PERGAMON

Pattern Recognition 32 (1999) 1099–1114

PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY

Recognizing on-line handwritten alphanumeric characters through flexible structural matching

Kam-Fai Chan, Dit-Yan Yeung*

Department of Computer Science, The Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong, Peoples's Republic of China

Received 13 April 1998; received in revised form 27 October 1998; accepted 27 October 1998

Abstract

Speed, accuracy, and flexibility are crucial to the practical use of on-line handwriting recognition. Besides, extensibility is also an important concern as we move from one domain to another which requires the character set to be extended. In this paper, we will propose a simple yet robust structural approach for recognizing on-line handwriting. Our approach is designed to achieve reasonable speed, fairly high accuracy and sufficient tolerance to variations. At the same time, it maintains a high degree of reusability and hence facilitates extensibility. Experimental results show that the recognition rates are 98.60% for digits, 98.49% for uppercase letters, 97.44% for lowercase letters, and 97.40% for the combined set. When the rejected cases are excluded from the calculation, the rates can be increased to 99.93%, 99.53%, 98.55% and 98.07%, respectively. On the average, the recognition speed is about 7.5 characters per second running in Prolog on a Sun SPARC 10 Unix workstation and the memory requirement is reasonably low. With this simple yet robust structural approach, we already have an effective and efficient on-line character recognition module. This module will be used as part of a larger system, a pen-based mathematical equation editor, which is being developed by the authors using a syntactical pattern recognition approach. © 1999 Pattern Recognition Society. Published by Elsevier Science Ltd. All rights reserved.

Keywords: Structure extraction; Structural primitives; Flexible structural matching; On-line handwritten character recognition

1. Introduction

Automatic recognition of on-line handwriting provides one of the most natural ways for human beings to interact with computers without having to learn any extra skill (e.g., typing). Research in this area has been active for more than three decades [1]. Different approaches, such as statistical, syntactic and structural, and neural network approaches, have been proposed.

Characters consist of line segments and curves. Different spatial arrangements of these elements form different characters. In order to recognize a character, we should first find out the structural relationships between the elements which make up the character. However, in practice, the syntactic and structural approach [2,3] suffers from several drawbacks. One of the major concerns is the need for robust extraction of primitives [4].

* Corresponding author. Tel.: +852-2358-6977; Fax: +852-2358-1477; E-mail: dyyeung@cs.ust.hk

In order to make on-line handwriting recognition feasible, at least three issues have to be considered: speed, accuracy, and flexibility [5]. High speed and accuracy are always the key characteristics of any system, while flexibility is also important here due to diverse writing styles of different writers. In other words, an ideal handwriting recognizer should be able to quickly and accurately recognize a reasonably wide range of handwriting input.

Most, if not all, handwriting recognition systems in existence are designed to recognize predefined character sets of finite size. Typically they include digits, uppercase letters and lowercase letters. However, when we move from one application to another, some characters may have to be added to the character set. However, many existing methods require that complete retraining from scratch be conducted and hence previous training efforts wasted (e.g., [6]). It would certainly be desirable to reuse what is already available as much as possible.

In this paper, we will propose a simple yet robust structural approach for recognizing on-line handwriting. Our approach is designed to achieve reasonable speed, fairly high accuracy and sufficient tolerance to variations. At the same time, it maintains a high degree of reusability and hence facilitates extensibility. First of all, we will review some related work. After introducing the structural primitives used in our proposed representation scheme, we will give an overview of the recognition process. Then, we will discuss how to extract structural primitives from the input and undertake some reconstruction steps in a robust manner. Afterwards, we will explain how to perform classification through model matching. We will also illustrate how to resolve ambiguities through some examples. Finally, we will present and discuss some experimental results, which are then followed by some concluding remarks.

2. Related work

Using the structural approach, two-dimensional patterns, such as characters, can be represented in at least two different ways. The first one is to use a representation formalism which is by nature of high dimensionality, such as an array, a tree or a graph [7,8]. The second one is to incorporate additional relationships between primitives into a one-dimensional representation form. Two well-known methods using the latter approach are the picture description language (PDL) [9] and the plex grammar [10].

Note that we need to consider the trade-off between expressive power and time complexity for processing when we choose any representation formalism. Graphs have the highest expressive power, but the detection of exact or approximate subgraph isomorphism is known to be intractable [11]. On the other hand, string matching is of polynomial time complexity, but its expressive power is much lower. When efficiency is our major concern, like in this paper, string representations are generally preferred. Hence, we will focus on string representations rather than high-dimensional representations.

Among the various string representations, the PDL and the plex grammar are usually used to describe how primitives are connected. These schemes may become very tedious when there exist large variations within the character classes. Berthod and Maroy's primitives [12] attempt to address the problem of high variability. However, their method does not make use of directional information. On the other hand, Freeman's chain code [13] and some extended schemes (e.g., [14]) use directional information to form primitives, although the resulting representations are often not compact enough.

2.1. Picture description language

The picture description language (PDL) [9] is one of the earliest attempts to describe pictorial patterns using a formal language. In PDL, each primitive has two connection points, i.e., *head* and *tail*. Four binary operators, denoted by $+$, $-$, $\times$ and $*$, are defined to combine a pair of PDL expressions. On the other hand, the unary operator $\tilde{}$ is used to reverse the head and tail of a primitive or a PDL expression. Fig. 1 shows how primitives are connected using these four binary operators.

By definition, a PDL expression is either

- a primitive, or

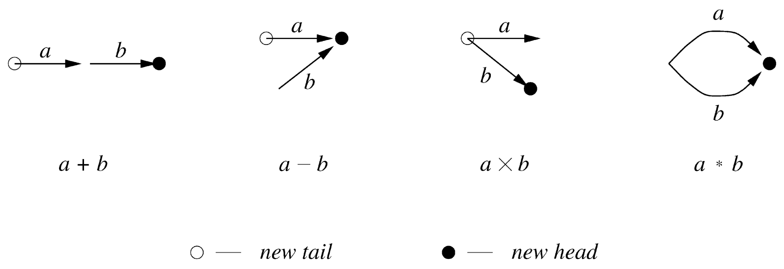


Fig. 1. Four binary operations in PDL.

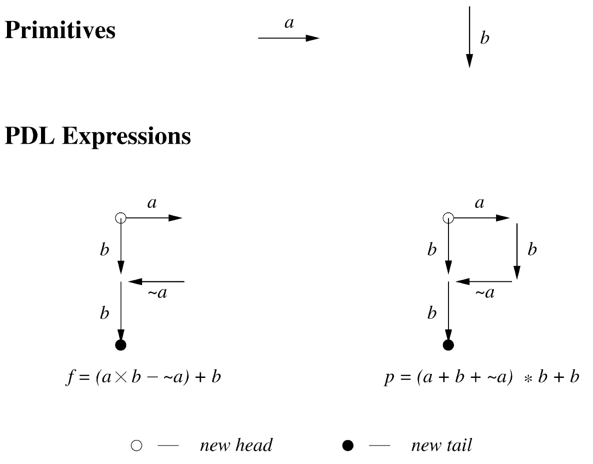


Fig. 2. Two example PDL expressions.

- a PDL expression preceded by a unary operator, or
- two PDL expressions connected by a binary operator.

One of the advantages of PDL is that pictorial patterns can be represented by strings, i.e., PDL expressions. Fig. 2 shows two PDL expressions which describe the structures of two characters, ‘F’ and ‘P’. However, PDL has its limitation that all connections between primitives or complex patterns are allowed at only two points. Also, the same pattern can sometimes be described by more than one PDL expression.

Intuitively, the ‘P’ in Fig. 2 could simply be represented as the concatenation of the PDL expression for ‘F’ and the primitive ‘b’. However, this combination is impossible because of the incompatibility between the heads and tails of the two structures. As a result, we need to describe the letter ‘P’ by another PDL expression.

