PERGAMON

Pattern Recognition 34 (2001) 1671–1684

PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY

www.elsevier.com/locate/patcog

Error detection, error correction and performance evaluation in on-line mathematical expression recognition

Kam-Fai Chan, Dit-Yan Yeung*

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 30 March 1999; received in revised form 22 June 2000; accepted 22 June 2000

Abstract

Automatic recognition of on-line mathematical expressions is difficult especially when there exist errors. In this paper, we incorporate an error detection and correction mechanism into a parser developed previously by us based on definite clause grammar (DCG). The resulting system can handle lexical, syntactic and some semantic errors. The recognition speed for 600 commonly seen expressions is quite acceptable, ranging from 0.73 to 6 s per expression on a modest workstation. In addition, we propose a performance evaluation scheme which can be used to demonstrate the effectiveness of both the symbol recognition and structural analysis stages by a single measure. © 2001 Pattern Recognition Society. Published by Elsevier Science Ltd. All rights reserved.

Keywords: Definite clause grammar; Error-correcting parsing; Error detection and correction; Hierarchical decomposition parsing; On-line mathematical expression recognition; Performance evaluation; Structural analysis

1. Introduction

With the recent advances in pen-based computing technologies, we already have all the necessary hardware to provide an input device for entering mathematical expressions into computers in a natural way, i.e., we simply write the expressions on an electronic tablet for the computer to recognize automatically. The key problem that remains is the automatic recognition of mathematical expressions, which is more on the software side.

Mathematical expressions are in general two-dimensionally structured patterns. Typically, they consist of special symbols and Greek letters in addition to English letters and digits. Moreover, characters and symbols may appear in various positions, possibly of different sizes. All these together make the recognition process very complicated even when all the individual characters and symbols can be recognized correctly.

However, in practice, we cannot expect that all the input expressions contain no errors. Error detection and correction are important steps, especially in parsing. A good error handler should report the presence of errors clearly and accurately. After recovering from each error, it should still be able to detect subsequent errors. At the same time, it should not significantly slow down the processing of correct expressions. Ideally some errors should even be corrected. Moreover, correct parts should not be treated mistakenly as errors and altered improperly.

With errors taken into consideration, different existing mathematical expression recognition systems may have very different performance. How should we rate a system as good? What criteria should we use to evaluate its performance?

Both symbol recognition and structural analysis have been extensively studied for decades. Mathematical expression recognition, which features both of them as the two major stages of the process, is a good subject for studying the integration of the two areas. However, the research area had not attracted too much attention in the past. It is only until recently that more researchers have started to pay more attention to this area. Detailed review of the literature can be found in Refs. [1,2].

* Corresponding author. Tel.: + 852-2358-6977; fax: + 852-2358-1477.

E-mail addresses: kchan@cs.ust.hk (K.-F. Chan), dyyeung@ cs.ust.hk (D.-Y. Yeung).

0031-3203/01/$20.00 © 2001 Pattern Recognition Society. Published by Elsevier Science Ltd. All rights reserved.
PII: S0031-3203(00)00102-3

In this paper, we will mainly focus on issues related to error detection, error correction and performance evaluation in on-line mathematical expression recognition. First, we will describe the different types of common errors that usually occur in mathematical expressions. Then, we will show how the errors can be detected and corrected. Afterwards, we will discuss some existing schemes and propose some new schemes for evaluating the performance of mathematical expression recognition systems. Finally, we will provide and discuss some experimental results which will then be followed by some concluding remarks.

2. An efficient syntactic approach to structural analysis of on-line handwritten mathematical expressions

Due to the complexity of mathematical expressions, systems used for recognizing them are better built based on some explicit grammar rules so that a clear and concise form is available for formal verification as well as subsequent extension. In addition, to ensure that a mathematical expression recognition system is useful in practice, its recognition speed is also an important factor to consider.

In Ref. [3], an approach, called hierarchical decomposition parsing, was proposed for obtaining the syntactic structures of mathematical expressions automatically. Hierarchical decomposition parsing is implemented in definite clause grammar (DCG), in terms of a set of replacement rules for parsing mathematical expressions. With DCG, we are not only able to define the replacement rules concisely, but their definitions are also in a readily executable form. Besides, the proposed method uses three major ideas, namely, left-factoring, binding symbol preprocessing, and hierarchical decomposition, to make parsing more efficient. Experiments done on some commonly seen error-free mathematical expressions show that the method can achieve quite satisfactory speedup.

In the following section, we will extend the parser developed by us in Ref. [3] to handle erroneous mathematical expressions as well. In particular, we will show how error detection and correction can be performed easily and effectively during parsing. Most importantly, all the procedures are clearly defined in the form of formal replacement rules.

3. Incorporating error detection and correction into mathematical expression recognition

Although error detection and correction are important steps, very few papers in the mathematical expression recognition literature have addressed these issues.

Dimitriadis et al. [4] put extra effort to detect and correct errors as, according to them, no attempts were made previously in this aspect. However, the error detection and correction methods used were quite simple. For example, some warning messages, such as “the root symbol should cover all of its terms”, may be given when the error is not fatal. However, some other errors, like “the function `tan` does not have arguments”, require the user to correct the input before the editor can proceed.

Lee and Wang [5] used some heuristic rules to correct recognition errors. For example, the expression “$x = 5in\ \theta$” will be converted to “$x = \sin\theta$” due to the similarity between ‘5’ and ‘s’. Other heuristic rules are also used, such as

- For every binary operator P, there must exist two operands that will generally be of the same typeface and size.
- There are no symbols in the subscript position of a numeral.
- Symbols in the same operand generally possess the same properties.

3.1. *Types of errors in handwritten mathematical expressions*

During the analysis of a mathematical expression, errors sometimes occur. In general, there are four types of errors:

- lexical errors (e.g., poorly written symbols),
- syntactic errors (e.g., an arithmetic expression with unbalanced parentheses),
- semantic errors (e.g., an operator applied to an incompatible operand), and
- logical errors (e.g., $1 + 1 = 3$).

Very often, error detection and correction are mostly centered around the parsing stage.

