot\vec{A}}{\vec{A}} .$$

In the formula, $\vec{C}$ is the vector from the start point of $\vec{A}$ to the midpoint of $\vec{B}$ and $U(x) = 0$ if $x < 0$, $U(x) = 1$ if $x \geq 0$.

From all these local distances, the final distance between the unknown and a template is obtained by a technique called "*DTW* matching". (The detail of *DTW* matching will be described in Chapter 3.) The unknown symbol is recognized as the one which has the minimum final distance.

3. Comments

In order to choose an algorithm for implementation, it is necessary to examine the following design goals.

3.1. Trainability

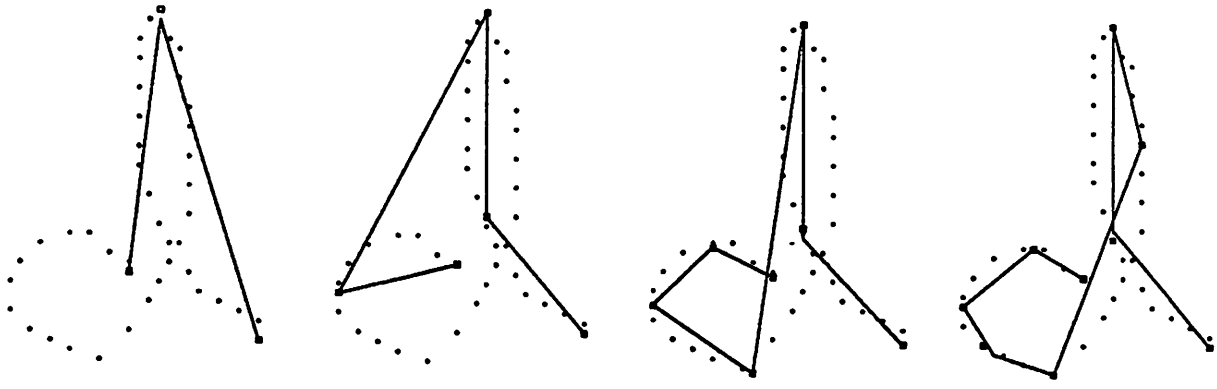


Figure 2.11 Segmentation in algorithm 7

As presented in Chapter 1, a primary concern in our *OHR* is trainability. The user should be able to train any symbol easily. This concern makes those ad hoc rule based syntactical algorithms not attractive. For instances, if a user wants to add a new symbol, how can the decision tree in algorithm 1 be automatically extended ? How can the transformation grammar and pattern grammar in algorithm 2 be automatically generated ?

The trainability concern also eliminates the statistical algorithms. This is because a statistical algorithm requires quite a few training samples to obtain a meaningful estimation of the statistics. To ask a user for many training samples makes the training procedure very tedious.

3.2. Sensitivity

A robust recognition algorithm should not be sensitive to small variations which are inevitable in our natural writing. However, the performance of some algorithms discussed above may drastically degrade with slight changes of the unknown symbol. For example, the two 8's in Fig. 2.12 can not be recognized by the decision tree in Fig. 2.2 because the 5th direction code of the Fig. 2.12 (a) '8' is 1 and the end point of the Fig. 2.12 (b) '8' is at the *lower* half of the symbol. The two *F*'s in Fig. 2.13 also do not match with the code of 'F' presented in algorithm 3. This is because the vertical stroke of the Fig. 2.13 (a) 'F' starts *right* of boundary A and the lower horizontal stroke of the Fig. 2.13 (b) 'F' *does not* cross boundary B.

In algorithm 6, the slope codes of the two 3's in Fig. 2.14 are

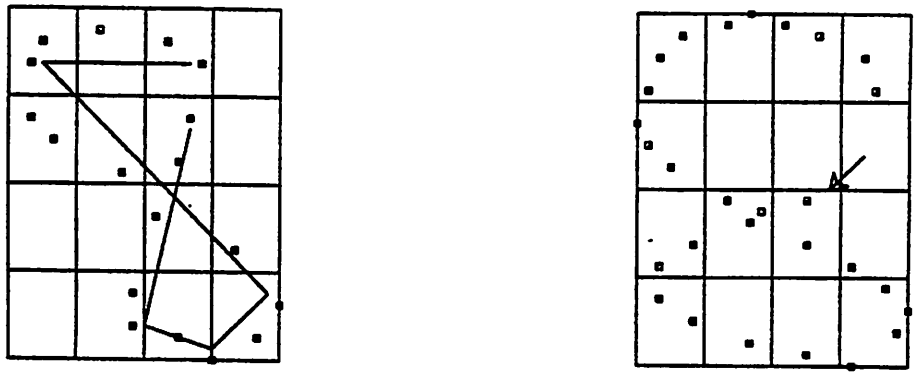


Figure 2.12 Limitations of algorithm 1

Figure 2.13 Limitations of algorithm 3

(1, 3, 5, 3, 5, 7) and
(1, 5, 3, 5, 6, 7).

With the distance function defined there, the distance between the two '3' is 2+2+2+1 = 7. This distance is unreasonably high. The reason is because the two 3's are not properly aligned.

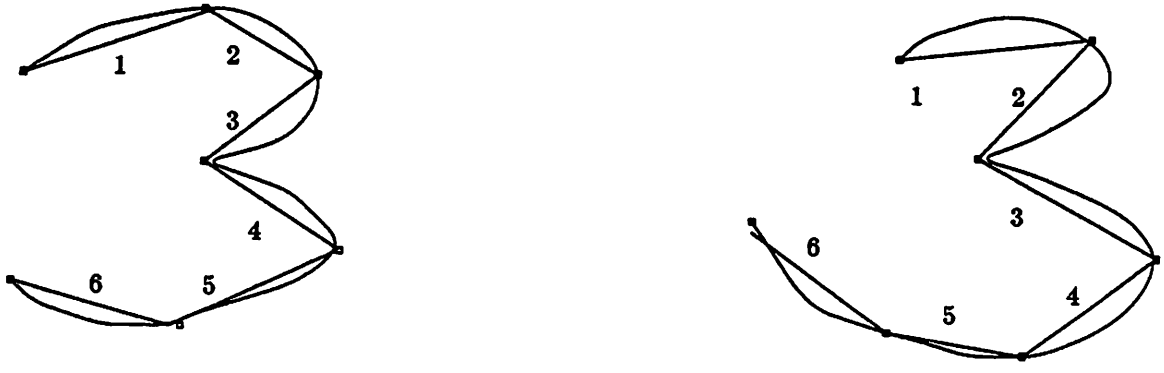


Figure 2.14 Limitations of algorithm 6

3.3. Trackability

Symbols are written by hand, which is one of the most controllable organs in human body. In order to achieve easy training, high accuracy, and fast response goals, this human factor has to be taken into account and avoid unnecessary overhead in algorithm. The algorithm must be intuitively simple enough so that a user can determine how to achieve a better recognition rate. In other words, it must be *user friendly.*

In the presented algorithms, it was found that some recognition features are very hard to cope with. For instance, it is very difficult for a user to imagine how to produce good (ϕ, d) in algorithm 4 or good Fourier descriptors in algorithm 5 for better recognition. In statistical algorithms, if there is a recognition error, the user has no way to determine what went wrong.

3.4. Computability

To make the *OHR* system a valuable data entry device, the recognition result must be carried out in a very short time after the symbol is finished. Although electronics technology has made microprocessors very fast, it is still impossible to cost effectively compute Fourier descriptors or the statistical discriminator $(X - M_c)^T \sigma^{-1} (X - M_c)$ in real-time using general purpose processors. The computation of local distances in algorithm 7 is also very substantial and hard to implement.

4. Conclusion

From the review and comments, the template matching approach was chosen because both algorithm 6 and 7 have shown that a symbol can be added to a dictionary without much training effort. The recognition feature used in algorithm 6 is easy to understand. The *DTW* used in algorithm 7 elegantly absorbs many writing variations.

To devise a better algorithm, the weaknesses in these two proposed algorithms have to be removed. Their merits have to be combined. Besides, to further improve the recognition accuracy, in addition to the slope sequence, the x sequence and y sequence should also be matched to check the similarity from other points of view. It is quite possible that these matching results

can remedy the inadequacy of each other.

As such, in the following chapters, the matching results of using the x, y, and slope sequences of a symbol will be first examined. Then, a decision procedure which is based on these matching results will be proposed.

References

1. G. F. Groner, "Real-time Recognition of HAndprinted Text," *Proceedings, FJCC*, pp. 591-601, 1966.

2. M. R. Ito and T. L. Chui, "On-line Computer Recognition of Proposed Standard ANSI Handprinted Characters," *Pattern Recognition*, vol. 10, pp. 341-349, 1978.

3. W. Newman and R. Sproull, *Principals of Interactive Computer Graphics*, 1978.

4. W. W. Loy and I. D. Landau, "An On-line Procedure for Recognition of Handprinted Alphanumeric Characters," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, pp. 422-427, July 1982.

5. G. H. Granlund, "Fourier Preprocessing for Hand Print Character Recognition," *IEEE Trans. Computer*, vol. C-21, pp. 195-201, February 1972.

6. G. Miller, "On-line Recognition of Hand-generated Symbols," *AFIP Proc. FJCC*, vol. 35, pp. 399-412, 1969.

7. D. Burr, "A Technique for Comparing Curves," *Proc. IEEE Conference on Pattern Recognition and Image Processing*, pp. 271-277, 1979.

8. U. Ramer, "An Interactive Procedure for the Polygonal Approximation of Plane Curves," *Computer Graphics and Image Processing*, vol. 1, pp. 244-256, 1972.

CHAPTER 3

TEMPLATE MATCHING

1. Preprocessing

When a symbol is being written on graphic tablet, the tablet periodically sends out the coordinates and up/down status of the stylus. This is the only information available to the *OHR* system. Among the sample points, only points at which the stylus is pressed down are used for recognition. Fig. 3.1(a) shows a plot of these points of a typical writing. From this plot, it can be seen that some preprocessing must be done to get meaningful features.

Along the main trace of a symbol, there are some fluctuating sample points. These points occur because of the high resolution (0.005 inch) and sampling rate (95 points/sec) of the tablet. Any unstable hand movement will create jaggy sample points. Therefore, the first required processing is to remove these noisy sample points.

Processing is also needed to provide uniform spatial sampling of the points. Since the temporal sampling rate of the tablet is fixed, there are more points in the portions in which the user writes slowly and fewer points in the portions in which the user writes quickly. The uneven spatial distribution of sample points may degrade the recognition accuracy.

The x and y coordinates from the tablet are not suitable for recognition because they are affected by the absolute position and the size of the symbol. They must be normalized.

1.1. Spatial filtering

The fluctuations in data points are removed by spatial filtering. A threshold is set for the minimum separation between two successive points. As illustrated in Fig. 3.1(b), all sample points which are inside the *window* set by the previous selected point are discarded.

At the start and end part of a stroke, more fluctuations are observed. This is because both the hand and the tablet are not as stable when the stylus is just pressed or lifted.[1] The window size of the first stroke point and the last stroke point is set at twice the regular size.

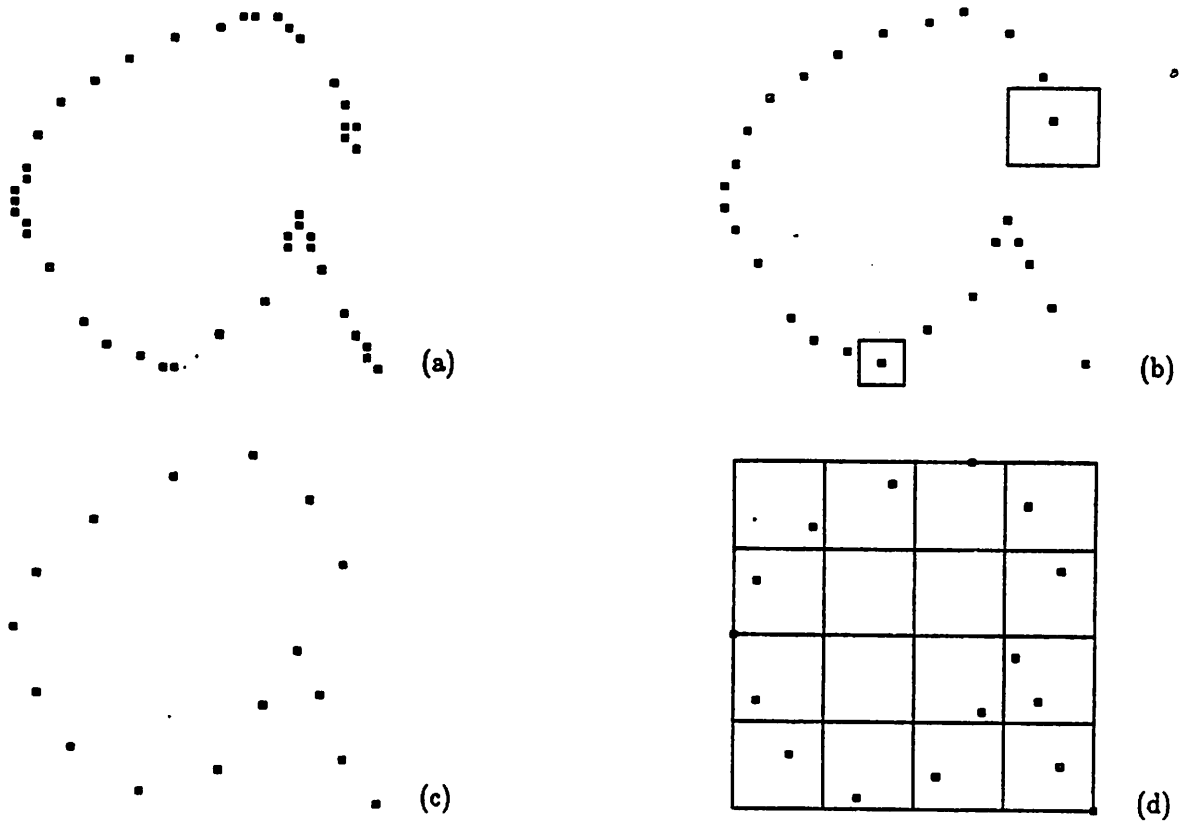


Figure 3.1 Preprocessing

1.2. Resampling

After spatial filtering, all points between the selected points are interpolated. Then, from the interpolated points, 16 equally spaced points are picked out as representative samples of the stroke. Fig. 3.1(c) shows the effect.

1.3. Normalization

Let $X_{\max}$, $X_{\min}$ denote the maximum and minimum x coordinates, $Y_{\max}$, $Y_{\min}$ denote the maximum and minimum y coordinates of the symbol, (x_i,y_i) denote its *ith* coordinate, and $(\tilde{x_i},\tilde{y_i})$ denote the normalized x,y coordinates. The $(\tilde{x_i},\tilde{y_i})$ is obtained by the following formula:

$$\tilde{x_i} = (x_i - X_{\min})\times\frac{W}{X_{\max} - X_{\min}} ,$$

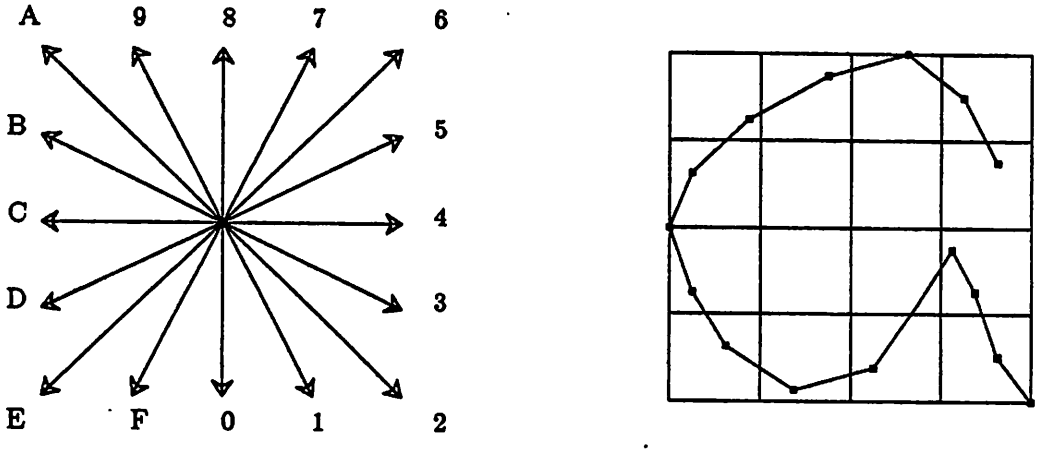
$$\tilde{y_i} = (y_i - Y_{\min})\times\frac{H}{Y_{\max} - Y_{\min}} .$$

In other words, the enclosing rectangle of a symbol is divided into $W\times H$ regions as illustrated in

Fig. 3.1(d). The x and y region indices of each point are used as its normalized coordinates.

1.4. Recognition features

The x code sequence, y code sequence, and slope code sequence are the three major features used for recognition. The x code and y code are simply the normalized x and y coordinates of a point. The slope code of a point is determined by the relative position of the point to the previous point as illustrated in Fig. 3.2. It can be derived either from the coordinates before normalization or from coordinates after normalization. However, because there are symbols, like *'1'* and *'/'*, which, although their aspect ratios are quite different, their normalized coordinates can be exactly the same. To distinguish them, the slope code has to be extracted from coordinates before normalization.



X = (3, 3, 2, 1, 0, 0, 0, 0, 0, 0, 1, 2, 2, 3, 3, 3, 3)

Y = (2, 3, 3, 3, 3, 2, 2, 1, 0, 0, 0, 1, 1, 1, 0, 0)

S = (0, 9, A, D, D, E, F, 1, 2, 3, 5, 6, 6, 1, 1, 2)

Figure 3.2 Recognition features

There are some other symbols, like *C* and *c*, whose x,y code sequences and slope code sequence are expected to be almost the same. The only difference is the symbol size. For these symbols, it is impossible to differentiate them unless there are lines on the tablet. Since the algo-

rithm is intended to allow arbitrary symbol size, these symbols are not distinguishable.

2. Distance accumulation

With the feature sequences of each symbol, the recognition algorithm is based on the following idea.

Let U_i denote the *ith* feature vector of the unknown, T_j denote the *jth* feature vector of the template, and $d_{i,j}$ denote the difference between U_i and T_j. The $d_{i,j}$ can be easily defined. One definition of $d_{i,j}$ is:

if $U_i = U_{x,i}$, $\quad d_{i,j}=dx_{i,j}=U_{x,i}-T_{x,j}$;

if $U_i = U_{y,i}$, $\quad d_{i,j}=dy_{i,j}=U_{y,i}-T_{y,j}$;

if $U_i = (U_{x,i},U_{y,i})$, $\quad d_{i,j}=dp_{i,j}=dx_{i,j}+dy_{i,j}$;

if $U_i = U_{s,i}$, $\quad d_{i,j}=ds_{i,j}=MIN(U_{s,i}-T_{s,j},\ Q-U_{s,i}-T_{s,j})$.

The Q in $ds_{i,j}$ is the total slope quantization levels. The $ds_{i,j}$ is defined as such because of the cyclic characteristics of the slope code as illustrated in Fig. 3.2.

Suppose there are I sample points of the unknown symbol and J points of the template. All $d_{i,j}$, $1\leq i\leq I$, $1\leq j\leq J$ can be tabulated as a *local distance matrix* as shown in Fig. 3.3. Let $D(T)$ denote the final distance between U and T. One reasonable way to obtain $D(T)$ is by accumulating the $d_{i,j}$ along a path wp in the local distance matrix, i.e.

$$D(T) = \sum_{(i,j)\in wp} d_{i,j} .$$

The physical meaning of wp is that it represents a "warping" of U and T. Fig. 3.4 illustrates the phenomena. The warping is necessary because the sample points in handwriting are not always perfectly aligned.

2.1. Warping path

It is obvious that there are enormous number of possible warping paths in the local distance matrix. In order to work out an algorithm which can find a satisfactory path for any symbol, the following four intuitive path constraints are imposed to limit the search space.

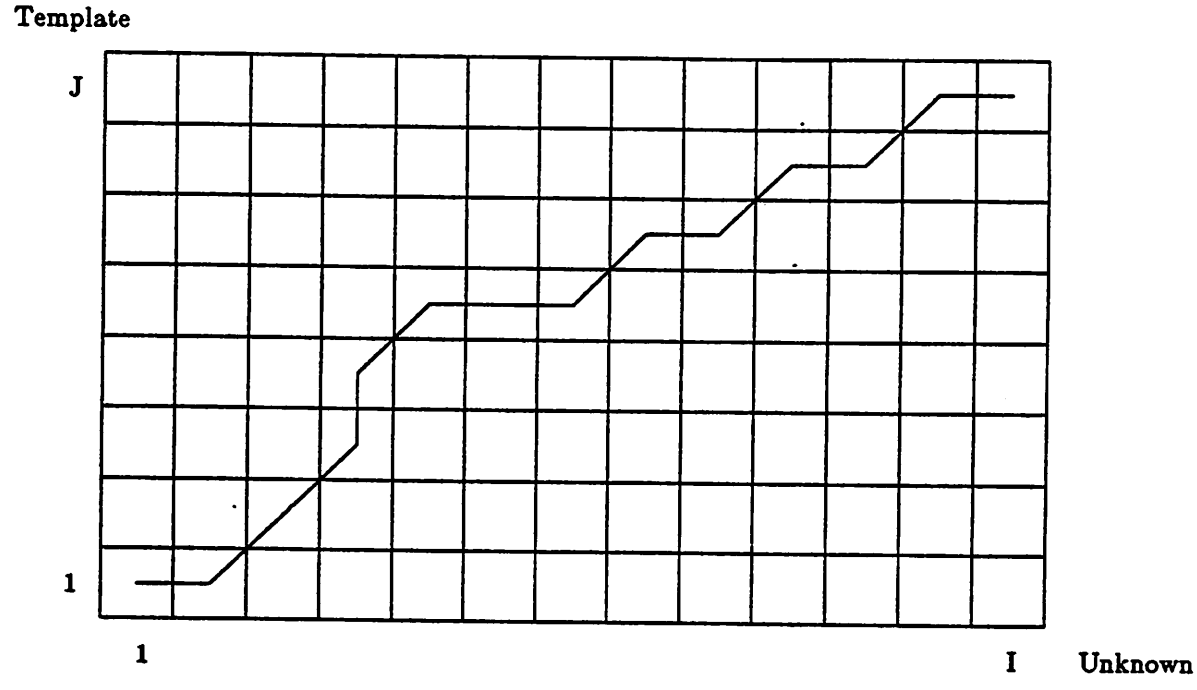


Figure 3.3 Distance accumulation

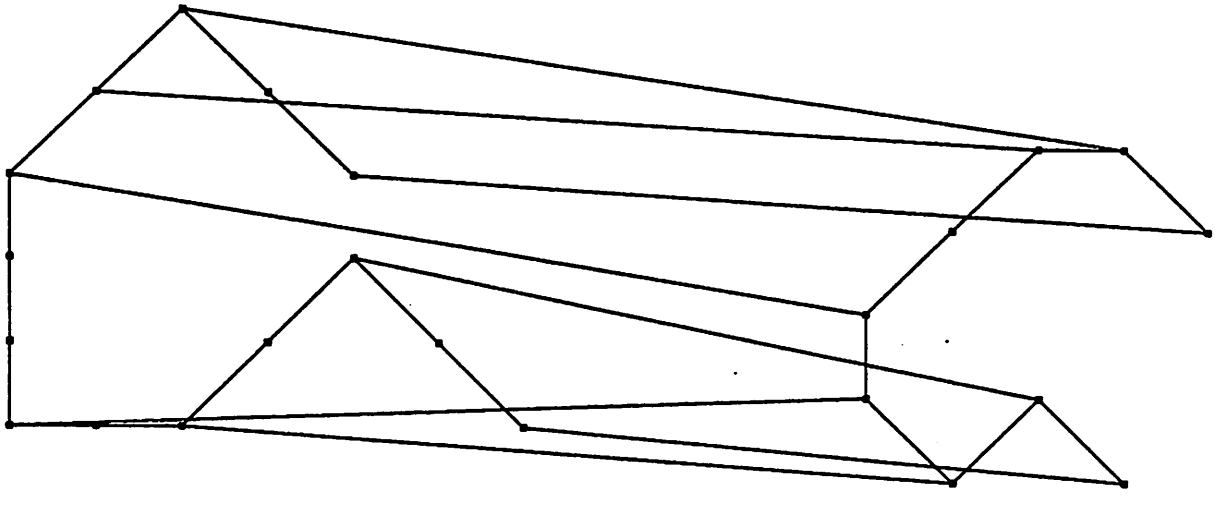


Figure 3.4 Physical meaning of the warping path

(1) The path should be monotonically increasing. Fig. 3.5 illustrates the reason. A non-monotonically increasing path implies that the best way to match the U and T is by aligning U_5 to T_7 and then U_6 to T_5. In natural writing, it is seldom found necessary to create this kind of reverse alignment. With reverse alignment, the distance between the two symbols in Fig. 3.5 will be 0, an undesirable result.

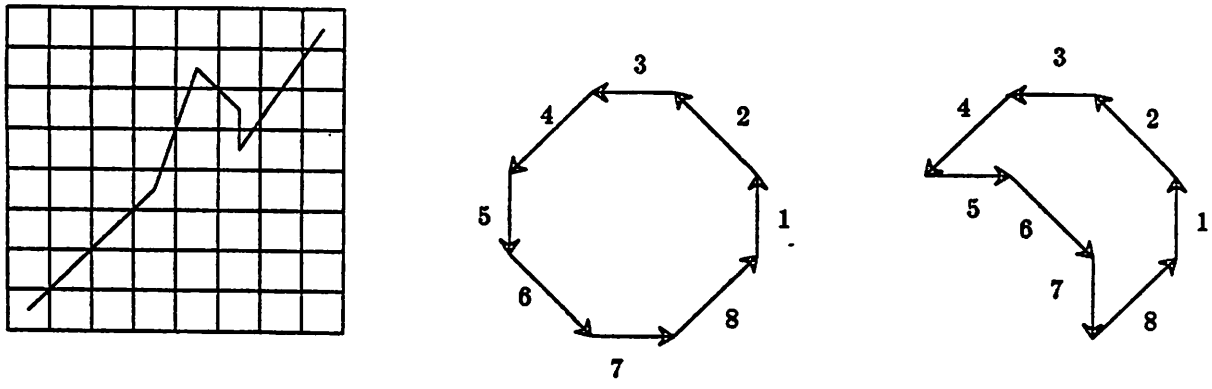


Figure 3.5 The warping path must be monotonically increasing.

(2) The path should be continuous. A path is defined to be continuous if and only if for every (i,j) on path, one of $(i-1,j)$, $(i-1,j-1)$ or $(i,j-1)$ is also on the path. The reason for this restriction is to force each U_i to be compared with at least one T_j and vice versa. Without this constraint, as shown in Fig. 3.6, unreasonable distance measures would be obtained because some of the critical sample points would be skipped.

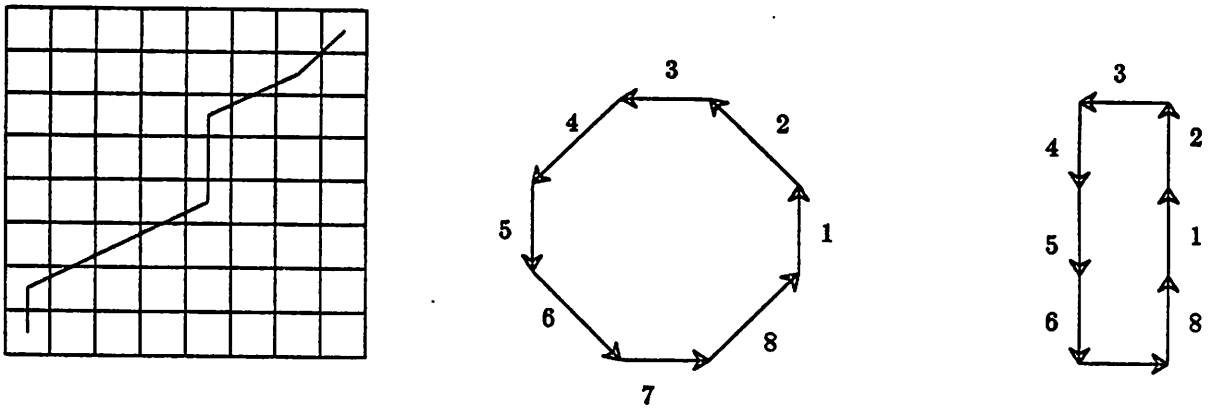


Figure 3.6 The warping path must be continuous.

(3) The path should start around (1,1) *and terminate around* (I,J). Fig. 3.7 shows a path which does not start from (1,1) and terminate at (I,J). The accumulated distance is unreasonable because the matching of sample points at the beginning and end of the symbols are skipped.

(4) The path should be close to the diagonal line. If two symbols are identical, the warping path should be the diagonal line in the distance matrix. Unless the distortion is quite bad, the optimal warping path should not deviate much from the diagonal line. If the path is allowed to

be far from the diagonal line, the distance between the two symbols in Fig. 3.8 would be 0.

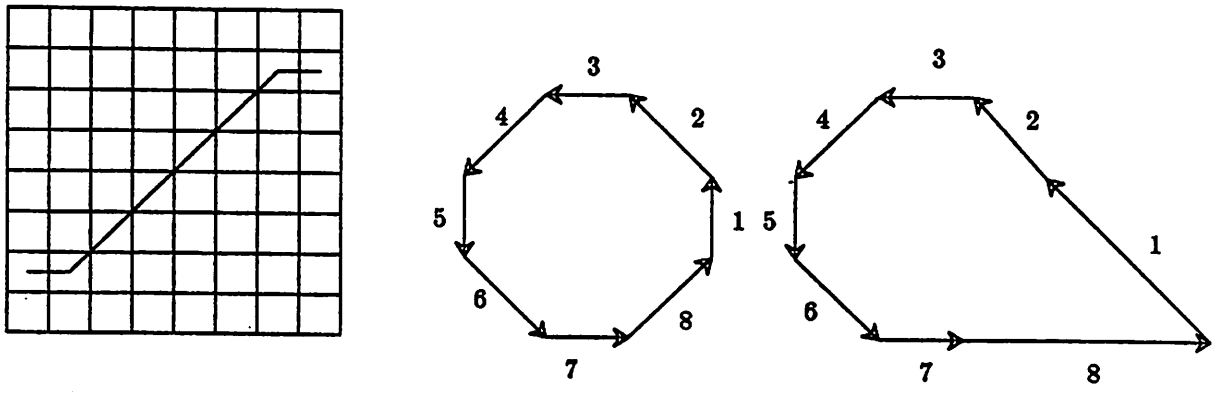


Figure 3.7 The endpoints of a warping path must be confined

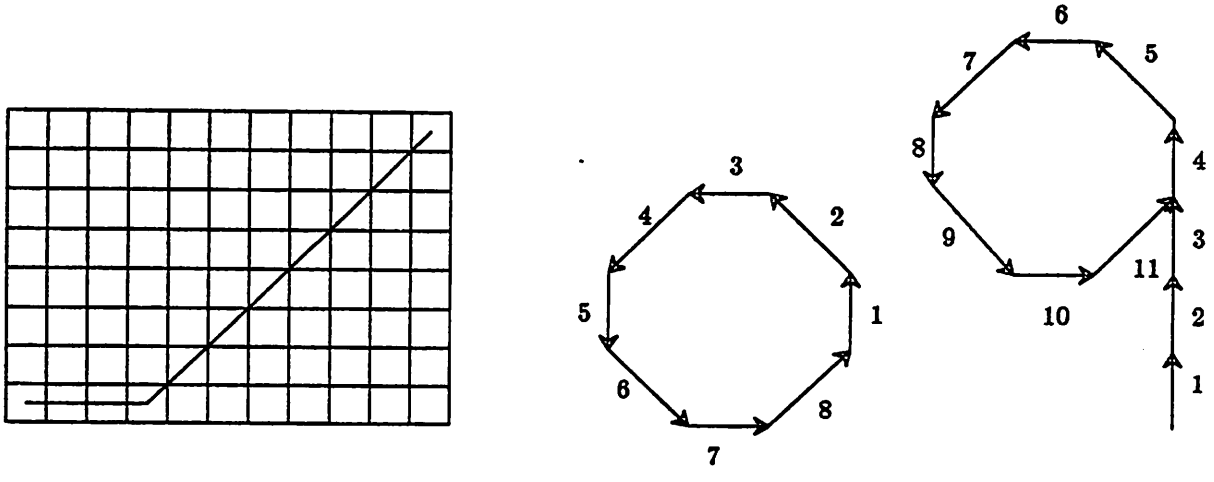


Figure 3.8 The warping path should be close to the diagonal line.

2.2. Diagonal warping path

Referring to Fig. 3.9, the most straight forward path which satisfies all of the requirements is the diagonal line, i.e.

$$D(T) = \sum_{i=1}^{I} d_{i,i\times\frac{J}{I}} .$$

This formula gives us a distance by comparing each unknown sample point with its linear counterpart in the template sequence. The advantage of using this path is that the number of local distance ($d_{i,j}$) computation and the accumulated distance ($D_{i,j}$) computation is linearly proportional to the number of sample points. The computational load is much less than that

encountered in the algorithms proposed in the next two sections. The disadvantage of using this path is, as illustrated in Fig. 2.14, that the alignment of the symbols is sometimes not linear, causing an unreasonable distance accumulation along the path.

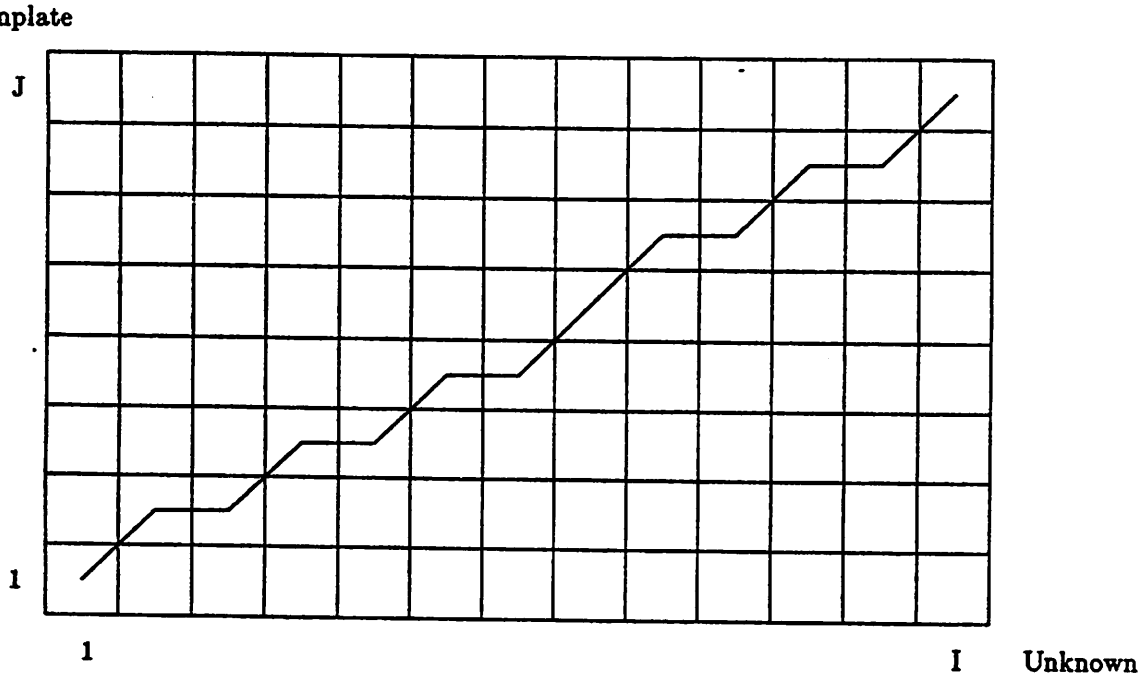


Figure 3.9 Diagonal matching path

2.3. Dynamic Time Warping (DTW) path

Define $D_{i,j}$ as the accumulated distance at (i,j). Since the warping path is required to be monotonically increasing and continuous, (i,j) can only be reached from $(i-1,j)$, $(i,j-1)$, or $(i-1,j-1)$. To make $D_{i,j}$ the minimum accumulated distance from (1,1) to (i,j), it should be obtained by

$$D_{i,j} = d_{i,j} + MIN(D_{i-1,j},D_{i-1,j-1},D_{i,j-1})$$

with the initial condition $D_{0,0} = 0$ and $D_{0,j} = D_{i,0} = \infty$, $1 \leq j \leq J$, $1 \leq i \leq I$.

This path, called *DTW* path, guarantees that $D(I,J)$ is the minimum distance from (1,1) to (I,J). It has been successfully applied in speech recognition.[2] Its advantage is that since the path is obtained by searching through all $I \times J$ elements in the distance matrix, the local misalignment is elegantly absorbed. There are also some disadvantages. The first disadvantage is that the *DTW* path is the path of the minimum accumulated distance. The use of this minimum

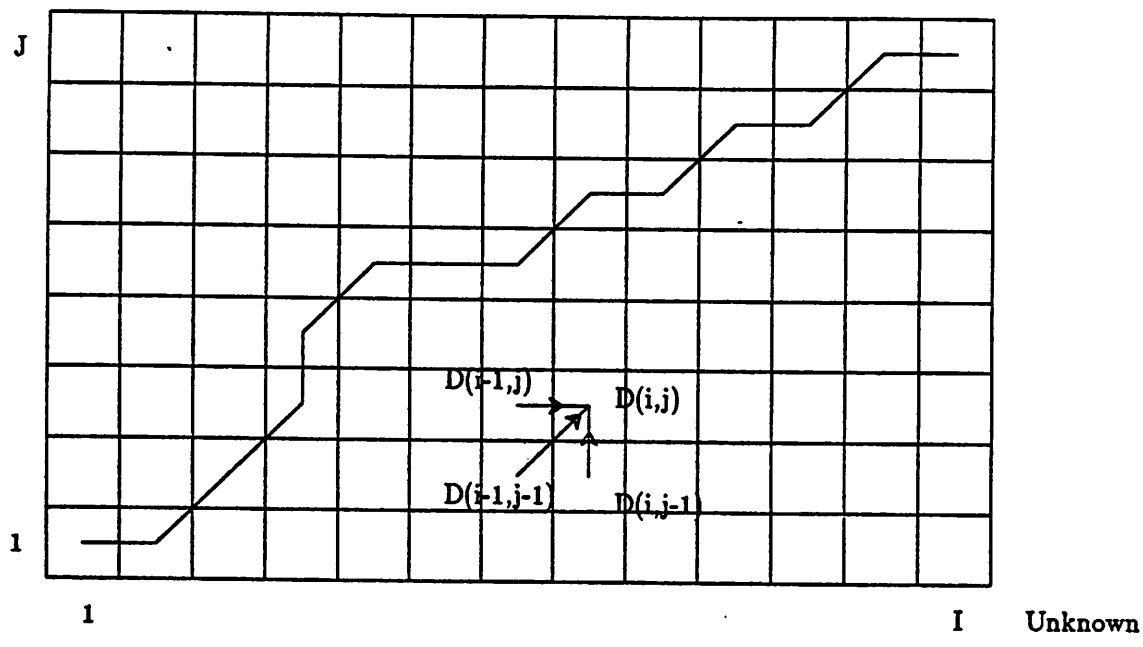


Figure 3.10 DTW matching path

accumulated distance may obscure the real difference between two symbols. For example, as illustrated in Fig. 3.8, the accumulated *DTW* distance of the slope codes of those two symbols is 0. Another disadvantage is having to compute all $d_{i,j}$ and $D_{i,j}$ for the path. The computational load is proportional to the square of the number of sample points.

The program for matching U with template T^k along the *DTW* path is as follows.

```
For i=1 to i=I
{  For j=1 to j=J^k
   {
      minimum_accumulated = MIN(D_j,D_{j-1},temp) ;
      D_{j-1} = temp ;
      temp = minimum_accumulated + d_{i,j} ;
   }
   D_{J^k} = temp ;
}
```

Notice that in the program, only an array of size J_k+1 is needed to store all of the intermediate accumulated distances, $D_{i,j}$.

2.4. Slope Constrained DTW (SC_DTW) path

To prevent the warping path from remaining horizontal or vertical too long, the path can simply be prevented from taking two consecutive vertical or horizontal steps as shown in Fig.