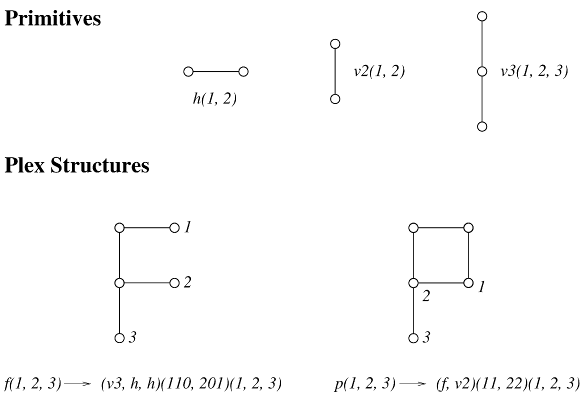
2.2. Plex grammar

The plex grammar [10] was designed to overcome the limitation of PDL by allowing more than two connection points.

The basic unit of the plex grammar is an *n-attaching point entity*, or simply *nape*. It is represented by an identifier and a list of attaching points. Structures formed by connecting the napes together are called *plex structures*. A plex structure consists of three components:

- a list of napes,
- a list of internal connections between napes, and
- a list of attaching points which can be used for joining the plex structure with other napes or plex structures.

Fig. 3 shows the structures of characters ‘F’ and ‘P’ in plex grammar. Three napes, h, v2 and v3, are used to



Primitives
h(1, 2)
v2(1, 2)
v3(1, 2, 3)
Plex Structures
1
2
3
1
2
3
f(1, 2, 3) ⟶ (v3, h, h)(110, 201)(1, 2, 3)
p(1, 2, 3) ⟶ (f, v2)(11, 22)(1, 2, 3)

Fig. 3. Two example plex structures.

create new plex structures. For example, the plex structure of the nape, f(1, 2, 3), is as follows:

$$(v3, h, h)\ (110, 201)\ (1, 2, 3).$$

The first part is the list of napes used in the structure. The second part gives information about the two internal connections:

- The first value, 110, means that a connection exists between point 1 of v3 and point 1 of the first h. Note that the second h is not involved in this connection, and hence the third digit is 0.
- The second value, 201, means that there exists a connection between point 2 of v3 and point 1 of the second h. Similarly, a value of 0 for the second digit shows that the first h is not involved in this connection.

The last part is the list of points which can be used to form connections with other napes or plex structures.

The plex grammar suffers from the same problem as PDL in that a pattern may be represented by several different plex structures corresponding to different orders in listing the napes.

2.3. Berthod and Maroy’s encoding scheme

In Berthod and Maroy’s encoding scheme [12], there are five primitives:

- straight line, denoted by **T**,
- positive (counter-clockwise) curve, denoted by **P**,
- minus (clockwise) curve, denoted by **M**,
- pen-lift, denoted by **L**,
- cusp, denoted by **R**.

Every character can be represented by a string of primitives. Fig. 4 shows some examples of characters which are encoded using Berthod and Maroy’s scheme.

Note that some resulting strings are ambiguous. For example, the code **P** has three interpretations {C, L, U}

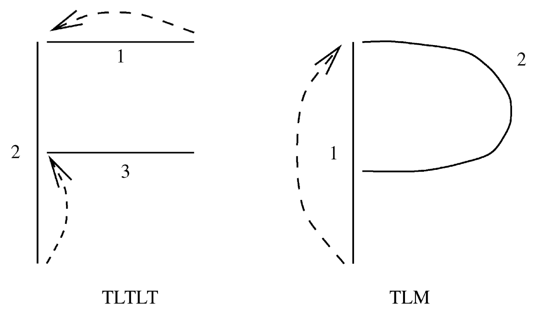


Fig. 4. Examples of some characters encoded using Berthod and Maroy's scheme.

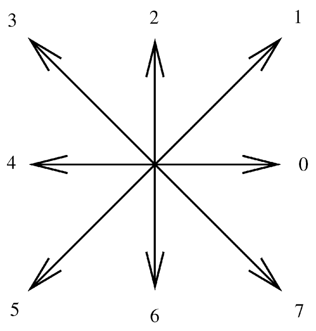


Fig. 5. Direction values used in Freeman's chain code.

and the code **TLTLT** has eight interpretations {A, F, H, I, K, N, Y, Z}. In the latter case, some of the characters are not really similar at all, e.g., 'H' has two vertical and one horizontal strokes while 'I' has one vertical and two horizontal strokes. The scheme is thus not very informative.

2.4. Freeman's chain code and some extended schemes

Freeman's chain code [13] is a widely used chain-coding method. It consists of eight values, 0–7, which indicate how the current point is connected to the next one. Fig. 5 shows the direction values in Freeman's chain code.

The extended Freeman and the improved extended Freeman encoding schemes [14] both use the eight direction values in Freeman's chain code as primitives. During the preprocessing stage, all values that are the same as their preceding ones in the chain code are removed so that the resulting chain code can be shorter and at the same time preserves all the critical information.

The extended Freeman encoding scheme pays special attention to the pen-lifting action by assigning a direction value (with a bar on it) to the pair of points between pen-up and its successive pen-down. As a result, the

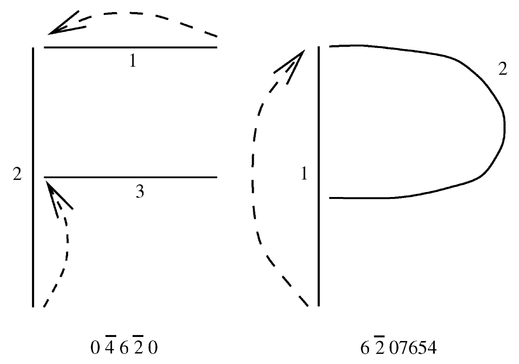


Fig. 6. Some example characters encoded using the extended Freeman encoding scheme.

number of primitives is increased to 16. Note that this extended scheme still creates some ambiguous cases, but the degree is much less than that of Berthod and Maroy's scheme. Fig. 6 shows some example characters encoded using the extended Freeman encoding scheme.

The improved extended Freeman encoding scheme reduces the number of primitives back to eight by representing $\bar{0}, \bar{1}, \ldots, \bar{7}$ simply as 0, 1, …, 7. According to the authors [14], the classification result remains the same as the one in the extended Freeman encoding scheme but its primitives are comparatively simpler.

3. Overview of the recognition process

3.1. Structural primitives

Characters are composed of line segments and curves. Every line segment or curve can be extended along a certain direction. A curve that joins itself at some point forms a loop. Hence, in our representation, we will use as primitives different types of line segments and curves with some directional information. Note that a single stroke may consist of several primitives. Basically, there are five types of primitives:

- line,
- up (curve going counter-clockwise),
- down (curve going clockwise),
- loop (curve joining itself at some point), and
- dot (a very short segment which may sometimes be just noise; we, however, cannot simply ignore it since it may be part of a character, like in 'i' and 'j').

To represent the directional information, we also employ Freeman's chain code [13] which consists of eight values, i.e., 0–7.

The grammar, G, which expresses our proposed structure in a string representation, is a 4-tuple $G = (V_T, V_N,$

P, S) where

- $V_T = \{$line, up, down, loop, dot, '{','}', ',', 0, 1, 2, 3, 4, 5, 6, 7$\}$, with 0–7 denoting the direction values in Freeman's chain code,
- $V_N = \{$Character, StrokeSet, Stroke, PrimitiveSet, Primitive, LineType, Direction$\}$,
- P is a set of production rules as follows:

$$P = \{$$

$$\begin{array}{rcl}
\text{Character} & \rightarrow & \{\text{StrokeSet}\} \\
\text{StrokeSet} & \rightarrow & \text{Stroke} \\
\text{StrokeSet} & \rightarrow & \text{Stroke, StrokeSet} \\
\text{Stroke} & \rightarrow & \text{PrimitiveSet} \\
\text{PrimitiveSet} & \rightarrow & \text{Primitive} \\
\text{PrimitiveSet} & \rightarrow & \text{Primitive, PrimitiveSet} \\
\text{Primitive} & \rightarrow & \{\text{LineType, Direction}\} \\
\text{Primitive} & \rightarrow & \text{loop} \\
\text{Primitive} & \rightarrow & \text{dot} \\
\text{LineType} & \rightarrow & \text{line} \mid \text{up} \mid \text{down} \\
\text{Direction} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7
\end{array}$$

$$\}$$

and
- the start symbol, S, is Character.