Quite a few strategies are available for correcting errors, such as panic-mode recovery, phrase-level recovery, error-correcting parsing, and global correction [6]. Panic-mode recovery is simple to use, but a considerable amount of input would be skipped without checking it for additional errors. Phrase-level recovery may replace part of the remaining input by some string to allow the parser to proceed when an error is detected. However, some misplaced strings may lead to infinite loops. Also, this method is not able to cope with situations in which the actual error has occurred before the point of detection. Theoretically, global correction can make as few changes as possible in processing an incorrect input string. The problem is that the time and space requirements of this approach are often too high for practical use. Although error-correcting parsing may lead to infinite loops too, the problem can be avoided by paying special attention when designing the grammar rules.

Hence, we decided to use error-correcting parsing techniques for detecting and correcting errors in the structural analysis stage.

3.2. Error-correcting parsing

The main idea of error-correcting parsing is to extend the grammar to include all the expected errors into its productions (i.e., grammar rules). As a result, the new grammar will cover not only the correct sentences, but also all the possible erroneous sentences that could occur. This method is applicable only when it is possible to anticipate the common errors that we may encounter. Since the domain of mathematical expressions in our research is well defined and limited, it is feasible to design an error-correcting parser for handling all the correct and incorrect expressions.

According to Ref. [7], there are three types of syntactic errors for string grammars, namely, substitution errors, deletion errors, and insertion errors. As a result, three types of transformation are suggested to correct the errors, as follows:

Let Σ be a finite alphabet and Σ^* be the closure of Σ. For two strings $x, y \in \Sigma^*$, a transformation is defined as a mapping $T: \Sigma^* \to \Sigma^*$ such that $y = T(x)$:

1. Substitution error transformation:

$$\omega_1 a \omega_2 \overset{T_S}{\vdash} \omega_1 b \omega_2 \quad \text{for all } a, b \in \Sigma,\ a \neq b.$$

2. Deletion error transformation:

$$\omega_1 a \omega_2 \overset{T_D}{\vdash} \omega_1 \omega_2 \quad \text{for all } a \in \Sigma.$$

3. Insertion error transformation:

$$\omega_1 \omega_2 \overset{T_I}{\vdash} \omega_1 a \omega_2 \quad \text{for all } a \in \Sigma,$$

where $\omega_1, \omega_2 \in \Sigma^*$.

When we incorporate the error-correcting mechanism into our existing parser, these types of transformation will be implicitly included into the grammar rules.

Error-correcting parsing is relatively easy to implement in our parser since we use DCG as the underlying formalism. Any extension added to the grammar can be put into use immediately without any extra effort.

3.3. Parsing with more hierarchical decomposition

The purpose of performing hierarchical decomposition is to partition an expression into sub-expressions. As a result, we can then parse all the sub-expressions separately. One advantage is to reduce the overhead of backtracking. Since the sub-expressions are much smaller in size when compared with the original expression, the time for backtracking can usually be greatly reduced. Another advantage is to maximize the structure obtained even when errors are encountered during parsing. As we know, parsing may fail entirely if the structure is not well-formed with respect to the grammar specified. In other words, nothing may be returned by the parser even when there exists only one simple error in the structure. By parsing all the sub-expressions separately, the effect can be kept local even when failure occurs. After obtaining all the sub-structures, we can then compose the final structure from a set of sub-structures. This is in the same spirit as the divide-and-conquer paradigm for problem solving.

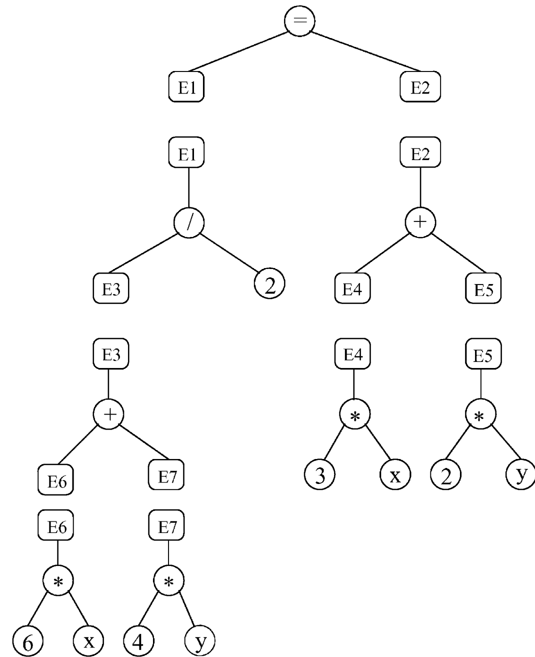


Fig. 1. Tree structures generated as a result of parsing with more hierarchical decomposition on the expression $(6x+4y)/2 = 3x+2y$.

The decomposition can often be applied hierarchically. For example, an expression can be divided into a list of smaller sub-expressions, and a smaller sub-expression can be further divided into a list of even smaller sub-expressions. Fig. 1 shows the resulting sub-structures for the expression $(6x+4y)/2 = 3x+2y$.

3.4. How to correct errors

Since the domain of mathematical expressions in our research is limited, it is, in principle, feasible to list all the possible errors and correct them one by one. In case some errors are overlooked, it is still possible and quite easy to

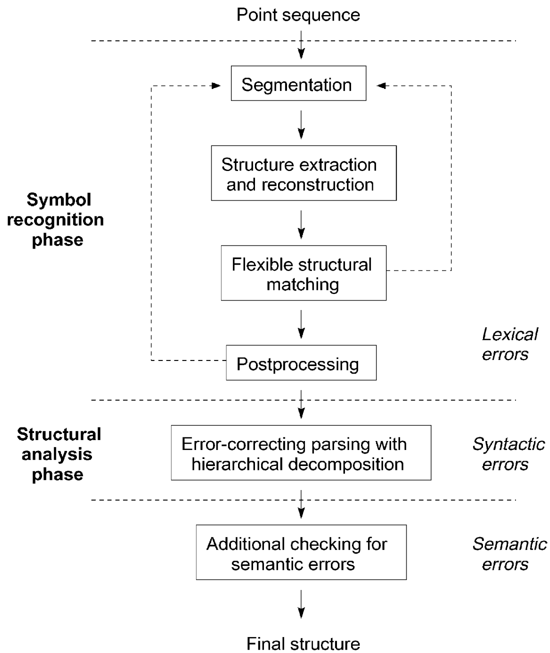


Fig. 2. Overview of the recognition process after incorporating the error detection and correction mechanism.