3.11. Let $f_{i,j}$ denote how the optimal path to (i,j) is reached. The $D_{i,j}$ on the *SC_DTW* path is obtained by

$$D_{i,j} = d_{i,j} + accu$$

$$accu = \begin{cases} MIN(D_{i,j-1}, D_{i-1,j-1}, D_{i-1,j}) & f_{i-1,j} \neq \rightarrow,\ f_{i,j-1} \neq \uparrow \\ MIN(D_{i,j-1}, D_{i-1,j-1}) & f_{i-1,j} = \rightarrow,\ f_{i,j-1} \neq \uparrow \\ MIN(D_{i-1,j-1}, D_{i-1,j}) & f_{i-1,j} \neq \rightarrow,\ f_{i,j-1} = \uparrow \\ D_{i-1,j-1} & f_{i-1,j} = \rightarrow,\ f_{i,j-1} = \uparrow \end{cases};$$

$$f_{i,j} = \begin{cases} \rightarrow & \text{if } accu = D_{i-1,j} \\ \uparrow & \text{if } accu = D_{i,j-1} \end{cases};$$

Template

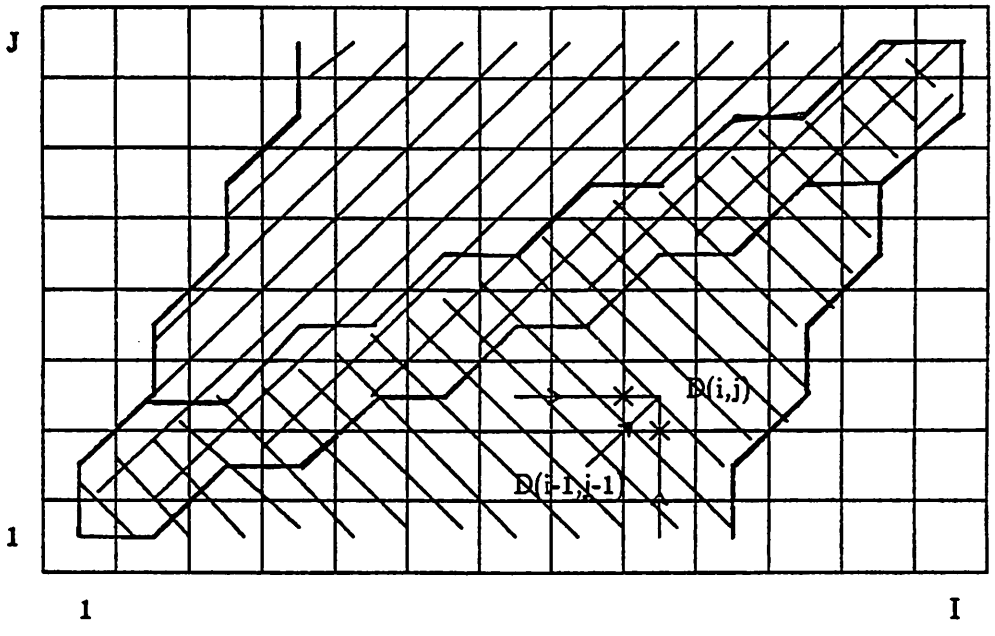


Figure 3.11 Slope constrained DTW

This slope constraint not only prohibits the path from staying horizontally or vertically too long, but also guarantees that the path not deviate far from the diagonal line. Fig. 3.11 shows if there is no two consecutive horizontal or vertical steps, the path will be confined in a region which is bounded by four lines with slopes 2 and 1/2. However, the *SC_DTW* path has some disadvantages. First, the *SC_DTW* sometimes forces the matching path to be non-optimal which results in errors. Second, if the number of sample points of one symbol is more than twice the number of sample points of the other symbol, the *SC_DTW* path can not reach (I,J) (see Fig. 3.12). This limitation makes the path unsuitable for symbols with large size differences.

Third, for each element (i,j) in the local distance matrix, extra computations are needed to determine $f_{i,j}$. Because the $f_{i,j}$ computation is proportional to the square of number of samples, it significantly increases the computational load.

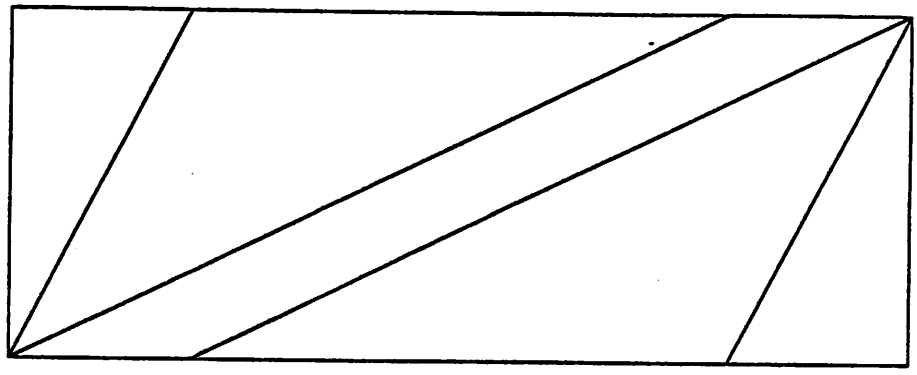


Figure 3.12 Limitation of SC_DTW

The implementation program of *SC_DTW* path is as follows.

```
For i=1 to i=I
{       For j=1 to j=J^k
        {       if (f_j ≠ → AND ftmp ≠ ↑)
                        accu = MIN(D_{j-1} , D_j , dtmp) ;
                if (f_j = → AND ftmp ≠ ↑)
                        accu = MIN(D_{i-1,j-1} , dtmp) ;
                if (f_j ≠ → AND ftmp = ↑)
                        accu = MIN(D_{j-1} , D_j ) ;
                if (f_j = → AND ftmp = ↑)
                        accu = D_{j-1} ;

                f_{j-1} = ftmp ;
                if(accu = D_j)    ftmp = → ;
                if(accu = dtmp) ftmp = ↑ ;
                if(accu = D_{j-1}) ftmp = / ;

                D_{j-1} = dtmp ;
                dtmp = accu + d_{i,j} ;
        }
        D_{J^k} = dtmp ;
}
```

Notice that only an array of size J_k+1 is needed to store all intermediate path history $f_{i,j}$.

3. Performance evaluation

In order to do a preliminary performance evaluation of various template matching algorithms, a small test data base was created. It consisted of 1500 writings from four subjects.

Each subject repeated 75 test symbols five times. The test symbols are:

$A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,$

$a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,$

$0,1,2,3,4,5,6,7,8,9,$

$+,-,*,/,=,>,<,(,),[,],\{,\}$.

They were chosen to represent symbols and strokes with different level of complexities. All the subjects were advised to keep the stroke sequence and stroke direction the same each time the symbol was written.

Since the data base is not very large, the performance comparisons based on it are subject to large statistical variations. The data base was used to only check whether there are major flaws in the algorithms.

In the first experiment, the performances of the following distance functions were compared.

Dx1: *local distance:* $dx_{i,j}$; *path: Diagonal ;*

Dy1: *local distance:* $dy_{i,j}$; *path: Diagonal ;*

Dp1: *local distance:* $dx_{i,j} + dy_{i,j}$; *path: Diagonal ;*

Ds1: *local distance:* $ds_{i,j}$; *path: Diagonal ;*

Dx2: *local distance:* $dx_{i,j}$; *path: DTW ;*

Dy2: *local distance:* $dy_{i,j}$; *path: DTW ;*

Dp2: *local distance:* $dx_{i,j} + dy_{i,j}$; *path: DTW ;*

Ds2: *local distance:* $ds_{i,j}$; *path: DTW ;*

Dx3: *local distance:* $dx_{i,j}$; *path: SC_DTW ;*

Dy3: *local distance:* $dy_{i,j}$; *path: SC_DTW ;*

Dp3: *local distance:* $dx_{i,j} + dy_{i,j}$; *path: SC_DTW ;*

Ds3: *local distance:* $ds_{i,j}$; *path: SC_DTW ;*

The following parameters were set the same for all these distance functions:

Number of sample points (I and J) *16,*

Width quantization levels (W): *16,*

Height quantization levels (H): *16,*

Slope quantization levels (Q): *16.*

Two rates were evaluated as performance indications. The first is the *recognition rate*. If the distance between the unknown symbol and its template is the smallest, it is counted in the recognition rate. The second is the *screen rate*. If the distance between the unknown symbol and its template is among the smallest three, it is counted in the screen rate.

Table 3.1 Performance of Diagonal warping path

	Dx1		Dy1		Dp1		Ds1	
	Rec	Scr	Rec	Scr	Rec	Scr	Rec	Scr
RWB	56.4	80.1	61.4	84.0	74.4	93.3	77.5	91.7
ENC	76.3	93.8	73.4	90.1	86.4	97.6	87.7	97.8
CCH	52.4	78.9	56.3	79.9	73.7	87.7	71.1	88.8
PYL	60.2	85.3	65.6	86.9	78.4	94.1	81.3	92.5
	61.3	84.6	64.2	85.2	78.2	93.1	79.4	92.5

Table 3.2 Performance of DTW

	Dx2		Dy2		Dp2		Ds2	
	Rec	Scr	Rec	Scr	Rec	Scr	Rec	Scr
RWB	46.2	68.7	56.1	80.3	90.9	97.0	89.8	96.8
ENC	61.0	86.1	65.2	84.2	95.4	99.2	96.0	99.4
CCH	43.8	69.4	51.6	77.4	86.9	94.1	87.7	95.2
PYL	45.4	73.0	61.3	86.9	93.0	98.6	93.3	98.4
	49.1	74.3	58.6	82.2	91.5	97.2	91.7	97.4

Table 3.3 Performance of SC_DTW

	Dx3		Dy3		Dp3		Ds3	
	Rec	Src	Rec	Src	Rec	Src	Rec	Src
RWB	55.1	77.1	65.3	86.9	89.8	97.3	89.8	98.1
ENC	75.2	92.8	73.5	89.6	95.7	99.2	97.0	99.7
CCH	53.5	78.7	61.6	83.7	85.6	94.4	88.0	94.9
PYL	57.9	83.2	70.5	91.7	93.8	98.9	92.8	98.1
	60.4	83.0	67.7	88.1	91.2	97.4	91.9	97.7

The experimental results are listed in Table 3.1, 3.2, 3.3. Several conclusions are obtained from the three tables.

(1) The *Dp* and *Ds* are much better than *Dx* and *Dy*. This is because both the *Dp* and *Dd* are derived from x and y coordinates. Using both features should be more accurate than using just one of them.

(2) The performances of *Dp* and *Ds* are similar.

(3) The *DTW* and *SC_DTW* distance functions are definitely better than *diagonal* path distance functions. The recognition errors of the *diagonal* distance functions confirm that misalignment of sample point really causes most of the errors.However, these errors are avoided by using the *DTW* and *SC_DTW* distance functions.

(4) The *SC_DTW* distance functions have essentially the same performance as the *DTW* distance functions.

3.1. Error analysis

From the errors made by *Dp*2 and *Ds*2 in last experiment, it was found there are five major causes. First, as shown in Fig. 3.13, many errors are caused by inconsistent *tails* or *hooks* at the start or end part of a stroke. It can be seen that the tails have serious impact on the normalized coordinates and hooks generate slope codes which have no counterpart in the template symbol.

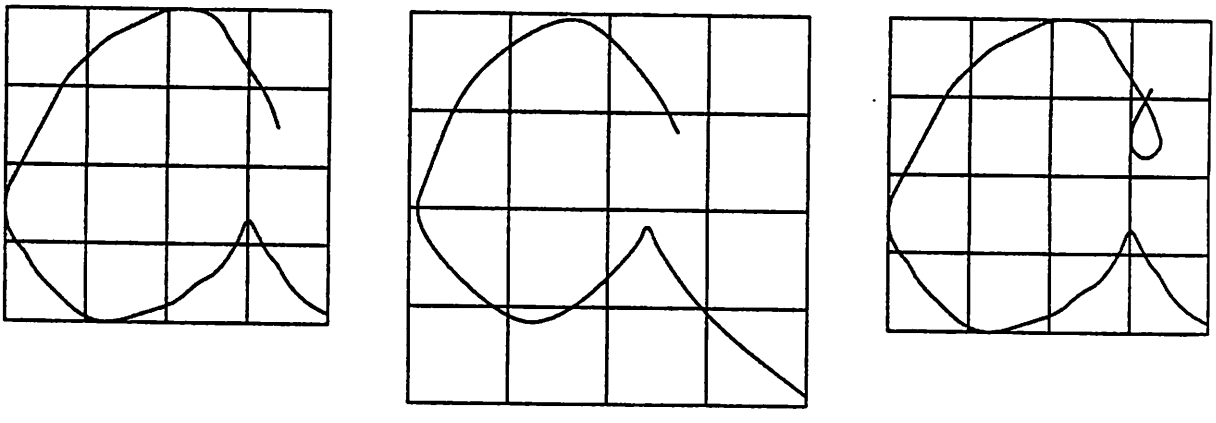


Figure 3.13 Symbols with stretches and hooks

Second, some errors are caused by symbols that are tilted to the left or right as shown in Fig. 3.14. The x,y code and slope code of a tilted symbol are obviously very different from a

normal symbol.

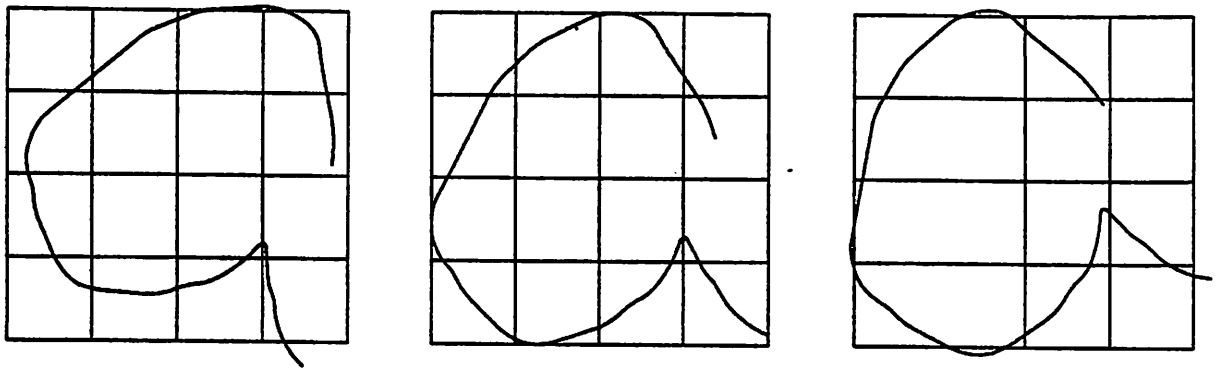


Figure 3.14 Tilted symbols

Third, each feature has its inherent weakness. As shown in Fig. 3.15, the quantized slope codes can not differentiate 'D' and 'P' well and the normalized x, y coordinates can not distinguish 'e' and 'l' well.

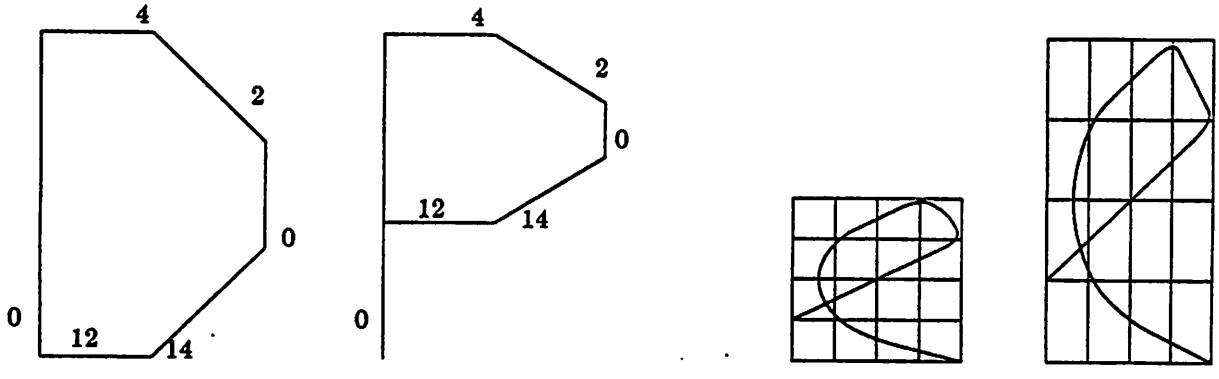


Figure 3.15 Limitations of recognition features

Fourth, as described in previous sections, the *DTW* path and *SC_DTW* path have their own inherent limitations.

Fifth, the distance accumulated from a critical segment is sometimes outweighed by distance accumulated outside the critical segment. This phenomena can be seen from the plots of local distances along the optimal warping path.[3] Fig. 3.16(a) shows the local distances of matching different occurrences of the same symbol. They are randomly around a low level. Fig. 3.16(b) shows the local distances of matching two quite different symbols. They are randomly around a high level. Fig. 3.16(c) shows the local distances of matching two quite similar

symbols. It can be seen most of the distances are around the low level and only a small portion of them are around the high level. This small portion is in fact the most critical segment to distinguish these two symbols. Unfortunately, it is not unusual to find that the distance accumulated from Fig. 3.16(a) is bigger than the distance accumulated from Fig. 3.16(c).

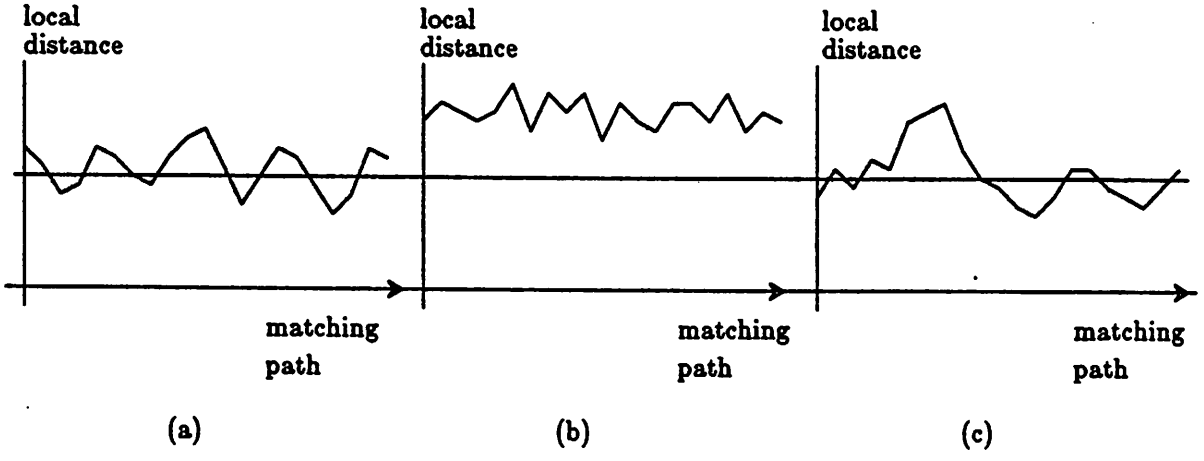


Figure 3.16 Local distances along the warping path

In this section, some improvements to the algorithm, which reduce the first two types of errors, are covered. Improvements which reduce the last three types of error will be discussed in next chapter.

3.2. Endpoint relaxation

To prevent the tails and hooks from causing errors, the endpoint requirements of the *DTW* path should be relaxed. Instead of forcing it to start from (1,1) and to terminate at (I,J), the $D_{i,0}$, $0 \le i \le \Delta$, and $D_{0,j}$, $0 \le j \le \Delta$ can be set at 0 and the final accumulated distance can be selected from $D_{i,J}$, $I-\Delta \le i \le I$ and $D_{I,j}$, $J-\Delta \le j \le J$. As shown in Fig. 3.17, if there is tail or hook within the Δ range at the start or end part of a writing, it will be ignored.

Experiments have been done with different relaxations. Table 3.4 lists the results of two of them,

Dd4: Dd2 with $\Delta = 2$,

Dp4: Dp2 with $\Delta = 2$,

Dd5: Dd2 with $\Delta = 4$,

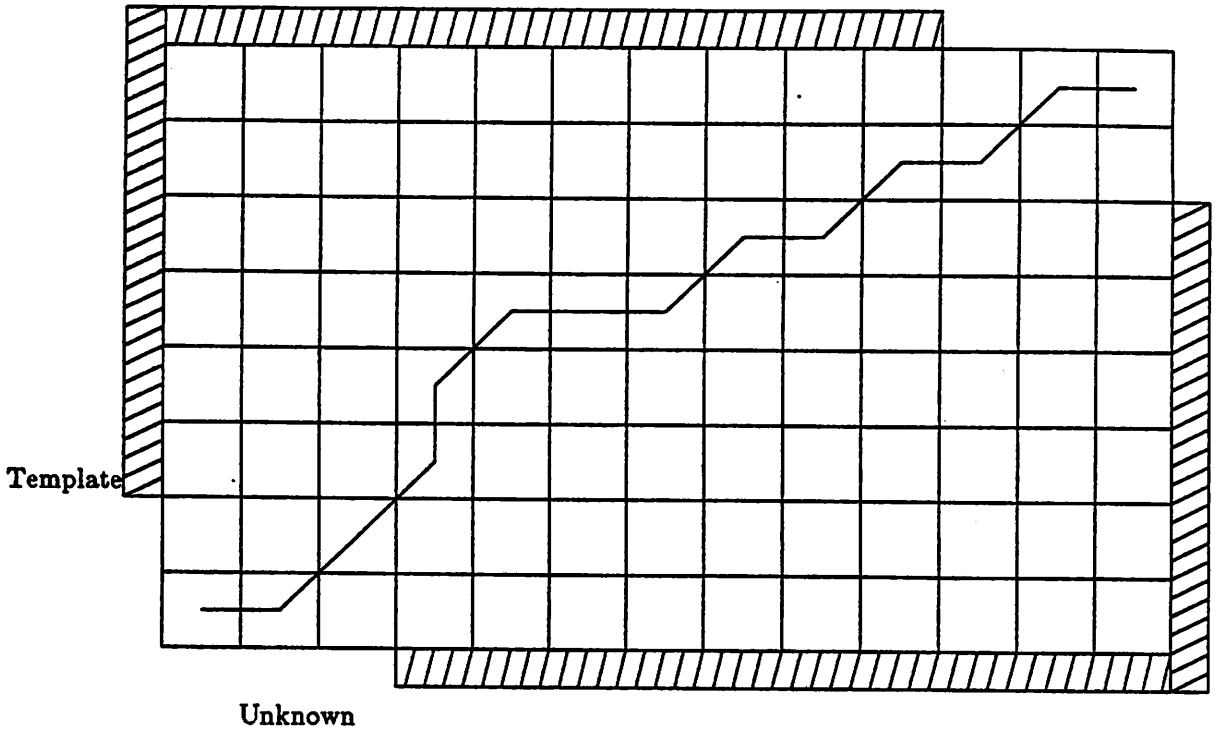


Figure 3.17 Endpoint Relaxation

Table 3.4 Endpoint relaxation

	Dd4		Dp4		Dd5		Dp5	
	Rec	Src	Rec	Src	Rec	Src	Rec	Src
RWB	89.0	96.2	90.6	96.8	89.3	95.4	89.6	97.3
ENC	95.4	99.2	94.9	98.6	96.2	99.4	95.2	98.4
CCH	86.6	94.4	86.6	94.6	89.0	94.6	85.6	94.9
PYL	92.5	98.1	92.5	98.4	92.2	97.8	93.3	98.6
	90.8	96.9	91.1	97.1	91.6	96.8	90.9	97.3

Dp5: Dp2 with $\Delta = 4$.

Unfortunately, there was no significant improvement. This is because, as illustrated in Fig. 3.18, there are some symbols whose primary differences are around the endpoints. Skipping these sensitive parts makes them even indistinguishable.

Since the tails and hooks could not be improved, users are advised to be careful around endpoints.

3.3. Curvature code

If a symbol is tilted, the curvature code, which is determined by the change of two consecu-

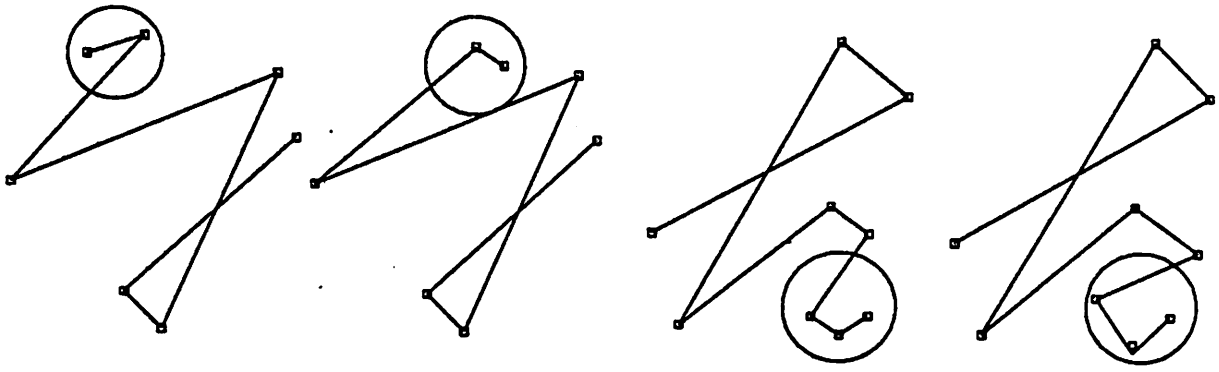


Figure 3.18 Limitations of endpoint relaxation

tive slope codes as shown in Fig. 3.19(a), is not changed. Fig. 3.19(b) illustrates an example. This fact suggests that maybe both slope code and curvature code should be used together in template matching to solve the tilt problem. In other words, the local distance $d_{i,j}$ should be defined as

$$d_{i,j} = W_s \times ds_{i,j} + W_c \times dc_{i,j} \ ,$$

in which the W_s and W_c are the weights of slope code and curvature code. The $dc_{i,j}$ is defined as

$$dc_{i,j} = MIN(U_{c,i} - T_{c,j},\ Q - U_{c,i} - T_{c,j}) \quad (Q \text{ is total quantization levels}) \ .$$

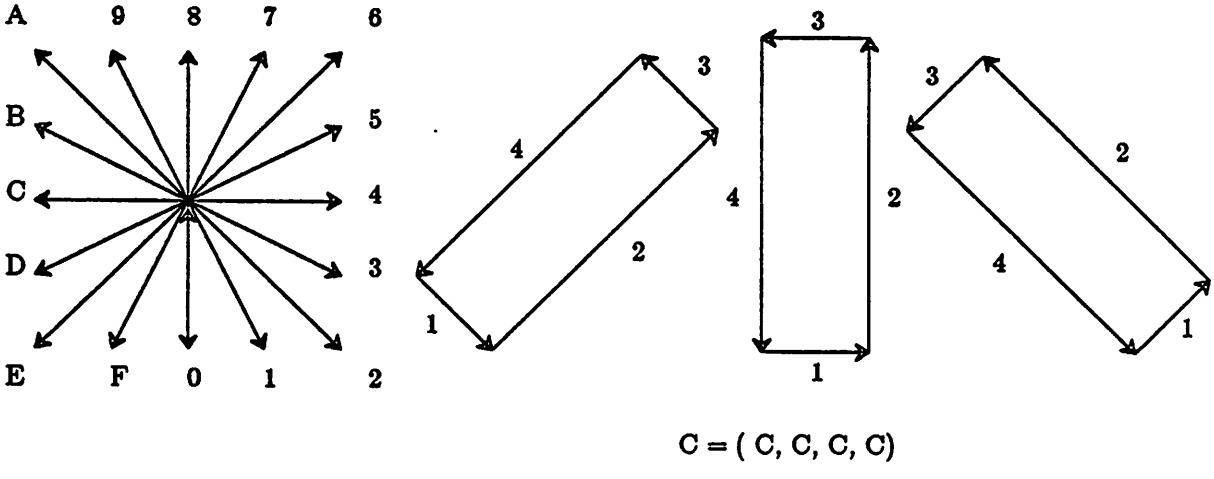


Figure 3.19 Curvature codes and tilted symbols

Experiments have been done with several weights. Table 3.5 lists the results of two of them,

Dc6: $W_s = W_c = 1$,

Dc7: $W_e = 2$, $W_c = 1$.

Unfortunately, the performance is much worse than *Dd2*.

Table 3.5 Using both slope and curvature codes				
	Dc6		Dc7	
	Rec	Top	Rec	Top
RWB	85.3	93.0	88.5	95.2
ENC	90.9	94.4	93.8	97.3
CCH	84.0	91.7	87.2	94.9
PYL	87.4	93.3	90.1	96.0
	86.9	93.1	89.9	95.8

The reason that the mixed local distance measure does not work well is because the curvature code, although it can accommodate large variations in the same symbol, has very poor capability to distinguish different symbols. As such, it deteriorates the performance of using slope code alone.

The tilt problem does not appear to be easily solved. The user should be careful not to tilt the symbols.

The test subjects were asked to avoid the tails, hooks, and tilts when writing. The recognition rate of the new data base improved as shown in Table 3.6.

Table 3.6 Performance without tilts								
	Dd2		Dp2		Dd3		Dp3	
	Rec	Scr	Rec	Scr	Rec	Scr	Rec	Scr
RWB	91.7	98.1	92.8	98.4	92.0	97.3	91.7	98.4
ENC	96.2	99.4	95.7	99.4	97.0	99.7	95.7	99.2
CCH	90.6	97.6	89.3	95.7	92.5	97.8	89.8	96.5
PYL	93.3	98.4	93.0	98.6	92.8	98.1	93.8	98.9
	92.9	98.3	92.7	98.0	93.5	98.2	92.7	98.2

3.4. Square local distances

To emphasize the critical segment of two symbols, the local distances could be squared, i.e. $dx_{i,j}^2 + dy_{i,j}^2$ for *Dp2* and $ds_{i,j}^2$ for *Dd2*, to penalize more heavily sample points which are quite different. Table 3.7 shows the results of using the squared local distance.

Dd8: Dd2 with local distance: $dd_{i,j}^2$;

Dp8: Dp2 with local distance: $dx_{i,j}^2 + dy_{i,j}^2$;

Dd9: Dd8 with local distance: $dd^2_{i,j}$;

Dp9: Dp8 with local distance: $dx^2_{i,j} + dy^2_{i,j}$;

Table 3.7 Square local distances

	Dd8		Dp8		Dd9		Dp9	
	Rec	Scr	Rec	Scr	Rec	Scr	Rec	Scr
RWB	92.0	98.1	92.8	98.6	92.2	97.0	91.7	98.4
ENC	96.2	99.4	95.7	99.4	97.0	99.4	95.4	99.2
CCH	91.2	97.8	89.8	96.5	92.8	98.1	90.4	97.3
PYL	93.6	98.1	93.3	98.9	93.0	97.8	94.1	98.9
	93.2	98.3	92.9	98.4	93.7	98.0	92.9	98.4

Unfortunately, using the squared local distances does not significantly change the rates. The reason is that the *DTW*, since it is an algorithm to find the minimum accumulated distance, is very good at avoiding the big distance elements in the local distance matrix.

4. Conclusion

In this chapter, it was described how to extract recognition features such that they are insensitive to fluctuations, symbol sizes, and writing speeds. Then, it was demonstrated that template matching along the *DTW* path and *SC_DTW* path provides better performance than matching along the *diagonal* path. It was also found that there is essentially no difference between using the *DTW* path and the *SC_DTW* path. The coordinate based distance function (i.e. *Dp2*) is as good as the slope based distance function (i.e. *Dd2*).

The recognition rate of template matching was not as high as desired. An effort was made to decrease recognition errors due to tails, hooks and tilts. However, it seems these errors could only be decreased with the user's cooperation. An effort was also made to give more weights to the critical segments. Unfortunately, the inherent *'go-for-the-minimum'* characteristics of the *DTW* still makes the results disappointing.

The screen rates from the experiments are all quite high. In next chapter, discussion will be continued to see how to incorporate the matching results of these distance functions for the final decision.

References

1. E. F. Yhap and E. C. Greanias, "An On-Line Chinese Character Recognition System," *IBM J. R & D*, pp. 187-195, May 1981.

2. L. Rabiner and S. Levinson, "Isolated and Connected Word Recognition - Theory and Selected Applications," *Trans. on Comm. IEEE*, vol. 29, no. 5, pp. 621-659, 1981.

3. L. Rabiner and J. Wilpon, "Isolated Word Recognition Using A Two-Pass Pattern Recognition Approach," *Proc. IEEE ICASSP*, pp. 724-727, 1981.

CHAPTER 4

DISAMBIGUATION

In Chapter 3, it was shown that although *DTW* template matching has very good screening performance, it does not distinguish ambiguous symbols well. An algorithm which can choose the correct symbol from the top runners of the template matchings is needed.

The most effective way to make the right decision is by checking some *ad hoc rules*. Fig. 4.1 illustrates several examples.[1,2] However, since these specialized rules have to be set by human being, this approach violates the design goal that the user should be able to easily train the system. It is preferable to have an algorithm which can automatically differentiate ambiguous symbols.

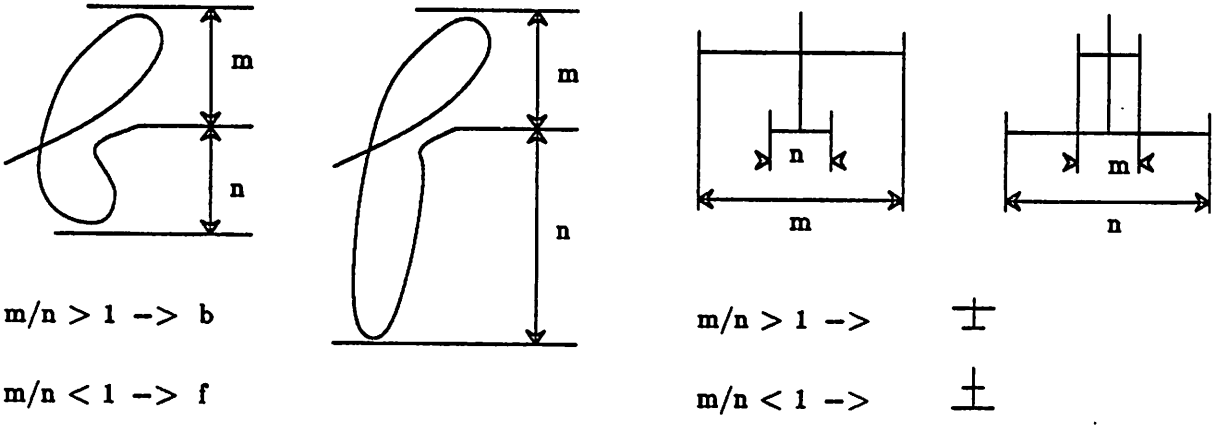


Figure 4.1 Ad hoc disambiguation rules

1. Clustering and template creation

In the algorithm presented in Chapter 3, a template is created directly from one training sample. In order to automatically obtain disambiguation rules for ambiguous templates, more than one training sample are needed for each template. If every training sample is used as a template, there will be too much redundancy in the templates because they may be very similar to each other. As such, an algorithm is needed to automatically group training samples into *clusters* and then create a template for each cluster.

Several algorithms have been proposed for this cluster seeking problem. Among them, the *UWA* (unsupervised clustering without averaging) algorithm, which is originally proposed for user-independent speech recognition, is most appealing.[3] It has two major advantages. First, it is a fully automatic procedure, that is, no human judgement is involved in the cluster seeking iteration. Second, from speech recognition experiments, it has been shown to converge quickly and work well.

1.1. UWA

Suppose there are n training samples, $\{t_1,t_2, \cdots ,t_n\}$, of a symbol. The distances between them are $(D(t_1,t_2),D(t_1,t_3), \cdots ,D(t_{n-1},t_{n-2}))$. Let $\Omega = \{t_1,t_2, \cdots ,t_n\}$. A clustering algorithm divides Ω into M disjoint clusters $\{\omega_1, \cdots \omega_M\}$ such that distances between a cluster center and all cluster members are less than threshold δ .

Several definitions have to be made before explaining the *UWA* algorithm. First, the *partial observation set*, Ω_{j+1}. Ω_{j+1} is the set of the training samples which have not been assigned to any cluster in $\omega_1,\omega_2, \cdots ,\omega_j$ during the cluster seeking iteration. i.e.

$$\Omega_{j+1} = \Omega - \bigcup_{i=1}^{j} \omega_i = \Omega_j - \omega_j \ .$$

Second, the *minimax* t_j of Ω_j. The t_j is defined as the training sample in Ω_j such that the maximum distance between it and any other training samples is the minimum, i.e.

$$t_j = MINIMAX(\Omega_j) = \underset{t_m \in \Omega_j}{ARGMIN} \underset{t_n \in \Omega_j}{MAX} D(t_m,t_n) \ .$$

The procedure of *UWA* algorithm is as follows.

```
UWA()
{
  Ω1 = Ω ;
  j = 1 ;
  While Ωj is not empty
  {
    ω0j = ∅ ;   /* ∅: Empty set */
    t1j = MINIMAX(Ωj) ;
    For every ti in Ωj
    {  If D(ti, t1j) ≤ δ)
          ti ∈ ω1j ;
    }
    k = 1 ;
```

```
        While ω_j^k is not the same as ω_j^{k-1}
        {
            k++ ;
            t_{j+1}^k = MINIMAX(ω_j^{k-1}) ;
            For every t_i in Ω_j
            {  If D(t_i, t_j^k) ≤ δ)
                    t_i ∈ ω_j^k ;
            }
        }
        ω_j = ω_j^k ;
        Ω_{j+1} = Ω_j − ω_j ;
        j++ ;
    }
}
```

Fig. 4.2 illustrates how this algorithm clusters the 9 training samples into 3 clusters.

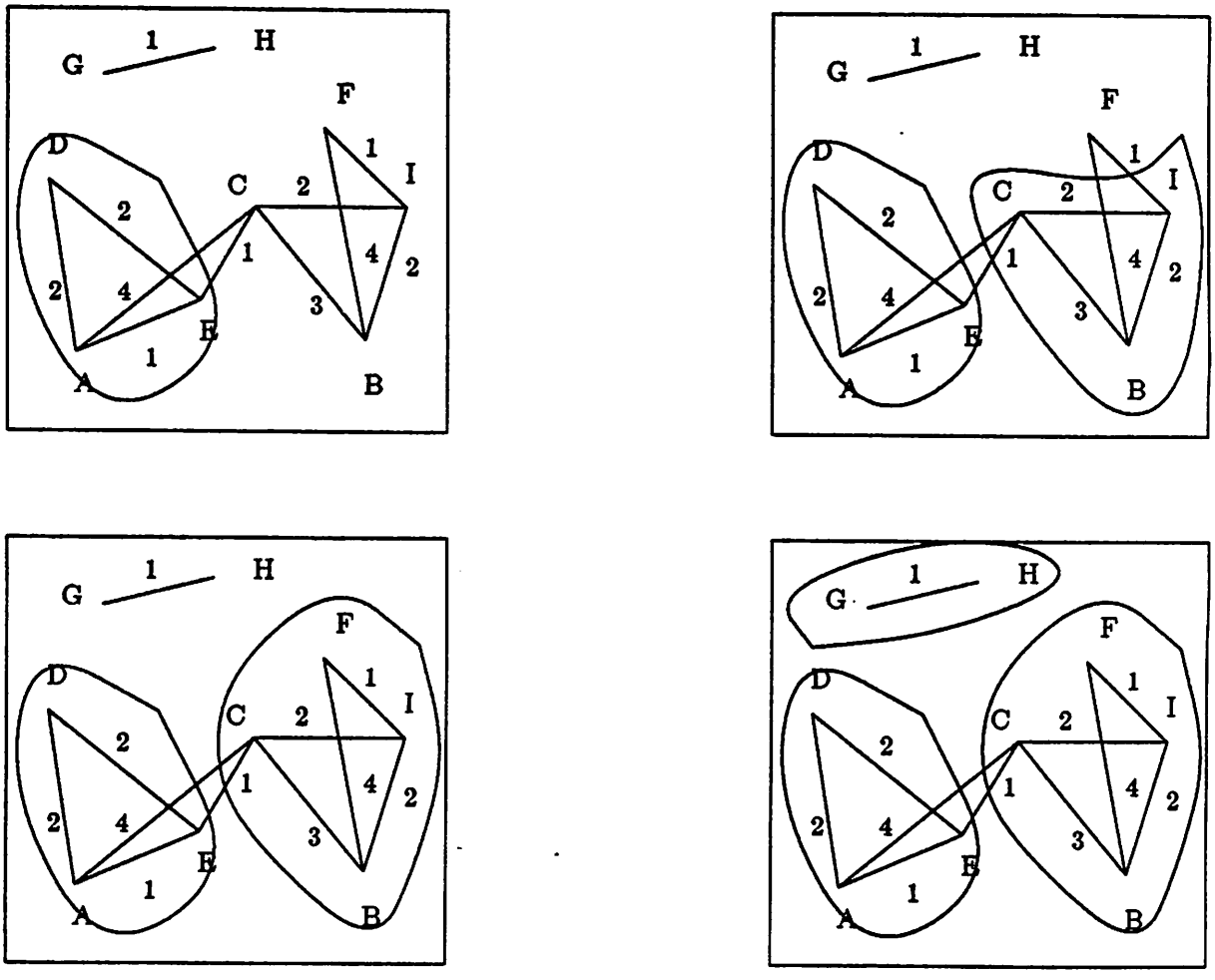


Figure 4.2 UWA clustering

1.2. Creating template

There are two ways to create the template of each cluster. First, the minimax of each cluster can be taken as the template. Second, the cluster members can be averaged to create a *synthesized* template. The first method is very straight forward. The second method, however, needs special attention during synthesis.