Note that no directional information is associated with isolated dots. Also, for simplicity, we do not associate directions with loops at this stage. The direction of a line or a curve depends on the starting and ending points. Fig. 7 shows some examples.

3.2. Model set

Before we can perform recognition, we need to define models for the corresponding character classes. Note that all models should be distinct. In other words, there exist no two models with the same number of strokes and the same sequence of primitives in each stroke. Also, different character classes may have different numbers of models depending on the complexity of their structures. Fig. 8 shows some examples.

As shown in Fig. 8, some models look very similar but have different numbers of strokes. In some cases, we may be able to combine some strokes together and hence reduce the number of models in the model set. However, for simplicity, we have not implemented this in our current work.

3.3. Recognition process

After we have written a character on the digitizer, what we get is only a sequence of points. In order to recognize the character, we must first extract the structural primitives from the point sequence to form a preliminary

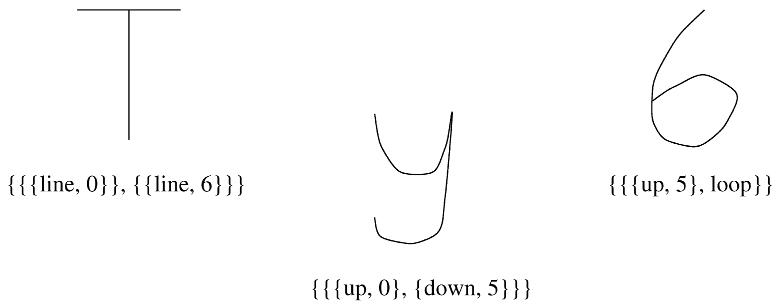


Fig. 7. Examples of the representations for some characters.

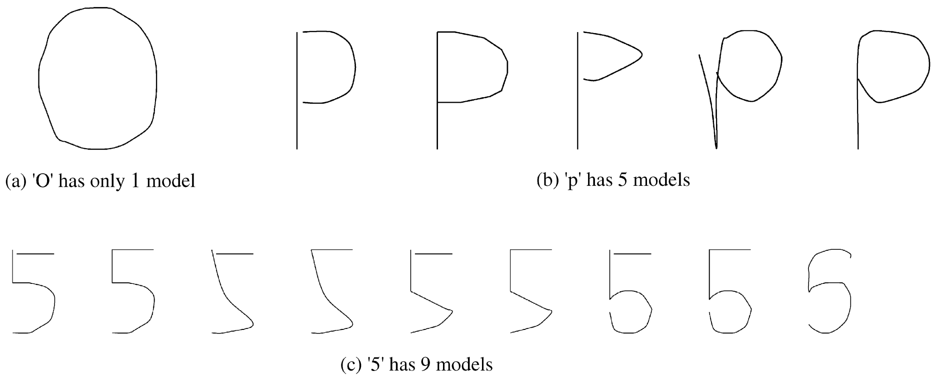


Fig. 8. Examples of the models for some character classes.

structure. Occasionally, some kinds of reconstruction may be required if certain conditions are met. With the finalized structure, we then compare it with the models in the model set and try to find a match. Here we apply flexible structural matching in order to increase the chance of finding a match. However, we sometimes may get ambiguous results and therefore need further classification. Fig. 9 summarizes the major stages of the recognition process.

4. Structure extraction and reconstruction

4.1. Structure extraction

In each character, there may be one or more strokes. Each stroke consists of a number of points that trace out a path on the writing surface from pen-down to pen-up in normal handwriting style. Every pair of consecutive points induces a direction. For points that follow the same direction or have a slight turn, we group them into one line segment. On the other hand, if there is a sharp turn along a stroke, we will represent it with multiple line segments. Fig. 10 shows the steps taken to extract the structure of a digit '3'.

In practice, some writers produce characters which are hard to recognize. Most cases are the result of different writing habits, for example, writing strokes in a reverse or unusual order. However, some may be due to the poor quality of the hardware. As a result, zig-zag line segments sometimes occur. To solve the problem, we can simply extract the mid-points of those zig-zag line segments and connect them together to form a stroke. Fig. 11 shows how zig-zag line segments are eliminated during structure extraction.

4.2. Structure reconstruction

After obtaining the preliminary structure, we sometimes may need to either combine some lines and curves

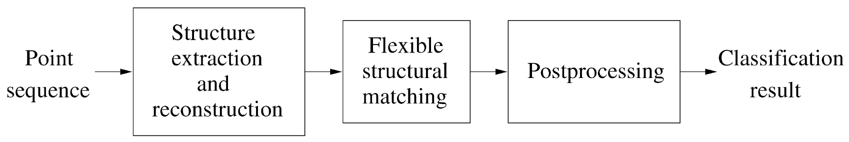

Point sequence
Structure extraction and reconstruction
Flexible structural matching
Postprocessing
Classification result

Fig. 9. Overview of the recognition process.

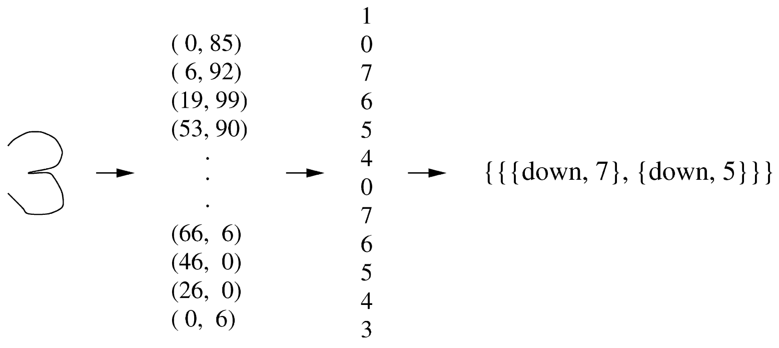


Fig. 10. Steps taken to extract the structure.

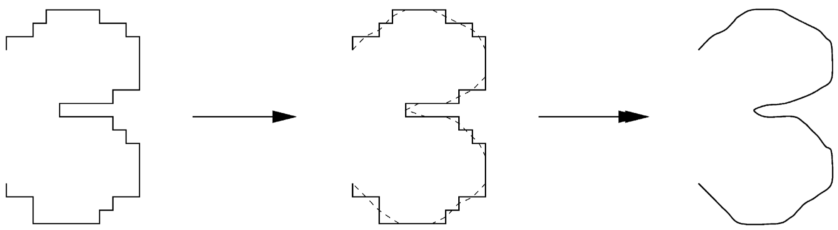


Fig. 11. Eliminating zig-zag line segments during structure extraction.

together under certain conditions, or further extract some sub-structures, such as loops, if they are detected in the original structure.

4.2.1. Combining lines and curves

Occasionally, a smooth stroke may be broken into parts due to either poor writing or low-quality hardware. In order to remedy this, we have to check each pair of consecutive primitives in a stroke. If some conditions are met, we will combine the two primitives together to form a new one. Table 1 shows some conditions under which consecutive primitives are combined.

Note that T_1 and T_2 denote the types of primitives and D_1 and D_2 represent their directions, respectively. The new primitive has *NewT* as its type and *NewD* as its direction (from the starting point of the first primitive to the ending point of the second primitive).

As shown in Table 1, in order to combine two lines together, their directions must be the same. Fig. 12 shows an example.

Combining curves is slightly more complicated. First of all, we have to determine the joining type which describes how the curves are connected. For example, if the direction of the last line segment of the first primitive is 2 and that of the first line segment of the second primitive is 4, it implies that the change is from 2 to 4. As shown in Fig. 13, there are two alternatives to change from 2 to 4. Here we always choose the shorter path, and therefore, JointType = find_joint_type (2, 4) = up.