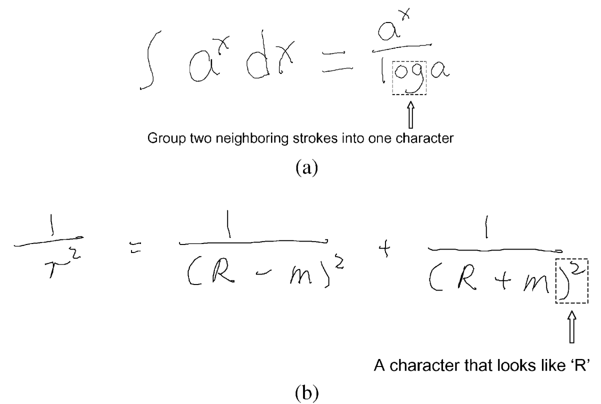


Fig. 3. Examples of lexical errors caused by poor segmentation: (a) resulting character does not look like any valid character, (b) resulting character looks like a valid but incorrect character.

make subsequent changes to the grammar due to the high extensibility of our DCG parser.

As mentioned in Section 3.1, there are mainly four types of errors, i.e., lexical errors, syntactic errors, semantic errors, and logical errors. Here, we do not attempt to tackle all of them. Instead, our focus is mainly on correcting lexical, syntactic and some semantic errors. How to correct logical errors will be left to our future work. Fig. 2 summarizes the recognition process after incorporating the error detection and correction mechanism. More details about the symbol recognition process can be found in Ref. [8].

3.4.1. Lexical errors

In general, there are two types of lexical errors. The first type is related to poor handwriting quality. It may be due to the writing style or the quality of the input device. Such errors are hard to correct. Even if we can do so, there is no guarantee that the "correction" is indeed correct.

Another type of lexical errors is often due to poor segmentation. In general, we are often required to segment all the symbols before the recognition stage. The most common action in segmentation is to group overlapping strokes together to form a symbol. However, we may sometimes be required to combine some completely separated strokes into a symbol. Theoretically, it is possible that a single character is incorrectly segmented into two or more units. This is unlikely to occur in our system, though, since the window size used for segmenting symbols is set to be bigger than the average size of all symbols in the expression.

When the writing is not neat enough, segmentation errors can easily occur. Such segmentation errors often lead to lexical errors. In other words, the character and symbol recognition module is not able to find a good match. However, in some cases, the resulting pattern may look like some other characters. Fig. 3 shows two such examples.

For the first case, we can simply repartition the character and perform the recognition process on the resulting characters. For the latter case, we have to add some postprocessing steps to distinguish between valid and invalid results. As shown in Fig. 3(b), the 'R' does not look like a normal one. By checking the difference in y-coordinate between the two endpoints of the strokes, we should have enough confidence to reject it. After rejection, it becomes an unmatched character. As a result, the same process we have just mentioned for the first case can also be applied.

3.4.2. Syntactic errors

One of the problems in parsing is that the whole process will fail if the input sentence is not well-formed according to the specified grammar. However, if we have pretty good ideas about the possible errors that may occur, we may simply extend the grammar to cover all possible correct and erroneous sentences. That is why we decided to use error-correcting parsing to detect and possibly correct errors in our system.

Syntactic errors may occur in different forms, for example, missing an argument after a function name, missing an operand, invalid implicit operator, and missing part of a binding or fence symbol. In the following, we

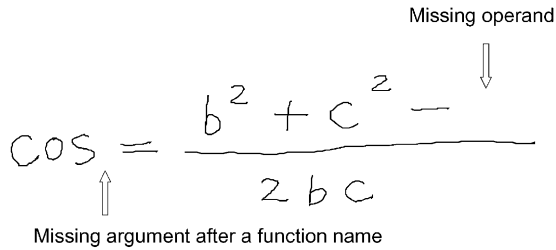


Fig. 4. An expression that has a missing argument after a function name and a missing operand.

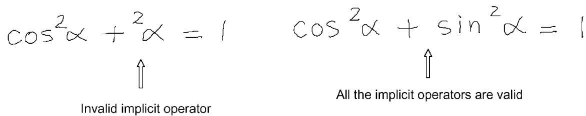


Fig. 5. Examples of expressions with and without an invalid implicit operator.

will discuss these errors in detail and propose ways to correct them.

Missing function arguments or operands: Sometimes we may forget to write some parts of an expression. If the missing part is the only element after a function name, in between two operators, or after the last operator, we will get syntactic errors. Fig. 4 shows an example with an argument and an operand missing.

To remedy this problem, we only need to add an *epsilon* symbol denoting an empty symbol into the structure.

Invalid implicit operators: One of the common characteristics of a typical mathematical expression is that it consists of some implicit operators. The common ones are for implicit multiplication, subscripting and exponentiation. Others are for the base of a function (e.g., $\log_2 N$) and the exponent of a function (e.g., $\cos^2 A$). However, not all the implicit operators can appear anywhere in an expression. Fig. 5 shows examples of some valid and invalid expressions.

Missing part of a binding or fence symbol: Binding symbols usually bind some other characters and symbols within their regions. What if nothing is found in the region? One simple example is an empty square root symbol, i.e., '$\sqrt{\quad}$'. In this case, we can simply treat the symbol just like a variable and report it in the output expression.

Fence symbols are often in pairs. If one of the two fence symbols forming a pair is missing, we will encounter a syntactic error. A simple way to recover from this kind of errors is to treat the unpaired fence symbol as a variable, just like the above case. However, in some cases, we may be able to correct the error by looking for some potentially misclassified characters or symbols. For example, when we find a single right parenthesis symbol without a corresponding left parenthesis, we can try to scan through the list of characters before it to find a possible left parenthesis that has been misclassified. If we can see some characters, like 'C' or 'L', there is a good chance that one character should in fact be the left parenthesis. Fig. 6 shows such an example.

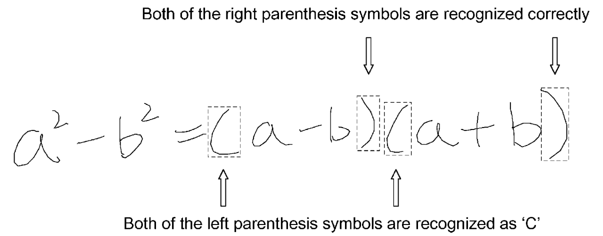


Fig. 6. An expression that contains two misclassified left parenthesis symbols.