Suppose $(t_1, t_2, \cdots, t_P)$ are members in the cluster ω, t_m is the final minimax of ω and T is the template going to be created. As seen in Fig. 4.3, the template of the cluster is synthesized as follows.

The procedure starts by obtaining the *DTW* warping paths between t_m and all t_p, $1 \leq t \leq P$. To do this, during the matching procedure, all $f_{i,j}$'s, where $f_{i,j}$ is the previous neighboring point on the optimal path to (i,j), are stored in the same manner as in the *SC_DTW* discussed in chapter 3. Then, from $f_{i,j}$, the optimal path between t_m and each t_p can be obtained by back-tracking from $f_{I,J}$.

With these matching paths, the *ith* sample point of T is the mean of all sample points aligned with the *ith* element of t_m, i.e.

$$T_i = \frac{1}{\#\ of\ samples\ aligned\ with\ t_{m,i}} \sum_{\{(p,j):\, t_{p,j}\ is\ aligned\ with\ t_{m,i}\}} t_{p,j} \ .$$

If the x,y codes are used, the average is the arithmetic average of the coordinates. If the slope code is used, the average is derived through vector addition as illustrated in Fig. 4.4.

1.3. Performance Evaluation

To evaluate the performance of the *UWA* algorithm, it was applied to the test data base. For each symbol, 4 samples of a symbol were used as training samples and the other one was used as the unknown. Since clustering provides templates which cover wider variations, the recognition rates are increased as shown in Table 4.1 and Table 4.2 for 3 different clustering thresholds (δ). The number of templates created under each threshold are also listed in the table. Comparing these results with those of Chapter 3, it can be seen that the templates

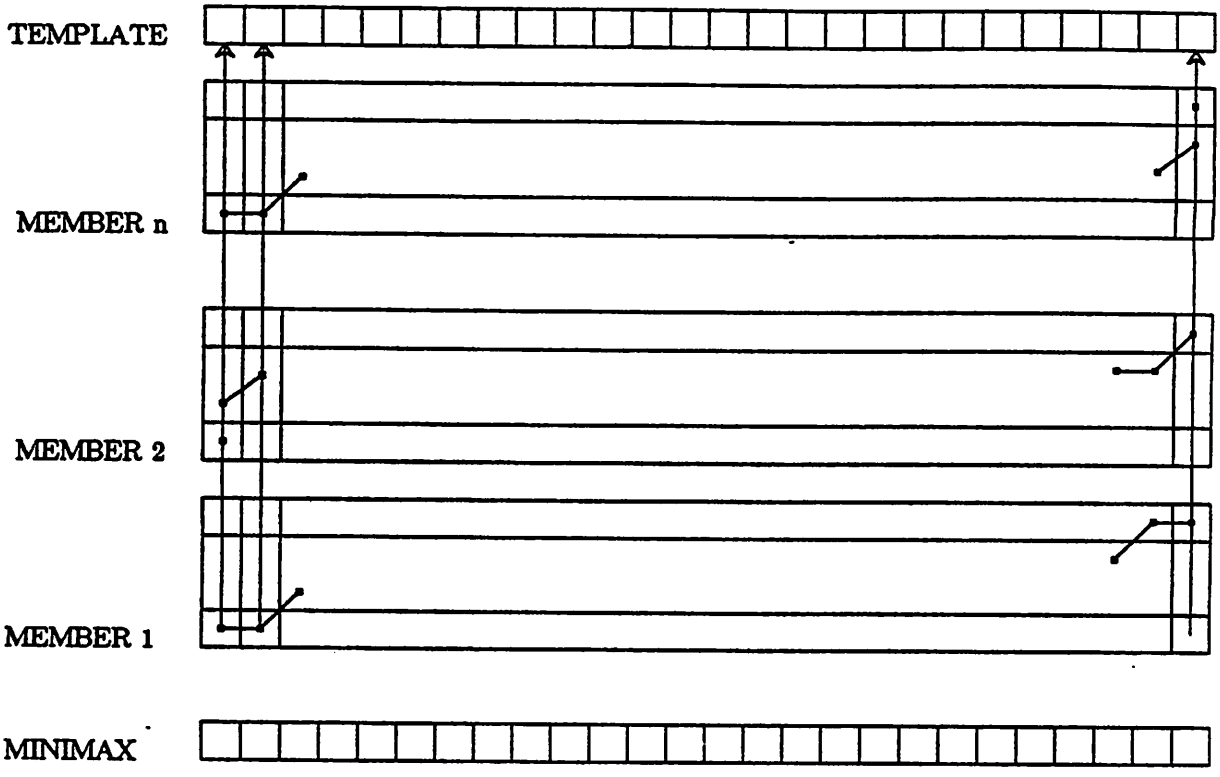


Figure 4.3 Synthesizing template

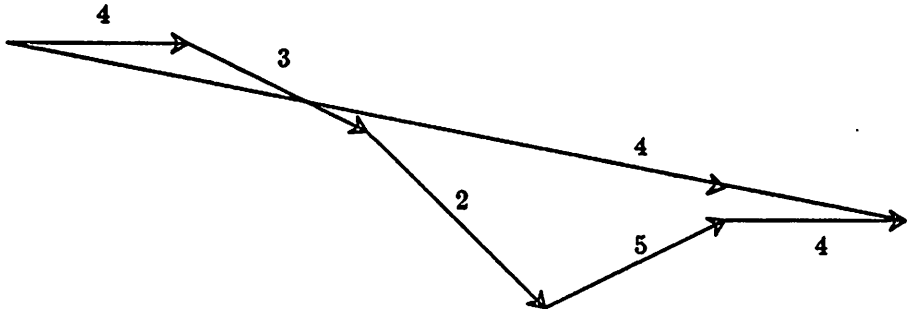


Figure 4.4 Average slope codes

created from multiple training samples significantly improve the recognition rates. The smaller the threshold, the more templates are created, and the better the recognition rates are achieved.

Table 4.3 is a tabulation of the recognition rates that were obtained with the synthesized templates. It can be seen that the recognition rates based on synthesized template are almost the same as the rates based on the minimax template. Therefore, it is not worthwhile to synthesize the templates.

Table 4.1 Number of templates and recognition rates for $Dd2$						
	δ=0.5		δ=1.0		δ=1.5	
	templates	rate	templates	rate	templates	rate
RWB	928	93.3	492	93.0	403	92.2
ENC	990	97.6	426	97.3	381	96.8
CCH	1169	93.3	604	92.5	420	91.7
PYL	988	94.6	434	94.9	389	94.4
	1018	94.7	489	94.4	398	93.7

Table 4.2 Number of templates and recognition rates for $Dp2$						
	δ=2.0		δ=2.5		δ=3.0	
	templates	rate	templates	rate	templates	rate
RWB	874	93.6	455	93.3	387	93.0
ENC	863	97.3	413	97.0	398	96.5
CCH	909	93.0	511	92.2	416	91.7
PYL	871	94.6	426	94.4	409	93.8
	879	94.6	451	94.2	402	93.7

Table 4.3 Recognition with synthesized templates				
	Dd12(δ=1.0)		Dp12(δ=2.5)	
	templates	rate	templates	rate
RWB	492	93.3	455	93.0
ENC	426	97.3	413	97.0
CCH	604	92.2	511	91.7
PYL	434	95.2	426	94.9
	489	94.5	451	94.1

2. Ambiguity checking

For two clusters $\omega_1 = \{ t_1^1, t_2^1, \ldots, t_{p_1}^1 \}$ and $\omega_2 = \{ t_1^2, t_2^2, \ldots, t_{p_2}^2 \}$. Let t_1 denote the template created from ω_1, t_2 denote the template created from ω_2, and $D_{a,b}$ denote the matching distance between training sample a and b. Fig. 4.5 is a plot with $D_{t_1,t}$ as the horizontal axis and $D_{t_2,t}$ as the vertical axis. If the distance function can differentiate between these two clusters well, the distribution should look like Fig. 4.5(a). Members of each cluster are far away from the 45° line. However, if the distance function can not separate the two clusters well, the distribution is like Fig. 4.5(b). Some members are around the 45° line and some may even cross the line. If a training sample crosses the 45° line, it means the distance between the training sample with its own template is even larger than the distance between it and the other template. As such, the *ambiguity* of two templates can be decided by counting how many training samples are in the

shaded "marginal" area. If the count is over a threshold, the two templates are determined to be ambiguous.

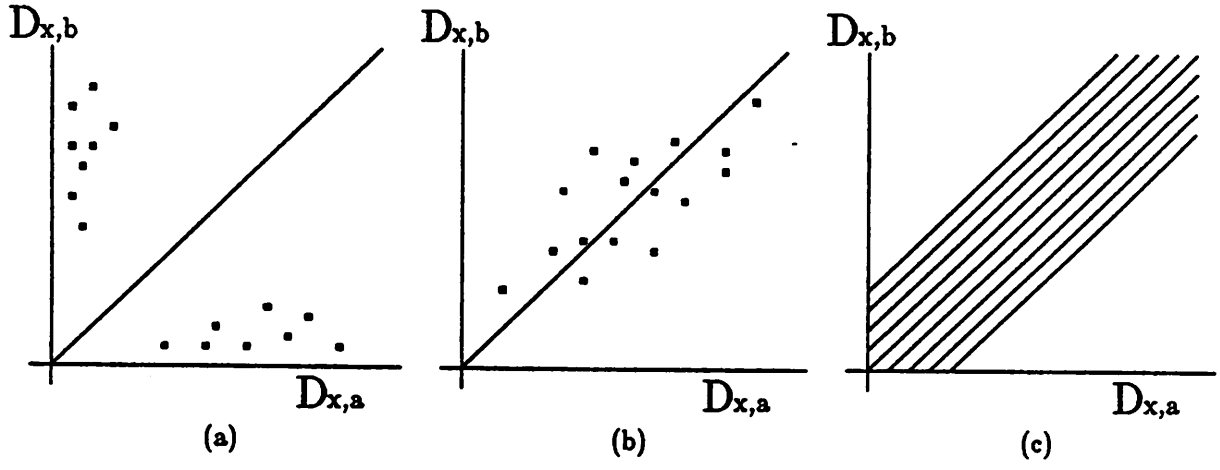


Figure 4.5 Determine whether two templates are ambiguous

3. Disambiguation techniques

3.1. Two-Pass *DTW*

As illustrated in Fig. 3.17, the deterministic distance of two ambiguous symbols is always accumulated from a very short critical segment. More penalty can be put on these segments by multiplying the local distance by a statistical weight. In speech recognition, it has been proposed to evaluate the distance by this formula:[4]

$$D(T) = \frac{\sum_{(i,j)\ on\ warping\ path} W_i \times d_{i,j}}{\sum_{(i,j)\ on\ warping\ path} W_i} .$$

For different ambiguous symbol pairs, the critical segments are obviously different. As such, the weights must be pair-dependent. For symbol T^a and T^b, one intuitive choice of W_i for this pair is

$$W_i^{a,b} = \frac{d_i^{a,a} + d_i^{b,b} - d_i^{a,b} - d_i^{b,a}}{\hat{\sigma}_i^{a,a} + \hat{\sigma}_i^{b,b} + \hat{\sigma}_i^{a,b} + \hat{\sigma}_i^{b,a}} .$$

The $d_i^{a,b}$ is the average and the $\hat{\sigma}_i^{a,b}$ is the variance of distances obtained from all the elements of the T^b training samples which match with the *ith* element in T^a and all the elements of the T^a training samples which match with the *ith* element in T^b. (These elements are in fact the same

as the elements used to synthesize the *ith* element of the template as illustrated in Fig 4.3.)

To use the weighted *DTW* for disambiguation, in the training phase, the symbols are first divided into *equivalent classes* according to the ambiguity. The weights of every symbol pair in each equivalent class are then determined. In the recognition phase, the unweighted template matching is performed as before, but the result is only used to decide which class the symbol is in. Then, all pairwise weighted *DTW* distance, given by:

$$D(T^a,T^b) = \frac{\sum_{(i,j)\ on\ warping\ path} W_i^{a,b} \times d_{i,j}^a}{\sum_{(i,j)\ on\ warping\ path} W_i^{a,b}} ,$$

are evaluated. Then, the total weighted *DTW* distance for each symbol, which is defined as:

$$D(T^a) = \sum_{b,b\neq a} D(T^a,T^b) ,$$

is calculated. The unknown is recognized as the symbol which has the minimum total weighted *DTW* distance.

This technique has worked very well for speech recognition. However, to use it for *OHR*, it was found that a substantial number of training samples were required to make a good estimation of $W_i^{a,b}$. Since it is desired to use an algorithm which can "learn" a new symbol after a short training period, this unweighted/weighted two-pass *DTW* algorithm, although it would be the best way to fix the inherent *DTW* weakness, is not suitable.

3.2. Prematching and Postmatching

From the recognition errors of *Dp*2 and *Dd*2, it was found that they seldom make the same mistake. For example, the '*D*' and '*P*' is often an error of *Dd*2, but *Dp*2 can differentiate them well. The '1' and '/' is often an error of *Dp*2, but *Dd*2 can differentiate them well. This fact suggests that it may be possible to have the two distance functions correct the errors of the other.

This idea was implemented by a prematching/postmatching scheme. One distance function is used for *prematching*. The top candidates, which are defined as templates whose distances are within a specified threshold δ from the minimum, are sent to the other distance function for

postmatching. The final decision is determined by the postmatching.

Table 4.5 is a list of the experimental results of the scheme under two matching thresholds (δ). In it, "R/W" means "(number of right corrections)/(number of wrong corrections)". "Right correction" means the top candidate from prematching is wrong but postmatching corrects it. "Wrong correction" means the top candidate from prematching is correct but the postmatching reverses it. "Rec" is the final recognition rate.

Table 4.5 Postmatching

	Dd2->Dp2			Dp2->Dd2		
	Scr	R/W	Rec	Scr	R/W	Rec
RWB	98.4	34/ 27	95.2	98.4	30/ 23	94.9
ENC	99.4	16/ 10	98.9	99.7	11/ 6	98.4
CCH	98.1	57/ 49	94.4	97.8	46/ 38	93.8
PYL	99.4	48/ 43	96.5	99.4	44/ 39	96.2
	98.8	155/129	96.2	98.8	131/106	95.8

Comparing Table 4.5 with Table 4.1 and 4.2, it is apparent that the new scheme results in a significantly better recognition rate.

3.3. Disambiguation function

3.3.1. What is disambiguation function

Fig. 4.6 illustrates some ambiguous template pairs. By visual observation, one may intuitively guess that if the slope sequence is used to differentiate (1, /), the accuracy should be better than using the x or y sequence. If the y sequence of the second stroke is used to differentiate (D, P), the accuracy should be better than when using slope sequences or x sequences. If the x sequence of the third stroke is used to differentiate (A, *), the accuracy should be better than when using any other feature sequence of any stroke.

These intuitive guesses imply one "fact": for two ambiguous symbols, at least one feature sequence of one stroke, can always differentiate them well. As such, instead of using $Dd2$ or $Dp2$ as general postmatching function, it would be better to choose one of the $Dx2$, $Dy2$ or $Dd2$ as the postmatching function specific to each ambiguous symbol pair.

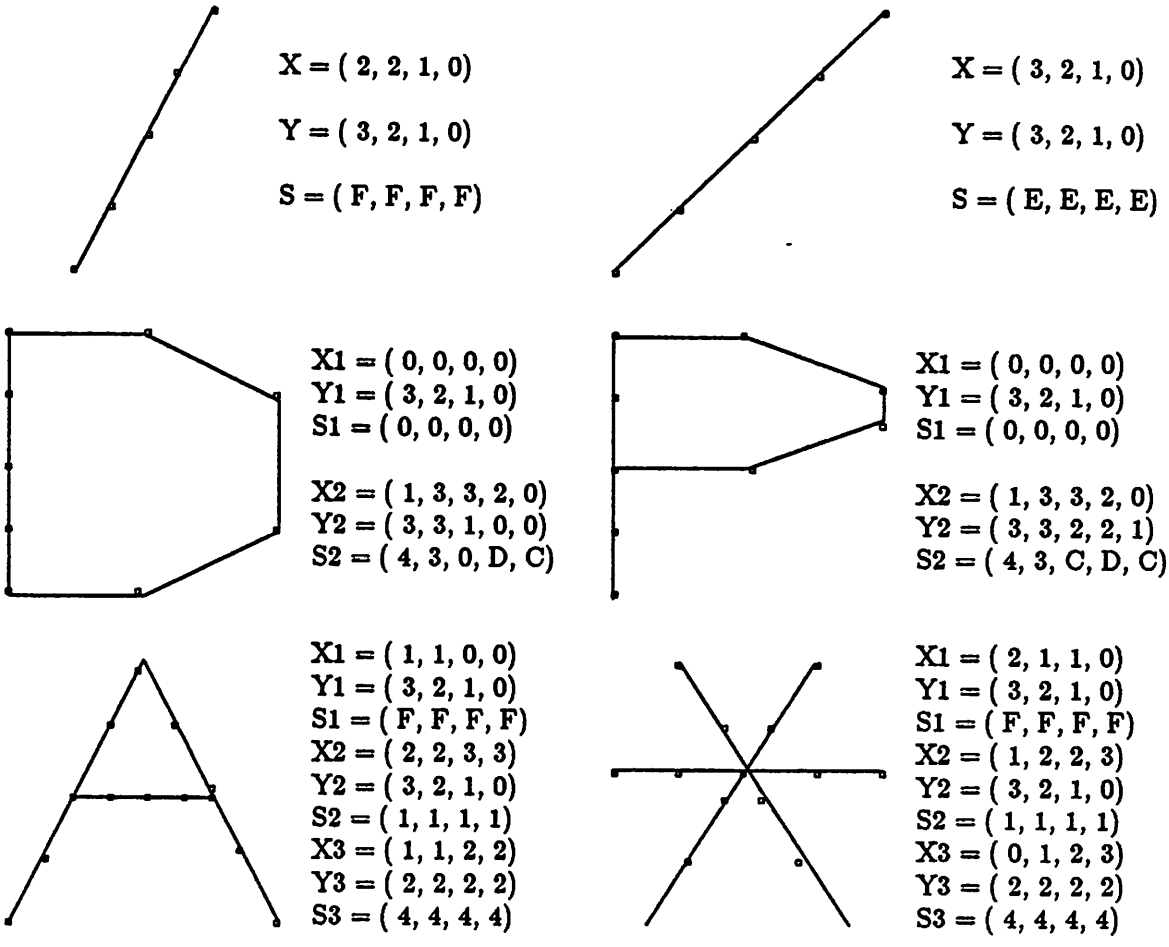


Figure 4.6 Intuitive disambiguation function

With this in mind, in the training phase, an algorithm is needed to decide which stroke is the critical stroke and which feature is the disambiguation key. In the recognition phase, an algorithm is needed to use these disambiguation keys for the final decision.

3.3.2. How to use disambiguation functions

Suppose the disambiguation function $F^{a,b}$ is obtained for template T^a and T^b in the training phase. The final decision in the recognition phase can be made based on the a *voting* of these functions. The way it works is as follows. After template matching, suppose both T^a and T^b are in the top candidates. Let $D_{F^{a,b}}(T^a)$ denote the distance between the unknown and template T^a when $F^{a,b}$ is applied. If $D_{F^{a,b}}(T^a) < D_{F^{a,b}}(T^b)$, T^a gets one vote. This test is performed on all of the pairs of the top candidates from template matching. The one which gets the most votes

is called the final winner.

3.3.3. How to determine disambiguation function

Suppose w_i and w_j are two ambiguous clusters. Let D^s denote the distance function D operated on the sth stroke and S denote the number of strokes of the two symbols. The candidates for disambiguation function are $Ds2^s$, $Dx2^s$, $Dy2^s$, $1 \leq s \leq S$. (The definitions of *Ds2, Dx2, and Dy2* are the same as in Chapter 3.) To select the disambiguation function from $Dd2^s$, $Dx2^s$, $Dy2^s$, the cluster members have to be used. Following are two algorithms.

3.3.3.1. By quality factor

For cluster pair w_i and w_j and distance function D, Fig. 4.7 shows a plot of the distribution of *intra-distance,* i.e. D_{t,t_i}, $t \in w_i$, D_{t,t_j}, $t \in w_j$, and *inter-distance,* i.e. D_{t,t_j}, $t \in w_i$, D_{t,t_i}, $t \in w_j$. If there are enough members in each cluster, the distribution of intra-distance and inter-distance should be Gaussian as illustrated. If D can differentiate these two clusters well, the two distributions will not overlap significantly as shown in Fig. 4.7(a). Otherwise, the two distributions will have excessive overlap as shown in Fig. 4.7(b). From this illustration, an intuitive *quality factor,* which represents the differentiating capability of D over ω_i and ω_j, is defined as

$$Q(D) = \frac{\mu_{inter} - \mu_{intra}}{[\sigma^2_{inter} + \sigma^2_{intra}]^{1/2}} .$$

In the definition,

μ_{intra} := average intra-distance,

μ_{inter} := average inter-distance,

σ^2_{intra} := variance of intra-distance,

σ^2_{inter} := variance of inter-distance.

The disambiguation function of ω_i and ω_j is determined by comparing the quality factors of functions $Dd2^s$, $Dx2^s$, $Dy2^s$. The one with the best quality factor is chosen as the disambiguation function.

However, applying this algorithm to our test data base did not produce good results. The

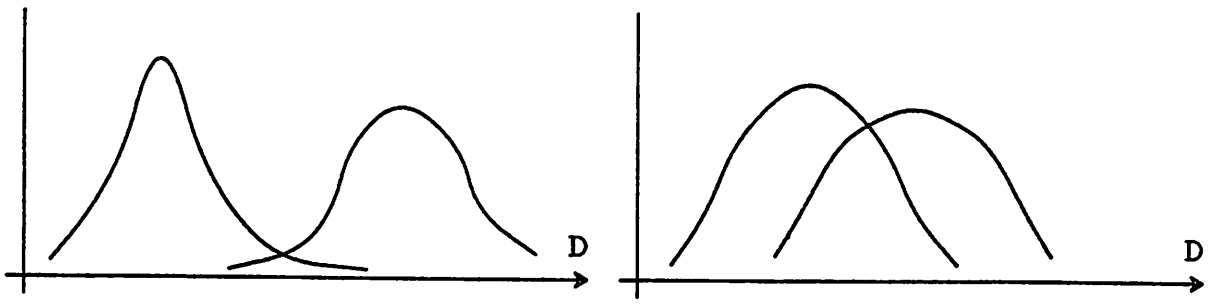


Figure 4.7 Inter- and intra-distance distribution

reason is the same as for the weighted DTW algorithm: it requires quite a lot training samples to obtain an accurate estimation of μ_{inter}, μ_{intra}, σ_{inter} and σ_{intra}. In order not to make the training tedious, this algorithm is not suitable for implementation.

3.3.3.2. By distance distribution

For cluster pair (w_i, w_j), it is not hard to tell whether the distance function D can differentiate them well by plotting $(D_{t,\mathcal{C}_i}, D_{t,\mathcal{C}_j})$ for all $t \in U_i \cup U_j$. Fig. 4.8 illustrates the results of applying D_x^1, D_y^1, D_s^1, D_x^2, D_y^2, D_s^2 over two symbols 'D' and 'P'. It is obvious the D_y^2, i.e. the distance of the y sequence of the second stroke, separates the two symbols better than all other functions.

Based on this concept, the disambiguation function can be selected by the following algorithm. For each distance function, the number of cluster members which cross the 45° line is counted and compared. Because crossing over the 45° line indicates that a recognition error occurs, the function which has the least number is chosen as the disambiguation function. If more than one distance function have the least number of errors, their inter-distance to intra-distance ratio, i.e. $\dfrac{\sum_{t \in w_j} D_{t,\mathcal{C}_i} + \sum_{t \in w_i} D_{t,\mathcal{C}_j}}{\sum_{t \in w_i} D_{t,\mathcal{C}_i} + \sum_{t \in w_j} D_{t,\mathcal{C}_j}}$ are compared. The one which has the largest ratio is chosen as the disambiguation function.

Experimental results showed that the disambiguation functions obtained by this distance

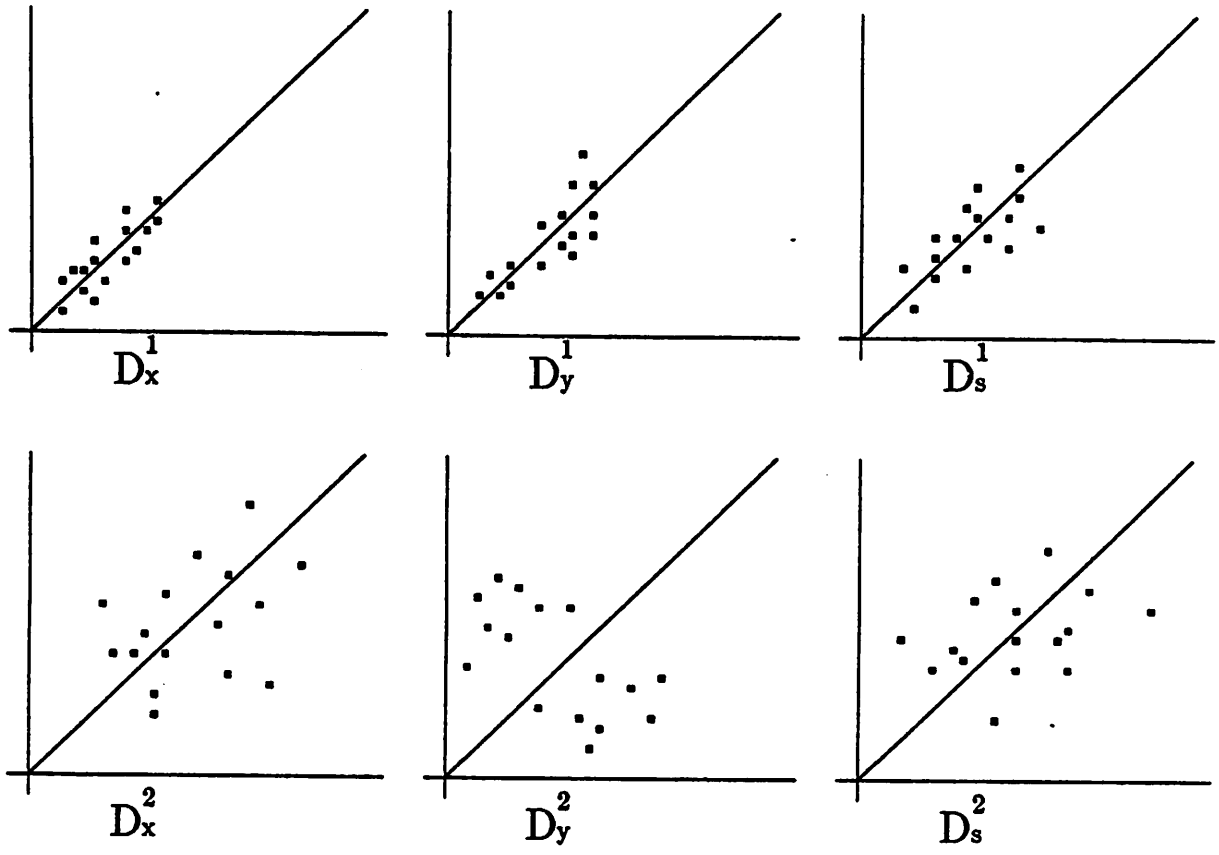


Figure 4.8 Determine disambiguation function from distance distribution

distribution algorithm are very close to those intuitive guesses as illustrated in Fig. 4.6. The recognition rates obtained when using these disambiguation functions are listed in Table 4.6. Comparing with those in Table 4.5, it can be seen that the ratio of right correction to wrong correction improved. The recognition rates are also significantly better.

Table 4.6 Performances of disambiguation functions

T.M.	Dd2		Dp2	
	R/W	Rec	R/W	Rec
RWB	16/ 3	96.8	17/ 5	96.2
ENC	7/ 0	99.2	10/ 2	99.2
CCH	19/ 6	95.7	21/ 9	94.9
PYL	16/ 4	98.4	16/ 7	97.3
	58/ 13	97.5	64/ 23	96.9

3.3.3.3. Using SC_DTW path and diagonal path for disambiguation

In the disambiguation scheme, the candidates passed to postmatching should be very similar to each other. As such, since *SC_DTW* path and *diagonal* path are closer to the diagonal line

than the *DTW* path, they may have better discriminating capability. Experiments have been done to use the *SC_DTW* distance functions, i.e. $Dd3^e$, $Dx3^e$, $Dy3^e$, and the ***diagonal*** distance functions, i.e. $Dd1^e$, $Dx1^e$, $Dy1^e$, as disambiguation functions. The results listed in the Table 4.7 show that there is no essential difference between operation with the *DTW* functions and *SC_DTW* functions. Recognition using the ***diagonal*** path functions is slightly inferior.

Table 4.7 SC_DTW and Diagonal path for disambiguation

	SC_DTW		Diagonal	
	R/W	Rec	R/W	Rec
RWB	17/ 2	97.3	16/ 3	96.8
ENC	7/ 0	99.2	6/ 1	98.6
CCH	20/ 4	96.5	18/ 5	95.7
PYL	16/ 4	98.4	15/ 4	98.1
	60/ 10	97.8	55/ 13	97.3

4. Conclusion

In this chapter, a clustering algorithm was first developed to automatically group training samples into clusters and create a template for each cluster. These cluster members are then used to determine whether two templates are ambiguous.

It has been tried to use weighted *DTW* to fix the inherent *DTW* errors. However, because this algorithm requires quite a lot training samples to obtain the weights, it is not practical. A postmatching scheme was then used for disambiguation. Although the recognition rate is improved, it would be better if the postmatching function can be specific to each ambiguous pair.

To obtain the disambiguation function specific to each ambiguous pair, an algorithm was developed to compare the differentiating capability of various distance functions. The one with the best performance is selected. Simulation results showed that these disambiguation functions are very close to those chosen by intuition. They also result in significantly improved recognition rates.

References

1. K. Ikeda, "On-Line Recognition of Hand-Written Characters Utilizing Positional and Stroke Vector Sequences," *Pattern Recognition*, vol. 13, no. 3, pp. 191-206, 1981.

2. K. Yoshida and H. Sakoe, "On-Line Handwritten Character Recognition for a Personal Computer System," *Trans. on CE. IEEE*, vol. 28, no. 3, pp. 202-209, 1982.

3. L. Rabiner and L. Wilpon, "Considerations in Applying Clustering Techniques to Speaker-Independent Word Recognition," *Journal of Acoustic Society America*, vol. 66, no. 3, pp. 663-673, Sept. 1979.

4. L. Rabiner and J. Wilpon, "Isolated Word Recognition Using A Two-Pass Pattern Recognition Approach," *Proc. IEEE ICASSP*, pp. 724-727, 1981.

CHAPTER 5

OPTIMIZATION

In preceding two chapters, the performance of the *OHR* algorithm has been brought to a level which is good enough for practical use. However, the recognition speed of the simulation program is very slow. In this chapter, modifications to the algorithm which improve the recognition speed will be discussed.

1. Real-time processing

Fig. 5.1 shows the execution timing diagram of the algorithm. From it, it can be seen that the writing time occupies a large portion of the elapse time. However, the recognition program does nothing but data acquisition during this period.

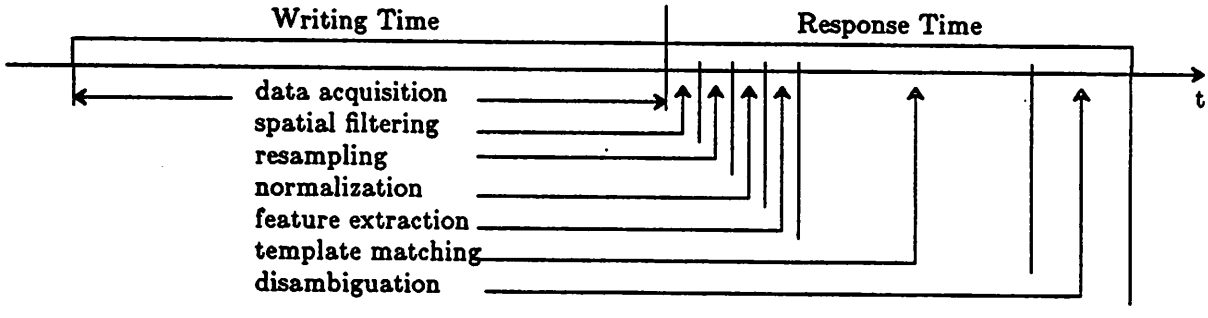


Figure 5.1 Timing diagram of simulation program

The sampling rate of tablet is approximately 100 points per second. Between two points there is about 10ms. If the data acquisition were changed from polling to interrupt driven, plenty of time between two samples would be available.

In an interrupt driven data acquisition scheme, an interrupt service routine is invoked to store the data sample in a queue. The main recognition procedure continuously checks the queue for data. When the queue is not empty, the data in it is processed.

Since the queue contains only the coordinates of the current point and previous points, in the sampling period, it is impossible to perform processing which requires information of future points. The spatial filtering can be performed since it only needs the current sample point and

its previous point. However, there is not much computation in spatial filtering. Most of the sampling period is still wasted.

The operations performed after spatial filtering include: resampling, coordinate normalization, and feature extraction. Chapter 3 has shown that resampling can not be performed until the whole stroke is finished. The coordinate normalization also can not be performed without knowing the maximum and minimum x, y coordinates of the whole symbol. To fully take advantage of the sampling period, the algorithm must be changed.

In Chapter 3, it was shown that the performances of $Dd2$ and $Dp2$ are equally good for template matching. For $Dp2$, the normalized coordinates must be obtained. As such, there is no way to start the DTW computation until the whole symbol is finished. On the other hand, the slope code used in $Dd2$ can be determined by comparing the coordinate of the incoming point with its previous sample point. As seen in Fig. 5.2, after a slope code has been obtained, one column of the local distance and accumulated distance can be computed. Since these distance computations are the most time consuming part of the algorithm, the time between data samples will be fully utilized.

Although the sampling period is fully utilized, there is one problem with this idea. Since the resampling and normalization of the unknown are not performed, the unknown slope sequence is now compared with template slope sequence which was obtained after resampling and normalization. Fig. 5.3 illustrates the potential problem. It is not certain that this matching can provide acceptable performance.

Without resampling, there are two major effects. First, the number of sample points in the unknown symbol is proportional to the symbol size and writing speed. Second, the distribution of sample points is not even. However, these two effects may not significantly impact the $Dd2$ performance. As shown in Fig 3.10, the $Dd2$ does not require the number of sample points of the unknown to be the same as the number of sample points of the template. It also shows that the affect of non-even spatial distribution of sample points can be minimized because the optimal path of $Dd2$ can stay horizontally or vertically for unlimited length.

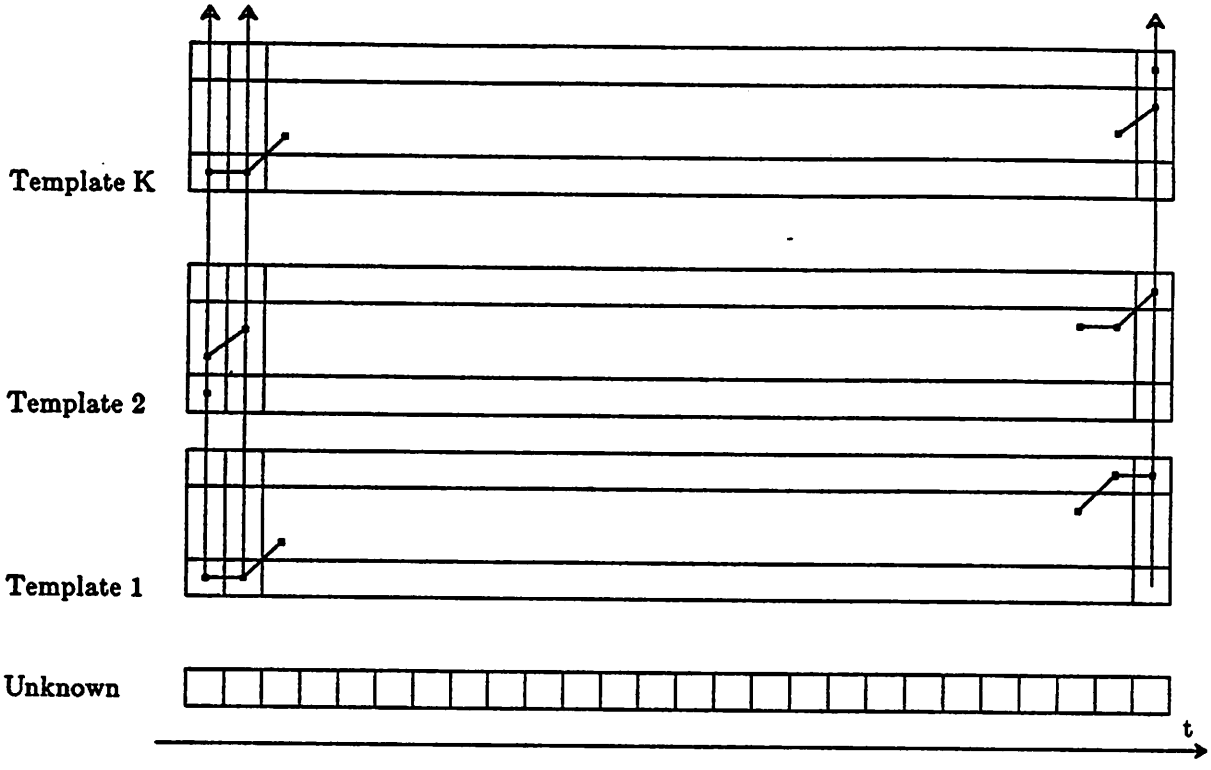
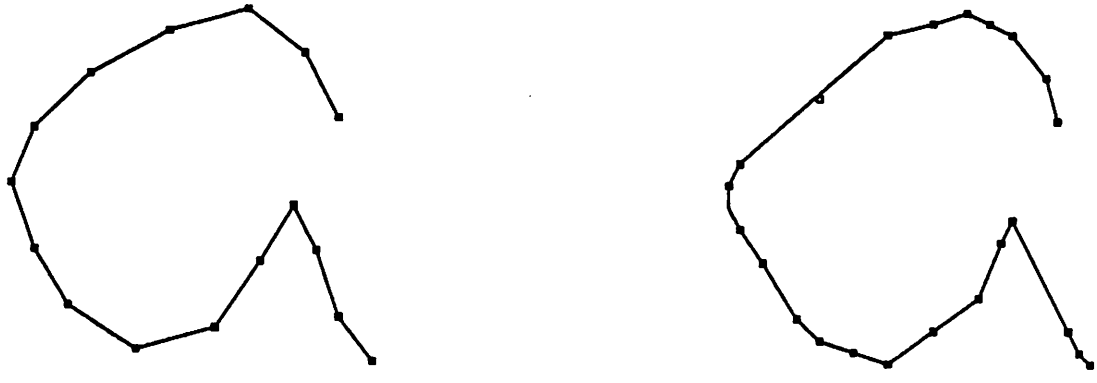


Figure 5.2 Real time DTW computation



resampled not resampled

Figure 5.3 Matching an non-resampled symbol with a resampled symbol

This new template matching algorithm was tested on the data base. Table 5.1 illustrates the comparison between the new *Dd2* template matching algorithm and the old one. Notice that the performance of the new algorithm is only slightly inferior, while drastically shortening the response time of the recognition program. The new timing diagram of the algorithm is as shown in Fig. 5.4.