By comparing the value of JointType with the types of both curves, we are then able to decide whether the two curves can be combined. Fig. 14 illustrates when two curves can and cannot be combined together.

After this stage, sets of characters which have only slight variations in point locations can be grouped under their corresponding structures. Fig. 15 shows some variations of the digit '3' which share the same structure.

4.2.2. Extraction of loops

As mentioned above, the only information from the input is just a sequence of points. By checking the relative positions of consecutive points, we can infer a chain code that reveals the directional change of points. However, such a chain code only allows us to detect lines and curves, but not loops. Fig. 16 shows how two different characters, 'C' and 'O', have the same preliminary structure.

In addition to the directional information between consecutive points, we may also measure the directional information of points with respect to the starting point. To obtain this additional code sequence, we use the same method for calculating direction values as above. The major difference here is that the value -1 will be returned when the distance between the current point and the starting point is less than a certain threshold. As a result, this indicates that a loop is detected. Fig. 17 shows how this additional code sequence helps to detect a loop.

In general, a loop can be found in three different locations of a stroke:

1. at the beginning,
2. in the middle, and
3. at the end.

Fig. 18 gives some examples.

When the loop is at the beginning of a stroke, the method described above can detect its occurrence. However, it does not help when the loop is either in the middle or at the end of a stroke. Fortunately, for these two cases, some forms of fluctuation often occurs in the additional code sequence, as shown in Fig. 18b and c.

Table 1
Some conditions for combining consecutive primitives

Primitive 1: $\{T_1, D_1\}$	Primitive 2: $\{T_2, D_2\}$	Condition	New primitive: $\{NewT, NewD\}$
{line, D_1}	{line, D_2}	$D_1 = D_2$	{line, D_1}
{up, D_1} or {down, D_1}	{up, D_2} or {down, D_2}	$T_1 = T_2$ = JointType	$\{T_1$, *NewD*$\}$

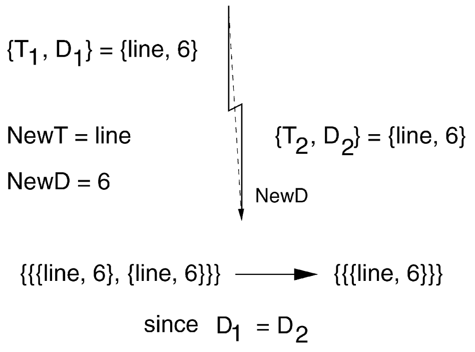


Fig. 12. Combining two lines into one.

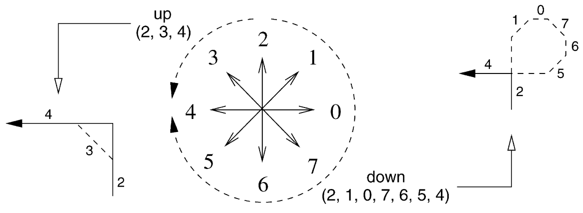
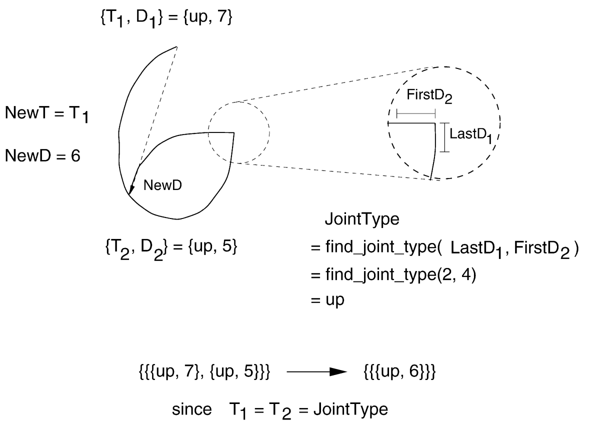
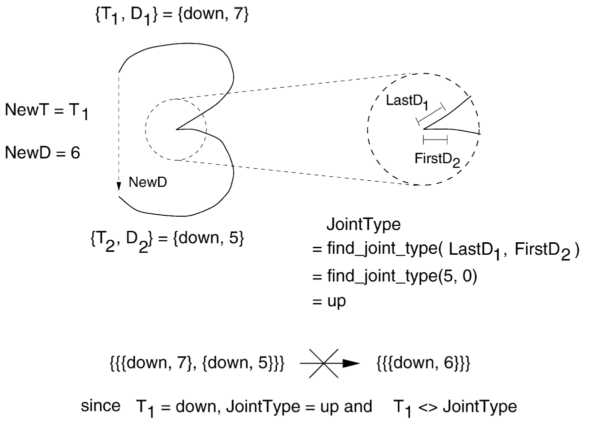


Fig. 13. Alternatives in changing from 2 to 4.


$\{T_1, D_1\} = \{\text{up}, 7\}$
NewT = T_1
NewD = 6
NewD
FirstD_2
LastD_1
$\{T_2, D_2\} = \{\text{up}, 5\}$
JointType
= find_joint_type(LastD_1, FirstD_2)
= find_joint_type(2, 4)
= up
{{{up, 7}, {up, 5}}} → {{{up, 6}}}
since $T_1 = T_2$ = JointType

(a) Two curves can be combined together


$\{T_1, D_1\} = \{\text{down}, 7\}$
NewT = T_1
NewD = 6
NewD
LastD_1
FirstD_2
$\{T_2, D_2\} = \{\text{down}, 5\}$
JointType
= find_joint_type(LastD_1, FirstD_2)
= find_joint_type(5, 0)
= up
{{{down, 7}, {down, 5}}} $\not\rightarrow$ {{{down, 6}}}
since T_1 = down, JointType = up and T_1 <> JointType

(b) Two curves cannot be combined together

Fig. 14. Examples illustrating when two curves can and cannot be combined together.

When the loop is in the middle of a stroke, we can check the distance between points, starting from the two ends of the stroke. When the distance decreases, we will continue to move inward until the distance is less than a certain threshold. When this happens, a loop is said to be found. For the case of having a loop at the end of a stroke, we apply a similar method. However, we will fix the ending point this time. Fig. 19 illustrates these two cases.

In general, loops can be detected by using one of the above methods, or a combination of some of them. For

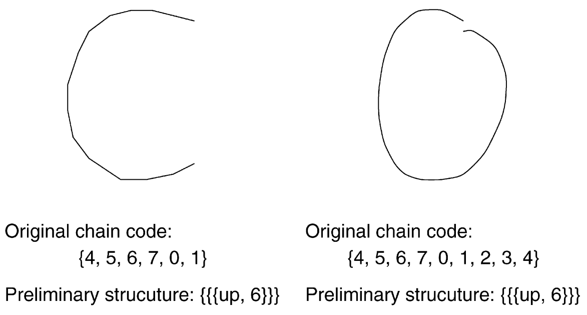


Fig. 16. Two different characters have the same preliminary structure.

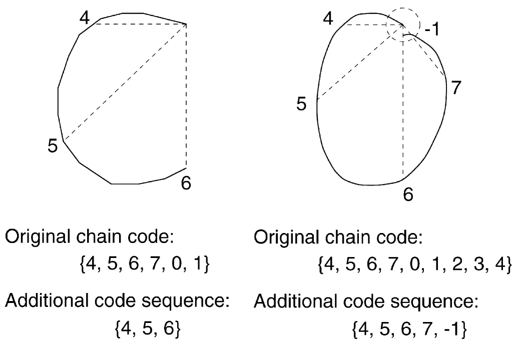


Fig. 17. Detecting the occurrence of a loop.

example, ‘g’ contains loops both at the beginning and in the middle (sometimes, at the end) of the stroke. We, therefore, require a combination of two methods for detecting them. Fig. 20 shows some examples of combining several methods to extract loops from a stroke.

Detection of loops is not always trivial as loops may appear in some characters in special ways. Sometimes, even where the stroke starts may affect the methods used to extract the loops. For example, if the starting position is near the center of the character ‘8’, two loops can easily be found as indicated by the two values of -1 found in the additional code sequence. However, if the starting point is near the top of the character ‘8’, only one value of -1 is found. Luckily, the fluctuations in both the original chain code and the additional code sequence give us hints for detecting such double loops. Fig. 21 illustrates these two examples.