This is certainly just a heuristic and does not guarantee to work in all cases. However, in practice, this method can correct many such errors.

3.4.3. Semantic errors

Mathematical expressions consist of characters and symbols. However, these elements cannot be put in arbitrary positions. Certain rules and conventions have to be followed. When some rules are violated, we may encounter ill-formed expressions.

With error-correcting parsing, we not only can parse correct expressions but can also parse incorrect expressions. In some cases, when we analyze the ill-formed structure, we may use those rules as heuristics to perform error correction after parsing. In the following, we will show some examples.

In general, we always write the constant coefficient before the variables. When we encounter an expression like "$y\ 1\ x$", we should have enough confidence to believe that the '1' in the expression should in fact be the division symbol '/'.

Some people do write the character 't' in a way that looks very similar to the symbol '$+$'. Hence, an expression like "$\tan\alpha$" may sometimes be recognized as "$+an\alpha$". Since the resulting structure of "$+an\alpha$" is rather unusual, we may choose to transform it back to "$\tan\alpha$".

Again, both of the above two cases use heuristics and have no guarantee that they will always work correctly. However, such heuristics do help to correct some erroneous expressions. Fig. 7 shows how error correction can be done for the two examples mentioned above.

4. Performance evaluation in mathematical expression recognition

It is only until recently that mathematical expression recognition has attracted more attention from the

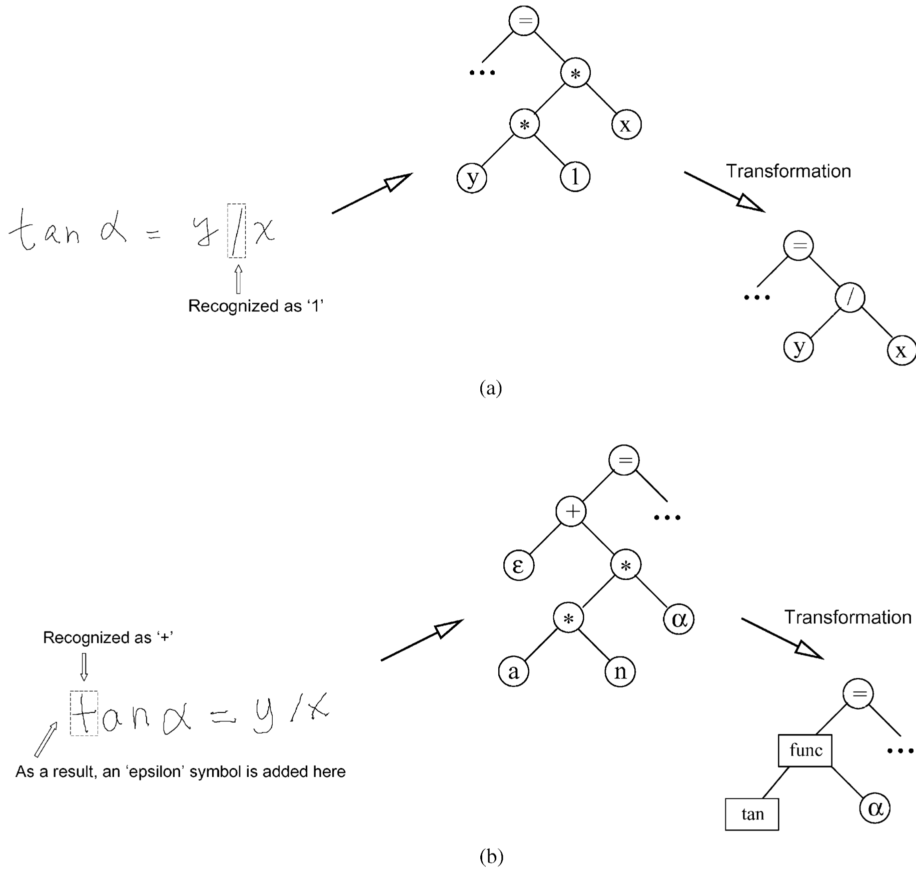


Fig. 7. Examples of error correction after parsing: (a) transform "y 1 x" into "y / x", (b) transform " + $an\alpha$" into "$\tan\alpha$".

research community. In the past, some researchers put their emphasis purely on the theoretical aspects without any experimental results reported. For those who did conduct experiments, their performance evaluation methods can roughly be grouped into three major categories:

1. Performing the test on a set of expressions and categorizing the results according to whether the expressions are correctly or incorrectly recognized [9].
2. Performing the test on a set of expressions and paying attention only to the symbol recognition rate [4,5,10–12].
3. Performing the test on some typical expressions [13,14]. Such expressions are usually written neatly by one or just a few writers. As a result, all the expressions can be recognized correctly.

4.1. Evaluation methods

Since mathematical expression recognition primarily consists of two major stages: symbol recognition and structural analysis, the evaluation methods are also based on these two stages.

4.1.1. Recognition of expressions

This method is simple and has been used by at least one researcher [9]. The recognition result will only fall into one of two categories, according to whether the expression is correctly or incorrectly recognized. The recognition rate is simply the ratio of the number of correctly recognized expressions to the total number of expressions tested:

$$R_e = \frac{\text{Number of correctly recognized expressions}}{\text{Total number of expressions tested}}.$$

However, with this method, a system that is able to recognize an expression with only one misclassified character will be treated the same as another system that is unable to recognize even one character for the same expression. Such limitation calls for measuring accuracy at a finer granularity.

4.1.2. Recognition of symbols

This is currently the most common method used to evaluate the performance of mathematical expression recognition systems. Focus is put only on the symbol recognition part. The recognition rate is the ratio of the number of correctly recognized symbols to the total number of symbols tested:

$$R_s = \frac{\text{Number of correctly recognized symbols}}{\text{Total number of symbols tested}}.$$

However, this method does not take the structural analysis performance into account, although structural analysis is known to be a crucial part of mathematical expression recognition. Hence, it again does not show the complete picture.

4.1.3. Recognition of operators

One of the potential applications of mathematical expression recognition is to add a natural interface to some existing mathematical software such as *Mathematica* and *Maple*. In order to achieve this purpose, we must be able to recognize not only all the characters and symbols in an expression, but also its structure. However, to the best of our knowledge, no scheme has so far been proposed to evaluate performance in this aspect.