Table 5.1 Resampling and template matching				
	without resampling		with resampling	
	Recog	Screen	Recog	Screen
RWB	96.5	98.1	96.8	98.4
ENC	98.6	99.4	98.6	99.4
CCH	95.2	97.6	95.7	98.1
PYL	97.6	98.9	98.1	99.4
	96.9	98.5	97.3	98.8

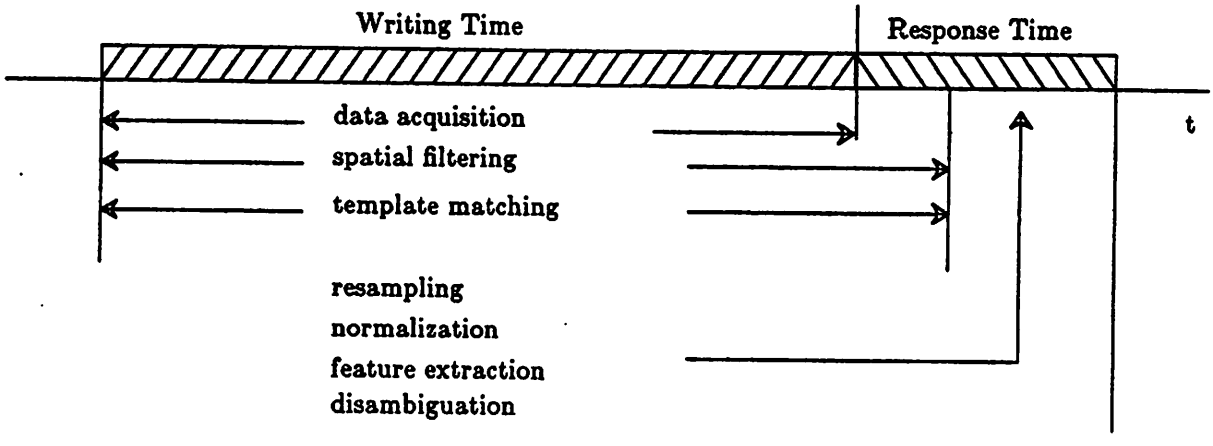


Figure 5.4 Timing diagram of real-time template matching

2. Pruning

If a template is very different from the unknown symbol, the distance computation should stop once it finds that the accumulated distance has grown too big. The *pruning* is implemented by checking the minimum of the whole column accumulated distance of each template. Since the accumulated distance of any element in future columns is definitely larger than the minimum of current column, a template is eliminated once the minimum is over a threshold. The program is as follows.

```
U_i is obtained and it is not end of stroke
For k=1 to k=K
{  If T^k is not eliminated
   {  column_minimum = ∞ ;
      For j=1 to j=J^k
      {  minimum_accumulated = MIN(D^k_j, D^k_{j-1}, temp) ;
         D^k_{j-1} = temp ;
         temp = minimum_accumulated + ds_{i,j} ;
         column_minimum = MIN(column_minimum, temp) ;
      }
      D^k_{j_t} = temp ;
```

If column_minimum > threshold, T^k is eliminated ;

}

}

3. Slope code

In implementation, the slope code is quantized as shown in Fig. 5.5 rather than Fig. 3.2. The reason for not using Fig. 3.2 quantization is because it is desirable to eliminate the time consuming division in the slope computation $\frac{(y_{i+1}-y_i)}{(x_{i+1}-x_i)}$. With Fig. 5.5 quantization, the slope code of the *ith* sample point can be determined by comparing $\Delta x = x_{i+1}-x_i$, $\Delta y = y_{i+1}-y_i$, $\Delta x*2$, and $\Delta y*2$, which only uses addition, subtraction, and shifting.

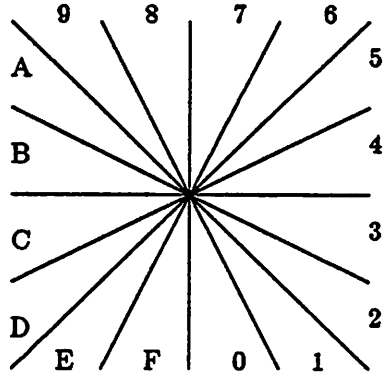


Figure 5.5 New slope code definition

4. Downsampling

Since the *DTW* is insensitive to the spatial distribution of sample points, it was investigated whether sample point with a slope code the same as the previous one could be eliminated. In other words, as shown in Fig. 5.6, whether the compressed slope code sequence can be used for recognition. The *DTW* distance computation is proportional to $I \times J$. Reducing the number of sample points should make the template matching proceed much faster.

4.1. Distance normalization

After downsampling, the number of sample points of each template is different. As illustrated in Fig. 5.6, the *'/'* has only one sample, the *'7'* has two, and the *'O'* has sixteen. Different number of sample points makes the matching path length inherently different. This

fact suggests that the accumulated distance should be somehow normalized to compensate the difference.

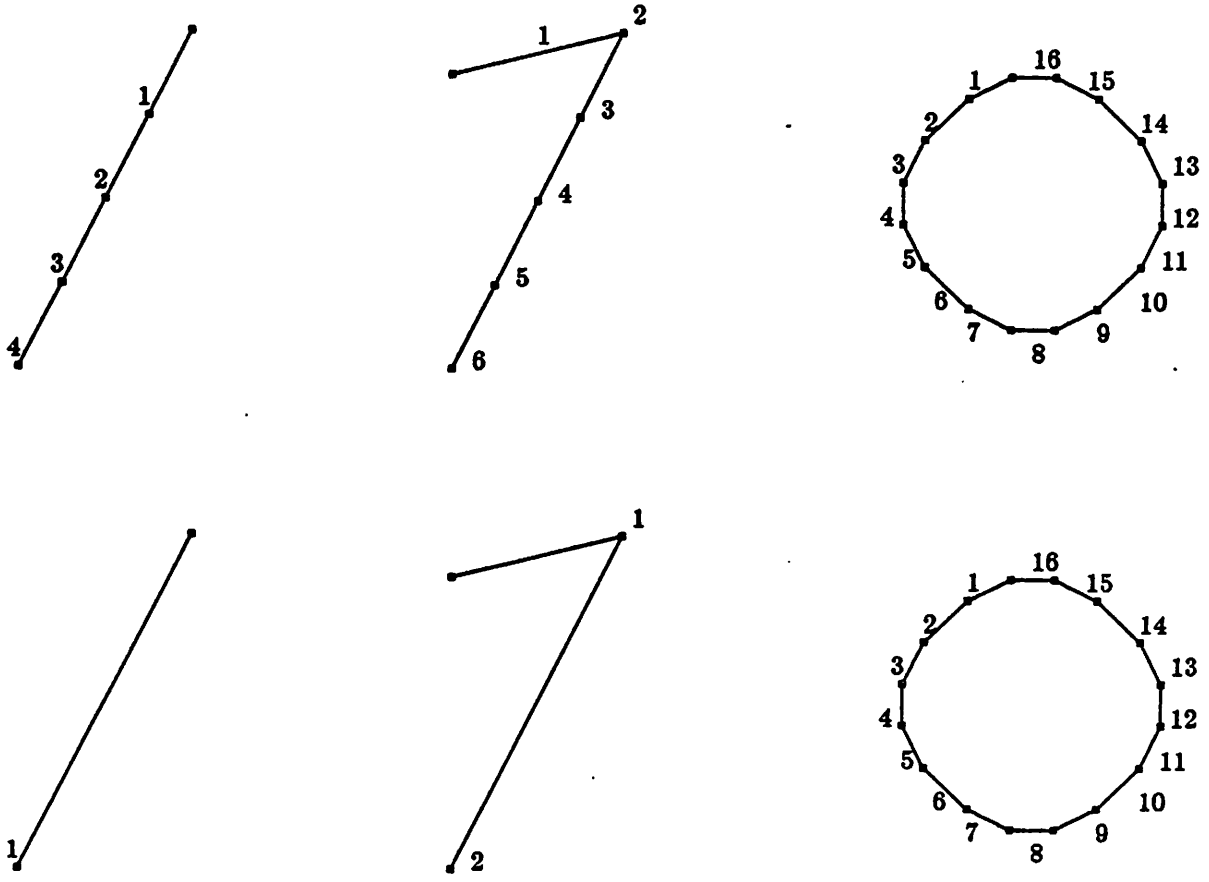


Figure 5.6 Downsampling

One obvious way to compensate the path length difference is to divide D^k by the length of warping path as shown in Fig. 5.7. In this case, instead of using the accumulated distance, the *per comparison* distance is used for similarity measurement.

However, it was found that this normalization has a severe drawback. As illustrated in Fig. 5.7, because the length of left path is longer than the length of the right path, the left normalized distance is lower than the right normalized distance. This is not fair because if the unknown symbol matches a template perfectly, the warping path should be the diagonal line. However, since the diagonal line has the shortest length from (1,1) to (I,J), it gets penalized more heavily than a crooked path.

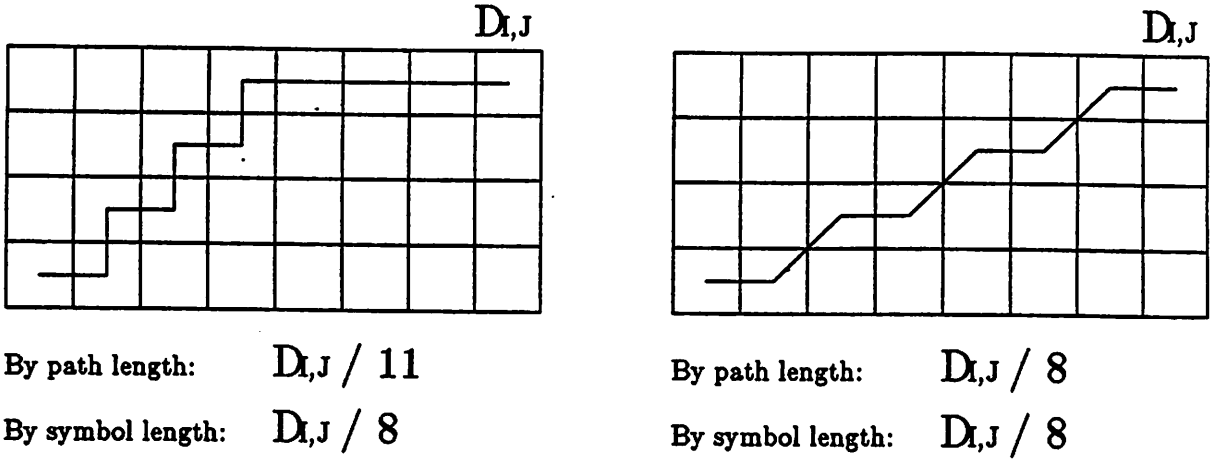


Figure 5.7 Distance normalization

In order to give less penalty to warping paths which are close to the diagonal line, the D^k is normalized by $MAX(I,J^k)$. The reason can be observed in Fig. 5.7. The length of the diagonal warping path is always equal to the longer of the unknown length and template length. If the distance is normalized to this maximum length, a path close to diagonal line gets a smaller penalty. This is because there are fewer local distance elements on a diagonal path than on a crooked path.

4.2. Slope sequence smoothing

One interesting phenomena was observed in downsampling. As shown in Fig. 5.8, if the stroke is written close to the border line of two quantization regions, the number of sample points can have big variation. A smoothing procedure needs to be performed to reduce these variations.

The smoothing operation is achieved by dividing each quantization level into two sublevels. If the sublevel code distance of two consecutive points is less than 2, the second point will not be used for matching. Fig. 5.8 shows how this procedure removes the fluctuations of border line strokes.

4.3. Performance

Table 5.2 shows the experimental results of matching the unknown symbol against the

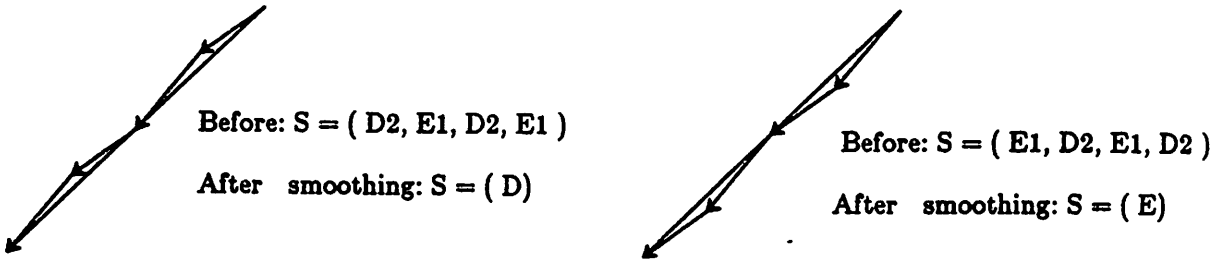


Figure 5.8 Sequence smoothing

downsampled templates. Notice that the distance definitely needs to be normalized. Normalizing the distance to the symbol length is much better than normalizing it to the path length. Although the overall recognition rates are not as good as before. the recognition speed is about 35% faster. This is because the number of sample points of each template is reduced by 20%.

Table 5.2 Downsampling of slope code

	No Norm.		Norm. to the length		Norm. to the longer	
	Recog	Screen	Recog	Screen	Recog	Screen
RWB	94.2	95.8	93.9	95.5	94.7	96.3
ENC	96.8	97.6	96.3	97.1	96.8	97.6
CCH	92.8	95.2	92.3	94.7	93.1	95.5
PYL	95.0	96.3	94.2	95.5	95.8	97.1
	94.7	96.2	94.1	95.7	95.1	96.6

Experiments were also performed in which the downsampling algorithm was applied to the disambiguation functions $Dx2$ and $Dy2$. If the x (or y) coordinate of the current point is the same as the x (or y) coordinate of the previous point, it is discarded. The downsampling reduces the number of sample points by approximately 10%. The disambiguation computation time is therefore reduced by approximately 20%. For these two functions, the recognition accuracy with downsampling is essentially the same as the accuracy without downsampling.

5. Coordinate normalization

After template matching, the coordinates of the unknown must be normalized before applying the disambiguation functions. The normalized coordinates are obtained by the formula

$$\tilde{x_i} = (x_i - X_{\min}) \times \frac{W}{X_{\max} - X_{\min}}$$

$$\tilde{y_i} = (y_i - Y_{min}) \times \frac{H}{Y_{max} - Y_{min}} .$$

In it, X_{max}, X_{min}, Y_{max}, and Y_{min} are the maximum and minimum tablet x,y coordinates of the symbol. W is the maximum x coordinate after normalization and H is the maximum y coordinate after normalization. If $X_{min} = Y_{min} = 0$, the normalization formula becomes

$$\tilde{x_i} = x_i \times \frac{W}{X_{max}} ,$$

$$\tilde{y_i} = y_i \times \frac{H}{Y_{max}} .$$

In linear interpolation, as shown in Fig. 5.9, the y_i corresponding to the x_i in between (0,0) and (X,Y) is:

$$y_i = x_i \times \frac{Y}{X} ,$$

which is the same as our normalization formula. There is a well known Bresenham algorithm[1] which can obtain all the integer interpolation points between (0,0) and (X,Y) incrementally without any multiplication, division, and floating-point operations. Referring to Fig. 5.9, assume $X > Y > 0$, the Bresenham algorithm moves along the longer side and compares ϵ and ϵ' for each increment. If $\epsilon > \epsilon'$, it takes point A. Otherwise, point B. The program of this algorithm is as follows.

```
X' = X + X,      Y' = Y + Y,
ε =  Y' - X,  ε' = ε - X,
x0 = y0 = i = 0;
while(xi < X)
{  i = i + 1;
   xi = xi-1 + 1;
   if(ε > 0)
   {  yi = yi-1 + 1;
      ε = ε + ε';
   }
   else
   {  yi = yi;
      ε = ε +  Y';
   }
}
```

The Bresenham algorithm can be used to create the *x mapping table* and *y mapping table* for coordinate normalization. As the above formula and Fig. 5.10 show, if X is replaced by X_{max}, Y is replaced by W, and x_i is incremented from 0 to X_{max}, the y_i obtained from Bresen-

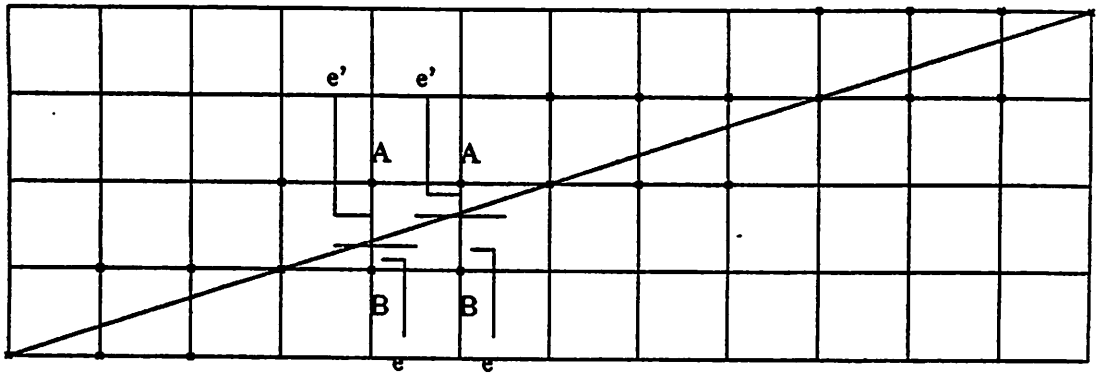


Figure 5.9 Bresenham algorithm

ham algorithm are exactly the $\tilde{x_i}$ for $x_i = 0$ to $x_i = X_{max}$. If X is replaced by Y_{max}, Y is replaced by H, and x_i is incremented from 0 to Y_{max}, the y_i obtained from Bresenham algorithm are exactly the same as $\tilde{y_i}$ for $y_i = 0$ to $y_i = Y_{max}$.

Let $Mx[x]$ and $My[y]$ denote the x mapping table and y mapping table created by Bresenham algorithm. The normalized coordinate of (x_i,y_i) is simply $(Mx[x_i],My[y_i])$. The multiplication and division involved in the coordinate normalization are eliminated.

References

1. J. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM System J.*, vol. 4, no. 1, pp. 25-30, 1965.

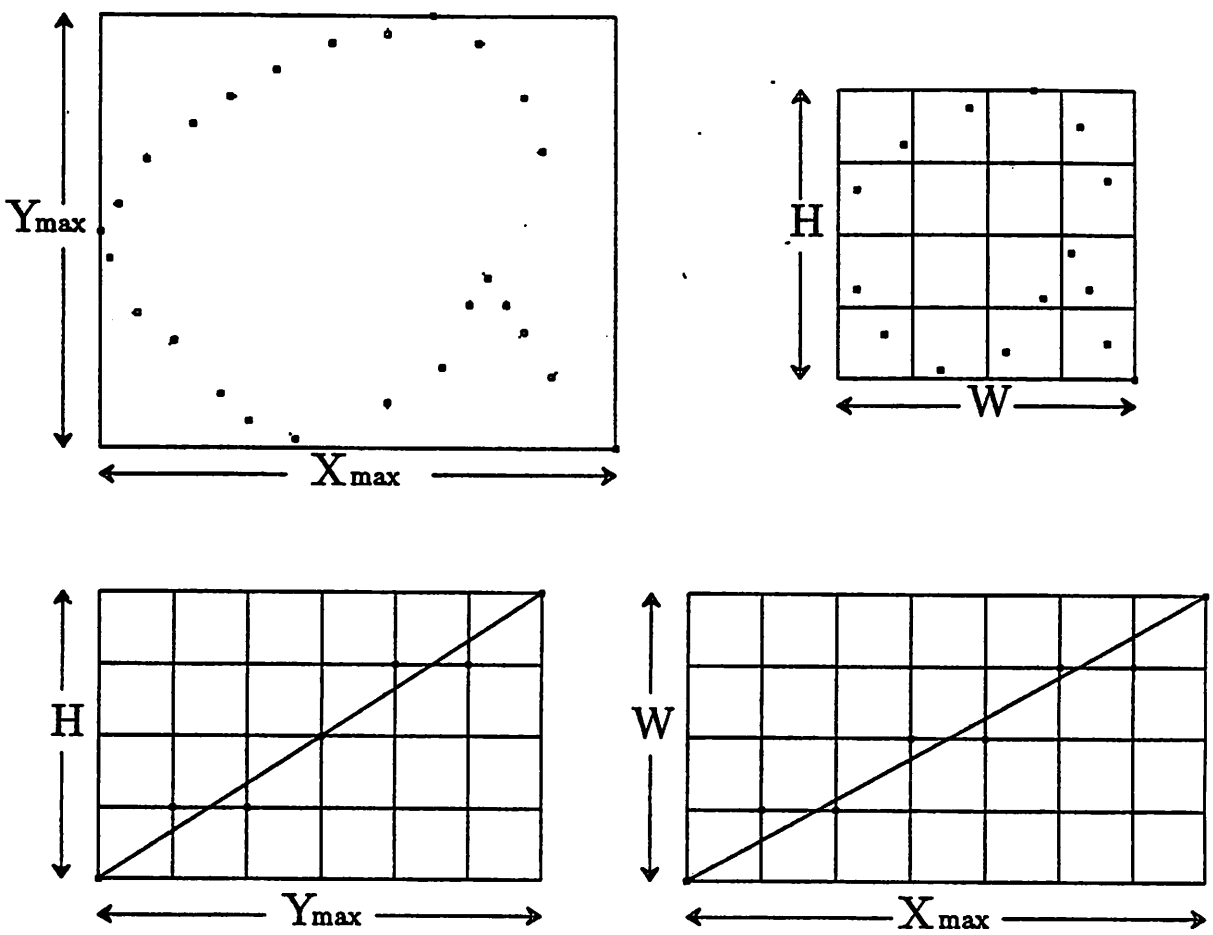


Figure 5.10 Using Bresenham algorithm for coordinate normalization

CHAPTER 6

IMPLEMENTATION

In previous chapters, the algorithms for the *OHR* system have been covered. In this chapter, the emphasis is on how to implement the system on an IBM PC. The implementation should achieve two goals. First, a user should be able to control existing application programs via handwritten symbols. Second, high recognition accuracy should be achieved without much training work by the user.

1. Keyboard emulator

To allow the user to be able to control an application program via *OHR*, the *OHR* has to run simultaneously with an application program. The most popular operating system used on the IBM PC is MS-DOS. Although MS-DOS is not a multi-tasking cperating system, it provides a function to install a program as an extension of *system service routine.*[1] Once a program is installed as system service routine, it is kept resident in RAM and can be called by any application program at any time. The same function also allows a user to install his own *interrupt service routine* to handle the interrupt from a peripheral. As such, the basic software structure of the *OHR* system is as illustrated in Fig. 6.1.

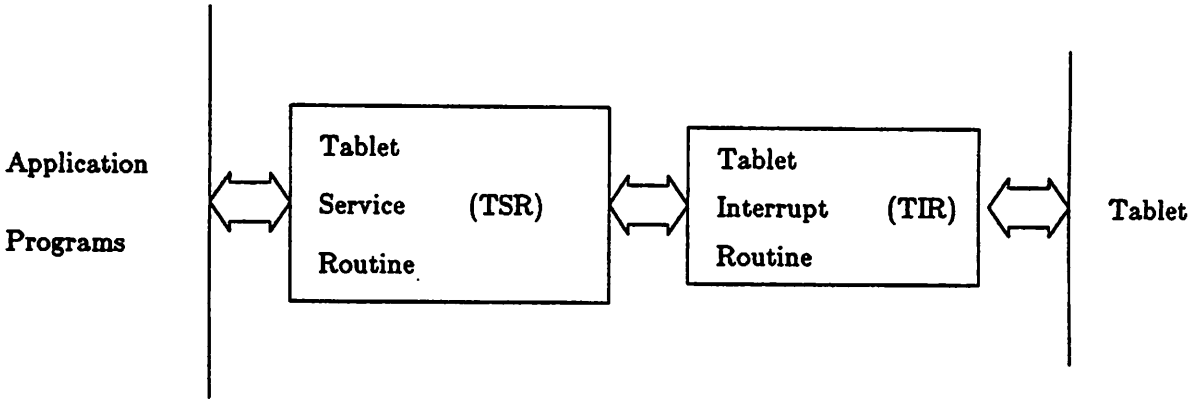


Figure 6.1 OHR software structure on IBM PC

1.1. Tablet Interrupt Routine (TIR)

The tablet is connected to a serial port. Once data is available from tablet, the *Tablet Interrupt Routine (TIR)* is invoked. The *TIR* converts the coordinate from tablet format to *OHR* format and then calls *Tablet Service Routine (TSR)* to put the coordinate into a queue for later processing.

1.2. Keyboard Service Routine (KSR)

In an IBM PC, the interface between application program and keyboard input is handled by two BIOS (Basic Input/Output System) routines. One is *Keyboard Interrupt Routine (KIR)* and the other is *Keyboard Service Routine (KSR)*. The *KIR* is invoked once a keystroke is pressed. It translates the keyboard scan code to ASCII code and then stores the ASCII code in a buffer. The *KSR* is called by application software to process the buffered ASCII code. Fig. 6.2 illustrates the scheme.

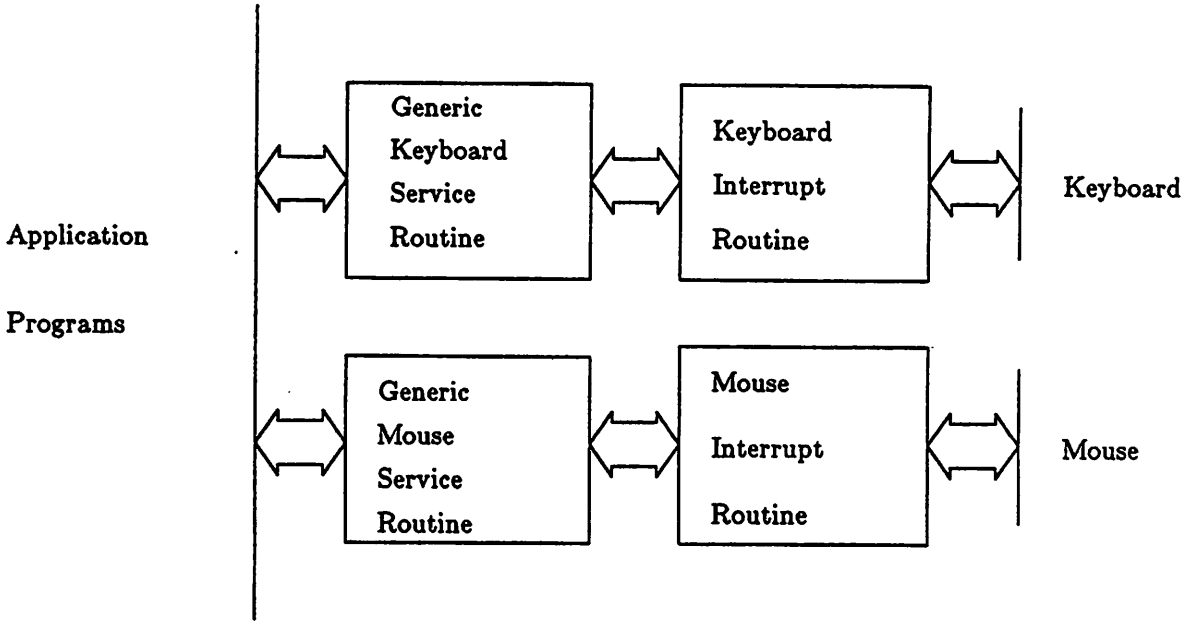


Figure 6.2 Standard IBM PC keyboard/mouse interface

MS-DOS allows a user to replace the generic *KSR* by his own *KSR*. Therefore, as shown in Fig. 6.3, the *OHR* puts the generic *KSR* in a private place and then install its own *KSR*. The new *KSR* calls the *Tablet Service Routine (TSR)* to do the following:

(1) Check whether there is a queued keystroke generated by the recognizer.

(2) If there is no keystroke from the recognizer, call the generic *KSR* to check whether there is queued keystroke from the keyboard.

(3) If there is no keystroke from the keyboard, check whether there are queued coordinates from *TIR*.

(4) If there are queued coordinates, recognize the symbol being written. Otherwise, go back to (1).

With this implementation, the application program can get keystrokes from both the recognizer and the keyboard, even though it does not know there is an *OHR*.

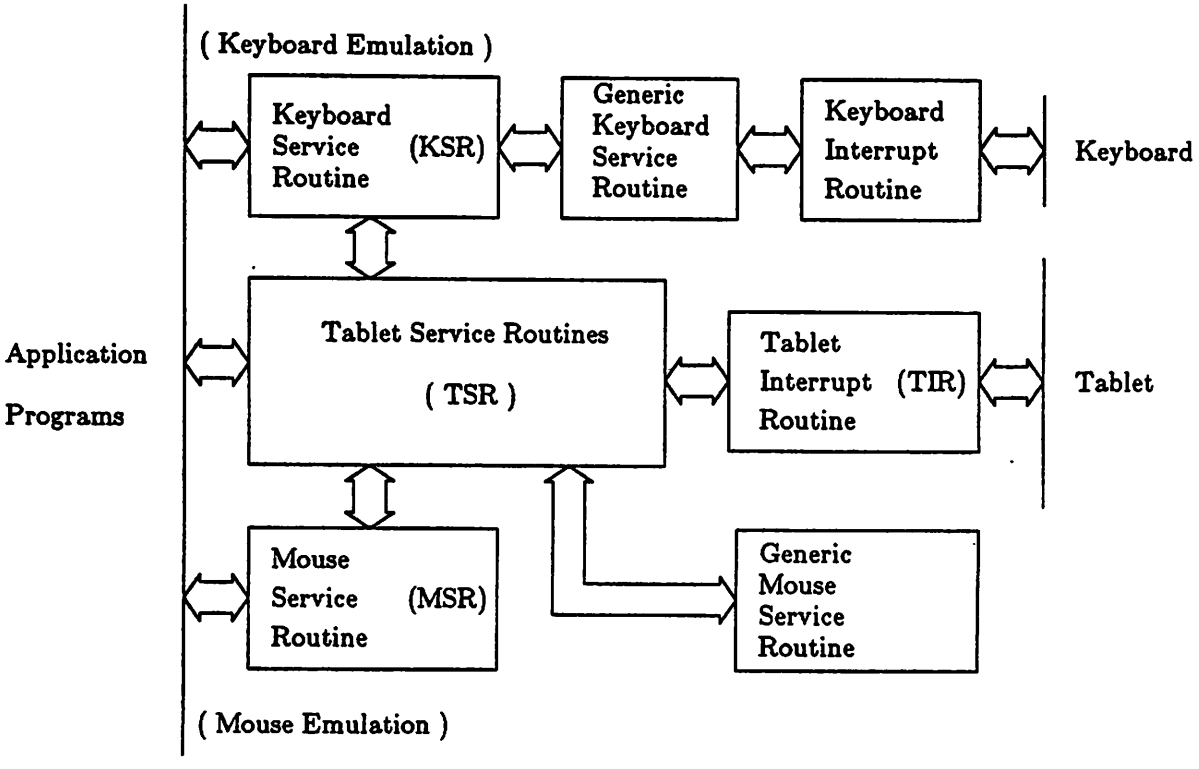


Figure 6.3 Retrofit IBM PC keyboard/mouse

2. Mouse emulator

Unlike the keyboard, the mouse is not supported at BIOS level.However, the communication protocol provided by Microsoft *Mouse Service Routine (MSR)* is considered as the mouse

interface standard. This protocol defines how an application program talks with the mouse service routine to get mouse information and to set mouse parameters.

Just as the keyboard service routine, there is another *mouse interrupt routine (MIR)* working together with *MSR*. Once the mouse is moved or any button is pressed, the *MIR* is invoked to process the event. The appropriate data of the event is then generated and put in a buffer. The mouse position on screen is also updated. Once the application software needs mouse information, it calls the *MSR* to check the mouse event. Fig. 6.2 shows the scheme.

The mouse is evaluated in a way similar to that of the keyboard. As illustrated in Fig. 6.3, the generic *MSR* is moved to a private place and replaced by new *MSR*. If the *TSR* is in mouse mode, the coordinates from *TIR* are first converted to the screen coordinate and then passed to the generic *MSR* to change the screen mouse location. If the application program needs the mouse information, the new *MSR* returns the information according to the events from stylus.

The Microsoft mouse has two buttons. The *OHR* stylus has only one internal switch. In order to emulate two buttons, two function boxes are used as shown in Fig. 6.4. After the user touches the LEFT/RIGHT function box, the clicking or dragging of the stylus are interpreted as the clicking or dragging of the mouse left/right button.

3. Tablet layout

Besides translating a written symbol to keystrokes or emulating mouse, the tablet can provide many unique user interface features. For example, in addition to the keystrokes of a symbol, the application program can also obtain the location and the size of it. Knowing the location, the application program can acquire the data which is being filled in certain field of a paper form on the tablet. This makes it unnecessary to complete a paper form first and then type the data into the computer. Knowing the size, the superscript and subscript of mathematical symbols can be easily entered.

Another useful feature is that some tablet area can be used for *function boxes*. Once a function box is touched, its preassigned keystrokes (*macro*) are sent to application programs as

though they were typed in from keyboard. It works like the function keys on the keyboard but has a significant advantage. Since the function boxes can be anywhere on the tablet, appropriate arrangement of the boxes gives a *logical and pictorial* illustration of the control structure of an application program.

Fig. 6.4 shows a simple tablet layout. The strip on the right edge are assigned for eight function boxes. The MODE is used to set the *OHR* either in "keyboard" mode or in "mouse" mode. If it is in the mouse mode, the coordinates from keyboard/mouse area are not processed by the recognition procedure. Instead, they are sent to application programs as though the stylus were a mouse. The LEFT and RIGHT functions are used to set which button the stylus is emulating. The keystrokes of the five other function boxes are programmable.

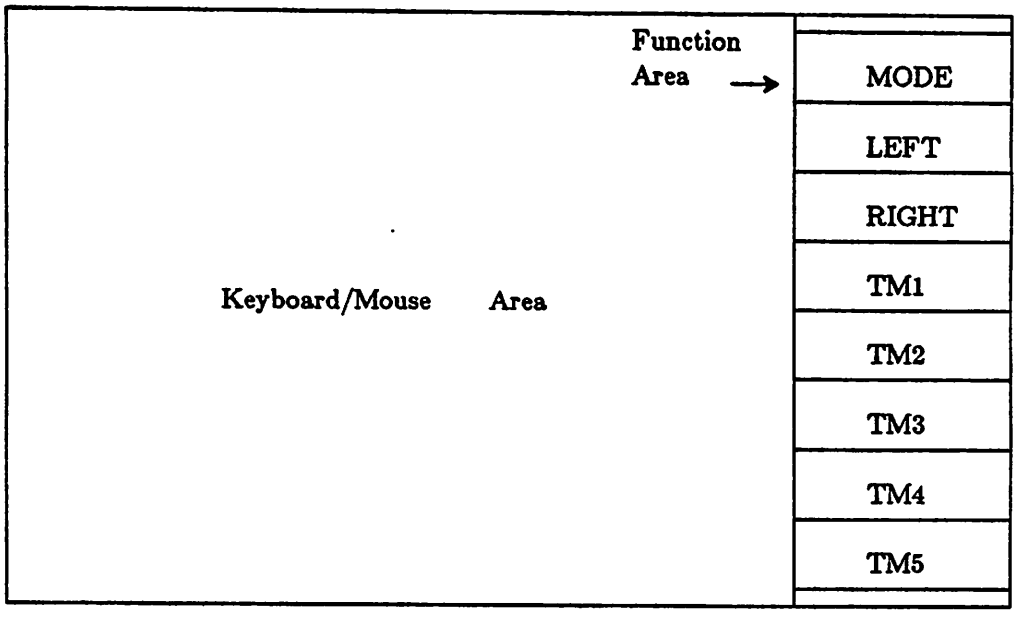


Figure 6.4 The tablet layout

Although this tablet layout is currently hard-coded in *OHR*, it can can be customized to any format by the appropriate tools.[2]

4. Tablet Service Routine (*TSR*)

The *TSR* is the "soul" of the system. It handles the service requests from application programs and manages data for the *OHR*. Fig. 6.5 illustrates this operation. The definitions of the

data objects are as follows.

TBLQ: a queue for tablet coordinates.

SYB: a structure which holds the features of the symbol just written.

DIC: an array of structures which holds the features of all templates.

KEYQ: a queue for keystrokes of recognized symbols or touch function.

The services provided by *TSR* are as follows.

clrTBLQ: Clear TBLQ

putTBLQ: Put a coordinate pair to TBLQ.

getTBLQ: Get a coordinate pair from TBLQ.

clrDIC: Clear DIC.

putDIC: Put a template to DIC.

getDIC: Get a template from DIC.

putMAC: Load definition of macros.

putKEYQ: Put a keystroke string to KEYQ.

peekKEYQ: Get a keystroke from KEYQ.

getKEYQ: Get a keystroke from KEYQ. If none, wait until getting one.

getRAW: Get the coordinates of the next symbol.

getSYB: Get the recognition features of the next symbol.

setDNSPL: Set whether to perform the downsampling.

5. Training procedure

Training of the *OHR* system is achieved by a set of utility programs: ***hinstal, htrain, hclus, hdisam, hload, and hanal*** . Fig. 6.6 shows the flow chart of how to use these programs. *Hinstal* is the program which installs *TIR*, *KSR* and *TSR*. It has to be run before any other utility program.

5.1. Initial training

For a specific application, the user first decides upon a symbol set which contains symbols

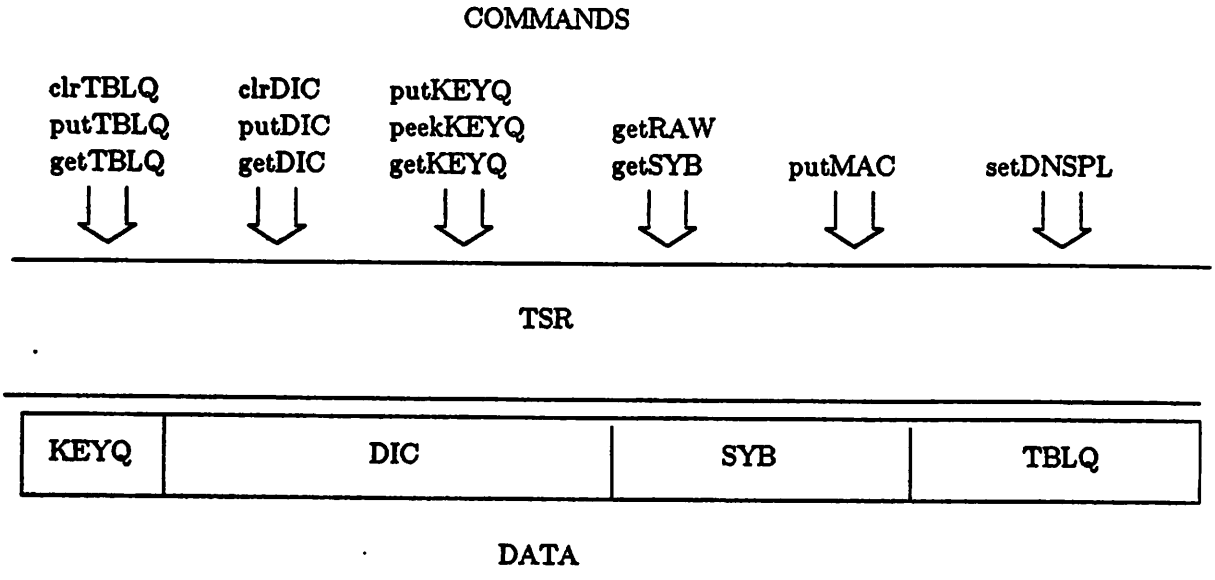


Figure 6.5 Structure of the *TSR*

he is going to use. For each symbol set, a ***directory*** must be created. In this directory, the name and the keystrokes of each symbol is defined in a file ***symbol.def***.