In the two cases above, we should notice that the two methods used are just variants of the method for detect-

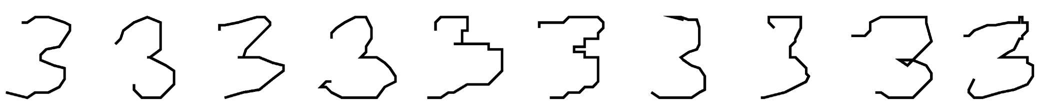


Fig. 15. Various occurrences which have the same structure as the digit ‘3’ in Fig. 10

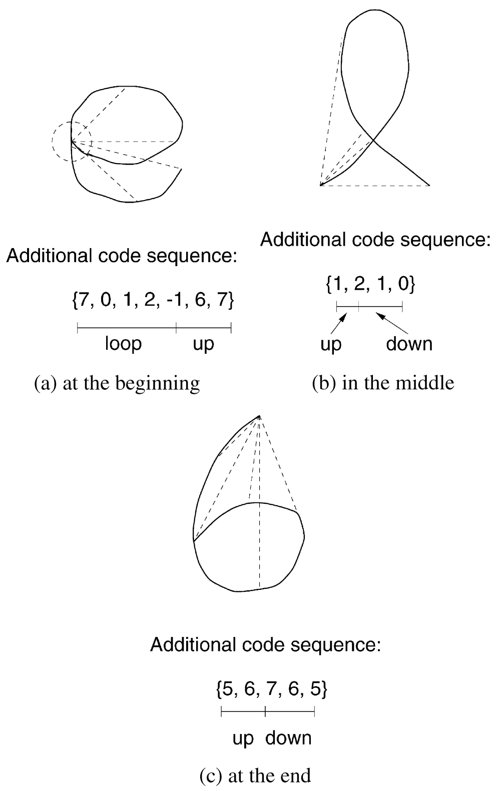


Fig. 18. Different locations in which a loop can occur within a stroke.

ing loops from the beginning of the stroke. In general, the three major methods as shown in Fig. 18 are already sufficient for finding loops in characters.

5. Flexible structural matching

After extracting the structure of a character (possibly with some reconstruction steps involved), we can then match it against a set of models. However, due to different writing styles and habits, variations within the same character class are not uncommon. In order to increase the recognition rate, those characters that do not have an exact match will be slightly varied in shape and direction in an attempt to find approximate matches.

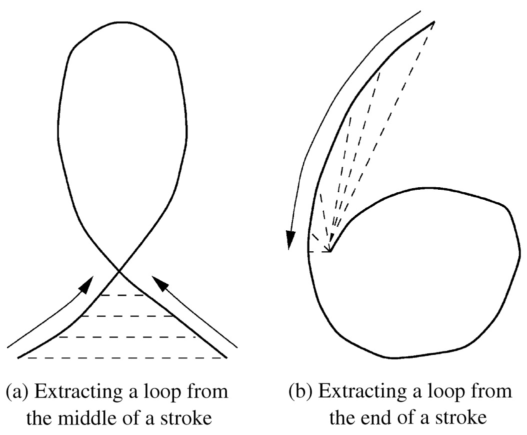

(a) Extracting a loop from the middle of a stroke
(b) Extracting a loop from the end of a stroke

Fig. 19. Extracting loops from the middle and the end of a stroke.

Most structural matching methods deal with graph representations directly [7]. Our method, instead, works on string representations. The following is our matching algorithm:

Algorithm. *Flexible structural matching*

1. Load the set of models in Z.
2. Extract the structure of the test character C.
3. Initialize the deformation level L to be 1.
4. Let S be the candidate set and $S = deform(L, C)$.
5. Let M be the match set and $M = match(Z, S)$.
6. If M is not empty, return M. Otherwise, $L = L + 1$.
7. If L is less than or equal to the maximum deformation level, go to step (4). Otherwise, exit and report failure of finding an exact match.

After loading the set of models and extracting the structure of the test character (as described in Section 4), we start to find a match or matches. First of all, we will simply compare the structure for the unknown character against the set of models to see whether at least one match can be found. If no match is resulted, we will deform the test structure in certain ways so as to increase the chance for finding matches. When all the deformation methods are exhausted and no matches are found, we then report failure.

Basically, there are four levels of structural deformation. Here we have to emphasize again that the search

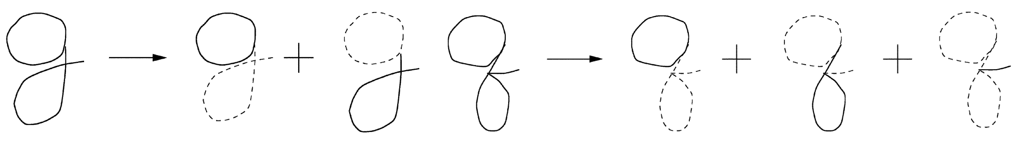


Fig. 20. Combination of several methods to extract loops from a stroke.

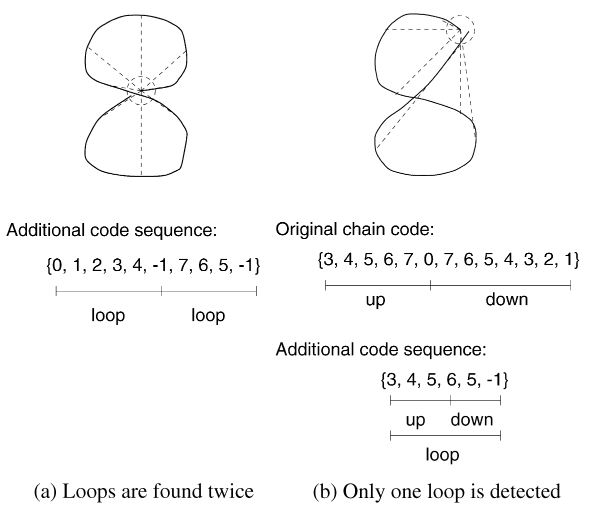
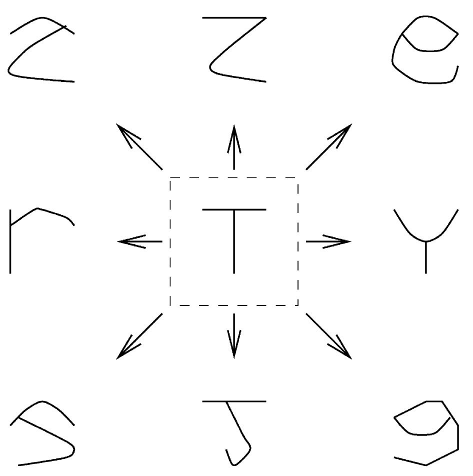


Fig. 21. Variants of the original method for detecting loops from the beginning of a stroke.

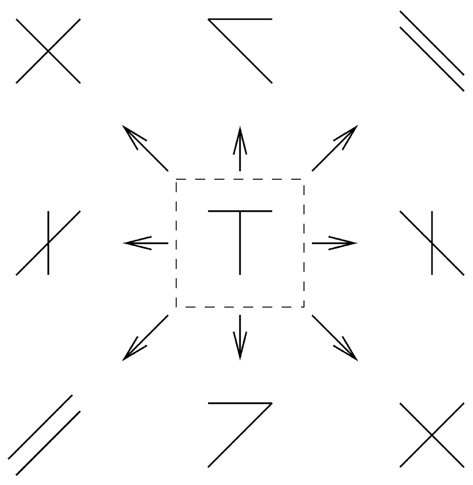
will stop once a match (or matches after deformation is performed) has been found. As an example, a regularly written 'T' is likely to get correct classification during the first level of matching. However, in order to illustrate the generality of our scheme, we will use this simple character anyway as an illustrative example in our following discussions:

1. *No deformation*: The test pattern has to be exactly the same as one of the models.
2. *Primitive type deformations*: When there is no exact match, we will vary the primitive type in an attempt to find an approximate match. In so doing, line may become either one of its two neighboring types, i.e., up and down (since line is midway between up and down). However, up can only become line, but not down (since line is the only neighbor of up). Similar restrictions also apply to down. As a result, a 'T' will have eight relaxed versions as shown in Fig. 22a.
3. *Directional deformations*: Similarly, we may also vary the direction. To do so, we find a neighboring code of the current one. For example, {line, 5} may become {line, 4} or {line, 6}. As a result, a 'T' will have eight relaxed versions, though two of them are the same, as shown in Fig. 22b.
4. *Simultaneous type and directional deformations*: When no exact pattern can be found during the previous relaxation steps, we may consider finding the nearest match by deforming both the primitive type and direction simultaneously. As a result, a much larger number of patterns will be covered. For example, a 'T' will have 80 possible deformed versions.