In mathematical expressions, there exist two kinds of operators, i.e., explicit and implicit operators. Common operators, such as arithmetic operators, are explicit operators that are denoted by distinct symbols. Unlike explicit operators, implicit operators do not have any physical form, but are reflected only by some spatial relationships between neighboring characters and symbols (i.e., operands) affected, for example, implicit multiplication, subscripting and exponentiation.

It is very common to represent the structure of a mathematical expression in the form of a structure tree that has operators as its internal nodes and characters as its leaves. Recognition of the structure of an expression can be viewed as a process of transforming the expression into its corresponding structure tree. Fig. 8 shows an example.

The recognition rate here is the ratio of the number of correctly recognized operators to the total number of operators tested:

$$R_o = \frac{\text{Number of correctly recognized operators}}{\text{Total number of operators tested}}.$$

Sometimes, it is quite difficult to count the number of operators simply from the expression due to the possible existence of implicit operators. An easier alternative is to count the number of internal nodes in its corresponding structure tree.

During the recognition of mathematical expressions, problems from an earlier stage may affect a later stage. For example, when one of the characters in a function

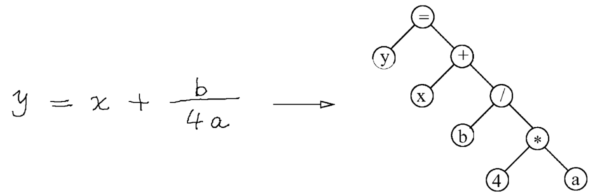


Fig. 8. Transforming an expression into its corresponding structure tree.

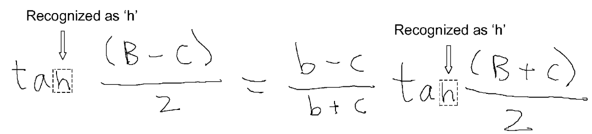


Fig. 9. One of the characters in a function name is misclassified.

name is misclassified, we have almost no way to construct such a function in the structure. Fig. 9 shows such an example.

Since the misclassification has been counted in the symbol recognition phase, it is unfair and unreasonable to doubly penalize by counting the subsequent errors caused by the initial misclassification. Hence, after the symbol recognition stage, whether the resulting structure is correct will depend on the results of symbol recognition. In the above example, when tan is misclassified as "t a h", we will treat those characters as "t a h" in the structural analysis phase and check if the resulting structure is consistent with this fact. In other words, the structure returned for "t a h" should usually be "$(t * a) * h$".

4.1.4. Integrated performance measure

Using two different recognition rates, i.e., separate recognition rates for symbols and operators for the evaluation of mathematical expression recognition systems, is somewhat troublesome. Moreover, both the symbol recognition and structural analysis stages are equally important. Hence, we propose to combine the two performance measures into a single integrated measure. The proposed integrated recognition rate is the ratio of the number of correctly recognized symbols and operators to the total number of symbols and operators tested:

$$R_i = \frac{\text{Number of correctly recognized symbols and operators}}{\text{Total number of symbols and operators tested}}.$$

In order to obtain high overall recognition rate, a system must be able to do both symbol recognition and structural analysis well.

(a)

(b)

(c)

Fig. 10. Examples of mathematical expressions collected from different writers: (a) two nicely written expressions, (b) two regularly written expressions, (c) a poorly written expression.

5. Experimental results and discussions

In this experiment, we perform tests on a number of different expressions which were extracted from a commonly referenced mathematical handbook [15]. Expressions are grouped into four domains, namely, elementary algebra, trigonometric functions, geometry and indefinite integrals. In each domain, we categorize the expressions into three different sizes, i.e, small, medium and large. Each size consists of five different expressions. There are 10 different writers. Totally, there are 600 expressions.

The writers were told to write the expressions in their usual style with the only condition that all the characters and symbols should be separated from each other. However, no constraints were imposed on the order of writing. In other words, a writer could even choose to write an expression backward as long as all the characters and symbols do not overlap each other.

As a result, most expressions turn out to be somewhat neat. However, due to some particular writing styles and unfamiliarity with the hardware, some expressions are not well written. One of the most common problems is that the center line of an expression is tilted. Fig. 10 shows some examples of the expressions collected.

Initially, the input is simply a sequence of points. After some segmentation steps, we then use the character recognition method proposed by us in Ref. [8]. All the characters and symbols recognized are represented as objects with associated attributes, including location, size, and identity. Note that the objects can be put in an arbitrary order for our subsequent processing.

The next step is to group the objects. Here we use a method similar to the one used in Refs. [16,17]. It mainly makes use of the idea of operator dominance [18]. For complex mathematical expressions, the techniques may have to be applied recursively. Afterwards, we perform hierarchical decomposition parsing along with some error detection and correction mechanism to obtain the final structure.

5.1. Detecting and correcting errors

The purpose of this experiment is to see how well we can perform error correction with the proposed method. Since it would become too tedious to report all cases here in detail, we decided to show only those typical ones with respect to their categories. They are listed as follows:

1. To correct lexical errors, we will focus on those with incorrect grouping of strokes.
2. To correct syntactic errors, we will cover those with part of the parenthesis symbols missing.
3. To correct semantic errors, we will report those with a ‘ + ’-like symbol in the function name ‘tan’.

Errors may occur in different forms. Some are very common while others may rarely be seen. Table 1 shows the frequencies of occurrence for some of the common error cases.

Almost one third of the expressions can be recovered by error-correcting parsing. About 5% of the symbols are segmented incorrectly. And very surprisingly, nearly half

Table 1
Frequencies of occurrence for some of the common error cases

Error case	Type of unit	Total number of units	Number of error cases	Error rate (%)
Parsing of mathematical expressions	Expression	600	196	32.67
Incorrect grouping of strokes	Character	11190	534	4.77
Missing part of parenthesis symbols	Pair of symbols	550	260	47.27
' + '-like symbol in tan	Function name	100	12	12.00

Table 2
Some results of the error correction process

Error case	Total number of error cases	Number of cases recovered or corrected	Recovery rate (%)
Parsing of mathematical expressions	196	196	100.00
Incorrect grouping of strokes	534	532	99.63
Missing part of parenthesis symbols	260	254	97.69
' + '-like symbol in tan	12	12	100.00

of the parenthesis symbols are not well written so that they are misclassified as some other characters and symbols. Also, it is not uncommon to have the 't' in tan written like a ' + ' symbol.