After specifying the symbols, ***htrain*** is used to obtain training samples for each symbol. Then, ***hclus*** is used to group training samples of each symbol into clusters and create templates. Then, ***hdisam*** is used to check the ambiguity between templates and set the disambiguation rules for ambiguous template pairs.

After ***hdisam***, the initial training is finished. A dictionary for this symbol set is created. The user can download the dictionary to *TSR* using ***hload***. The ***hload*** also downloads the file ***macro.def***, which specifies the keystrokes of each touch function box, to *TSR*.

After the dictionary is downloaded, the user can start his application program and "write" to it !

5.2. Adaptive training

The the accuracy of the trainable *OHR* system is primarily hurt by two phenomena. First, for some symbols, the user writes the training samples very consciously. However, this consciousness causes the training samples to be different from later samples. Second, for ambiguous symbols, the disambiguation rules are sometimes not accurate because there are not enough

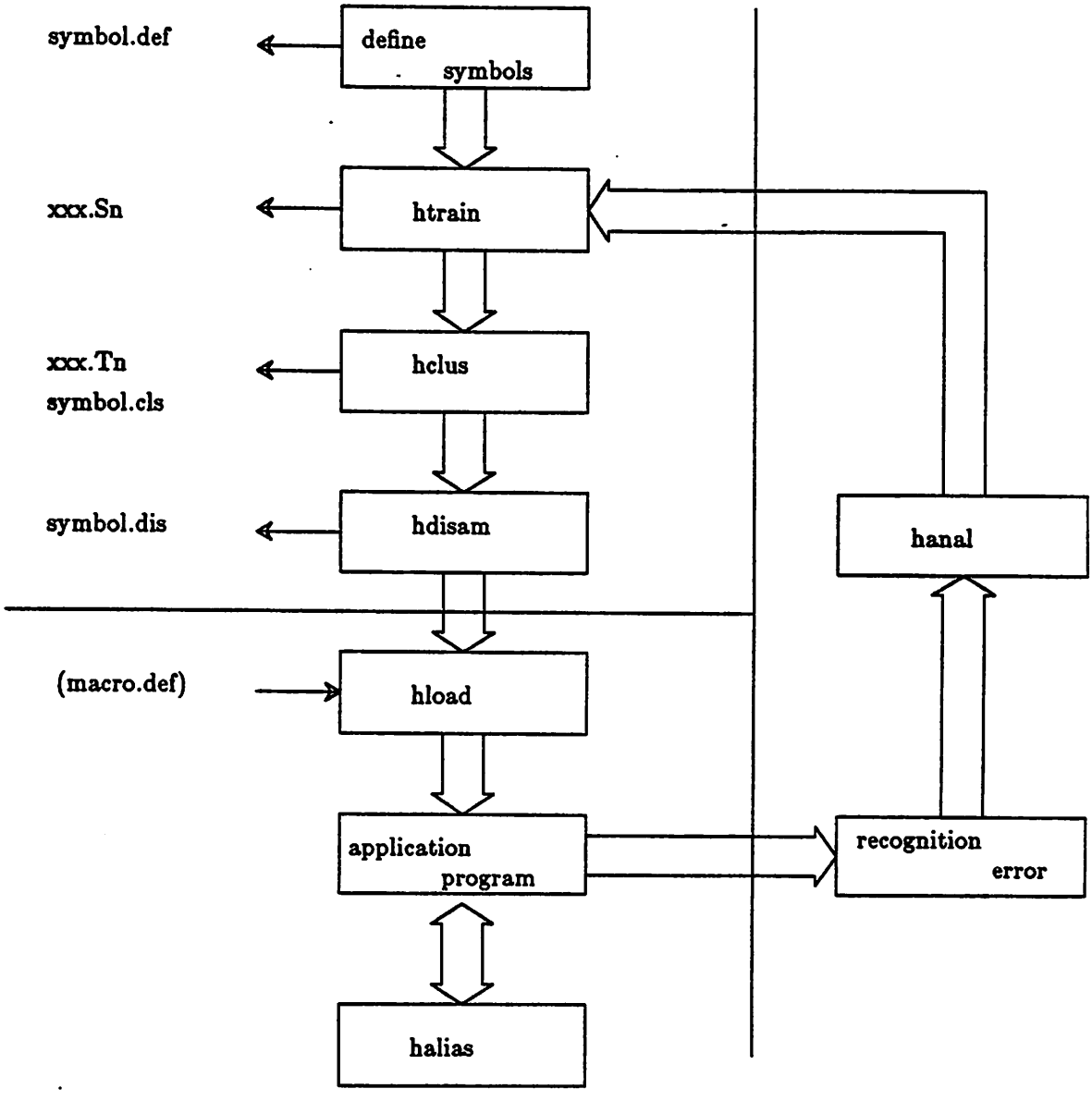


Figure 6.6 Procedures to use the *OHR*

training samples.

To fix the second phenomena, the user must provide more training samples during the initial training. However, because of the first phenomena, it seems useless to ask for too many training samples during the initial training.

To solve this dilemma, a "*train-on-error*" approach is used to achieve a high recognition rate without much training overhead. At the initial training phase, the user only gives minimum training samples of each symbol based on his feeling of how he is going to write the symbol.

Then, during the recognition phase, if an error occurs, he can first use the *hanal* to see the writing trace of the symbol he just wrote and the suspicious templates. If the error is due to inconsistent writing, the user can decide to either not make the same "mistake" or to include what was written as another training sample. If he wants the latter, he can use the "*htrain -adapt*" to extract the just written symbol as a new training sample.

As shown in Fig. 6.7, the "*hanal*" provides the matching details of all distance functions. Based on this information, the user may find the disambiguation rule is not optimal and can use the "*htrain -inter*" to provide more training samples for the ambiguous symbols.

After enough new training samples are collected, "*hclus*" and "*hdisam*" are used to update templates and disambiguation rules.

5.3. Temporary training

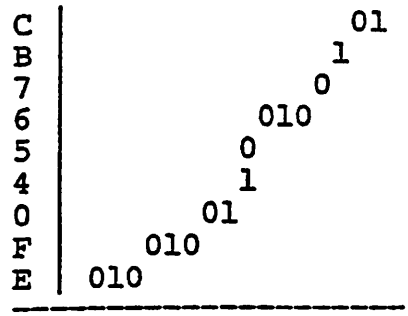
It is often useful to allow the user to temporarily add one symbol to represent a string of keystrokes during an application program. For example, if the text string "UNIVERSITY OF CALIFORNIA, BERKELEY" happens very often during a user's editing, he would like the *OHR* to send out the whole string when he writes '*uc*' as shown in Fig. 6.8. This feature was implemented in "*htrain -alias*". (It is named after the *alias* command in UNIX.) This program asks for only one training sample of the temporary symbol and then downloads it to *TSR* as a template. No clustering and disambiguation are performed. If the user wants to permanently add the symbol in the symbol set, he has to put it in *symbol.def* file and run *htrain*.

6. Performance

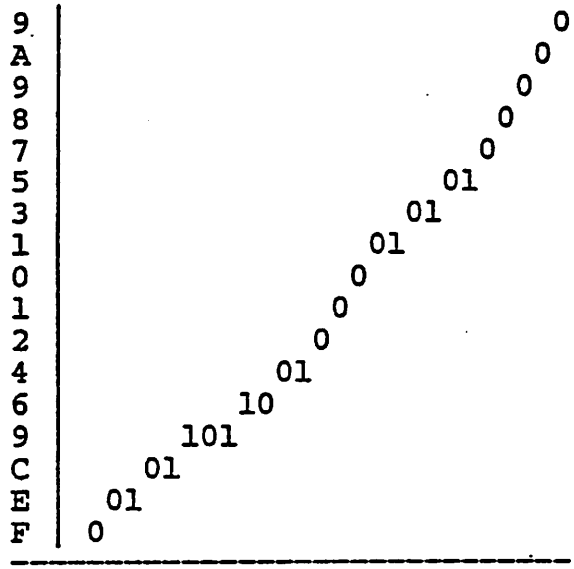
Experimental results showed this implementation can achieve very high recognition rate without much training. The recognition rate can be close to 100% if the user does the adaptive training well. In addition to the accuracy, other performance figures are:

(1) The recognition speed is about 75ms per template stroke without downsampling and 55ms per template stroke with downsampling.

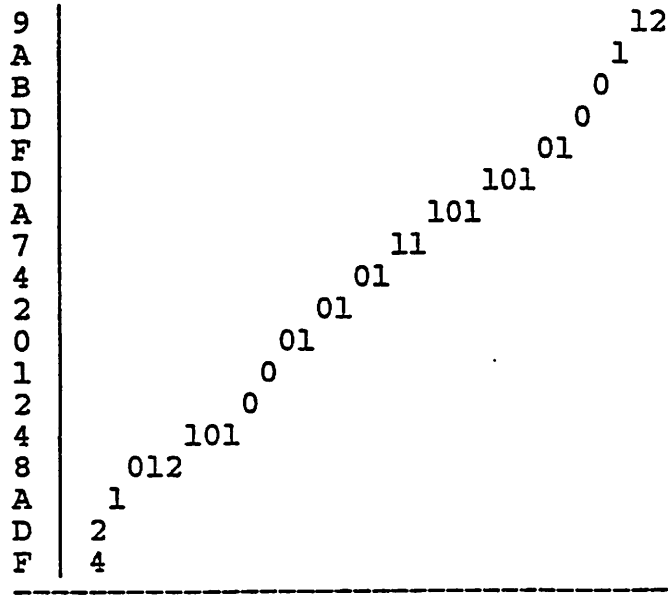
6: Slope sequence of stroke 1: Before norm: 7 After norm: 6



6: Y sequence of stroke 1: Before norm: 9 After norm: 5

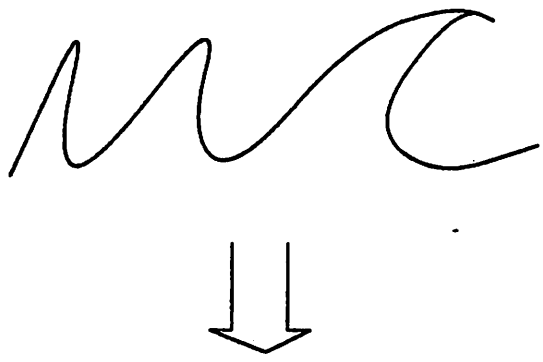


6: X sequence of stroke 1: Before norm: 26 After norm: 13



```
0:  Slope sequence of stroke 1:  Before norm:  6   After norm:  5

C |                 01
B |                1
7 |               0
6 |            010
4 |           1
F |          0
0 |         0
F |      010
E | 010
--------------------
    EFEF0F0F56767ACD

0:  Y sequence of stroke 1:  Before norm:  21   After norm:  12

E |                              5
C |                            2
A |                          1
7 |                       101
4 |                     01
2 |                   01
0 |                 01
1 |               0
3 |             01
5 |           11
8 |        101
B |      01
D |    01
F | 01
--------------------------------
    FEDCBA98764321O123456789A9

0:  X sequence of stroke 1:  Before norm:  20   After norm:  10

8 |                              01
A |                            1
C |                          11
F |                        01
D |                      01
B |                   101
8 |                 01
5 |              101
2 |            01
0 |          01
1 |        0
3 |      01
5 |    01
6 |  0
8 | 3101
------------------------------------
    B98765432101234568ABCDEFEDB987
```

Figure 6.7 Matching details from hanal



UNIVERSITY OF CALIFORNIA, BERKELEY

Figure 6.8 On-line trained temporary symbol

(2) The memory needed for a template is about 100 bytes per template stroke.

(3) The resident driver (*TIR, KSR, and TSR*) takes 85K bytes memory.

References

1. "MS-DOS Technical Reference Manual," *MicroSoft.*

2. "Handwriter System Manual," *Communication Intelligence Corp.*, 1985.

CHAPTER 7

CURSIVE SCRIPT

1. Introduction

Because word can typically be written in cursive script much faster than when printed, it would be desirable to make the *OHR* system be able to recognize cursive scripts.

A simple way to handle cursive script is to treat the whole script as a discrete symbol. In this case, the discrete symbol recognition algorithm can be applied without any change. However, this approach is applicable only if there are not too many words. Otherwise, the training becomes impractical.

In order to reduce the training effort, another challenging approach is to recognize scripts based only on letter templates. In this case, instead of thousands of words, the user need only train 26 letters.

In this chapter, algorithms and implementations based on the second approach will be investigated.

2. Review

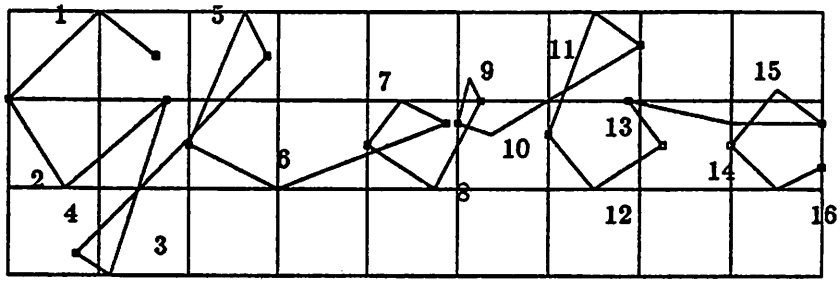
There have been many algorithms proposed for cursive script recognition. Almost all of them consist of three steps. First, the cursive script is divided into segments. Then, a recognition algorithm is applied to recognize each segment. Finally, the segment recognition results are put together for the best concatenation.

In this section, four algorithms are reviewed. Again, since none of them has gone through extensive testing, only the approaches and not the results can be compared.

2.1. Algorithm 1:[1]

As shown in Fig. 7.1, this algorithm decomposes a cursive script into a sequence of upstrokes and downstrokes at points with x or y coordinates that are either a local maximum or a local minimum. For each downstroke or upstroke, a set of 12 parameters, which are derived

from a physical model of handwriting generation, are extracted.[2] These 12 parameters can classify a downstroke into one of the 22 predefined categories.



(a)

Type 22	0.35	0.71					0.08	0.31
Type 21	0.66	0.32					0.57	0.62
Type 2	0.17	0.23					0.76	0.13
Type 1	0.84	0.94					0.35	0.44
script strokes	1	2	3	4			15	16

(b)

	Stroke String	Likelihood
Word 3	4-20-8-16-12-20-5-15-10	0.17
Word 2	21-18-15-12-9-6-3-12	0.71
Word 1	1-3-5-7-9-11-13-15-17-19-21	0.23

(c)

Figure 7.1 Algorithm 1

In the training phase, parameter statistics of each downstroke category are obtained. Concatenations of the downstrokes of all target words are also stored in dictionary. In the recognition phase, the *a posterior* probabilities of all downstrokes are evaluated. The cursive script is then represented as a likelihood matrix as illustrated in Fig. 7.1(b). From the matrix, the likelihood value between the unknown script and each dictionary word can be obtained by summing

up the likelihood values of the downstrokes of that word. The word with the maximum total likelihood is recognized as the unknown script.

2.2. Algorithm 2:[3]

In this algorithm, 6 primitives and 16 features are first defined to describe a script. The 6 primitives are (1) R (L): horizontal right (left) extrema, (2) T (B): vertical local maximum (minimum), (3) C: cusp, (4) I: inflection, (5) X: intersection point, and (6) P (N): positive (negative) curvature. The 16 features are (1) { P B X }, (2) { N T N }, (3) { P T P }, (4) { N B N }, (5) { N C P }, (6) { P C P }, (7) { P X T P }, (8) { N B X N }, (9) { P X P }, (10) { N B T X N }, (11) { N I P B }, (12) { P I N B }, (13) { B N I P }, (14) { B P I N }, (15) enclosure and (16) pen up/down . Because the primitives are very easy to identify, it is not difficult to convert a script into a primitive sequence. From the sequence, a feature network can be generated as illustrated in Fig. 7.2(b).

In the training phase, the feature sequences of all letters are compiled into a dictionary. Fig. 7.2(c) illustrates some dictionary entries. In the recognition phase, the feature network of the unknown script is parsed to find all letters and concatenations.

If more than one word can be derived from the feature network, the relative heights of letters are checked to see whether the profile of the unknown script matches the profile of each possible word. For example, the *hugo* in Fig. 7.2(a) can not be recognized as *hugb* because *o* is between two baselines. It can not be recognized as *uugo* either because the *h* is over the upper baseline.

2.3. Algorithm 3:[4]

In this algorithm, several complicated algorithms are first applied to normalize the shape of a cursive script. The purpose of the normalization is to reduce the irregularities of (1) zone baselines, (2) global slant, (3) local position variations, and (4) local slant variations. Fig. 7.3 illustrates the affects of each normalization step. It can be seen that after normalization, the global slant has been corrected, the local maximum and minimum are brought to the baselines, and the

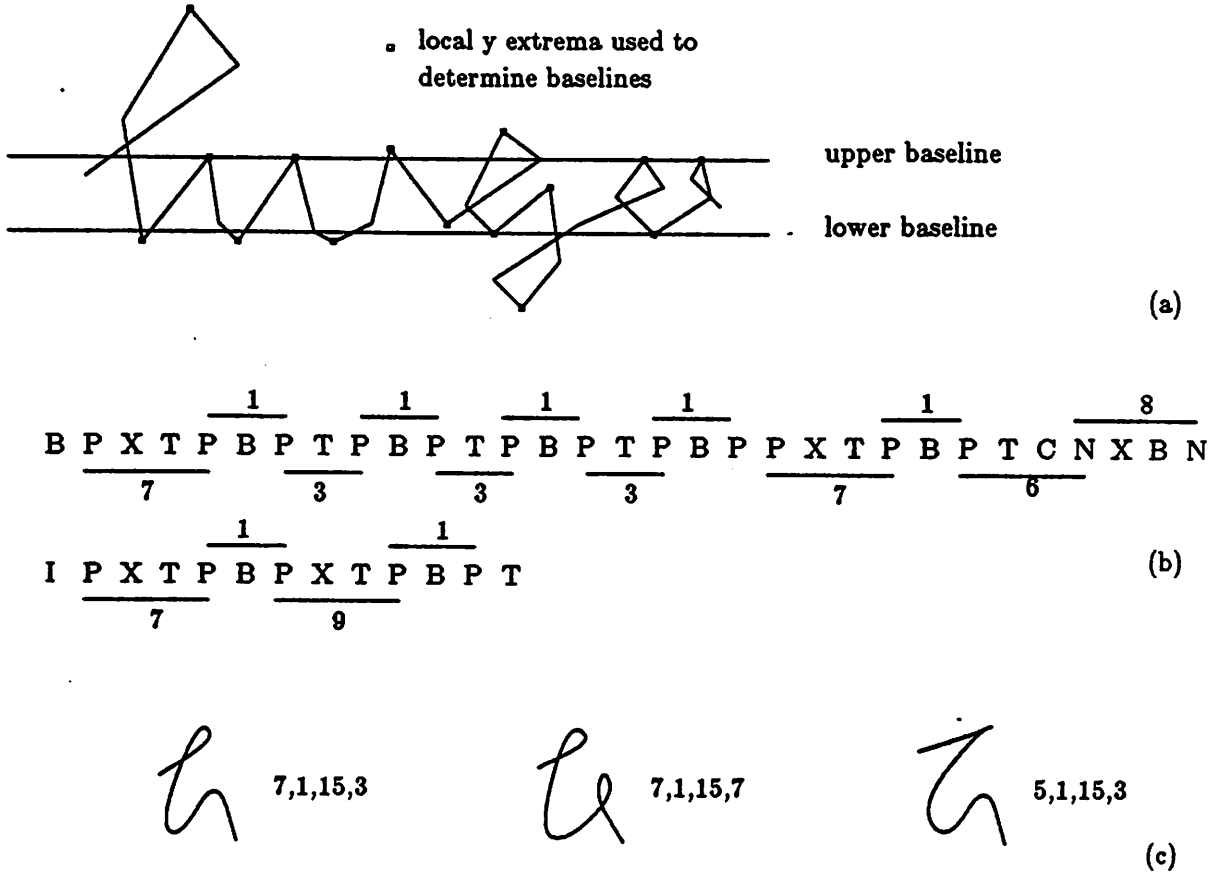


Figure 7.2 Algorithm 2

descending strokes are vertically aligned.

The y coordinate and slope code of each sample point are extracted as recognition features. In the training phase, letter templates are created by manually extracting letters from normalized training scripts. A vocabulary file which contains a list of all allowed words is also created. In the recognition phase, a two-pass *DTW* algorithm is applied to find the best word in vocabulary. The detail of the two-pass *DTW* algorithm will be discussed later.

2.4. Algorithm 4:[5]

This algorithm assumes the scripts are written on a 1/4-inch lined paper. The recognition features extracted from each sample point are the normalized y coordinate and the slope angle. In the training phase, the initial templates of each letter are created by writing each letter discretely. Then, these initial templates are used to extract letters from training scripts. The

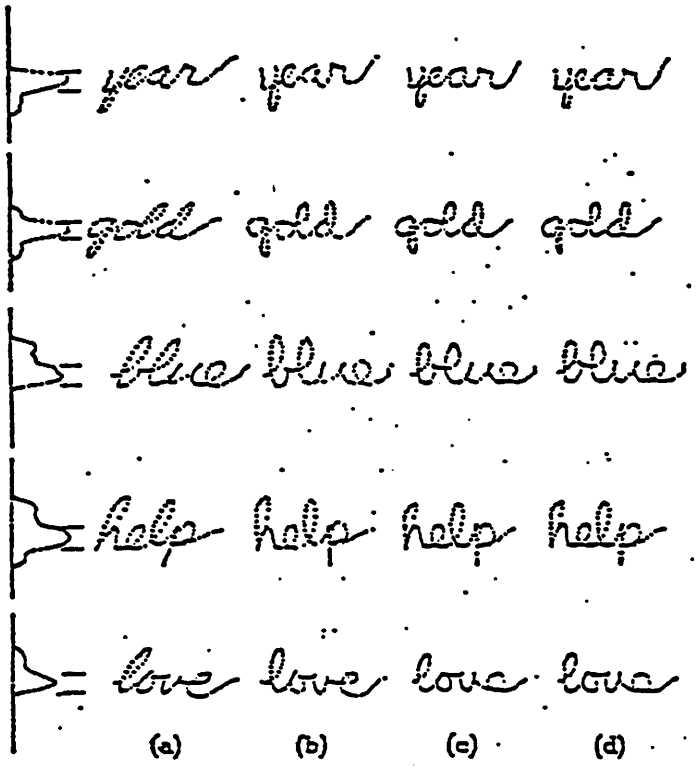


Figure 7.3 Preprocessing in algorithm 3

extracted letters, together with initial templates, are all used for recognition.

In the recognition phase, a one-pass *DTW* algorithm is applied to the unknown script and all templates. The one-pass *DTW* algorithm can find the concatenation of templates which matches the unknown cursive script best. The details will be discussed later.

In order to improve the performance of the one-pass *DTW* algorithm, two restrictions are applied. The first is to prohibit concatenation at cusps and corners. This restriction is required because these points seldom should be concatenation points. The second restriction is to use digram (letter pair) statistics to define the penalty of concatenating two letters. For example, concatenating '*v*' and '*q*' should be penalized more than concatenating '*u*' and '*q*'. This is because '*v*' almost never follows '*q*' in a word.

2.5. Comments

In algorithm 1, there are two weaknesses. First, the recognition features are too complicated. Second, training would be very tedious because the number of training samples required by the statistical classifier is large.

In algorithm 2, there are also two weaknesses. First, the user must train every word to create its feature sequence in the dictionary. Second, since the algorithm is based on recognition of primitives, it is very sensitive to local writing variations as discussed in chapter 2.

Algorithm 3 and 4 demonstrate that the *DTW* algorithm can be applied for cursive script recognition. Algorithm 4 is especially interesting because it is very similar to the discrete symbol recognition algorithm. Since very high recognition has been achieved by this algorithm, the complicated shape normalization procedures in algorithm 3 may not be necessary.

As such, the *DTW* matching based algorithms which were developed for discrete symbol recognition in previous chapters were further studied to see how they could be extended for cursive script recognition.

3. DTW for cursive script

In order to simplify the problem, it is assumed that the user writes the "body" of a script first and then followed by the second stroke of the letters *i, j, t, and x.* For the script illustrated in Fig. 7.4, stroke 2, 3, and 4 can be written in any order but must be after stroke 1. In this section, only recognition of the "body", which is a concatenation of the first stroke of letters, will be covered.

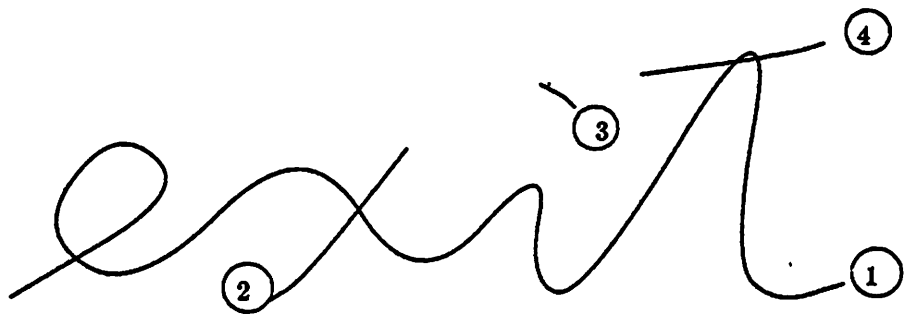


Figure 7.4 Delayed strokes

3.1. Two pass DTW algorithm

Fig. 7.5 shows a script and four templates. If slope code is used as the sole recognition feature, Fig. 7.6 shows the local distance matrices between the script and templates. An algorithm is needed to find the constituent letters of the script from these matrices.

When an unknown discrete symbol was matched with a template, the *DTW* path had to start from (1,1) in the local distance matrix. This is because U_1 is the start point of the unknown symbol and it should match with T_1, the first point of the template T. This requirement is implemented by initializing the accumulated distance $D_{i,0}$ to be ∞ for $1 \leq i \leq I$.

If $D_{i,0}$ is initialized to 0 instead of ∞, the path to $(i,1)$ is definitely from $(i,0)$ since there is no accumulated distance. Therefore $(i,1)$ is a start point of warping paths which ignore (U_1, ... U_{i-1}). This trick has been used for endpoint relaxation as shown in Fig. 3.18. If the endpoint relaxation is extended and all $D_{i,0}$, $1 \leq i \leq I$, are initialized to be 0, the matching paths can start from any $(i,1)$, $1 \leq i \leq I$. Define f_i as the start point of the warping path which ends at

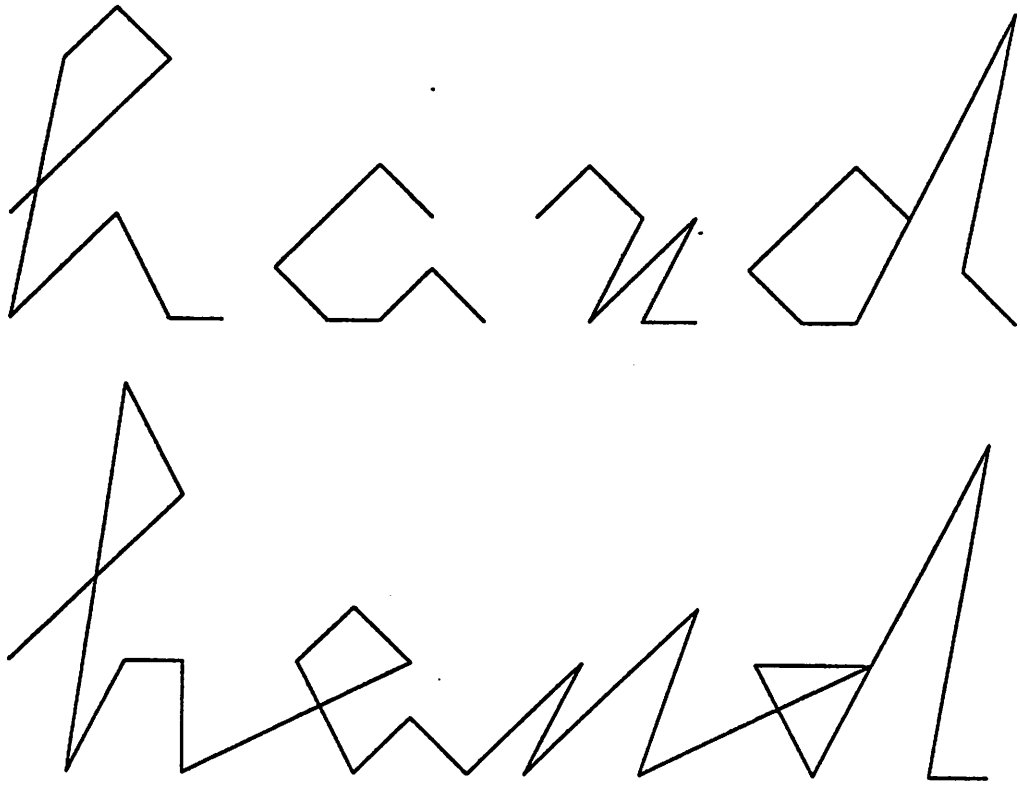


Figure 7.5 Letter templates and a script

(i,J). The physical meaning of the path from $(f_i,1)$ to (i,J) is: if U_i is the end point, the segment $(U_{f_i}, \ldots U_i)$ matches the template best and the distance is $D_{i,J}$.

This phenomena is just what is needed to identify letters in cursive script. Fig. 7.7 shows the accumulated distance matrices with all $D_{i,0}$ being set to 0. Notice in Fig. 7.8 that the paths corresponding to the "dips" of the top row distances reveal the segments in the script which match the template when slopes are used.

With all the matching paths and distances from Fig. 7.7, a directed graph can be constructed. Fig. 7.9 illustrates part of the graph for the example. The branch B_i^k in the graph corresponds to the path which ends at i while matching the script with template T^k. Its distance is $D_{i,J}^k$. It can be seen from Fig. 7.9 that the script can be recognized as a concatenation of templates which yields the minimum accumulated distance in the graph.

The problem of finding the optimal concatenation can be again solved by dynamic programming. Let G_i denote the minimum accumulated distance at U_i and f_i^k denote the start

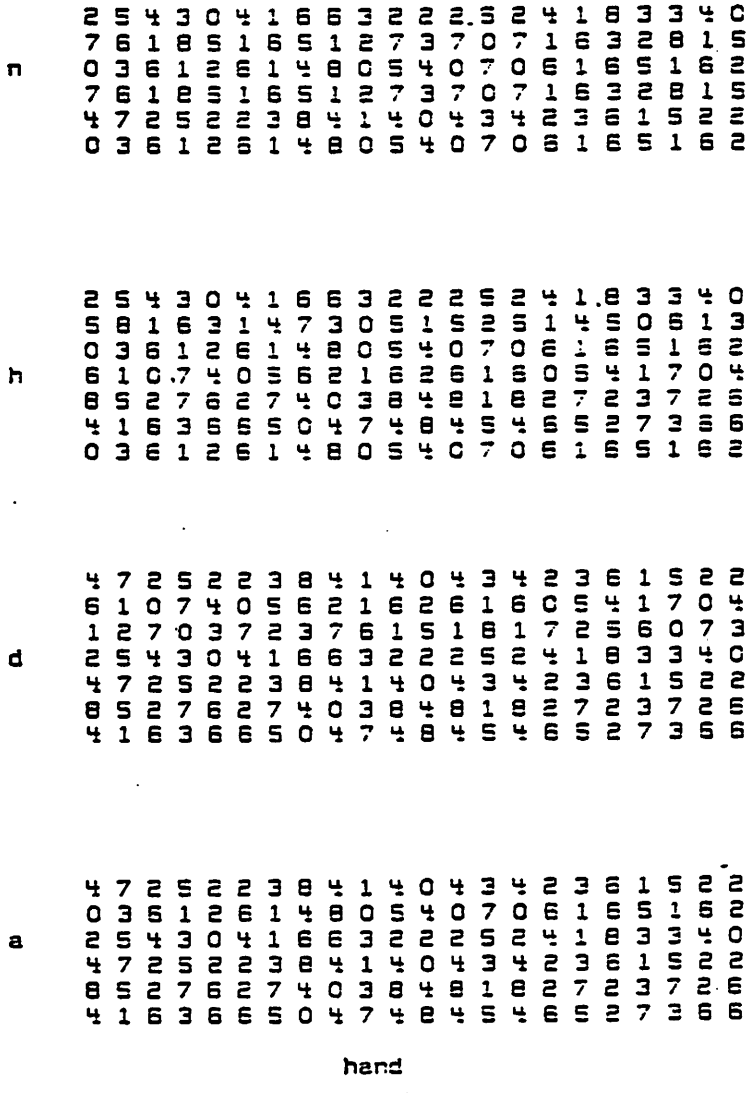


Figure 7.8 Local distances for script recognition

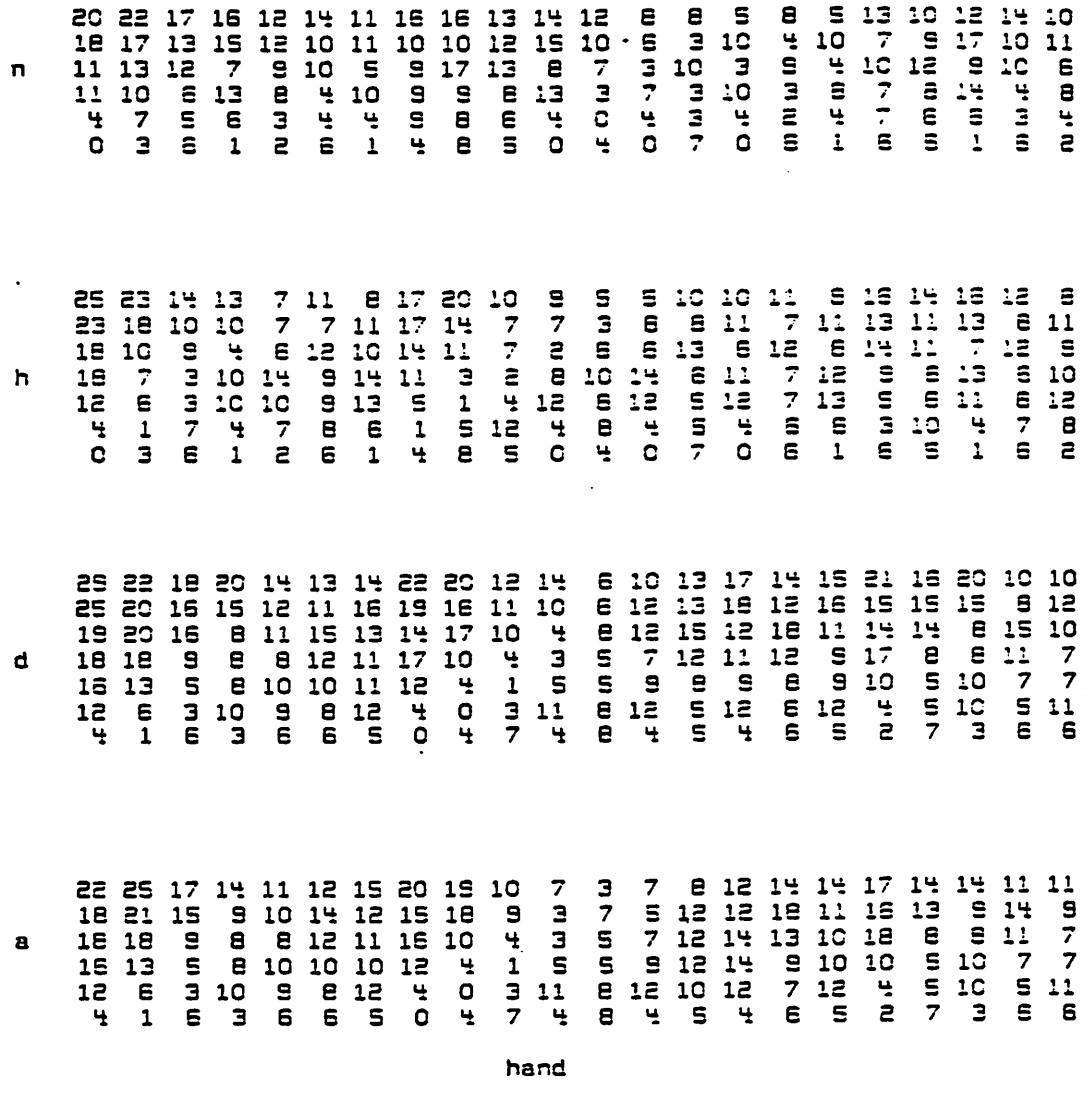


Figure 7.7 The first pass accumulated distances

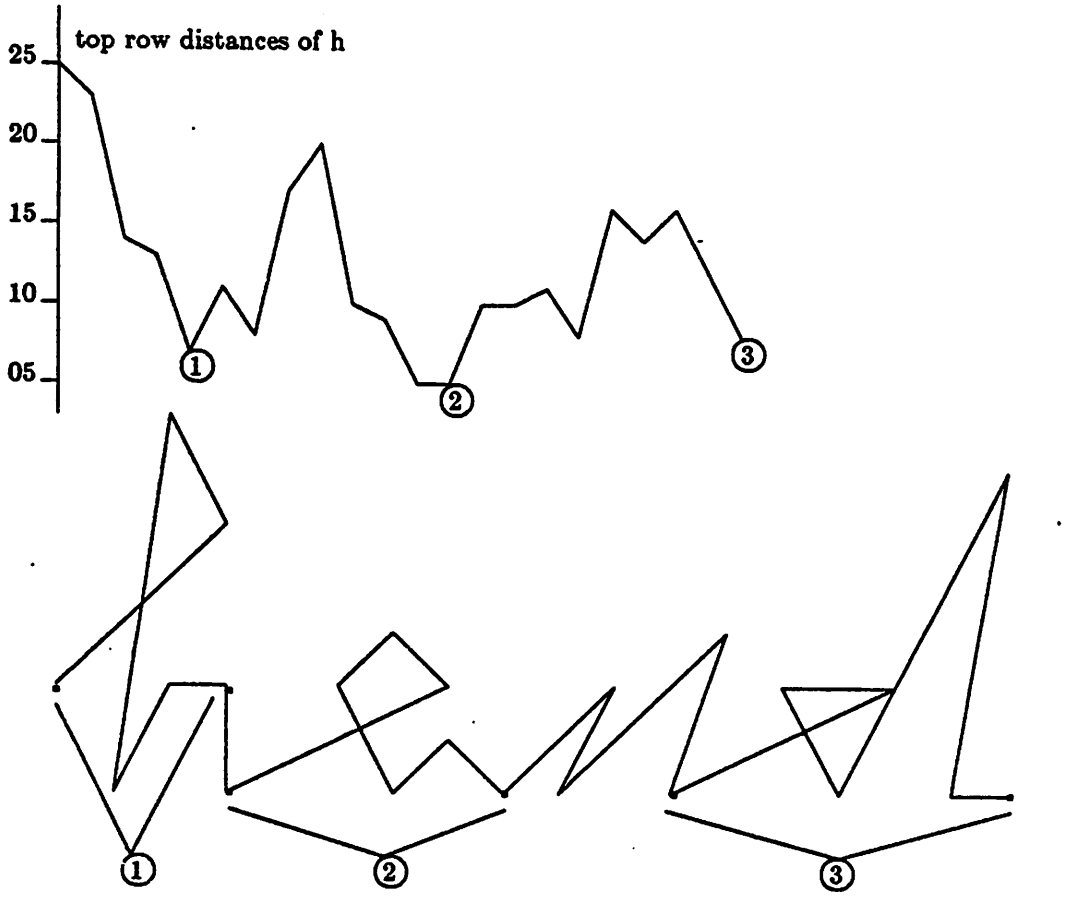


Figure 7.8 Segments of the top row dips

point of B_i^k. G_i is obtained by:

$$G_i = \underset{1 \le k \le K}{MIN} (D_i^k + G_{f_i^k}) \ ,$$

with $G_0 = 0$. After obtaining all G_i, the script is recognized by backtracking the B_i^k which renders G_I.