Note that many false-positive cases may be resulted if too much flexibility is allowed. Hence, we may need some



(a) Relaxed versions of 'T' as a result of applying type deformations



(b) Relaxed versions of 'T' as a result of applying directional deformations

Fig. 22. Examples of structural deformations.

additional steps to verify the answer if domain-specific information is available to narrow down the matching results.

By using flexible structural matching, some previously unmatched characters are able to find a match. This increases the recognition rate and at the same time decreases the rejection rate. Fig. 23 shows some more variations of the digit '3' classified under the same structure as the '3' in Fig. 10.

6. Postprocessing

With flexible structural matching, some ambiguities may occur. Here we have a choice either to report all the ambiguous cases as answer, or to add some postprocessing steps to determine the most probable choice based on additional information.

In general, there are two major types of ambiguities:

1. Some character classes are represented using the same structure. For example, 'D' and 'P' are often classified under the structure {{{line, 6}}, {{down, 6}}}. In order to distinguish between them, we may consider the relative position of the vertical stroke and the curve. Fig. 24 outlines the algorithm for doing so.
2. Some character classes have similar appearances so that ambiguous results are often produced, for example, '1' and '7', 'A' and 'H', 'u' and 'y', and so on. Fig. 25 outlines the algorithm for distinguishing between 'u' and 'y'. For this case, the difference between the heights of the two primitives becomes crucial.

7. Experimental results and discussions

7.1. Experimental results

In our experiment, we used an on-line handwriting dataset collected by the MIT Spoken Language Systems Group [15]. It is a subset of the full set for isolated alphanumeric characters only. There are 62 character classes (10 digits, 26 uppercase and 26 lowercase letters) in our set. Each character class has 150 different entries written by 150 different people. Totally, there are 9300 characters. More than half of them are regularly written. The remaining ones are those either with noise in the data, poorly written, deliberately written in some strange and unusual way, or with zig-zag line segments. Fig. 26 shows some examples of the characters in the dataset.

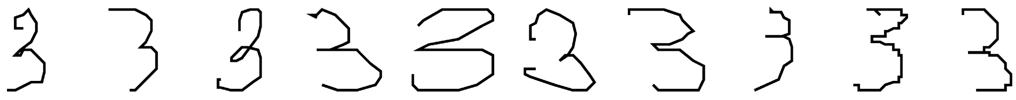
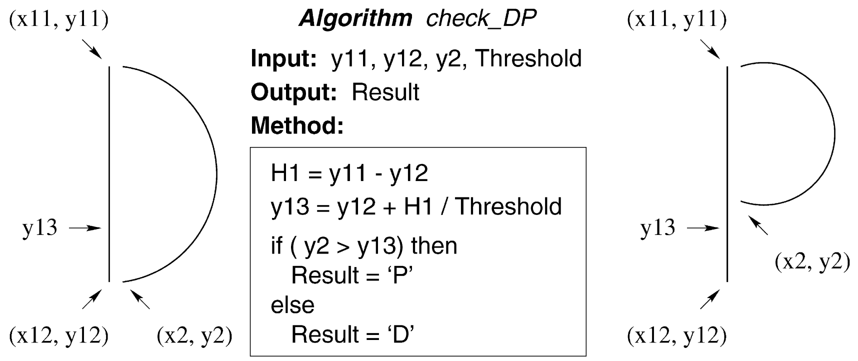
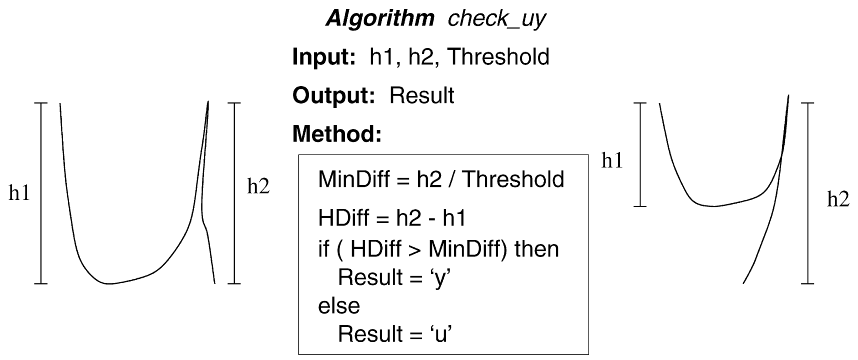


Fig. 23. Digits grouped under the same structure as the digit '3' in Fig. 10 after flexible structural matching.



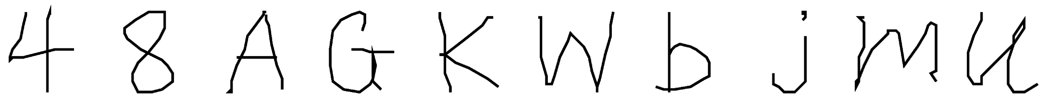
```
Algorithm check_DP
Input: y11, y12, y2, Threshold
Output: Result
Method:
    H1 = y11 - y12
    y13 = y12 + H1 / Threshold
    if ( y2 > y13) then
      Result = 'P'
    else
      Result = 'D'
```

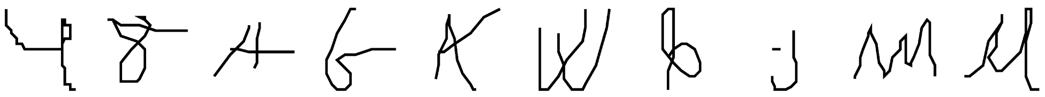
Fig. 24. Outline of an algorithm for distinguishing between 'D' and 'P'.



```
Algorithm check_uy
Input: h1, h2, Threshold
Output: Result
Method:
    MinDiff = h2 / Threshold
    HDiff = h2 - h1
    if ( HDiff > MinDiff) then
      Result = 'y'
    else
      Result = 'u'
```

Fig. 25. Outline of an algorithm for distinguishing between 'u' and 'y'.


(a) Regularly written characters


(b) Poorly written characters

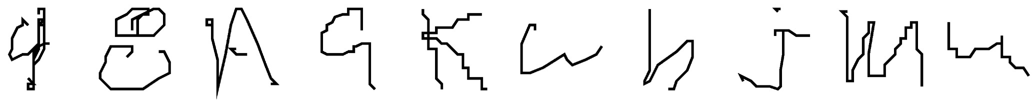

(c) Characters that may be ambiguous or have noise in the data

Fig. 26. Some examples of the characters in the data set.

Table 2
Recognition results for different character sets

	Correct	Incorrect	Rejected
Digits	$\frac{1479}{1500}$ (98.60%)	$\frac{1}{1500}$ (0.07%)	$\frac{20}{1500}$ (1.33%)
Uppercase letters	$\frac{3841}{3900}$ (98.49%)	$\frac{18}{3900}$ (0.46%)	$\frac{41}{3900}$ (1.05%)
Lowercase letters	$\frac{3800}{3900}$ (97.44%)	$\frac{56}{3900}$ (1.43%)	$\frac{44}{3900}$ (1.13%)
All	$\frac{9058}{9300}$ (97.40%)	$\frac{178}{9300}$ (1.91%)	$\frac{64}{9300}$ (0.69%)

For all character classes, we have developed algorithms accordingly to resolve ambiguities during the postprocessing steps. As a result, there are only three possible outcomes in the recognition: correct, incorrect, and rejected. Table 2 summarizes our results.