Our proposed methods for error detection and correction seem to be quite effective, at least in this experiment. As shown in Table 2, some of the error cases can be fully recovered while the recovery rates for others are above 97%.

5.2. *Different schemes for performance evaluation*

The purpose of this experiment is to explore how different schemes can effectively deliver the above results. In addition, we also show how different types of errors may occur in practice.

5.2.1. *Recognition of expressions*

Table 3 summarizes the recognition results for all expressions written by 10 different writers. There are only two possible results, correctly or incorrectly recognized.

Table 3
Recognition results in R_e for 10 different writers

	Recognition rate for each expression size			Overall recognition rate (R_e) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	100.00	100.00	100.00	100.00
Minimum	80.00	75.00	65.00	73.33
Average	95.50	92.00	78.50	88.67

Table 4
Recognition results in R_s for 10 different writers

	Symbol recognition rate for each expression size			Overall recognition rate (R_s) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	100.00	100.00	100.00	100.00
Minimum	98.92	98.53	97.81	99.24
Average	99.72	99.63	98.85	99.40

The mean is 88.67% and the medium is 89.17%, which are quite close to each other. On the average, only one error occurs in each expression and the highest number of errors in a single expression is four.

5.2.2. *Recognition of symbols*

Table 4 shows the recognition results of all symbols in the expressions written by 10 different writers. Notice that the result varies from writer to writer. The recognition rate ranges from almost 98–100%. Fig. 11 shows some examples of the misclassified cases.

Through analyzing some error cases, it is quite obvious that some characters are more likely to cause problems than others. The best example is the character 'C' (or 'c'). As we can easily notice, the difference between the appearances of 'C' and 'c' is very small, especially for handwritten ones. If we do not check their neighboring characters, it may not be possible to distinguish between them. In fact, it is easy for the writer to write them in an ambiguous style such that they cannot be distinguished easily, unless the writer pays special attention to the difference in size between the two characters when writing them. Fig. 12 shows such an example. The expression is supposed to be the following:

$$\tan\frac{(B-C)}{2}=\frac{b-c}{b+c}\tan\frac{(B+C)}{2}.$$

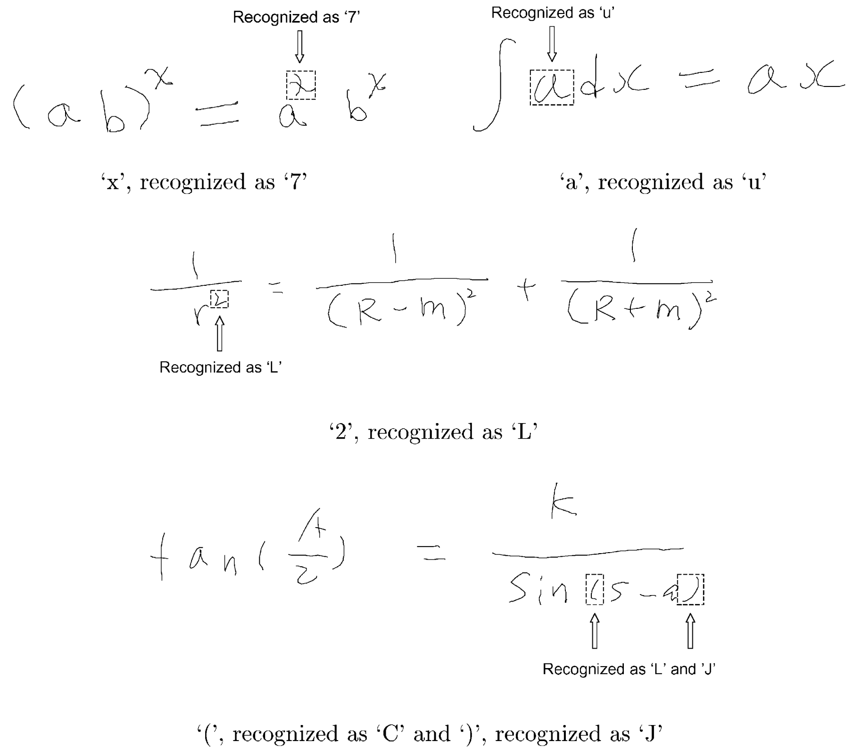


Fig. 11. Examples of some misclassified characters.

Fig. 12. An example illustrating that the characters 'C' and 'c' can easily become indistinguishable.

Recognized as 'H'

Recognized as 'H'

Fig. 13. Is it an 'H' or '1 + '?

However, all the C's look pretty much the same. As a result, we may have to treat them either as all C's, or all c's.

As shown in Section 3.4.1, some lexical errors caused by poor segmentation can be recovered. However, this is not always the case. Fig. 13 shows an example in which the recognized character itself is rather ambiguous.

One of the characteristics and hence difficulties of mathematical expressions is that characters and symbols can be of different sizes. One common way to preprocess them before recognition is to first normalize their size. While some characters looked all right before normalization, their appearance may seem to be different after the process has been applied even when the aspect ratio is kept the same as in our system. Fig. 14 shows one example in which '1' looks more like a left parenthesis symbol after normalization.

Fig. 14. The subscript '1' looks more like a left parenthesis symbol after normalization.

5.2.3. Recognition of operators

Table 5 shows the recognition results of all operators in the expressions written by 10 different writers. Note that the recognition rates are comparatively higher that those of symbol recognition. Fig. 15 shows some examples of misclassification in operator recognition.

In some cases, the writing direction may affect the correct recognition of operators. Fig. 16 shows an example in which misclassification of operators occurs due to the tilted writing direction. In that particular expression, the subexpression "$a^2 - x^2$" is misclassified as "a^{2-x^2}".

5.2.4. Integrated performance measure

Sometimes we may encounter errors in both symbol and operator recognition. Fig. 17 shows such an example.

Instead of having two different measures, we can simply use our integrated measure to evaluate the performance as follows:

$$R_i = \frac{26 + 16}{27 + 18} = \frac{42}{45} = 93.33\%.$$

Table 6 shows the overall recognition results of the expressions written by 10 different writers using the integrated performance measure defined above. Notice that the rates can effectively show similar distribution as those of symbol and operator recognition.