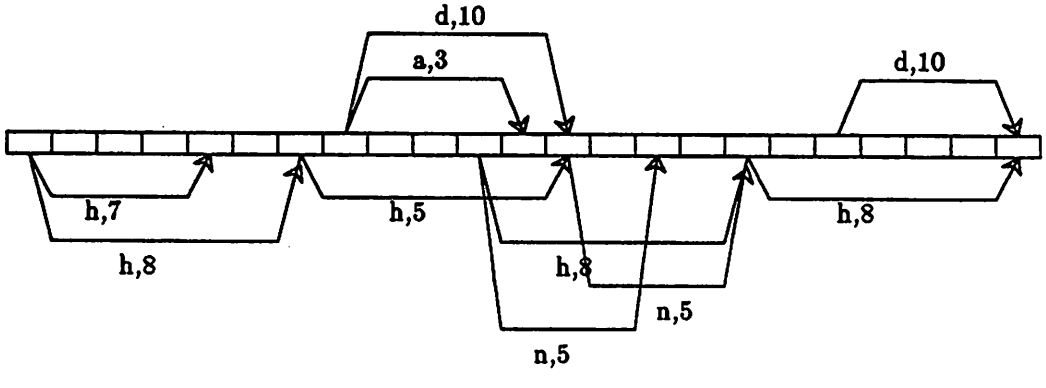


Figure 7.9 Cursive script recognition from directed graph

3.2. Single pass DTW algorithm

Fig. 7.10 illustrates another view of the cursive script recognition problem. It shows the ideal warping paths of matching the script '*hand*' with its letter templates. Let $D^k_{i,j}$ denote the accumulated distance at (i,j) in the local distance matrix of U and T^k. If the distance accumulation is allowed to go beyond template boundary, i.e. $D^k_{i,1}$ is determined by

$$D^k_{i,1} = d^k_{i,1} + MIN(\underset{1 \le k \le K}{MIN} D^k_{i,J^k} , D^k_{i-1,1}) \ ,$$

$D^k_{i,j}$ would be the optimal accumulated distance at (i,j,k) when comparing $(U_1, U_2, ..., U_i)$ with all possible template concatenations up to T^k_j.

Fig. 7.11 shows the accumulated distances obtained by this algorithm. The $D^4_{15,4}$ (which is obtained by comparing $(U_1, \cdots, U_{15})$ with a concatenation of *h*, *a*, and the first 4 elements of *n*), is less than the distance of comparing $(U_1, \cdots, U_{15})$ with any other possible concatenations up to T^4_4.

This algorithm can be thought of as a combination of the two passes of the previous algorithm. The script is recognized by backtracking the matching path which renders the minimum D^k_J. It is very attractive because it takes only one dynamic programming pass. Since dynamic programming is very time consuming, this algorithm should be much faster than the two-pass *DTW* algorithm.

3.3. Extract templates from scripts

There is always some difference between a discrete letter and a script letter. As illustrated in Fig. 7.5, the ligatures which link letters into a script are hard to produce when the letters are written discretely. In both one-pass and two-pass *DTW* algorithms, the end point of one template and the start point of the next template are forced to be consecutive. This makes the recognition vulnerable to the ligatures. To solve the problem, the letter templates should have natural ligatures. This is achieved by extracting the templates from scripts. The procedure is as follows:

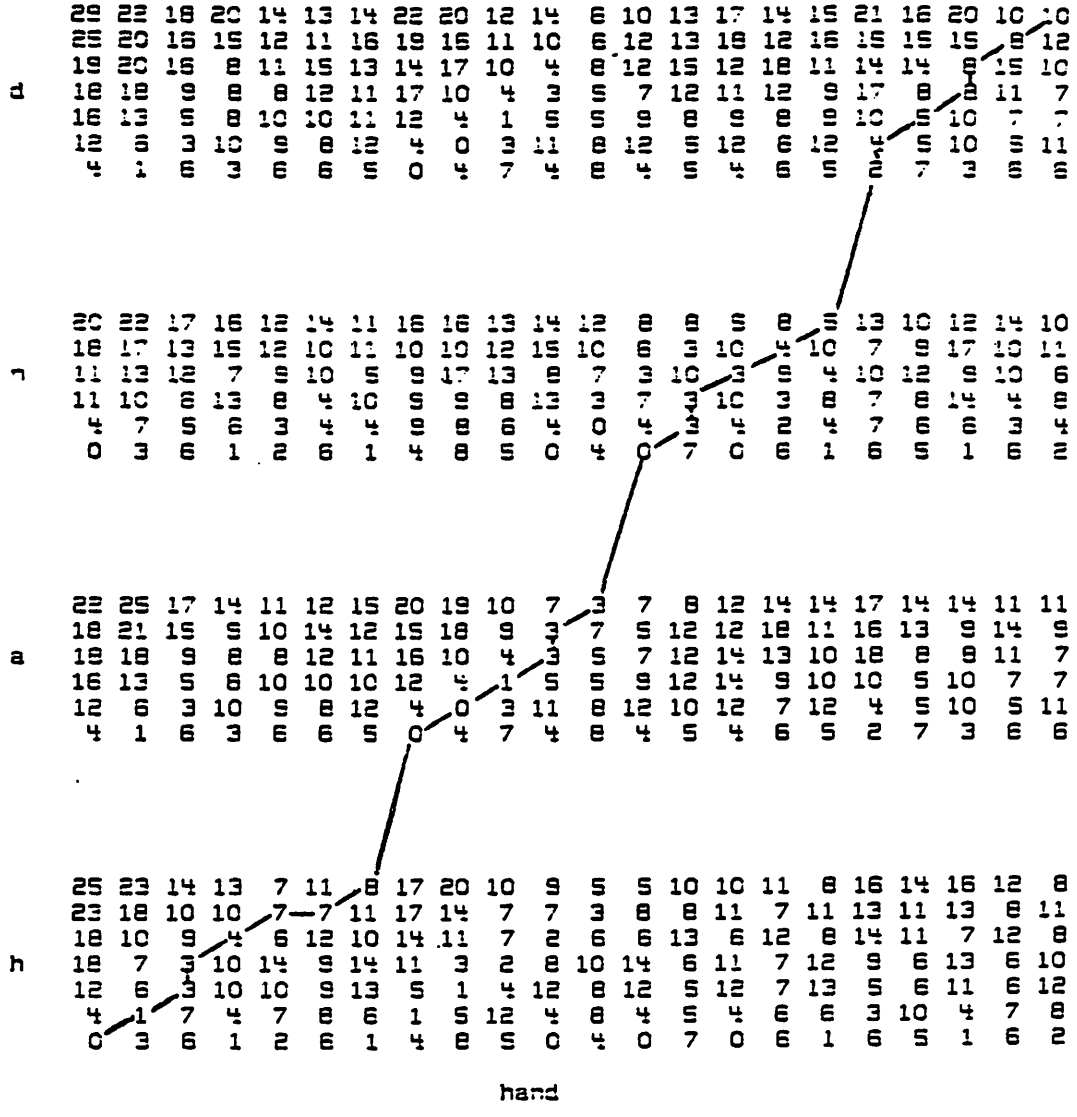


Figure 7.10 The optimal global matching path

	25	27	22	24	18	17	18	26	26	20	22	14	18	23	25	23	24	30	30	34	26	26
	25	20	20	19	16	16	20	24	22	19	18	14	20	21	27	21	26	30	29	31	24	28
	19	20	20	12	15	19	18	20	25	18	12	16	14	22	21	29	26	29	30	24	31	31
d	18	21	13	12	12	16	17	23	18	12	11	13	15	20	21	24	23	31	24	24	29	28
	16	16	9	12	14	16	18	20	12	9	13	13	17	19	20	22	25	26	21	26	29	30
	12	9	7	14	20	15	20	12	8	11	19	23	23	18	24	22	29	20	21	29	30	36
	4	5	11	14	19	13	16	8	12	19	22	23	15	20	20	26	27	19	25	29	34	32

	20	22	17	16	12	14	11	17	25	22	23	23	23	25	12	19	18	24	26	28	32	31
	18	17	13	15	12	10	16	19	19	21	25	21	27	20	21	15	21	23	25	33	31	36
n	11	13	12	7	9	15	14	18	23	21	18	22	20	21	14	20	20	26	29	30	35	31
	11	10	6	13	15	13	19	15	16	19	24	20	22	14	21	19	24	24	25	31	29	34
	4	7	5	10	12	14	15	20	16	17	21	18	15	14	16	18	21	27	23	26	29	30
	0	3	9	10	12	13	12	12	20	25	18	19	11	18	16	22	23	22	27	27	33	29

	22	25	21	18	15	18	19	25	25	18	15	11	15	16	20	22	25	30	30	30	27	29
	18	21	19	13	14	19	17	21	26	17	11	15	13	20	20	26	24	29	29	25	30	30
a	16	21	13	12	12	16	17	23	18	12	11	13	15	20	21	24	23	31	24	24	28	28
	16	17	9	12	14	15	18	20	12	9	13	13	17	19	20	22	25	26	21	26	28	30
	12	10	7	14	20	15	20	12	8	11	19	23	23	16	24	22	29	20	21	29	30	36
	4	5	11	14	19	13	16	8	12	19	22	23	15	20	20	26	27	18	25	29	34	32

	25	23	14	13	7	11	8	17	23	21	20	16	15	21	21	22	19	27	30	30	31	27
	23	18	10	10	7	7	11	18	20	15	18	14	19	19	22	18	22	27	27	32	27	30
	18	10	9	4	6	12	13	17	22	18	13	17	17	24	17	23	23	29	30	25	32	33
h	18	7	3	10	14	14	19	22	14	13	19	21	25	17	22	22	27	29	25	32	31	35
	12	6	3	10	16	18	22	16	12	15	23	26	19	16	24	22	29	24	25	32	31	37
	4	1	7	10	16	18	15	12	16	23	22	26	19	16	20	22	25	22	29	29	33	34
	0	3	9	10	12	13	12	12	20	25	18	19	11	18	16	22	20	22	26	27	33	28

hand

Figure 7.11 The one-pass accumulated-distances

(1) The user creates templates of all letters in the same way as in discrete symbol recognition.

(2) The user provides several training scripts.

(3) For each training script, the *DTW* is applied to match the script with templates of its letters. For example, a training script *hand* is matched with templates of *h*, *a*, *n*, *d*.

(4) If the matching result is correct, the matching path is backtracked to obtain the concatenation points between letters. These concatenation points decompose the training script into letter segments. These segments are stored as new training samples of their letters.

(5) After obtaining several letter training samples from training scripts, the clustering program *hclus* is used to obtain new letter templates.

Fig. 7.12 illustrates the four letter segments extracted by matching the word *'hand'* with templates *'h'*, *'a'*, *'n'*, and *'d'*. It can be seen the ligatures are extracted out with letters. Since these ligatures are extracted by the same procedure which will be used for recognition, recognition based on them should be better than the recognition based on discretely created templates.

3.4. Performance evaluation

From the experience in discrete symbol recognition, the template matching has to be pipelined with data acquisition for faster response. Therefore, only the slope code can be used as the recognition feature for template matching. The program for script template matching is in fact almost the same as the one for discrete symbol recognition, except that the $D^k_{i,0}$ has to be set differently.

In the first experiment, the performances of both two-pass and one-pass *DTW* algorithms were tested over 130 scripts. These scripts were carefully written with lines on tablet. However, the recognition rates of both algorithms were very poor, about 50%.

In the second experiment, the normalized y coordinate was used as an additional recognition feature and the local distance was changed to

$$d_{i,j} = U_{y,i} - T_{y,j} + L(U_{s,i} - T_{s,j}) .$$

The y coordinate normalization is defined as illustrated in Fig. 7.13. 16 levels are used to make

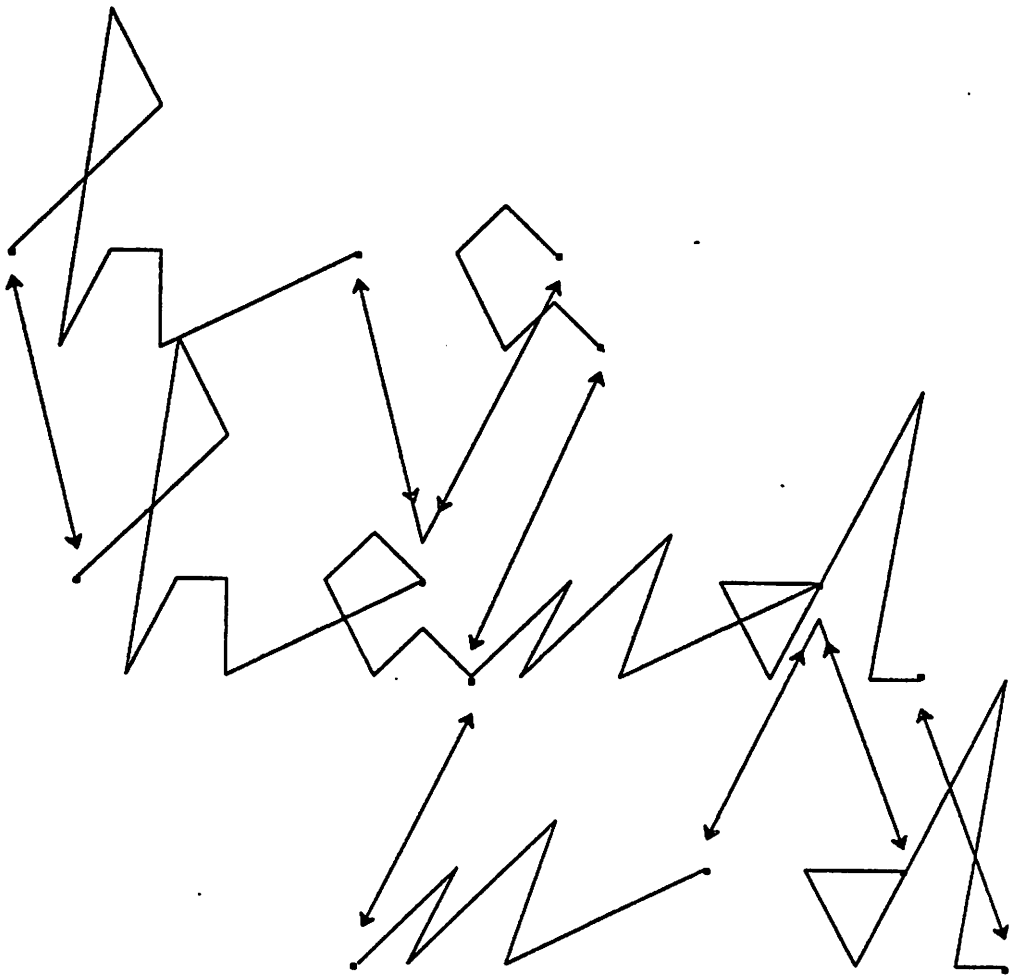


Figure 7.12 Extract templates from script

the y coordinate have the same weight in the local distance as the slope code. However, the experimental results showed that the recognition rates of both *DTW* algorithms could be improved to only 60%.

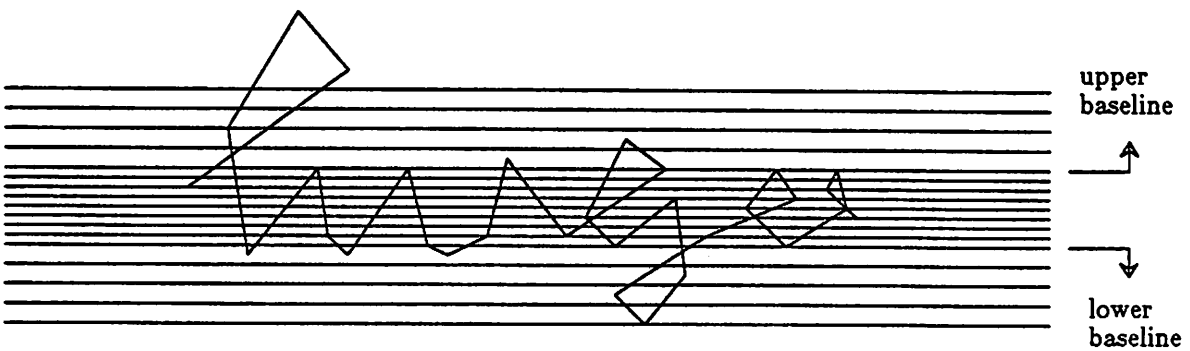


Figure 7.13 Y coordinate quantization

It was found that the short templates, i.e. templates like '*c*' and '*r*' which have few sample points, always caused recognition errors. This fact suggests that the accumulated distance should be normalized.

The way the cursive script accumulated distance is normalized is shown in Fig. 7.14. In the two-pass *DTW* algorithm, the top row distance is normalized to the maximum of the template length and the segment length. In the one-pass *DTW* algorithm, when $\underset{1\le k\le K}{MIN} D^k_{i,J^k}$ is compared with $D^k_{i-1,1}$ to determine whether to allow concatenation, the D^k_{i,J^k} is normalized to $MAX((\sum_{\{p:\ T^p \text{ is on } D^k_{i,J^k}\ path\}} J^p), i-1)$ and the $D^k_{i-1,1}$ is normalized to $MAX((\sum_{\{p:\ T^p \text{ is on } D^k_{J^k_{i-1},1}\ path\}} J^p)+1, i-1)$. The reason for these normalizations is the same as those discussed in chapter 5.

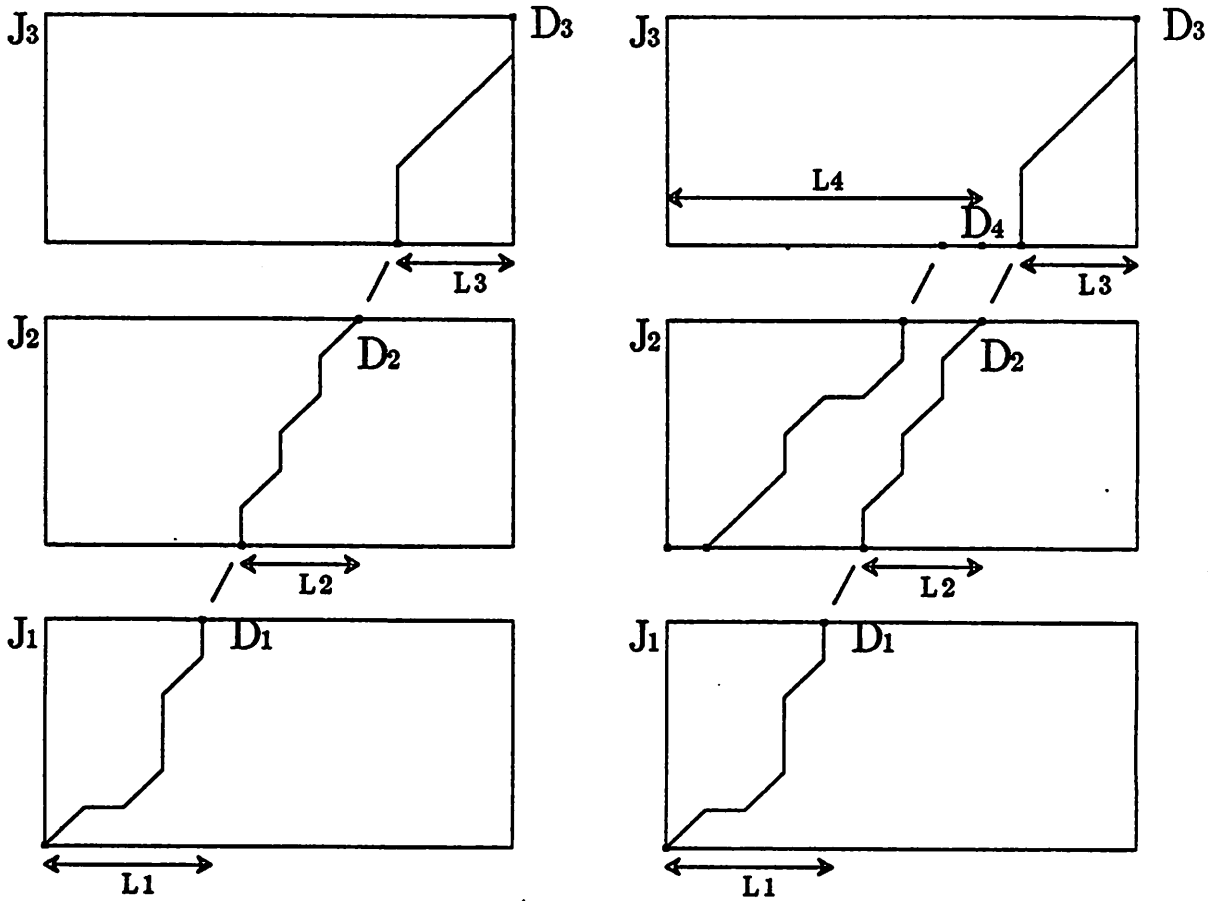
However, with all the normalization effort, the recognition rates were still far from satisfactory. They were about 68%.

Although the results of these experiments were very disappointing, it was found that the behaviors and performances of the one-pass *DTW* matching and the two-pass *DTW* matching were very similar. This implies that the one-pass *DTW* algorithm can replace the two-pass *DTW* algorithm.

4. Syntax-directed template matching

The unsatisfactory recognition results were caused by two problems. First, from experiments conducted in chapter 3, *DTW* template matching alone could achieve only 92% recognition rate for cursive lower case letters. If a script contains four letters, the possibility it would be recognized is $(0.92)^4 = 72\%$. Second, the boundaries between letters in a script are not explicit. The script '*hand*' can be easily recognized as '*hewd*' as shown in Fig. 7.15.

Because of these two problems, the unknown script is usually recognized as a word which does not make any sense. For example, *'back'* is recognized as *'fach'* and *'gun'* is recognized as *'grur'*.



D3/MAX(J3,L3)
D2/MAX(J2,L2)
D1/MAX(J1,L1)

(a) two-pass DTW

D4/MAX(J2+1,L4)
D2/MAX(J1+J2,L1+L2)

(b) one-pass DTW

Figure 7.14 Distance normalization in script recognition

An effort was made to solve these problems by requiring that the output of the recognizer must be syntactically correct. Let $STR^k_{i,j}$ denote the template concatenation up to (i,j,k) in the local distance matrix. (In Fig. 7.11, the $STR^4_{11,4}$ is "*hn*".) To force the recognition result to be an allowed word, the syntax check could be performed at the final stage:

$$U = \underset{\{k:STR^k_{I,J^k} \text{ is in dictionary}\}}{ARGMIN} D^k_{I,J^k} .$$

However, this method has one serious problem. It is too late to find the word is wrong. For example, suppose the unknown is *'hand'* and somehow the best concatenation ended at *'d'* is *'hewd'*. Although *'hewd'* is determined to be incorrect, there is no way to recover the correct

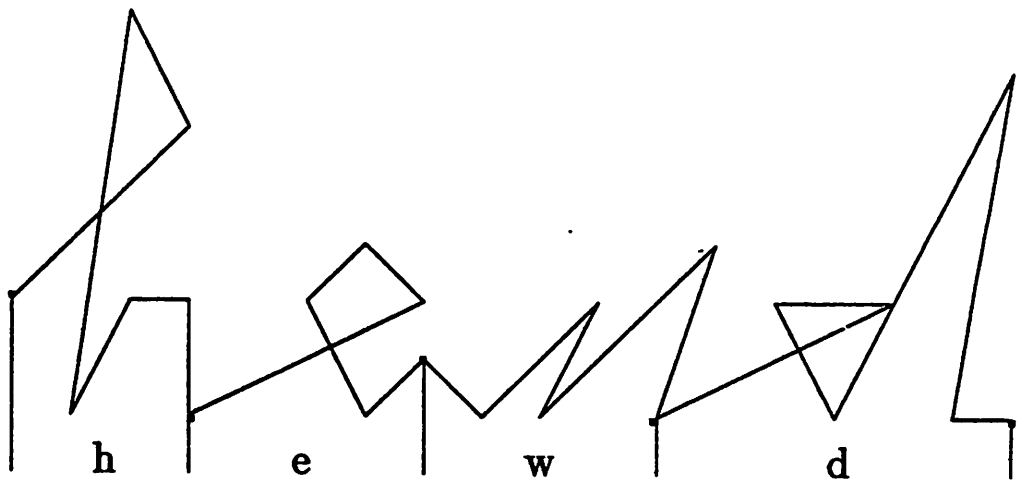


Figure 7.15 Misrecognition due to no explicit endpoints

answer. It would be better if the *'hew'* can be prevented at the concatenation stage.

A better algorithm is as follows. In the one-pass *DTW* algorithm, the concatenation happens at $(i,1,k)$ only when

$$D^k_{i-1,1} > \underset{1\le k\le K}{MIN} D^k_{i,J^k} .$$

Once this is true, the T^k is concatenated with $STR^{\gamma}_{i,J^{\gamma}}$, $\gamma = \underset{1\le k\le K}{ARGMIN} D^k_{i,J^k}$. Therefore, the syntax check can be performed by checking whether T^k is allowed to be concatenated with $STR^{\gamma}_{i,J^{\gamma}}$. In other words, the concatenation can happen only when

$$D^k_{i-1,1} > \underset{\{k' : T^k \text{ can concatenate with } STR^{k'}_{i,J^{k'}}\}}{MIN} D^{k'}_{i,J^{k'}} .$$

Under the syntax check, the concatenation at (8,1,4) in Fig. 7.11, although the concatenation provides less distance, can not happen because *n* never occurs after *h* in all vocabulary words.

To facilitate the implementation, in the training phase, the syntax of all allowed words are constructed as a tree. Fig. 7.16 illustrates the tree for a vocabulary { *al, andy, anny, bo, bob, dave, dean, emma, emmy* }. It can be seen that each node in the syntax tree is a permissive prefix. In the recognition phase, the syntax check algorithm forces each $STR^k_{i,j}$ at a node.

Experimental results showed that the syntax check works extremely well. It boosts the recognition rate to 98%. Since the recognition rate is so high, the slope sequence alone was

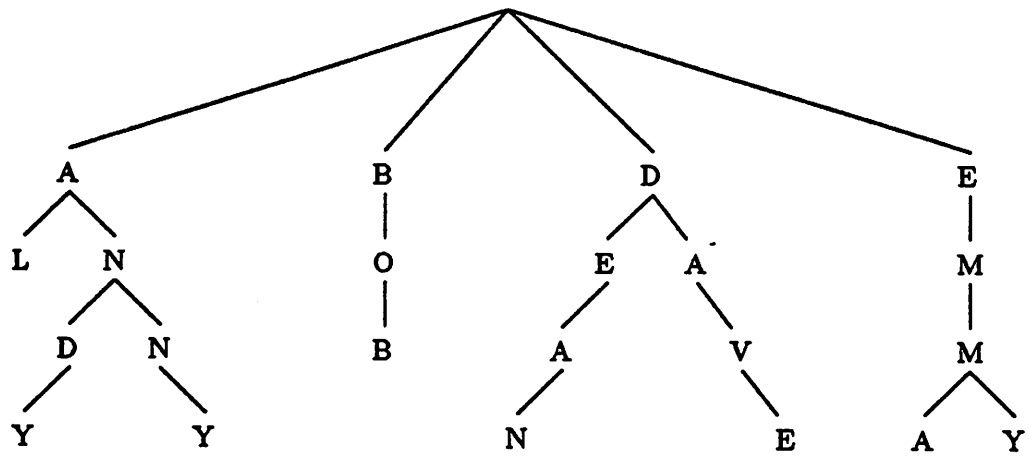


Figure 7.16 Syntax tree

tried on some sloppy scripts of various sizes. The recognition accuracy is always above 90%.

5. Improving the syntax-directed algorithm

Although the syntax directed single-pass *DTW* algorithm works very well for "regular" vocabularies, it has four limitations.

(1) For each template, this algorithm only gives similarity between the unknown script and the best matching of words which end with its letter. This algorithm does not provide the similarities between the unknown script and other words. For example, if both '*fell*' and '*bell*' are in vocabulary and the *DTW* computes that '*bell*' is the best word ended at '*l*', there is no way to tell what is the distance between the script and '*fell*'.

(2) If the prefix of a word is eliminated, the word has no chance to be recovered. This phenomena can be illustrated in Fig. 7.17. Suppose the '*b*' and '*f*' in Fig. 7.17 are two other templates along with the templates in Fig. 7.5. Suppose there are only two words {*bad, fan*} in the vocabulary. Although the distance between the script '*bad*' in Fig. 7.17 and the concatenation of *'b', 'a', 'd'* is less than the concatenation of *'f', 'a', 'n'*, it will be recognized as '*fan*' by the syntax-directed algorithm. This is because the distance between the '*b*' in '*bad*' and the '*b*' template is worse than the distance between it and the '*f*' template. This forces the '*a*' to concatenate with the '*f*' and the word '*bad*' has no chance of being evaluated again.

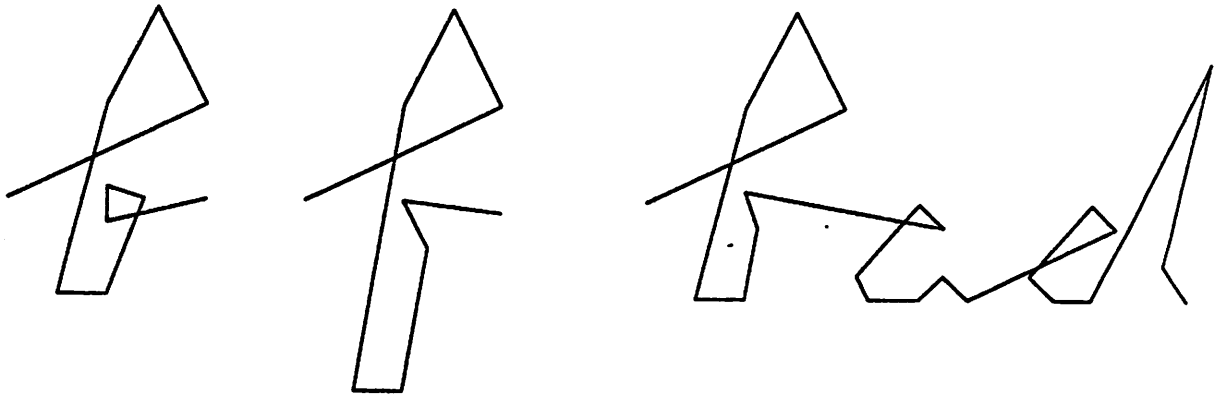


Figure 7.17 Error due to bad start of a script

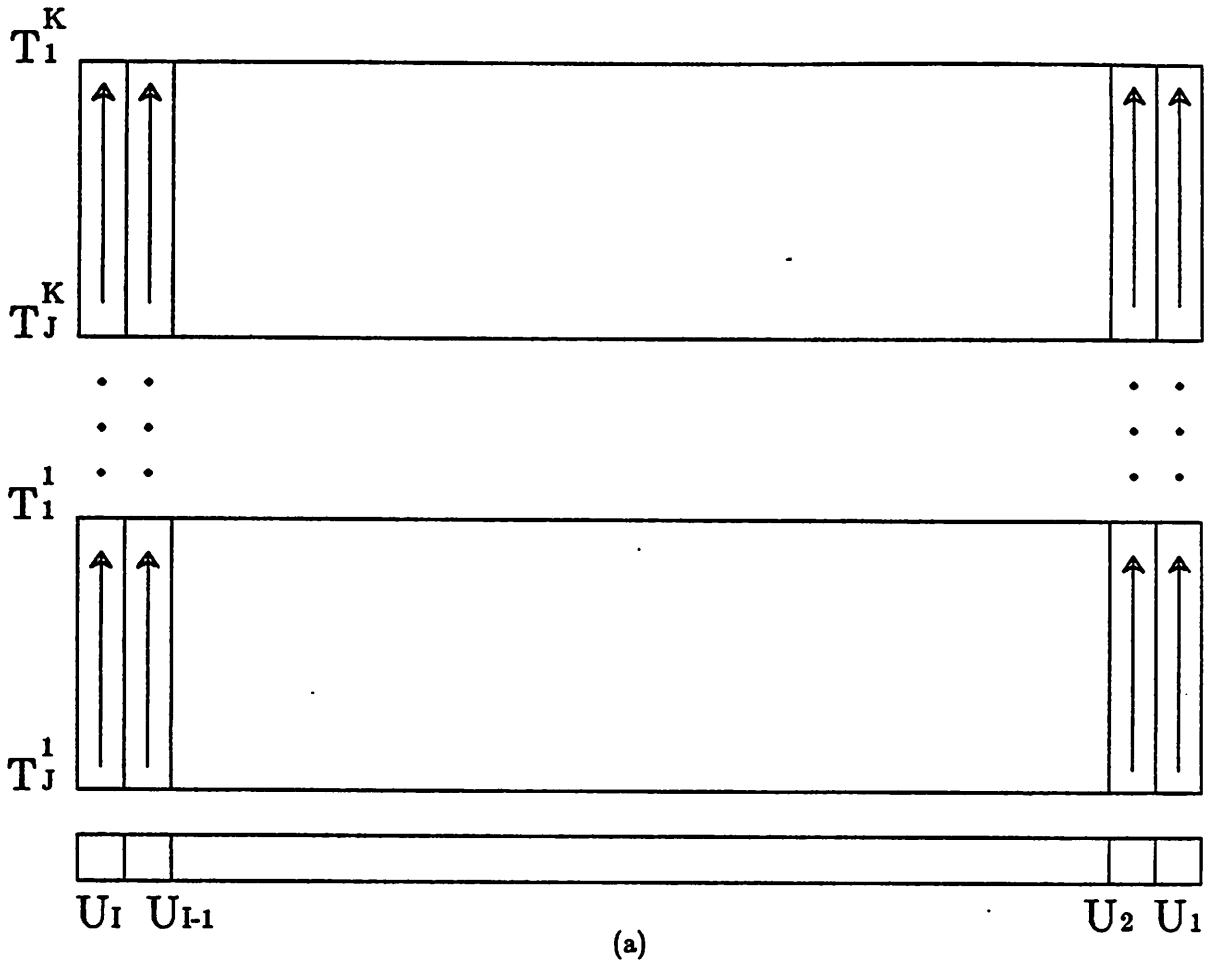
(3) The syntax check only guarantees the final concatenation is a permissive prefix instead of a complete word. For example, if both '*home*' and '*hem*' are in vocabulary, the algorithm may come up with that the best concatenation ended by '*m*' is '*hom*' instead of '*hem*'. Even though '*hom*' is not a word in vocabulary.

(4) If normalized y coordinate are used with slope code for matching, unless the user carefully controls the sizes of the letters, the recognition rate does not differ much from using slope code alone. However, the computational load is much higher and real-time matching can not be performed. Therefore only the slope can be used for template matching. But, the slope sequence can not differentiate some letters, like (*a*, *d*), (*b*, *f*), and (*e*, *l*), well.

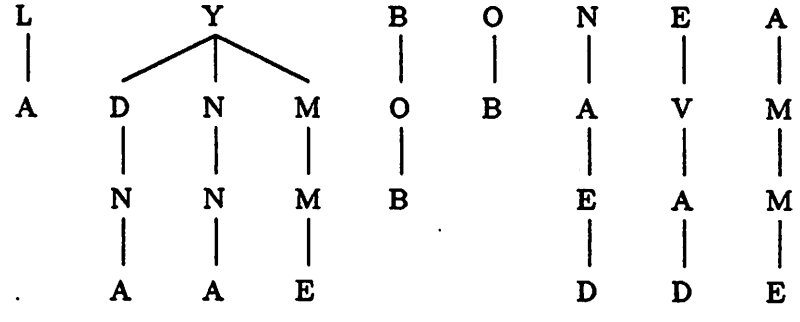
In order to explore solutions of these limitations, a "tough" vocabulary was created which contains {*feel*, *fell*, *bell*, *flea*, *fled*, *bled*}. The syntax directed one-pass *DTW* algorithm has demonstrated poor performance when recognizing these words.

5.1. Playback

To overcome the first and second limitation, it has been tried to match the unknown script with each template backwards. As illustrated in Fig. 7.18, the playback matching starts from U_I and ends at U_1. The distance accumulation computation with T^k starts from $T^k_{J^k}$ and ends at T^k_1. The concatenation of the playback matching is directed by another syntax tree which is formed by words in vocabulary but reversely spelled.



(a)



(b)

Figure 7.18 Playback matching of script

From playback matching, three new clues can be obtained.

(1) For words ending with the same letter but starting with different letters, the playback matching can reveal the distances of both. For example, the distance between the unknown script and the word '*fell*' can come out at *f* and the distance between the unknown

script and word *'bell'* can come out at *b* in playback matching.

(2) Words which are eliminated due to a "bad" prefix can be recovered. This is because the structure of the syntax tree for playback matching is completely different from forward matching.

(3) *DTW* is not a reciprocal procedure. The result of playback gives a distance from another point of view.

Experimental result showed that for the "tough" vocabulary, the playback matching substantially improves the first three limitations of forward matching and the recognition rate is increased to 75%. However, the inadequacy of slope sequence still degrades the recognition rate.

One drawback of the playback matching is that it can not start until the whole script is finished, and hence doubles the recognition time.

5.2. Profile check

It was found that the heights of letters which can not be distinguished well by the slope sequence are quite different. Therefore, the weakness of slope sequence may be corrected by checking the profile of a script. Two algorithms have been tried for this purpose.

As shown in Fig. 7.19, the profile of a script letter was divided into four types: *a*-type, *b*-type, *f*-type, and *g*-type. With the types of all letters, a *profile string* is assigned to each word. For example, the profile string for *bled* is *(b b a b)* and the profile string for *fled* is *(f b a b)*.

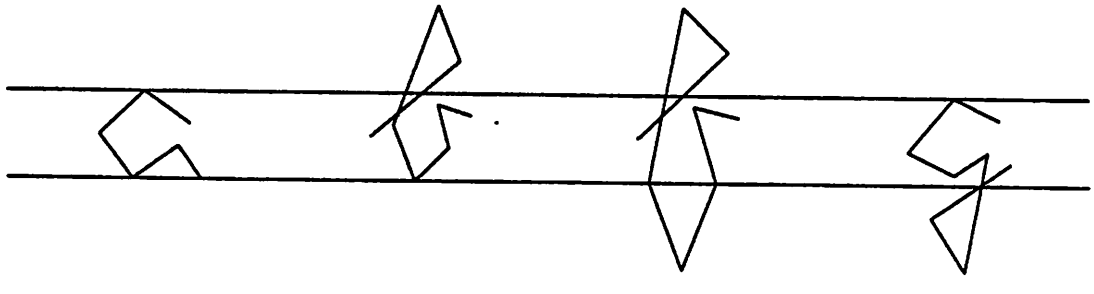


Figure 7.19 Profile types

To check the profile, the baselines of the unknown script must be first determined. Then the baselines are used to determine the profile type of each letter. If the profile string of a word

is not the same as the profile string of the unknown, that word is eliminated. For example, the script *bled* can not be recognized as *fled* if its profile string is detected as *(b b a b)*.

To determine the baselines, Berthod proposed in algorithm 2 to use local Y-maximum and Y-minimum. Assume there are N local Y-maximum in a script, the upper baseline is at the $\left[\frac{N}{2}\right]$ Y-maximum. The lower baseline is obtained by the same procedure but applied to the local Y-minimum. Burr proposed in algorithm 3 to use the Y-histogram. The upper and lower baselines are at the first two points in the histogram whose values are 75% of the histogram maximum. Both algorithms have been tried out and, unfortunately, the experimental results showed that no fixed threshold (like $\left[\frac{N}{2}\right]$ or 75%) could work well over a wide range of scripts.

Since the baselines can not be obtained reliably, another possibility is to use a *profile transition type* instead of profile type to characterize the profile. As shown in Fig. 7.20, 9 profile transition types were defined: *d (pull up), q (pull down), p (push up), b (push down), / (shift up), \ (shift down), < (expand), > (shrink), and - (no change)*. These transition types are determined by comparing the Y-maximum and Y-minimum of a letter with the TU (top upper), TL (top lower), BU (bottom upper), and BL (bottom lower) thresholds of the preceding letter as illustrated in Fig. 7.20. These thresholds are set at 1/4 of its height above or below its Y-maximum or Y-minimum.

With these profile transition types, a new profile string can be assigned to each word in vocabulary. For example, the string for *bled* is *(- b d)* and the string for *fled* is *(p b d)*. The new profile string is used in the same way as before. It eliminates words whose profile string is not the same as the unknown script. For example, suppose '*fled*' is chosen as the word which, from all words ended by *d*, matches the unknown script best. If it is found that the profile string of the concatenation for '*fled*' is *(- b d)*, then '*fled*' would not be considered since its profile string should be *(- b d)*.