Note that some characters, for example, '0', 'O' and 'o', have exactly the same structure in their corresponding character sets. When they are mixed together in the combined set, it is intrinsically impossible to determine the correct class unless we have additional contextual information. Hence, in our experiment, we treated this kind of ambiguity as correct recognition. In other words, if the result indicates that the test character '0' may be either '0', 'O' or 'o', we regard this as a correct decision. Table 3 shows all such examples.

In general, incorrect recognition is sometimes due to the ambiguous nature of the characters. Fig. 27 shows some examples. Rejection, on the other hand, is often the result of abnormal writing style, e.g., '4', '8', 'A', and 'j' in Fig. 26c.

Table 3
Characters in different character sets that have the same structure

	Pairs	Triples
All models share the same structure(s)	(C, c), (O, o), (P, p), (S, s), (V, v), (W, w), (X, x)	*nil*
Some models share the same structure(s)	(M, m), (U, u), (Y, y)	(I, l, 1), (O, o, 0)

If we do not count the rejected cases, the result is shown in Table 4.

Note that rejection could simply be avoided by adding the otherwise rejected case as a new model for its class. Caution should be taken though, as the number of models will increase and some models may be so specific that they are only responsible for very few (mostly just one) examples. Table 5 shows the numbers of models used in different character sets.

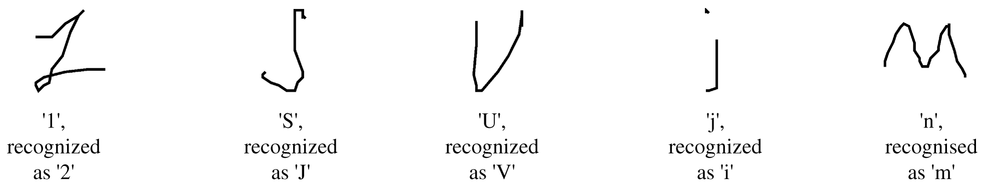


Fig. 27. Examples of some incorrectly recognized characters.

Table 4
Reliability rates for different character sets obtained by excluding the rejected cases from the calculation

	Reliability rate[a]
Digits	$\frac{1479}{1480}$ (99.93%)
Uppercase letters	$\frac{3841}{3859}$ (99.53%)
Lowercase letters	$\frac{3800}{3856}$ (98.55%)
All	$\frac{9058}{9236}$ (98.07%)

[a] The *reliability rate* refers to the percentage of *correct* classification when rejected cases are excluded from the calculation.

Table 5
Number of models used in different character sets

	Number of models per character set		
	Minimum	Maximum	Average
Digits	2	10	5.40
Uppercase letters	1	14	4.96
Lowercase letters	1	10	4.27
All	1	14	3.90

Table 6
Similar pairs in different character sets

	Similar pairs	Total number
Digits	(0, 6), (1, 7), (4, 9)	3
Uppercase letters	(A, H), (B, R), (C, G), (C, L), (D, P), (J, T), (K, R), (M, W), (N, U), (N, V), (N, Y), (U, V), (U, Y), (X, Y)	14
Lowercase letters	(a, d), (a, g), (a, q), (a, y), (b, h), (b, p), (c, e), (c, l), (d, j), (d, q), (d, u), (e, l), (f, t), (g, q), (g, s), (g, y), (h, n), (h, u), (i, j), (m, w), (n, r), (n, u), (n, v), (n, y), (r, v), (u, v), (u, y), (x, y)	28
All	All of the above, plus (B, 8), (D, b), (G, 6), (G, a), (G, b), (J, 5), (L, h), (T, 5), (Z, 2), (Z, 7), (a, 9), (b, 6), (g, 9), (q, 9), (s, 9), (y, 4), (y, 9)	62

7.2. Discussions

Here are some observations from the experimental results:

1. Different character sets have different misclassification rates. The digit set has the smallest while the combined set has the largest. To a great extent, the misclassification rate is correlated with the number of similar pairs in a character set, as shown in Table 6.
2. The number of rejected characters for the combined set is far less than the sum of those numbers for the individual sets. This is due to the wider coverage when all models are combined together. However, some previously rejected characters are misclassified. Some examples are shown in Fig. 28.
3. When we combine all the models together, it seems that the average number of models in the combined set should be within the range of the minimum and maximum. However, as shown in Table 5, the average obtained in our experiment is even less than the minimum (i.e., the average number of models in the lowercase letter set). This is due to the elimination of some duplicate models. As we mentioned above, some characters, like '0', 'O' and 'o', 'C' and 'c', 'M' and 'm', etc., have the same models in their corresponding sets. When we combine all sets together, the duplicates should be removed.
4. When writing a character, variations can occur in different ways, for example, in number of strokes, in stroke order, and in stroke direction. One simple way to tackle this problem is to add additional models. However, in order to keep the number of models

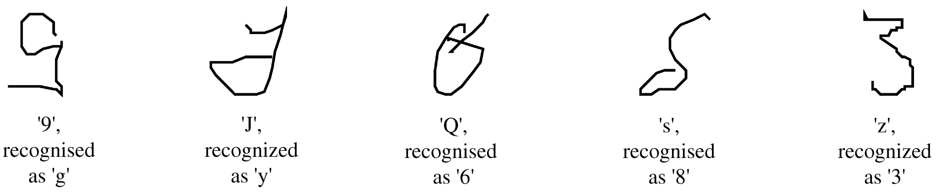


Fig. 28. Examples of some misclassified characters in the combined set that were previously rejected in their corresponding sets.

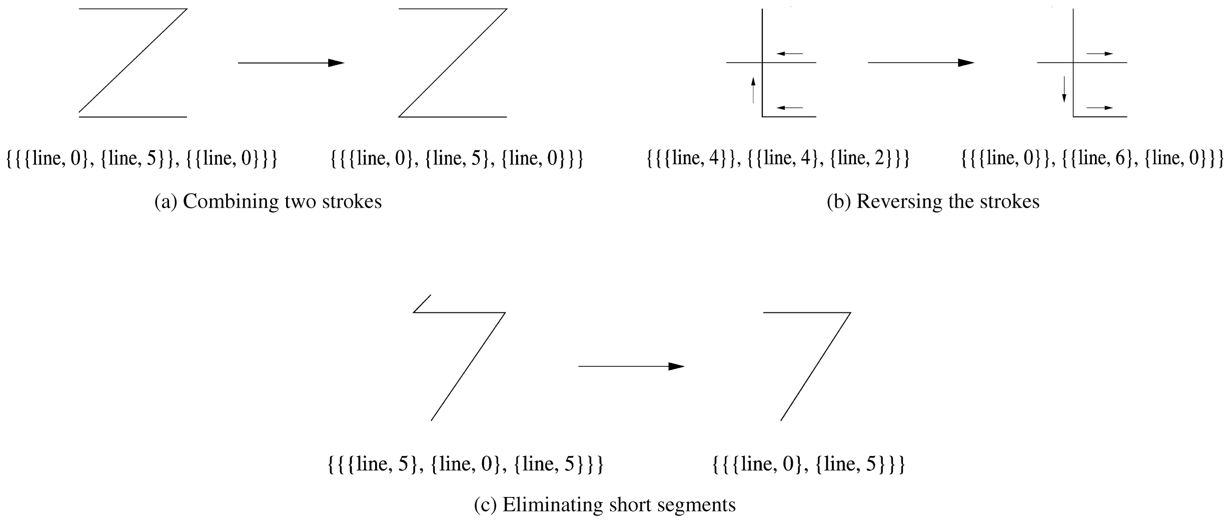

{{{line, 0}, {line, 5}}, {{line, 0}}}
{{{line, 0}, {line, 5}, {line, 0}}}
(a) Combining two strokes
{{{line, 4}}, {{line, 4}, {line, 2}}}
{{{line, 0}}, {{line, 6}, {line, 0}}}
(b) Reversing the strokes
{{{line, 5}, {line, 0}, {line, 5}}}
{{{line, 0}, {line, 5}}}
(c) Eliminating short segments

Fig. 29. Results of applying some heuristics.

reasonably small and at the same time increase the chance to find a match, an alternative way is to apply heuristics to the unmatched test pattern and see if the resulting pattern matches one or some of the models. Some feasible heuristics, as depicted in Fig. 29, are:

(a) If the ending point of a stroke is very close to the starting point of another stroke, we can combine the two strokes together.
(b) If the primitives in a stroke are all in some unusual directions (going up or going left), we may try to reverse the writing directions of the stroke.
(c) Some short segments may be the result of noise. By eliminating them from the test pattern, we may find a match.