Table 5
Recognition results in R_o for 10 different writers

	Structural recognition rate for each expression size			Overall recognition rate (R_o) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	100.00	100.00	100.00	100.00
Minimum	98.33	99.50	98.67	98.83
Average	99.67	99.82	99.67	99.72

5.3. Recognition performance before error correction

In the previous section, we used four different evaluation schemes to show the recognition performance of our system. In particular, our proposed integrated performance measure is able to show the effectiveness of both the symbol recognition and structural analysis stages by a single measure.

Now, we will use the same measure to show the recognition performance before error detection and correction. Table 7 shows the overall recognition rates for 10 different writers when no error detection and correction are included.

As mentioned above, parsing may fail when there exists even only one simple error in the structure. If there is no error detection and correction, the recognition rate

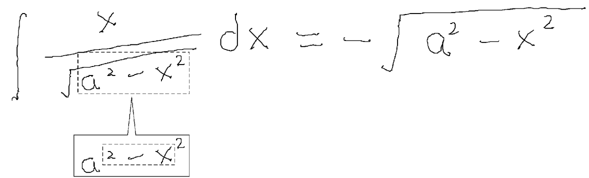
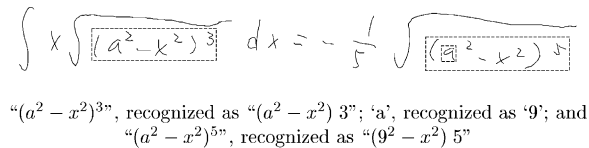
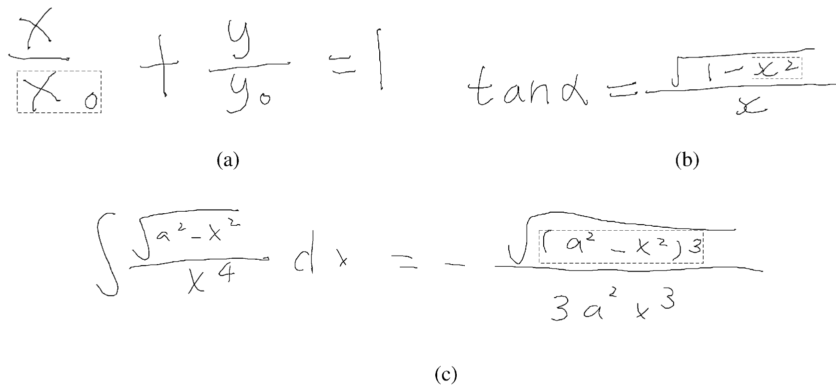


Fig. 16. "$a^2 - x^2$", recognized as "a^{2-x^2}".



"$(a^2 - x^2)^3$", recognized as "$(a^2 - x^2)$ 3"; 'a', recognized as '9'; and "$(a^2 - x^2)^5$", recognized as "$(9^2 - x^2)$ 5"

Fig. 17. An example with misclassification errors in both symbol and operator recognition.



(a) (b) (c)

Fig. 15. Examples of some misclassification cases in operator recognition: (a) "x_0", recognized as "$x\,0$", (b) "x^2", recognized as "$x\,2$", (c) "$(a^2 - x^2)^3$", recognized as "$(a^2 - x^2)$ 3".

will then become zero whenever syntactic errors occur. Hence, some of the rates shown are indeed very low.

Actually, we may perform error detection without subsequent correction. The main idea is to replace the erroneous part by other legal token to allow the parser to proceed. For example, we can convert an unmatched left parenthesis and treat it as a variable so that the parsing process can be completed. However, for this experiment, we will not perform other error correction techniques mentioned in Section 3.4, such as repartitioning the unclassified symbol in order to find some possible matches, looking for missing part of fence symbols, or checking the resulting structure after parsing for possible error correction. Table 8 shows the overall recognition rates for 10 different writers when error recovery without correction is performed.

Note that error recovery alone can already increase the recognition performance quite significantly. The main reason is that partial structures may be returned even with the erroneous cases. As the average number of errors in each expression is not high, the resulting recognition rate is often above 80%. As a result, the overall recognition performance with error recovery is much better. When using more error correction techniques, the recognition performance is then further improved.

Table 6
Overall recognition rates in R_i for 10 different writers

	Overall recognition rate for each expression size			Overall recognition rate (R_i) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	100.00	100.00	100.00	100.00
Minimum	98.73	98.87	98.19	98.60
Average	99.70	99.70	99.14	99.51

Table 7
Overall recognition rates in R_i for 10 different writers when there are no error detection and correction

	Overall recognition rate for each expression size			Overall recognition rate (R_i) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	97.97	88.64	72.61	84.94
Minimum	82.24	42.86	19.70	54.19
Average	90.94	69.02	45.93	68.63

5.4. Recognition speed

In order to demonstrate the potential of our system for practical use, we tabulate the time taken for recognizing expressions of different sizes in different domains. Our recognition system implemented in Prolog runs on a Sun SPARC 10 workstation. The timer starts after the sequence of points is read by the system and ends when the final structure is returned. In other words, the time taken includes segmentation, symbol recognition, grouping of symbols, parsing and error correction. Table 9 summarizes the result.

Table 8
Overall recognition rates in R_i for 10 different writers when error recovery without correction is performed

	Overall recognition rate for each expression size			Overall recognition rate (R_i) (%)
	Small (%)	Medium (%)	Large (%)	
Maximum	99.72	98.09	97.84	98.07
Minimum	94.71	90.35	89.69	91.64
Average	97.48	95.18	93.19	95.29

Table 9
Time required for recognizing different expressions

Expression domain	Time in seconds required for recognizing mathematical expressions								
	Small size			Medium size			Large size		
	Min.	Med.	Max.	Min.	Med.	Max.	Min.	Med.	Max.
Elementary algebra	0.73	1.11	1.83	1.03	1.60	2.35	2.07	3.46	6.00
Trigonometric functions	0.80	1.15	1.70	1.22	1.99	2.87	1.92	2.71	3.98
Geometry	0.75	1.05	1.42	1.17	1.79	2.53	1.67	2.62	3.70
Indefinite integrals	0.60	1.12	1.93	1.42	2.08	2.70	2.30	2.96	4.37