This algorithm works better than the baseline algorithm because it uses the heights of adjacent letters rather than the whole script. It is easier for a user to do "local" height control

rather than "global" control.

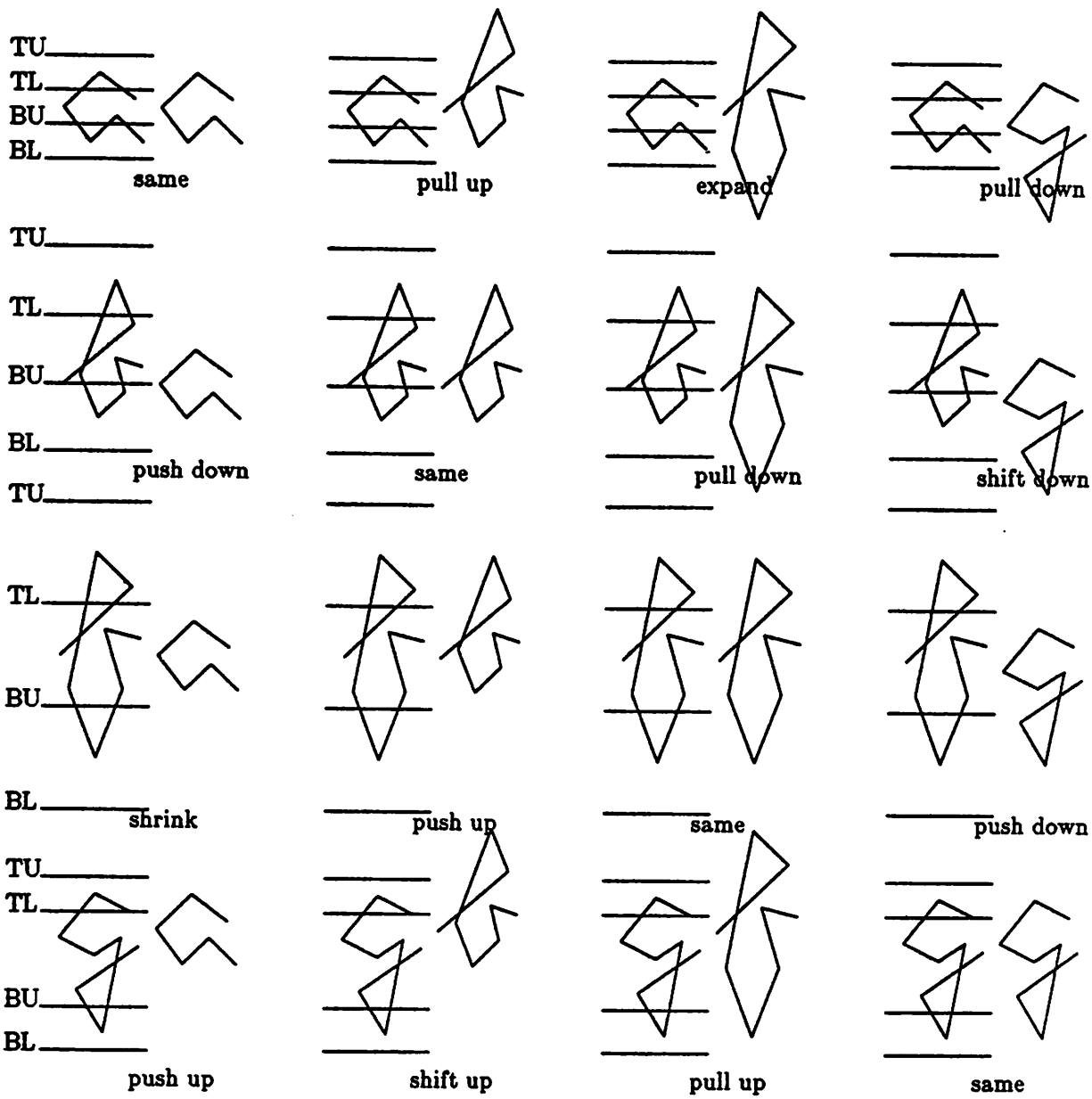


Figure 7.20 Profile transitions

Experimental result showed that the effects of this algorithm are mixed. On one hand, if the user can control the script profile well, this algorithm effectively compensates the weakness of slope sequence. The recognition rate for the "tough" vocabulary is enhanced to 90%. On the other hand, if the user does not control the script profile well, the profile check results in recognition error because the profile strings do not match.

If there are no printed lines on the tablet, it is not easy to control the profile. However, if there are printed lines on the tablet, it is not hard to control the profile after a short practice. The rule of thumb is to stretch the Y-maximum and Y-minimum of each letter to the lines as shown in Fig. 7.21.

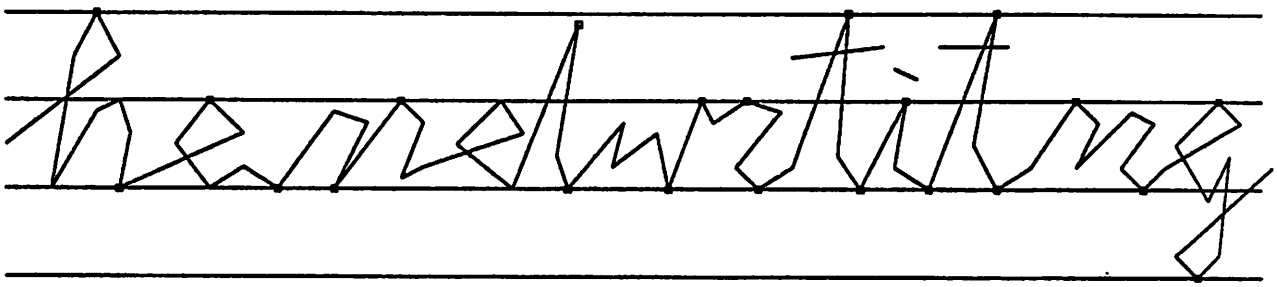


Figure 7.21 Avoid profile errors

Since the profile check needs the maximum and minimum y coordinate of each letter segment in a script, the template matching program was changed and the maximum and minimum y coordinate associated with each matching path are obtained in a similar way as the $f_{i,j}$ in *SC_DTW* (chapter 3). This modification achieves a faster response by doing as much computation as possible in the sampling period.

6. Revisit two-pass *DTW* algorithm

Although the playback matching and the profile check significantly increase the recognition rate of ambiguous scripts, they also increase the execution time of the algorithm. In addition, the matching distance of *each* word is needed in order to determine the error source and design a disambiguation scheme.

To obtain the matching distance of each word, those local distance matrices which are obtained from the first pass of the two-pass *DTW* algorithm, i.e. the matrices in Fig. 7.7, were reviewed again.

The local distance matrices are computed in parallel with data acquisition. They are available right after the script is finished. Instead of using the the second pass of the two-pass *DTW* algorithm presented before, the following algorithm was used to obtain the distance of each

word.

Let $T(l)$ denote all templates for the *lth* letter. For word $W=(L_1L_2\cdots L_W)$ (L_w is a letter), a depth first distance concatenation is performed starting from $T(L_W)$ to $T(L_1)$. During concatenation, a "grace period" is allowed between letters in order to solve the ligature problem. The profile transition is also checked. Once a violation is found, that concatenation is eliminated. If all concatenations end up too far from U_1, or get to U_1 before evaluating the first letter ($T(L_1)$), this word is also eliminated.

For example, suppose the vocabulary is {*an, and, hanh, hand* }, and five templates, two *a* and one *d*, *h*, and *n*, are available. Suppose the concatenation graphs for these words are as illustrated in Fig. 7.22. The graph of the word '*an*' is not acceptable because the transition from '*n*' to '*a*' is (b) but it should be (-). The word *'and'* is eliminated because it ends too far from the script start point. The word '*hand*' is chosen as the winner because it has the minimum concatenation distance.

Experimental results showed that the accuracy of the new syntax directed concatenation algorithm is as good as the syntax directed matching algorithm with playback matching and profile check. However, because there is no playback matching, the response time is improved significantly.

After the profile check, there are not many words left. For each word, not too many branches are expected unless the user has many templates for a letter. The required computation time is not excessive.

7. Profile check mark

With the profile check, the user has to be careful about the relative heights of letters in a script. Sometimes this extra attention is annoying because most words in a regular vocabulary can be recognized without profile check. In the algorithm presented in the last section, since the distance evaluation and profile check are performed on a word-by-word basis, the profile check need not be performed over all vocabulary words.

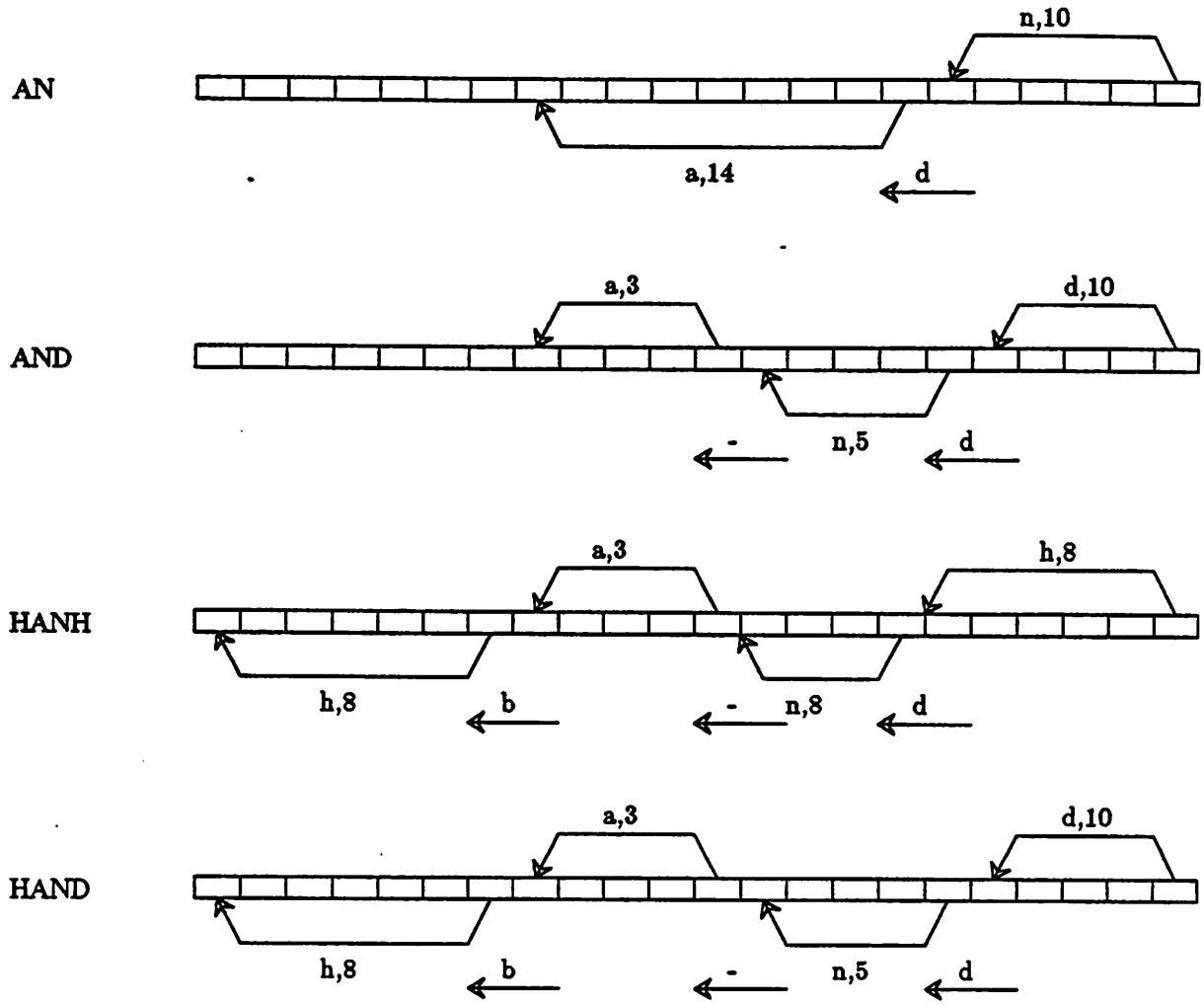


Figure 7.22 Modified two-pass *DTW* algorithm

As such, it is up to the user to decide which words need the profile check. For those ambiguous words, the user just puts a ***profile check mark*** after the words in the vocabulary file. In the recognition phase, the profile check is only performed for words with the check mark.

8. Delayed strokes

The delayed strokes are classified into three categories: ., –, and / as illustrated in Fig 7.23. With this definition, a ***stroke string*** can be assigned to each word in the vocabulary. For example, the string of the word *'quit'* is (. –) and the string of the word *'exit'* is (/ . –).

During recognition, the "body", i.e. first stroke, of the unknown script is first matched with the first stroke of each template using the single-pass *DTW* algorithm discussed above. Then,

the type of each delayed stroke is determined by a simple criterion as illustrated in Fig. 7.23. These types are then rearranged as a stroke string in ascending order of the x coordinate of the middle point of each delayed stroke. This stroke string is used to eliminate candidates which do not have the same stroke string. After the elimination, the one with the minimum template matching distance is selected as the word of the script.

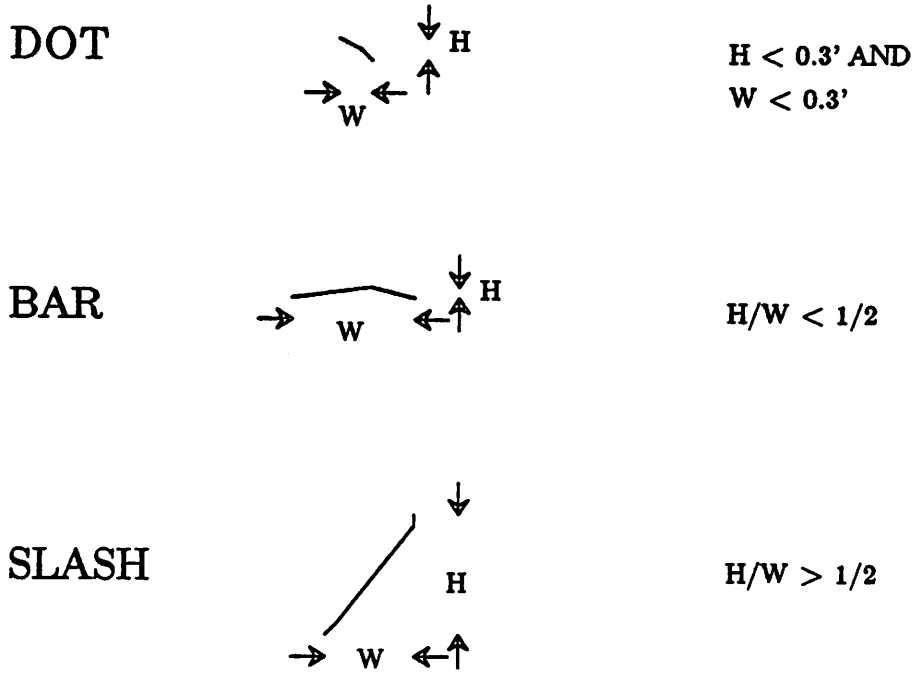


Figure 7.23 Recognition of delayed strokes

For example, suppose the vocabulary is { *erect*, *excel* }. A user's script *erect* can not be recognized as *excel* if the second stroke is correctly recognized. This is because the stroke string of *erect* is (−) and the stroke string of *excel* is (/).

Notice that this algorithm allows the user to write the delayed strokes in arbitrary order, as long as he finishes the script body first.

9. Implementation

Because of the limitation of the size of a RAM resident program, the cursive script recognition program is not included in the *TSR* as in the discrete symbol recognition case. Instead, a dedicated program *cscript* is provided for algorithm experiment. In *cscript*, four commands are

provided: *vocab*, *train*, *recog*, and *analy*. *Vocab* is used to list all words in vocabulary. The usage of the other three commands are explained in next section.

10. Adaptive training

Based on the "*train-on-error*" approach, the system can achieve a high recognition rate if the user performs training when recognition errors occur. The adaptive training procedure is similar to the discrete symbol recognition case as shown in Fig. 6.6. Following is a brief description.

(1) Create templates of all letters using *htrain* and *hclus* as discussed in last chapter. (The *hdisam* is not needed because the disambiguation rules are not used in cursive script recognition.)

(2) For each vocabulary, create a file which contains all the words.

(3) If the user wants to obtain letter templates which are extracted from words in the vocabulary, he can use the *train* command in *cscript*. After new letter training samples are extracted from training scripts, use *hclus* to create templates.

(4) After the initial training, load the templates and vocabulary. Then, use the *recog* command to start testing.

(5) If a recognition error occurs, the user can first examine whether the error is due to the insufficiency of the slope sequence, e.g. *fell* is recognized as *bell* or *feel*. If this is the case, the user should put a profile check mark after those ambiguous words in the vocabulary file. Then, be careful about the profile when writing those words. If the cause of the error is not obvious, the user can use the *analy* command to compare the just written script with suspicious words. The *analy* can show the breakdown of the script and the matching distance of each segmented letter. If a letter in this script is quite different from its templates, the user should use the *'train -adapt'* command to extract these letters out and store them as new letter training samples.

(6) After obtaining several new letter training samples, use *hclus* to consolidate these training samples and create new templates.

11. Conclusion

In this chapter, an algorithm was successfully developed which can recognize cursive scripts based on templates of letters. The algorithm is an extension of the *DTW* matching algorithm used for discrete symbol recognition. Because of the wide variations of scripts, syntax information is used to achieve a high recognition rate. The experimental results showed that with syntax check, the high recognition accuracy can be achieved without too much training.

Although the simple syntax-directed one-pass *DTW* matching algorithm works well for regular words, it does not work well for ambiguous words. Playback matching and profile checking were tried to remedy the problem. However, these two remedies can not satisfactorily solve the problem.

It is finally realized that the best way to recognize a script is to obtain and compare the matching distance between an unknown script and each vocabulary word. This is done by replacing the dynamic programming concatenation algorithm in the second stage of the original two-pass *DTW* algorithm by a simple heuristic concatenation algorithm. By allowing grace period in concatenation and only performing profile check on specified words, not only better recognition rate is achieved, but also fewer restrictions are put on the user.

This algorithm provides almost 100% recognition rate if the user performs the adaptive training properly. It was implemented in a package that a user can easily do the adaptive training. In term of recognition speed, the response time of our IBM PC implementation is about 0.3 seconds per vocabulary word.

References

1. P. Mermelstein and M. Eden, "Experiments on Computer Recognition of Connected Handwritten Words," *Information and Control*, pp. 255-270, 1964.

2. M. Eden, "Handwriting and Pattern Recognition," *IRE Trans. Information Theory*, vol. IT-8, pp. 160-166, 1962.

3. M. Berthod and S. Ahyan, "On-Line Cursive Script Recognition : A Structured Approach With Learning," *Proc. International Conference on Pattern Recognition*, 1980.

4. B. Burr, "A Normalizing Transformation For Cursive Script Recognition," *Proc. IEEE Conference on Image Processing and Pattern Recognition*, 1982.

5. C. C. Tappert, "Cursive Script Recognition by Elastic Matching," *IBM J. R & D*, pp. 765-771, Noverber, 1982.

CHAPTER 8

INTEGRATION WITH SPEECH RECOGNITION

Because of the ability to tolerate variations between two sequential patterns, the *DTW* matching algorithm has played a major role in the success of both the discrete symbol recognition and the cursive script recognition. However, the large number of computations required by the *DTW* severely limit the number of templates which can be matched within a reasonable response time. In this chapter, a *DTW* processor, which is originally designed for speech recognition, will be used to solve this problem. Since the handwriting recognition algorithm is so similar to the speech recognition algorithm, the implementation of the *OHR* on a speech recognition system will be discussed.

1. MARA speech recognition system

MARA is a speech recognition system built at UCB which uses a specially designed integrated circuit for *DTW*.[1] Fig. 8.1 shows its architecture. It is a single board system which is plugged in SUN workstation. To use it, the user must first specify a vocabulary and give training *utterances* of each word in vocabulary. These training utterances are used to create templates. During recognition, all trained templates are first downloaded to the template RAM in *MARA*. Then, the system monitors whether an utterance is spoken. Once an utterance is detected, its feature sequence is sent to the *DTW* processor for distance computation until the end point is detected.

The *DTW* matching for speech works as follows. The recognition feature used in *MARA* is *frame* sequence. A frame consists of digitized short time spectrum of 16 bands. Let U_i denote the *ith* frame of the unknown utterance and T_j as the *jth* frame of a template,

$T_j = (T_{j,0}, T_{j,1}, \ldots T_{j,15})$.

The local distance between U_i and T_j is defined as:

$$d_{i,j} = \sum_{k=0}^{k=15} (U_{i,k} - T_{j,k})^2 \ .$$

The accumulated distance is computed in the same way as in *OHR*, i.e.

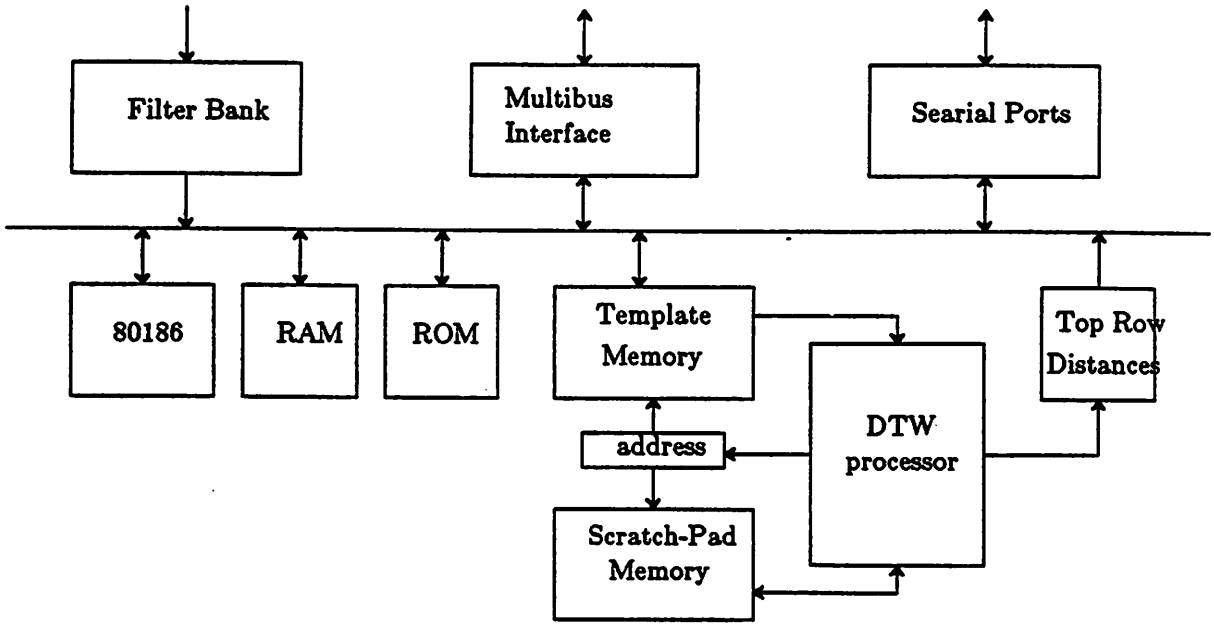


Figure 8.1 MARA architecture

$$D_{i,j} = d_{i,j} + MIN(D_{i-1,j}, D_{i-1,j-1}, D_{i,j-1}).$$

Once a frame of the unknown utterance is sent to it, the whole column of accumulated distances is updated as depicted in Fig. 5.2. This computation is carried column by column until the end point of the utterance is detected. Then, the template with the minimum accumulated distance is recognized as the unknown. The appropriate keystrokes are then sent to the application program.

In the *DTW* matching, although there are tremendous number of computations, it is found that the computations are very repetitive. For this type of computation, an ASIC (application-specific integrated circuit) often obtains several orders of magnitude better performance over a general purpose microprocessor.

The *DTW* processor is designed to take advantage of the parallelism and regularity in both the local distance and the accumulated distance computation. Its architecture is as illustrated in Fig. 8.2. There are 4 adders to handle the local distance computation, 3 comparators to handle the accumulated distance selection, another adder to handle distance accumulation, a subsystem to keep track of the effective path length for distance normalization, and a subsystem to handle

the addresses for data I/O. For each data fetch, a template frame together with its accumulated distance and effective path length are all brought in for parallel processing. Because of this architecture, a new accumulated distance and its effective path length can be updated in 0.8μs when operating with 5 MHz clock.

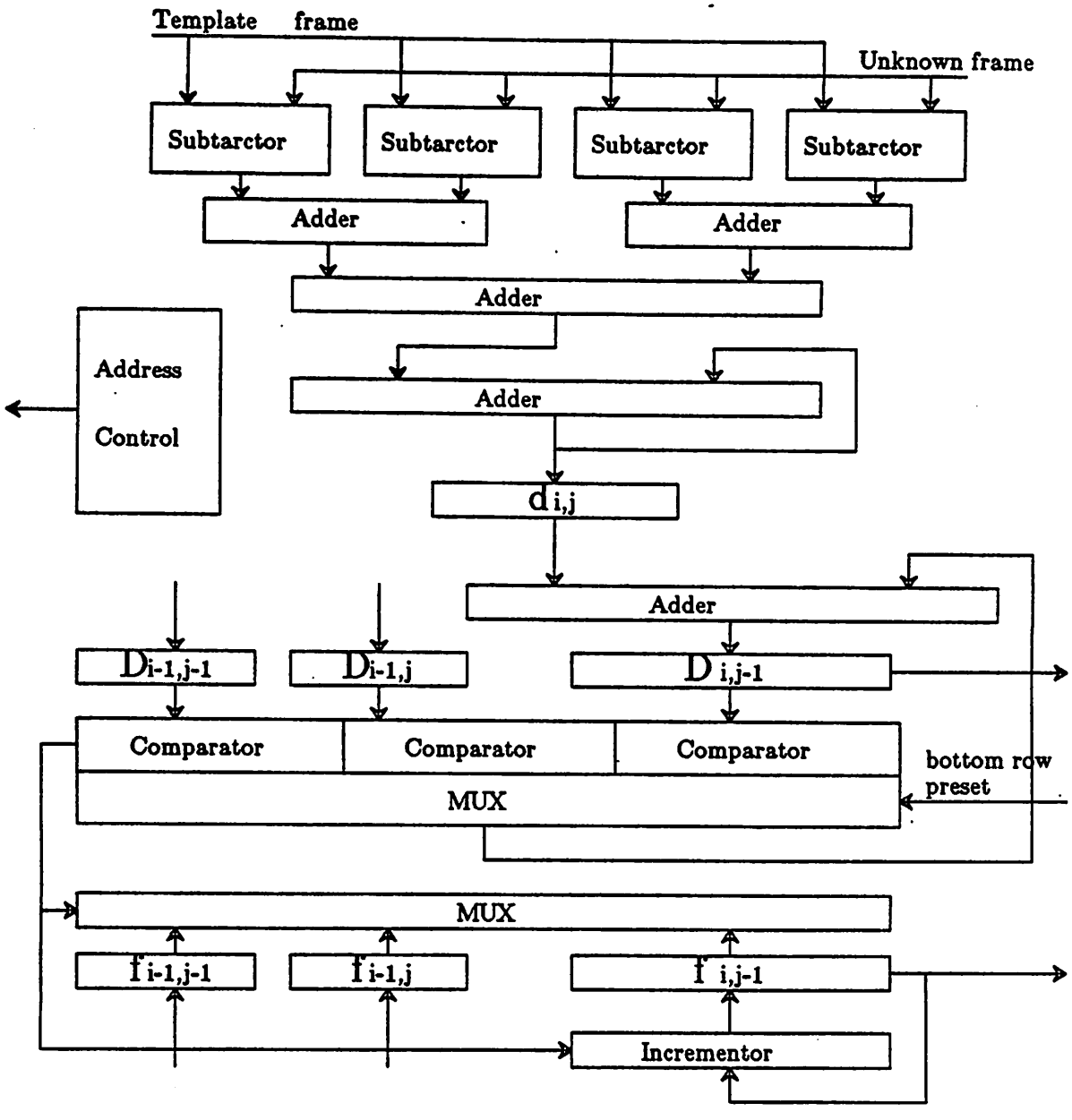


Figure 8.2 DTW processor architecture

It has been demonstrated that *MARA* can handle a 1000 word vocabulary in real-time. Implementation of the same algorithm on a VAX750 can only handle 20 words in real-time.

In each column computation, the bottom row distance (D_0^k) of each template is set by the microprocessor. This feature makes the *DTW* processor be able to handle both the single-pass and two-pass *DTW* algorithms for connected speech recognition.

2. Using MARA for OHR

2.1. Discrete symbols

After seeing the similarity between the speech recognition algorithm and handwriting recognition algorithm, it is apparent that they can share some hardware, especially the *DTW* processor. To use the *DTW* processor for handwriting recognition, the first problem encountered is how to use it to compute the local distances.

If the recognition feature is normalized x or y coordinate, the x or y coordinate can be treated as data of the first band. By setting the data of other bands to 0, the local distance obtained from the *DTW* processor:

$$\sum_{k=0}^{k=15}(U_{k,i} - T_{k,j})^2 = (U_{0,i}-T_{0,j})^2 = (U_{x,i}-T_{x,j})^2 \ ,$$

is just the local distance needed.

If the recognition feature is slope code, the implementation is not so straight forward. This is because the distance of two slope codes is measured by Lee Metric which is not the same as the Euclidean distance computed by the processor.

However, the Lee Metric can be made compatible with the Euclidean distance by converting the slope code into a spectral format. Fig. 8.3 shows the conversion table. Sending the spectral format of two slope codes to the *DTW* processor, the local distance obtained from it is exactly the Lee Metric. For example, the spectral format of slope code 2 is

(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) ,

and the spectral format of the slope code 14 is

(0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0) .

The local distance computed from *DTW* processor is

$$\sum_{k=0}^{k=15}(U_{i,k} - T_{j,k})^2 = 4 \ ,$$

which is exactly the Lee Metric distance between slope code 2 and 14.

		SPECTRUM 0 1 2 3 4 5 6 7 8 9 A B C D E F
	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	2	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	3	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
	4	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
	5	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
S	6	1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
L	7	1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
O	8	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
P	9	0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
E	A	0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
	B	0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0
	C	0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
	D	0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0
	E	0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
	F	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

Figure 8.3 Convert slope code to spectral format

This code conversion makes the *DTW* processor available for *OHR* without any circuitry change. The sampling period of the tablet is 10ms. The *DTW* processor computes one accumulated distance in 0.8μs. Therefore, 12500 sample points can be handled within sampling period. A typical template has around 15 sample points. As such, 800 templates can be handled in real-time by one *DTW* processor. This number should be adequate for many applications.

After solving the local distance computation problem, the *OHR* was implemented on *MARA* is as follows. The tablet is connected to a serial port of *MARA*. An interrupt service

routine is installed to handle the data acquisition. The microprocessor extracts the recognition features and converts them to the right format.

If a multi-stroke symbol is treated as a single stroke symbol as shown in Fig. 8.4, if the disambiguation algorithm need not be performed, *MARA* can be used for *OHR* without any change. In this case, a spectral code of a slope is treated as a frame. A symbol is treated as an utterance. The symbol templates can be stored and matched in the exactly same way as the utterance templates in speech recognition.

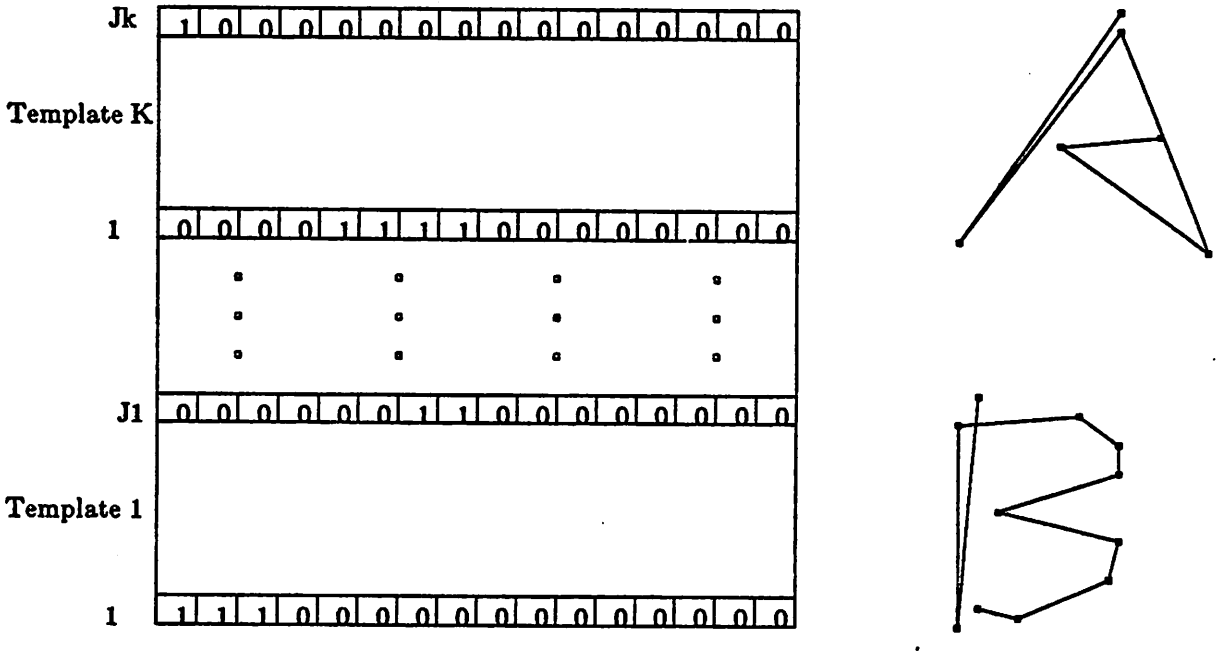


Figure 8.4 Use MARA for OHR with no disambiguation

However, if the disambiguation must be performed, the *MARA* has to be slightly modified to use the *DTW* processor for disambiguation. The template memory should be divided into several blocks as shown in Fig. 8.5. For different distance functions, the microprocessor tells the *DTW* processor where to get the template data. For example, at the template matching stage, the microprocessor tells the *DTW* processor to get templates from the *1st stroke* block of *slope* feature. When the second stroke is detected started, the microprocessor tells the *DTW* processor to take templates from the *2nd stroke* block of *slope* feature. At the disambiguation stage, according to the disambiguation rules, the microprocessor sends the *DTW* processor the proper

feature sequence of the unknown symbol and also tells it which block in the template memory should be matched with.

Y	Stroke 3	Template 2	
	Stroke 2	Template 2	
		Template 1	
	Stroke 1	Template 3	
		Template 2	
		Template 1	
X	Stroke 3	Template 2	
	Stroke 2	Template 2	
		Template 1	
	Stroke 1	Template 3	
		Template 2	
		Template 1	
SLOPE	Stroke 3	Template 2	
	Stroke 2	Template 2	
		Template 1	
	Stroke 1	Template 3	
		Template 2	
		Template 1	

Figure 8.5 Organization of template memory

2.2. Cursive script

For cursive script recognition, because there is no disambiguation, the templates are arranged as in Fig. 8.4. Under this arrangement, the template matching stage is essentially identical to that in the connected speech recognition part of *MARA*.

After template matching, the microprocessor computes the final distance of each word in vocabulary and does the profile check. For the profile check, the y extrema of each path are required. To facilitate this computation, circuitry can be added to the *DTW* processor such that

the y extrema of each path are obtained in a similar way as the effective path length. With this implementation, the y extrema are available to the microprocessor right after template matching. This saves the microprocessor a substantial amount of time.

3. Conclusion

Although the *OHR* has not been put on the latest version of *MARA*, the same integration idea has been tried out on *MARILYN*, which is the preceding version of *MARA* with similar architecture.[2] It works very well and impressively fast. It is very exciting to see both speech and handwriting can be handled by one hardware accelerator. This implies that the future PC can have two new user interface dimensions by adding one recognition system.

References

1. R. Kavaler, "The Design and Evaluation of a Speech Recognition System for Engineering Workstations," *Ph.D Thesis*, University of California, Berkeley, 1986.

2. D. Mintz, *An Implementation of A Speech Recognition System*, University of California, Berkeley, 1983.

CHAPTER 9

SUMMARY AND CONCLUSIONS

1. Algorithm

In this project, two new algorithms have been successfully developed for a trainable on-line handwriting recognition system. The first one is for discrete symbols. The second one is for cursive scripts.

1.1. Discrete Symbol Recognition

Fig. 9.1 shows the block diagram of the algorithm for discrete symbol recognition. The recognition is performed by matching the x, y, and slope sequences of an unknown symbol against trained symbol templates. There are always inevitable variations in our writings. This problem was solved by using the dynamic time warping (*DTW*) technique. Experimental result showed that this technique can absorb variations very well.

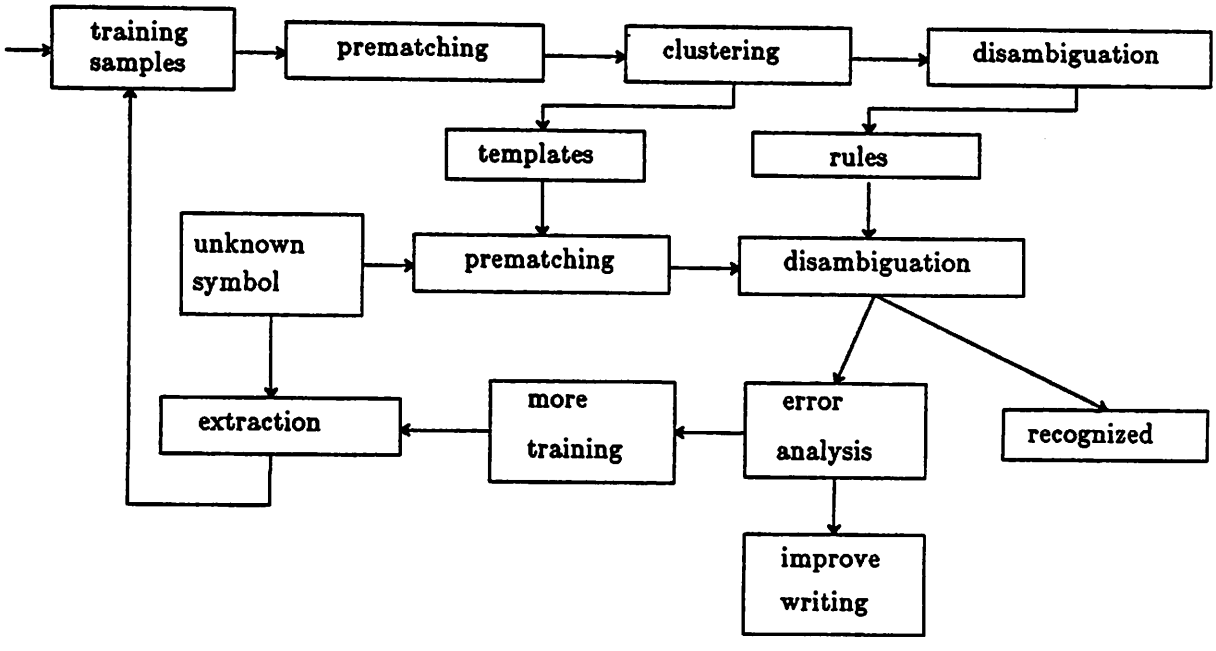


Figure 9.1 Discrete Symbol Recognition

Without good templates, it is impossible to achieve high recognition rate. The only way to obtain good templates is to ask the user for adequate training samples. In order not to make the

training tedious, the training samples are collected in two phases. During the initial training phase, the user can provide a few training samples which he feels cover all his writing variations. During the recognition phase, if a recognition error happens, the user can extract the misrecognized writing out and use it as a training sample of its symbol. The training samples are consolidated by a clustering algorithm to create templates.

Although the *DTW* algorithm is very good at absorbing writing variations, it has severe limitations in differentiating ambiguous symbols. To solve the problem, the recognition is divided into two stages. In the prematching stage, the slope matching is performed. If the prematching can not determine a definite winner, the disambiguation is performed on all of the top candidates.