5. In this research, we first work on three separate character sets, i.e., digits, lowercase and uppercase letters. We then combine all three sets together. An advantage of our approach is that we do not have to design everything from scratch when we move from individual sets to the combined set. Instead, we only have to handle those characters which bear similarities across character sets. Fig. 30 shows the relationships between models formed from different character sets. The shaded region is the portion which requires extra work when the combined set is formed.

   Since the next stage of our research is to include some mathematical symbols such as $+$, $-$, $*$, $/$, $\sum$, $\sqrt{}$, and $\int$, we also performed a preliminary experiment by adding the models of those symbols to the alphanumeric character set. As expected, we obtained a high recognition rate of 92.74% without the need for any modification because there are very few similar pairs between the mathematical symbols and those characters in the previously combined set. Further improvement would be expected by adding procedures to resolve ambiguities.
6. The matching algorithm used in this paper is very simple and easy to understand. However, when the test structure is a complicated one, it may become inefficient during the matching steps. Some heuristics, e.g., designing a good cost function to guide the search, may help in reducing the search time. It, however, is beyond the scope of this paper and will be investigated in our future research.

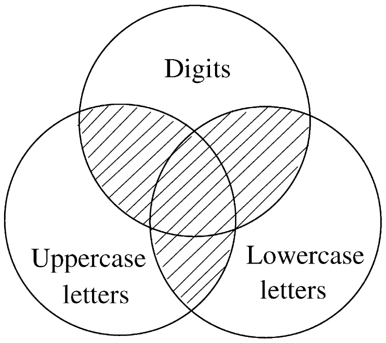


Fig. 30. Relationships between models formed from different character sets.

8. Conclusion

Nowadays, relatively few researchers use the structural approach for character recognition. Our experiment shows that, by making use of structural information contained in a character together with a simple, flexible matching mechanism and some additional post-processing procedures, we can indeed achieve fairly good recognition results. On the average, the recognition speed is about 7.5 characters per second running in Prolog on a Sun SPARC 10 Unix workstation and the memory requirement is reasonably low. In addition, our approach allows easy extensions to an existing system.

There are some more advantages with our approach:

1. Since our approach is a model-based one, all the patterns have semantically clear representations that can be used for subsequent manual verification.
2. Training is not necessary, though it may be introduced later to automate model construction possibly with some optimality criteria used. New models may be added any time, though some effort has to be put on resolving conflicts between the new models and some existing ones.

However, at this stage, model creation is not automatic yet. In other words, we still have to manually design the set of models in advance. Fortunately, automatic extraction of models from data is feasible in our scheme and it will be one of our future directions to pursue.

In summary, with this simple and robust structural approach, we already have an effective and efficient on-line character recognition module. This module will be used as part of a larger system, a pen-based mathematical equation editor [16], which is being developed by the authors using a syntactical pattern recognition approach and will be reported in depth separately.

Acknowledgements

This research work is supported in part by the Hong Kong Research Grants Council (RGC) under Competitive Earmarked Research Grant HKUST 746/96E awarded to the second author.

References

[1] C.C. Tappert, C.Y. Suen, T. Wakahara, The state of the art in on-line handwriting recognition, IEEE Trans. Pattern Anal. Machine Intell. 12 (8) (1990) 787–808.
[2] K.S. Fu, Syntactic Pattern Recognition and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[3] T. Pavlidis, Structural Pattern Recognition, Springer, New York, 1977.
[4] S. Lucas, E. Vidal, A. Amiri, S. Hanlon, J.C. Amengual, A comparison of syntactic and statistical techniques for off-line OCR, in: R.C. Carrasco, J. Oncina (Eds.), Grammatical Inference and Applications (ICGI-94), Springer, Berlin, September 1994, p. 168–179.
[5] C.C. Tappert, Speed, accuracy, and flexibility tradeoffs in on-line character recognition, in: P.S.P. Wang (Ed.), Character and Handwriting Recognition: Expanding Frontiers, World Scientific, Singapore, 1991, pp. 79–95.
[6] S.W. Lee, Y.J. Kim, A new type of recurrent neural network for handwritten character recognition, Proc. 3rd Int. Conf. on Document Analysis and Recognition, Montreal, Canada, 1995, pp. 38–41.
[7] T. Pavlidis, Structural descriptions and graph grammars, in Pictorial Information Systems, Springer, New York, 1980.
[8] A. Rosenfeld, Array, tree and graph grammars, in Bunke and Sanfeliu [18], chapter 4, pp. 85–115.
[9] A.C. Shaw, A formal picture description scheme as a basis for picture processing systems, Inform. Control 14 (1969) 9–52.
[10] J. Feder, Plex languages, Inform. Sci. 3 (1971) 225–241.
[11] H. Bunke, Hybrid pattern recognition methods, In Bunke and Sanfeliu [18], chapter 11, pp. 307–347.
[12] M. Berthod, J.P. Maroy, Learning in syntactic recognition of symbols drawn on a graphic tablet, Comput. Graphics Image Process. 9 (1979) 166–182.
[13] H. Freeman, Computer processing of line drawing images, ACM Comput. Surveys 6 (1) (1974) 57–98.
[14] P.S.P. Wang, A. Gupta, An improved structural approach for automated recognition of handprinted characters, in: P.S.P. Wang (Ed.), Character and Handwriting Recognition: Expanding Frontiers, World Scientific, Singapore, 1991, pp. 97–121.
[15] R. Kassel, A comparison of approaches to on-line handwritten character recognition. Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science, June 1995.
[16] K.F. Chan, D.Y. Yeung, Towards efficient structural analysis of mathematical expressions, in: A. Amin, D. Dori, P. Pudil, H. Freeman (Eds.), Advances in Pattern Recognition, Springer, Berlin, 1998, pp. 437–444. Joint IAPR

International Workshops, SSPR'98 and SPR'98, Sydney, Australia, 11–13 August 1998.
[17] P.S.P. Wang (Ed.), Character and Handwriting Recognition: Expanding Frontiers, World Scientific, Singapore, 1991.
[18] H. Bunke, A. Sanfeliu (Eds.), Syntactic and Structural Pattern Recognition – Theory and Applications, World Scientific, Singapore, 1990.

About the Author—KAM-FAI CHAN received a B.Sc. degree from the Radford University and an M.Sc. degree from the University of South Carolina, both in computer science. He is currently a Ph.D. candidate in the Department of Computer Science at the Hong Kong University of Science and Technology. His major research interests include pattern recognition, logic programming and Chinese computing.

About the Author—DIT-YAN YEUNG received his B. Sc. (Eng.) degree in electrical engineering and M. Phil. degree in computer science from the University of Hong Kong, and his Ph.D. degree in computer science from the University of Southern California in Los Angeles. From 1989 to 1990, he was an assistant professor at the Illinois Institute of Technology in Chicago. He is currently an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology. His current research interests are in the theory and applications of pattern recognition, machine learning, and neural networks. He frequently serves as a paper reviewer for a number of international journals and conferences, including *Pattern Recognition*, *Pattern Recognition Letters*, *IEEE Transactions of Pattern Analysis and Machine Intelligence*, *IEEE Transactions on Image Processing*, and *IEEE Transactions on Neural Networks*.