5.5. Discussions

After recognizing 600 expressions in the experiment, we summarized the results with four different schemes above. Here are some observations:

1. The writing quality differs very much from writer to writer. Apparently, it is easier to achieve high recognition rates with neat writing. Hence, for more extensive evaluation, we should try to perform the tests on as many writers as possible.
2. The longer the expression is, the lower the overall recognition rate becomes. This is shown in Table 3 as small expressions get the highest recognition rates. The same pattern is generally also found in Tables 4–6, especially if we consider the number of perfect classification cases in each class. At the symbol recognition level, the observation makes sense when we consider that the number of characters and symbols in longer expressions is much more than that of shorter expressions. As a result, the chance to have misclassified cases is also getting bigger. Although we do normalize it by the total number of characters and symbols in the expression to reduce the effect, the trend for the longer expressions to get lower recognition rates still exists. At the structural analysis level, it is speculated that people tend to write more neatly when the expression is short.

6. Conclusion

Recognition of handwritten mathematical expressions is not trivial as characters and symbols of different sizes and with subtle spatial relationships are often found in such expressions. The task becomes even more difficult when there exist errors.

In this paper, we incorporate some error detection and correction mechanism into an existing parser. It helps us to minimize the chance of parsing failure for erroneous input and at the same time increase the overall accuracy. Besides handling some lexical and syntactic errors, our system, in some cases, can also perform error correction after parsing to correct semantic errors. As a result, it can achieve fairly high recognition rates on some neatly written expressions. At the same time, the recognition speed for 600 commonly seen expressions is also quite acceptable, ranging from 0.73 to 6 s on a modest workstation by today's standard.

In addition, we propose a simple performance evaluation scheme which can show the effectiveness of both the symbol recognition and structural analysis stages by a single measure.

Although our current system only works on on-line mathematical expressions, it is not hard to modify the system to handle off-line data as well. However, some issues in mathematical expression recognition have not yet been addressed in this paper, such as resolving ambiguities, using more contextual information in error detection and correction, handling logical errors, etc. These are topics of our future research.

Acknowledgements

This research work is supported in part by the Hong Kong Research Grants Council (RGC) under Competitive Earmarked Research Grants HKUST 746/96E and HKUST 6081/97E awarded to the second author.

References

[1] D. Blostein, A. Grbavec, Recognition of mathematical notation, in: H. Bunke, P. Wang (Eds.), Handbook of Character Recognition and Document Image Analysis, World Scientific, Singapore, 1997, pp. 557–582.

[2] K.F. Chan, D.Y. Yeung, Mathematical expression recognition: a survey, Int. J. Document Anal. Recognition, to appear. An earlier version of the paper can be obtained at the following URL: `ftp://ftp.cs.ust.hk/pub/-techreport/99/tr99-04.pdf`.

[3] K.F. Chan, D.Y. Yeung, An efficient syntactic approach to structural analysis of on-line handwritten mathematical expressions, Pattern Recognition 33 (3) (2000) 375–384.

[4] Y.A. Dimitriadis, J.L. Coronado, Towards an ART based mathematical editor, that uses on-line handwritten symbol recognition, Pattern Recognition 28 (6) (1995) 807–822.

[5] H.-J. Lee, J.-S. Wang, Design of a mathematical expression recognition system, Pattern Recognition Lett. 18 (1997) 289–298.

[6] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[7] K.-S. Fu, Syntactic pattern recognition, in: T.Y. Young, K.-S. Fu (Eds.), Handbook of Pattern Recognition and Image Processing, Vol. 1, Academic Press, San Diego, 1986, pp. 85–117 (Chapter 4).

[8] K.F. Chan, D.Y. Yeung, Recognizing on-line handwritten alphanumeric characters through flexible structural matching, Pattern Recognition 32 (7) (1999) 1099–1114.

[9] A. Beláid, J.-P. Haton, A syntactic approach for handwritten mathematical formula recognition, IEEE Trans. Pattern Anal. Mach. Intell. 6 (1) (1984) 105–111.

[10] L.H. Chen, P.Y. Yin, A system for on-line recognition of handwritten mathematical expressions, Comput. Process. Chinese Oriental Languages 6 (1) (1992) 19–39.

[11] H.-J. Lee, M.-C. Lee, Understanding mathematical expressions using procedure-oriented transformation, Pattern Recognition 27 (3) (1994) 447–457.

[12] A. Kosmala, G. Rigoll, On-line handwritten formula recognition using statistical methods, Proceedings of the Fourteenth International Conference Pattern Recognition, Brisbane, Australia, 1998, pp. 1306–1308.

[13] M. Okamoto, B. Miao, Recognition of mathematical expressions by using the layout structures of symbols,

Proceedings of the First International Conference on Document Analysis and Recognition, Saint-Malo, France, 1991, pp. 242–250.

[14] H.M. Twaakyondo, M. Okamoto, Structure analysis and recognition of mathematical expressions, Proceedings of the Third International Conference on Document Analysis and Recognition, Montreal, Canada, 1995, pp. 430–437.

[15] D. Zwillinger (Ed.), CRC Standard Mathematical Tables and Formulae, 30th Edition, CRC Press, Boca Raton, 1996.

[16] M. Okamoto, A. Miyazawa, An experimental implementation of a document recognition system for papers containing mathematical expressions, in: H.S. Baird, H. Bunke, K. Yamamoto (Eds.), Structured Document Image Analysis, Springer, Berlin, 1992, pp. 36–53.

[17] J. Ha, R.M. Haralick, I.T. Phillips, Understanding mathematical expressions from document images, Proceedings of the Third International Conference on Document Analysis and Recognition, Montreal, Canada, 1995, pp. 956–959.

[18] S.K. Chang, A method for the structural analysis of 2-D mathematical expressions, Inform. Sci. 2 (3) (1970) 253–272.

About the Author—KAM-FAI CHAN received his B.Sc. degree from Radford University, M.Sc. degree from the University of South Carolina, and Ph.D. degree from the Hong Kong University of Science and Technology, all in Computer Science. He is currently a postdoctoral research associate 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 on Pattern Analysis and Machine Intelligence*, *IEEE Transactions on Image Processing*, and *IEEE Transactions on Neural Networks*.