The disambiguation algorithm is based on the assumption that any two ambiguous templates can be effectively differentiated by matching either x or y or slope sequence of one of the strokes. To determine the disambiguation rule, i.e. which distance function can differentiate best, the cluster members of both templates are used to test the performances of all distance functions. The one with the best performance is selected. In the recognition phase, the disambiguation functions of all the prematching top candidates are asked to cast their votes. The template which gets the most votes is the final winner.

With appropriate adaptive training, this algorithm can virtually recognize any discrete symbol with near 100% accuracy.

1.2. Cursive Script Recognition

Fig. 9.2 shows the block diagram of the cursive script recognition algorithm. The recognition is accomplished through three stages. The first stage is matching. By slightly modifying the *DTW* algorithm for discrete symbols, all possible letter segments in the unknown script and their distances can be found.

In the second stage letters are connected. For each word in the specified vocabulary, its likelihood is determined by a heuristic search and concatenation algorithm. It has been shown

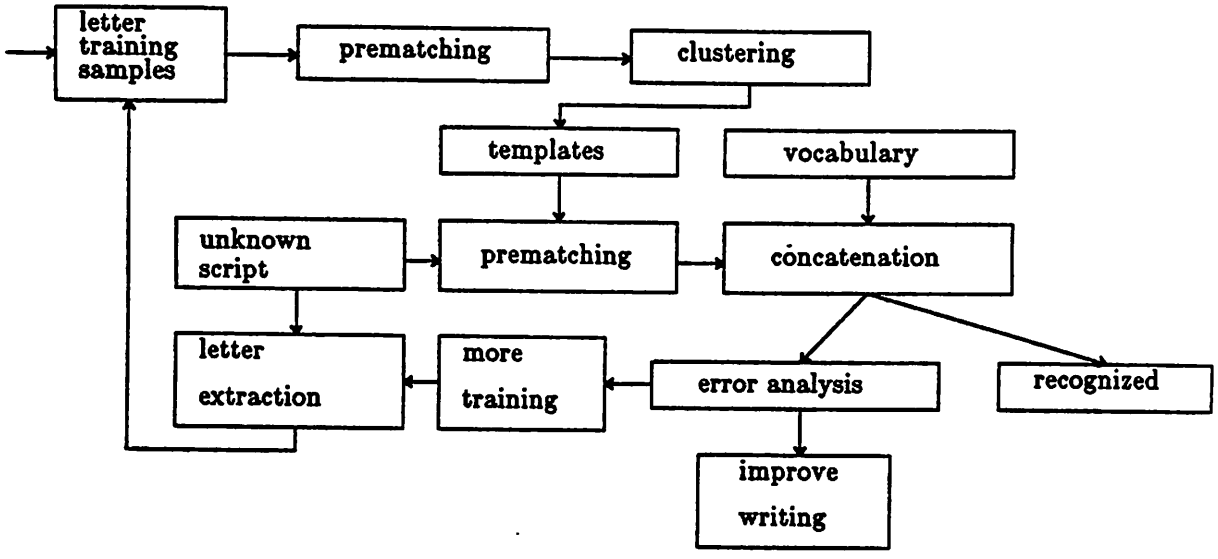


Figure 9.2 Cursive Script Recognition

that with appropriate concatenation relaxation, this algorithm works as well as the time-consuming dynamic programming concatenation.

The third stage is disambiguation. Two features are used for this purpose. One is the delayed strokes, i.e. the dot in i, j, the bar in t, and the slash in x. The other is the relative letter profile. If a word does not have the same delayed strokes and profile as those of the unknown script, it will be rejected. The relative letter profile is used because there are some letters which can not be accurately differentiated by the slope matching. Since the profile check requires the user to control the relative letter height, it is only performed on words specified by the user.

To ensure the best templates can be adaptively obtained, the letter training samples are also collected in two phases. First, the user writes each letter discretely and creates templates in the same way as in discrete symbol recognition. Then, if a script is not correctly recognized during recognition, the letter extraction algorithm can decompose the just written script and use the extracted segments as new training samples of their letters. With these new training samples, the clustering algorithm can generate letter templates which are more suitable for cursive script recognition.

If the adaptive training is correctly performed and the profile checks are appropriately set, experimental results have shown that this algorithm can recognize scripts of a specified

vocabulary with almost 100% accuracy.

2. Implementation

Both the discrete symbol recognition algorithm and the cursive script recognition algorithm have been implemented on an IBM PC for further application exploration. Fig. 9.3 shows the structure of the implementation. There are three levels in the system. The first level is a tablet server. It resides in memory to provide tablet related services to application programs. The application programs can get its services through standard PC system call. The second level is a keyboard emulator and a mouse emulator. These emulators pass the recognized symbol keystrokes and stylus coordinates to application programs as if these data were from the keyboard and mouse. These emulators are needed because the existing software can only take input from keyboard and mouse. The third level is utility programs. These programs are used to facilitate the adaptive training.

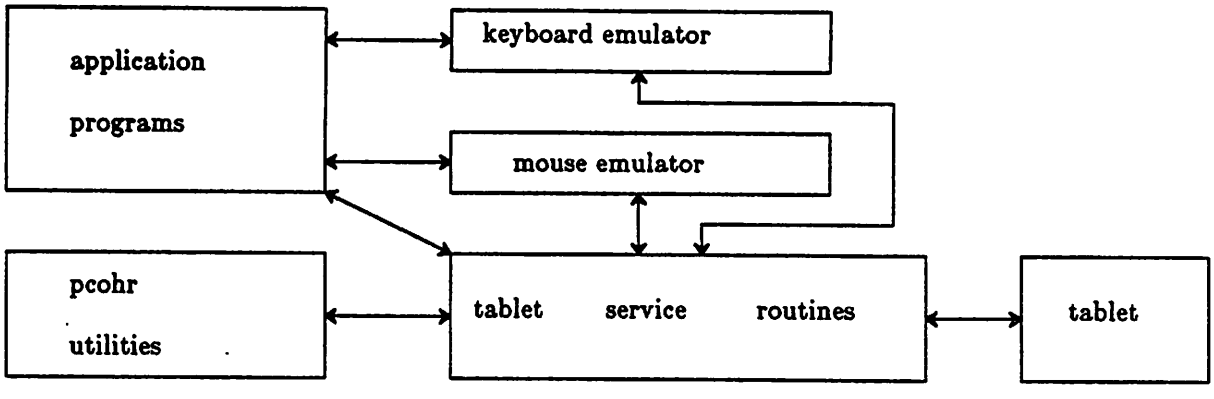


Figure 9.3 PCOHR Structure

Most of the system software is written in C with a small portion in ASSEMBLY. It runs under MS-DOS. With a 4.77MHz 8088, the recognition speed for discrete symbol is about 100ms per template. The recognition speed for cursive script recognition is about 300ms per template. The space needed for each template is about 200 bytes.

The *DTW* template matching, which used extensively in the system, is computationally intensive. Part of the *OHR* system has been implemented on a dedicated *DTW* processor based speech recognition system. It turns out the dedicated processor significantly improves the

execution of the *DTW* template matching and 800 templates can be handled in real time. It has been demonstrated that the algorithms for speech recognition and handwriting recognition are so similar that the future PC can in fact use one processor for both types of recognition.

3. Application

The developed system has been used for some simple application experiments. From the limited experience, it was found that the *OHR* system has two essential impacts. One, it significantly improves the entry of symbols which are now entered by typing a string of hard-to-remember keystrokes. This improvement is extremely important for computerizing languages for which keyboard is not an acceptable data entry device. Two, it significantly improves the entry of freehand drawing. This enhancement has the potential to bring computer from text oriented to graphics oriented, which is more natural to human being.

However, since the user can not look at the screen while he is writing, the current PC setup can not make much use of the *OHR* system. The full strength of the *OHR* can only be revealed when it is integrated with a notebook PC as shown in chapter 1. Hopefully the dream PC will be available soon and all the PC users can write and sketch to their computers.

References

*** APPENDIX: PCOHR User's Guide ***

(1) Introduction:

PCOHR is an on-line handwriting recognition system for IBM PC. It enables a user to interface with IBM PC software by writing and drawing on a tablet.

(2) Equipments:

. IBM PC with serial port COM1.
. Seiko digitizing tablet.
. IBM PC Color Graphics Adapter.
. PCOHR.

(3) Set up:

. Connect Seiko tablet to COM1.
  Make sure only the 4,7,8 switches on the tablet are ON.
. Make sure all PCOHR commands are in your command path.
. Use "hinstal" to install the tablet server. Without the server, programs can not talk with the tablet.

(4) Discrete Symbol Recognition (DSR):

- Suppose we want to train a symbol set "DEMO" which contains D, P, 0, 6, \(*a, \(*g, NAND, NOR, and three Chinese characters "from", "field", "good". A directory has to be created for this symbol set. In this directory, create a "symbol.def" file as illustrated in Fig. 1. In Fig. 1, the first column specifies the root file name of each symbol. The second column specifies the keystrokes of each symbol.

```
D     D
P     P
0     0
6     6
ALPHA     \(*a
GAMMA     \(*g
NAND      NAND
NOR NOR
FROM      from
FIELD     field
GOOD      good
```

Fig. 1 Symbol.def

- After "symbol.def" is specified, use "htrain" to give training samples of the symbols. For each symbol, the program first asks for how many training samples you are going to give. Then, it prompts you to write each of them. At this moment, try to decide how many writing variations you have for this symbol and how many training samples you would like to give for each variation. We know this is difficult because writing a symbol is almost an unconscious behavior. But don't worry. The PCOHR has very good adaptive training capability. All we need now is just some training to get the system started.

- The "htrain" displays the writing trace of each training sample. Make sure it looks OK before saving it.

  Fig. 2 shows the traces of some training samples.

Press RETURN to continue

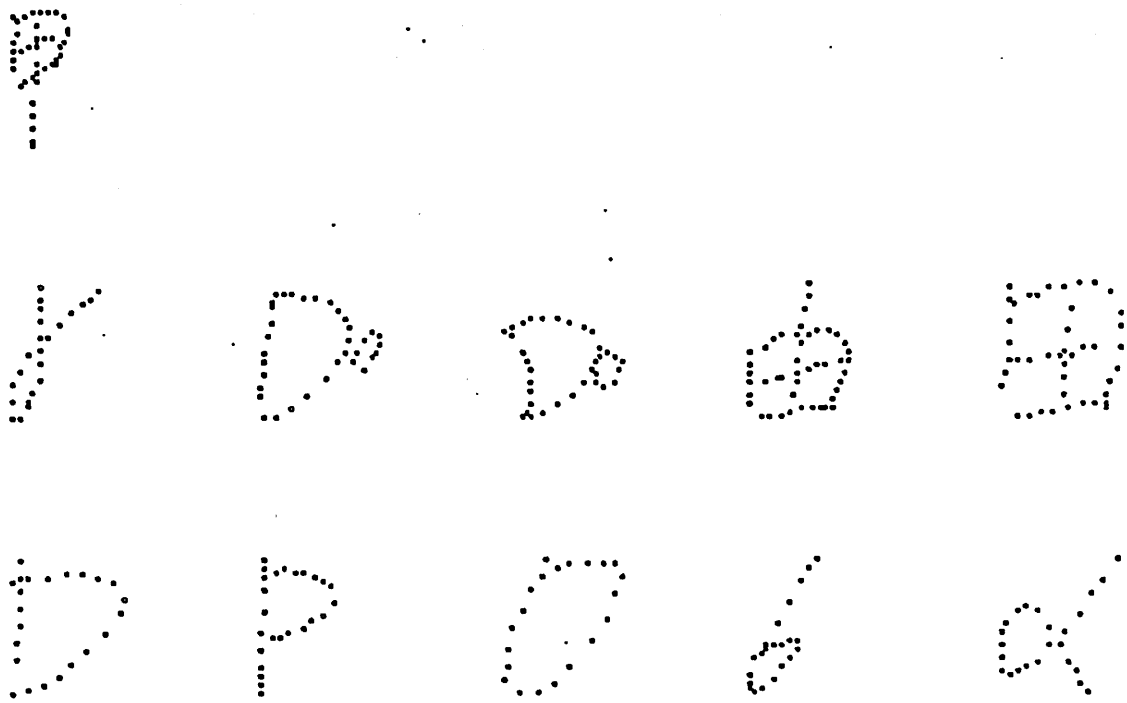


Fig. 2 Writing traces of symbols

- After giving all the training samples, use "hclus" to group training samples into templates. If the -plot option is specified, i.e. "hclus -plot", "hclus" displays the templates and their cluster members. Fig. 3 shows an example.

  The clustering results are saved in the file "symbol.cls" as illustrated in Fig. 4.

- After templates are created, use "hdisam" to find out the disambiguation rules for ambiguous templates. If the -print option is specified, i.e. "hdisam -print", "hdisam" prints out the intermediate results.

  The disambiguation results are saved in the file "symbol.dis". Fig. 5 illustrates the file for our symbols.

Press RETURN to continue

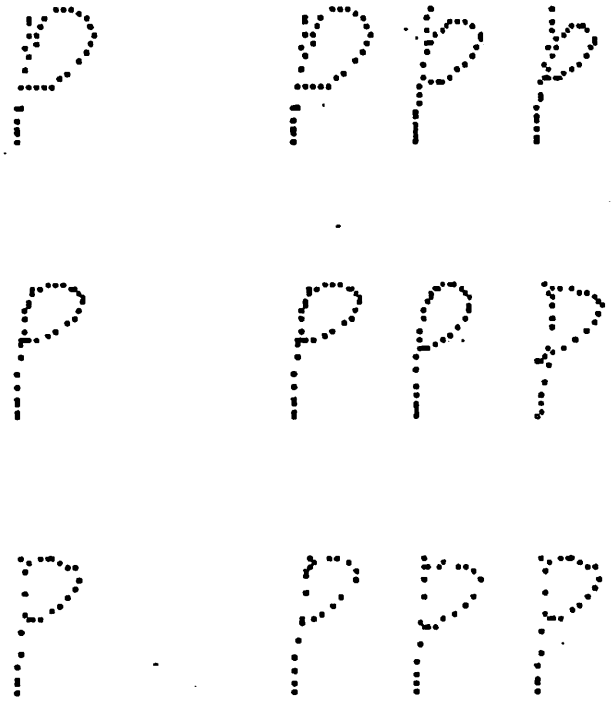


Fig. 3 Clustering

```
SYMBOL: 0
TEMPLATE: 0.T1
MEMBER: 0.S1 0.S2 0.S3

SYMBOL: 6
TEMPLATE: 6.T1
MEMBER: 6.S1 6.S2 6.S3

SYMBOL: ALPHA
TEMPLATE: ALPHA.T1
MEMBER: ALPHA.S1 ALPHA.S2 ALPHA.S3

SYMBOL: D
TEMPLATE: D.T1
MEMBER: D.S1 D.S2 D.S3

SYMBOL: FIELD
TEMPLATE: FIELD.T1
MEMBER: FIELD.S1 FIELD.S2 FIELD.S3

SYMBOL: FROM
TEMPLATE: FROM.T1
MEMBER: FROM.S1 FROM.S2 FROM.S3

SYMBOL: GAMMA
TEMPLATE: GAMMA.T1
MEMBER: GAMMA.S1 GAMMA.S2 GAMMA.S3

SYMBOL: GOOD
TEMPLATE: GOOD.T1
MEMBER: GOOD.S1 GOOD.S2 GOOD.S3

SYMBOL: NAND
TEMPLATE: NAND.T1
MEMBER: NAND.S1 NAND.S2 NAND.S3

SYMBOL: NOR
TEMPLATE: NOR.T1
MEMBER: NOR.S1 NOR.S2 NOR.S3

SYMBOL: P
TEMPLATE: P.T1
MEMBER: P.S1 P.S2 P.S3
TEMPLATE: P.T2
MEMBER: P.S7 P.S8 P.S9
TEMPLATE: P.T3
MEMBER: P.S4 P.S5 P.S6
```

Fig. 4 symbol.cls

```
TEMPLATE: 0.T1

TEMPLATE: 6.T1
0.T1 1y

TEMPLATE: ALPHA.T1

TEMPLATE: D.T1

TEMPLATE: FIELD.T1

TEMPLATE: FROM.T1
FIELD.T1 3y

TEMPLATE: GAMMA.T1

TEMPLATE: GOOD.T1
FIELD.T1 3y FROM.T1 3y

TEMPLATE: NAND.T1

TEMPLATE: NOR.T1
NAND.T1 1x

TEMPLATE: P.T1
D.T1 2y

TEMPLATE: P.T2

TEMPLATE: P.T3
```

Fig. 5 symbol.dis

- After the templates are disambiguated, they are ready for recognition. Use "hload -dir DEMO" to download all the templates in directory DEMO.

- After the templates are downloaded, you can start your application program and write to it !

- If the system makes the same recognition error very often, it is time to give the system more training. To do adaptive training, you first type command "hanal -plot" to see the traces of the just written symbol and suspicious templates.

  For example, the first '6' in Fig. 6 is misrecognized as '0'. By typing "hanal 0 6 -plot", you can see the misrecognized '6' and the templates of '0' and '6'.

- If "hanal" shows your writing is quite different from its template and you think this writing is very common to you, type command "htrain -adapt". "Htrain" will ask what symbol the writing is, then extracts it out as a new training sample of that symbol.

- If your writing looks should be recognized, you can type command "hanal 0 6 -print" to see the matching details. Fig. 7 shows the analysis of the Fig. 6 recognition error.

  The matching details usually give a very good revelation of what is the best disambiguation rule. In Fig. 7, we can see the slope distance function fails to make the right recognition but the y distance function can distinguish '0' and '6' very well.

  If the disambiguation rule of the two symbols is not the same as what you think it should be, use command "htrain -adapt" to extract the writing out as a new training sample of its symbol.

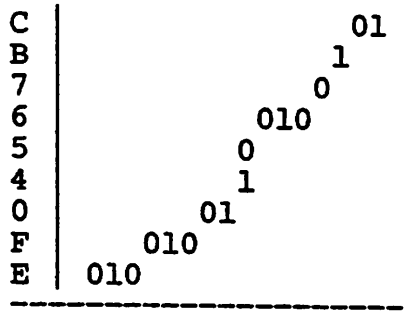
- After obtained several new training samples, type command "hclus" and "hdisam" to update the templates and disambiguation rules.

- If the training procedure is well performed, PCOHR can achieve near 100% recognition rate without too much tedious training overhead.

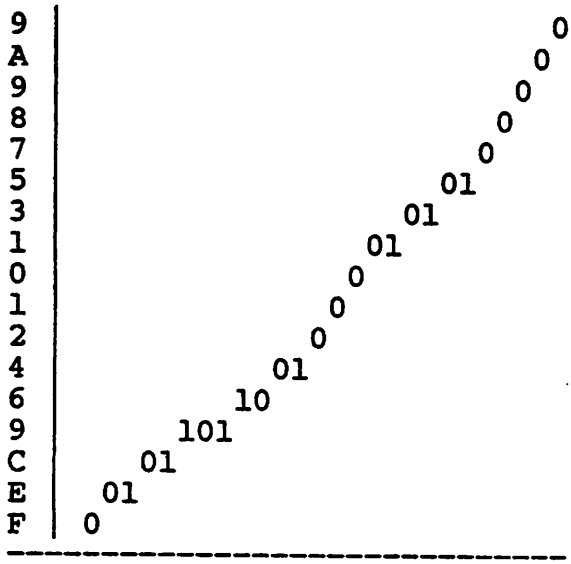
Press RETURN to continue

Fig. 6 A misrecognition

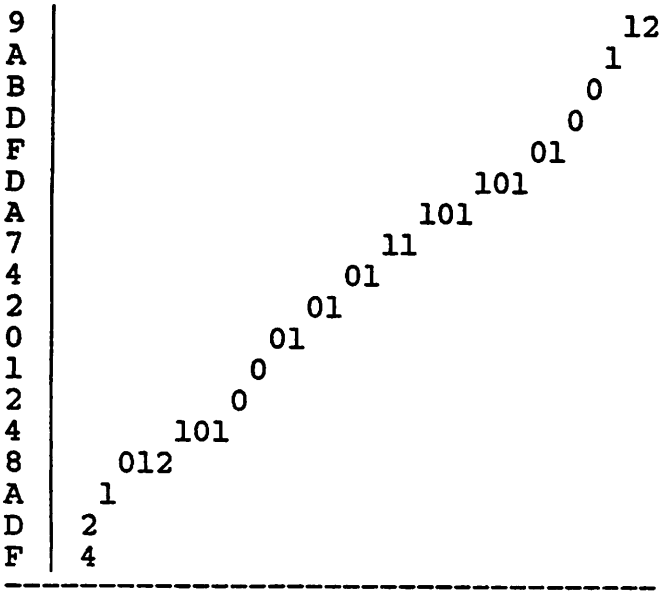
6: Slope sequence of stroke 1: Before norm: 7 After norm: 6



6: Y sequence of stroke 1: Before norm: 9 After norm: 5



6: X sequence of stroke 1: Before norm: 26 After norm: 13



```
0:  Slope sequence of stroke 1:  Before norm:  6   After norm:  5

C |                 01
B |               1
7 |             0
6 |          010
4 |        1
F |       0
0 |     0
F |  010
E | 010
--------------------
    EFEF0F0F56767ACD

0:  Y sequence of stroke 1:  Before norm:  21   After norm:  12

E |                          5
C |                        2
A |                      1
7 |                    101
4 |                  01
2 |                01
0 |              01
1 |            0
3 |          01
5 |        11
8 |     101
B |   01
D |  01
F | 01
------------------------------
    FEDCBA98764321012345689A9

0:  X sequence of stroke 1:  Before norm:  20   After norm:  10

8 |                              01
A |                            1
C |                          11
F |                        01
D |                      01
B |                    101
8 |                  01
5 |                101
2 |              01
0 |            01
1 |          0
3 |        01
5 |      01
6 |    0
8 | 3101
----------------------------------
    B98765432101234568ABCDEFEDB987
```

Fig. 7 Matching details from hanal

(5) DSR Limitations:

- This system is very vulnerable to slope inconsistencies. Unconscious "hooks" at the start and end of a stroke and "circles" in the middle of a stroke, no matter how small they are, hurt the recognition accuracy very much. The user should be careful not to create these small "hooks" and "circles".

- The system is not sensitive to symbol size. A user can write a symbol in any size. This feature, however, makes the system not be able to distinguish symbols whose only difference is size, e.g. 'C' and 'c'.

- For symbol recognition, it takes approximately 0.1 second per template. For "hdisam", it takes 0.6*(n+m) second to process two clusters with n and m members.

- The symbol can not have more than 15 strokes.

- In this release, only printable keystrokes can be assigned to a symbol.

(6) DSR Commands:

. htrain [syb] [-adapt]
[ks -alias]

no option: If there are symbols which do not have any training sample, htrain will ask for it.
syb: Htrain will ask for more training samples for the symbol "syb".
-adapt: Htrain will extract the just written symbol out and store it as a new training sample of the symbol "syb".
ks -alias: Htrain will ask for only one training sample of a temporary symbol and use "ks" as its keystrokes. This training sample is then downloaded to the server as a new template.

(The training samples are saved as files with file name "xxx.Sn". "xxx" is the root file name specified in "symbol.def". "n" is a number automatically assigned.)

. hclus [-print] [-plot]

Hclus performs clustering on training samples of each symbol in "symbol.def". The results are stored in "symbol.cls". The templates are saved as files with file name "xxx.Tn". "xxx" is the root file name specified in "symbol.def". "n" is a number automatically assigned.

-print: The intermediate clustering results are printed out.
-plot: The template and members of each cluster are plotted out.

. hdisam [tpl1 tpl2] [-print]

no option: Hdisam performs disambiguation for all templates in "symbol.cls". The results are saved in "symbol.dis".
tpl1 tpl2: Hdisam performs disambiguation only for tpl1 and tpl2. The disambiguation result is printed out but not saved in "symbol.dis".
-print: The intermediate disambiguation results are printed out.

. hload -dir FOO

The templates in "FOO\symbol.dis" are downloaded to the tablet server. If "symbol.dis" does not exist, templates in "symbol.cls" are downloaded. In this case, no disambiguation will be performed during recognition. The "FOO\macro.def" is also downloaded for touch functions.

. hanal [sybl] [syb2] ... [-plot] [-print]

-print: Hanal prints the detail matching paths and distances between the just written symbol and the specified symbol templates. The output can be redirected to a file for reviewing.
-plot: Hanal plots the traces of the just written symbol and the specified symbol templates.

. hdsp [fl] [f2] ... [-plot] [-print]

The fl, f2 ... are file names of either training samples or templates.

-plot: Hdsp plots the traces of the specified files.
-print: Hdsp prints the recognition features of the specified files.

(7) Cursive Script Recognition (CSR):

. Because of the IBM PC system limitation, the cursive script recognition routine is not included in the tablet server as the discrete symbol recognition routine. Instead, it is packed together with the training and analysis routines in the program "cscript". To use "cscript", the first step is to create a directory which contains templates of discrete letters. You can create these templates by using "htrain" and "hclus" as described in DSR.

. After these templates are created, you have to specify the words you are going to use in a vocabulary file. Fig. 8 illustrates an example. The "*" after some words is profile check mark which will be explained later.

```
date
time
tree
exit
ver
dir
cls
adam
bob
feel *
bell *
fell *
```

Fig. 8 Vocabulary file

. Suppose the letter templates are in a directory LTR and the vocabulary file is TEST. Type command "cscript -dir LTR -voc TEST" to get into the cursive script recognition program.

. There are four commands in cscript: vocab, recog, analy, and train. Vocab simply prints out what words are in the vocabulary and their stroke string and profile string. Stroke string contains the types of delayed strokes in the descending order of the x coordinate of the middle point. For example, the stroke string of "exit" is (/ . -), meaning Slash, Dot, Bar. The profile string of "exit" is (- - d), meaning the profile transition from e to x is '-', x to i is '-' and i to t is 'd'. Fig. 9 lists the definition of the type letters.

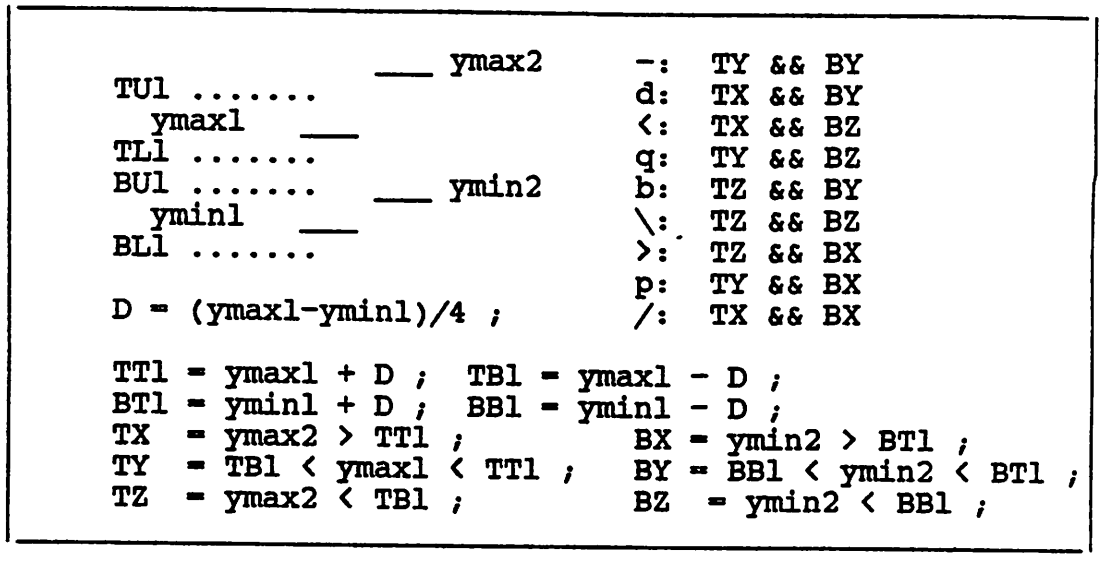


Fig. 9 Letter profile transition

. To test the recognition, type command "recog". This command prompts you to write the test script. When you start writing, the proceeding dots on screen indicate recognition is in progress. After you finish writing, the recognized word will be printed out after a while.

- If the script is misrecognized, you can type the command "analy wrd" to see the intermediate results obtained from comparing the script with the word "wrd". After "analy wrd", the command "analy -break" plots how the script is broken down when compared with "wrd". The command "analy -toprow ltr" shows the matching details between the script and templates of the letter "ltr".

  Fig. 10 shows the "body" of the script "peter", its letters extracted by the recognition routine, and the letter templates. We can see letters extracted from a script are quite different from the templates which were created discretely. Fig. 11 shows the top row records of 'e'. There are two dips at (47) and (88). The dip at (47) is from 31 with distance 16. The dip at (88) is from 69 with distance 17. They correspond to the two 'e' in "peter".

- The recognition error is always due to two causes. One, the matching can not distinguish some letters well. Two, the letter in script is very different from the templates. If "analy" shows the error is due to the first cause, you should add the profile check marks after the ambiguous words, e.g. "feel", "fell", and "bell" in the TEST vocabulary. If "analy" shows the error is due to the second cause, you can use the command "train wrd" to break the misrecognized script down to letter segments. "Train wrd" will store these segments as the new training samples of their letters.

- Since it is hard to create the script ligatures when the letter is written discretely, it is recommended to extract the letter training samples from scripts.

- After obtaining several extracted training samples, use "hclus" to create new templates. The "hdisam" is not needed since the disambiguation rules are not used for script recognition.

- If the adaptive training is well done, the recognition rate can be close to 100%.

Press RETURN to continue

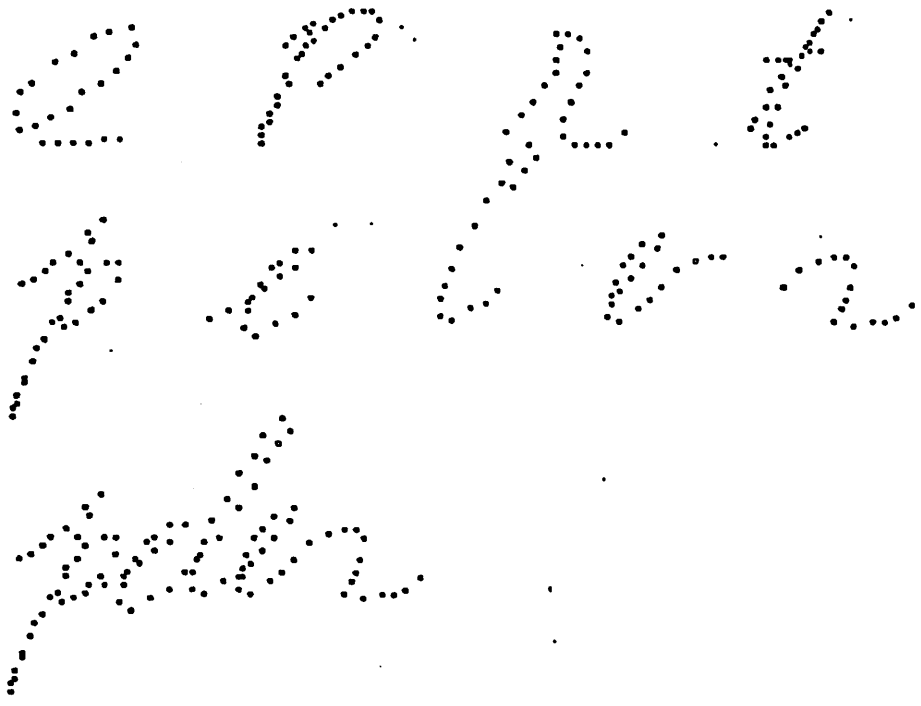


Fig. 10 Template matching of a script "body"

(0)	0	132	(1)	0	56	(3)	0	56	(4)	0	56
(5)	0	56	(6)	0	57	(7)	0	26	(8)	0	28
(9)	0	24	(10)	0	24	(11)	0	25	(12)	0	23
(13)	0	21	(14)	0	20	(15)	0	19	(16)	0	16
(17)	0	15	(18)	0	16	(19)	0	17	(20)	0	18
(21)	0	18	(22)	0	19	(23)	0	20	(24)	0	19
(25)	0	19	(26)	23	42	(27)	23	44	(28)	23	46
(29)	23	48	(30)	23	50	(31)	23	37	(32)	23	37
(33)	23	38	(34)	23	40	(35)	23	38	(36)	23	36
(37)	23	36	(38)	31	49	(39)	31	41	(40)	31	41
(41)	31	34	(42)	31	34	(43)	31	27	(44)	31	25
(45)	31	17	(46)	31	14	(47)	31	14	(48)	31	14
(49)	31	15	(50)	31	15	(51)	31	16	(52)	31	16
(53)	31	17	(54)	31	18	(55)	31	18	(56)	31	20
(57)	52	45	(58)	52	38	(59)	52	38	(60)	52	38
(61)	52	38	(62)	52	33	(63)	52	33	(64)	52	30
(65)	52	25	(66)	52	17	(67)	52	15	(68)	52	15
(69)	52	16	(70)	52	16	(71)	52	16	(72)	52	17
(73)	52	18	(74)	52	22	(75)	69	38	(76)	69	40
(77)	69	38	(78)	69	33	(79)	69	29	(80)	69	30
(81)	69	20	(82)	69	19	(83)	69	19	(84)	69	19
(85)	69	19	(86)	69	19	(87)	69	18	(88)	69	17
(89)	69	19	(90)	69	21	(91)	85	46	(92)	85	46
(93)	85	46	(94)	85	36	(95)	85	34	(96)	85	34
(97)	85	32	(98)	85	30						

Fig. 11 Finding letters in a script

(8) CSR Limitations:

- For script recognition, it takes approximately 0.3 sec per vocabulary word. Please be patient to the recognition response.

- For a script, the user has to complete the "body" of a script first and then the extra strokes, i.e. the dot of 'i', 'j', the bar of 't' and the slash of 'x'.

- The recognition algorithm has difficulty to distinguish letters with similar slope sequences, e.g. 'a' and 'd', 'b' and 'f', 'l' and 'e'. These errors can only be effectively fixed by profile check. However, the profile check requires the user to have careful control over the relative letter height in a script.

(9) CSR Commands:

. vocab: It prints out all the words in current vocabulary, together with their stroke strings and profile strings.

. recog: It asks for a script and prints out the recognition result.

. analy wd
  [-break]
  [-toprow ltr]:

  wd: It compares the just written script with the word "wd". The intermediate recognition results are printed out.
  -break: It plots the decomposition results of the last "analy wd".
  -toprow ltr: It prints the top row matching results of the letter "ltr" of the last "analy wd".

. train wd [-adapt]

  wd: It asks for a script of wd and extracts letter segments from it. The extracted segments are stored as new letter training samples.
  -adapt: It extracts letter training samples from the just written script.

(10) Miscellaneous features:

- Fig. 12 shows a simple tablet layout for PCOHR. The tablet is divided into two areas: keyboard/mouse area and function area. In the function area, three boxes are reserved for keyboard/mouse switching, left button setting, and right button setting. The rest boxes are used for "macros".

- If the stylus is in keyboard mode, the coordinates from tablet are processed by the recognition routine in the server. The keystrokes of the recognized symbol are sent to the client program as if they were typed in from keyboard.

- If the stylus is in mouse mode, the coordinates from tablet are sent to the client program in a format as the signals from mouse. When the stylus is moved/clicked/dragged, the client will get signals as if the mouse were moved/clicked/dragged.

  There is only one switch in the stylus. To use it as a two-button PC mouse, you have to set the stylus to be the left button or the right button by touching the corresponding function box.

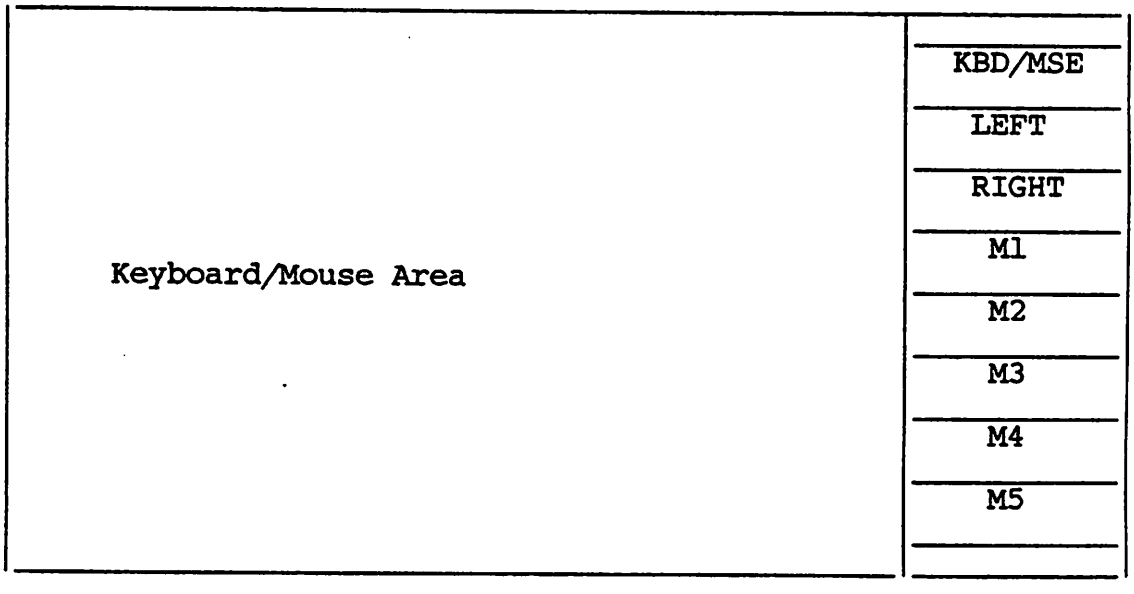


Fig. 12 Tablet layout

. The user can specify the keystrokes of each function box in the file "macro.def" and download them to the tablet server by "hload". Once a function box is touched, the corresponding keystrokes are sent to the client program.

Fig. 13 shows an example of "macro.def". When M1 is touched, the client program will get "Hello, World !" from tablet server as if they were typed in from keyboard.

```
Hello, World !
htrain
hclus
hdisam
hanal
```

Fig. 13 Macro.def

(11) Tablet Server:

. Fig. 14 illustrates the structure of the PCOHR system.

Fig. 14 PCOHR structure

. The tablet server is a memory resident program. On one hand, it handles the tablet interrupt. On the other hand, it provides all tablet related services to the client programs.

. Four major data are managed by the server. They are:

(1) TBLQ, which holds the data from tablet,
(2) SYB, which holds the recognition features of the unknown symbol,
(3) DIC, which holds the recognition features of templates,
(4) KEYQ, which holds the keystrokes ready to send.

. The client program gets services from tablet server by calling INT 60 and specifying parameters as follows.

AH = 01: Clear TBLQ.
AH = 02: Put a coordinate pair to TBLQ.
Input: CX = x, DX = y, AL = stylus status.
(Stylus status: 1 -> up, 2 -> down, 4 -> away.)
AH = 03: Get a coordinate pair from TBLQ.
Output: CX = x, DX = y,
AL = stylus status.
(0 means no pair available.)
AH = 04: Clear DIC.
AH = 05: Load DIC.
Input: ES:SI = starting address, CX = length.
AH = 06: Get DIC.
Output: ES:SI = starting address, CX = length.
AH = 07: Load macros.
Input: ES:SI = starting address, CX = length.
AH = 08: Get a keystroke from KEYQ.
Output: AL = keystroke.
(0 means no keystroke available.)
AH = 09: Peek a keystroke from KEYQ.
Output: AL = keystroke.
(0 means no keystroke available.)
AH = 10: Get the coordinates of the next symbol.
Output: ES:SI = starting address, CX = length.
AH = 11: Get the features of the next symbol.
Output: ES:SI = starting address, CX = length.
AH = 13: AL = 1, turn on downsampling,
AL = 0, turn off downsampling.

. After the server is installed, the keyboard and mouse emulators are also installed. Any IBM PC software which uses standard keyboard/mouse I/O routines (i.e. INT 16 and INT 33) automatically becomes a client of the server. The keystrokes of a recognized symbol and the events of the stylus are sent to the client as if they were data from keyboard and mouse.

(12) The C compiler used for PCOHR is Computer Innovation C86 V2.30. The assembler is Microsoft Macro Assembler.