Available online at www.sciencedirect.com
ScienceDirect
Computers & Graphics 31 (2007) 554–567
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

An efficient graph-based recognizer for hand-drawn symbols

WeeSan Lee[a], Levent Burak Kara[b], Thomas F. Stahovich[c,*]

[a]Computer Science Department, University of California, Riverside, CA 92521, USA
[b]Mechanical Engineering Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA
[c]Mechanical Engineering Department, University of California, Riverside, CA 92521, USA

Abstract

We describe a trainable, multi-stroke symbol recognizer for pen-based user interfaces. The approach is insensitive to orientation, non-uniform scaling, and drawing order. Symbols are represented internally as attributed relational graphs describing both the geometry and topology of the symbols. Symbol definitions are statistical models, which makes the approach robust to variations common in hand-drawn shapes. Symbol recognition requires finding the definition symbol whose attributed relational graph best matches that of the unknown symbol. Much of the power of the approach derives from the particular set of attributes used, and our metrics for measuring similarity between graphs. One challenge addressed in the current work is how to perform the graph matching efficiently. We present five approximate matching techniques: stochastic matching, which is based on stochastic search; error-driven matching, which uses local matching errors to drive the solution to an optimal match; greedy matching, which uses greedy search; hybrid matching, which uses exhaustive search for small problems and stochastic matching for larger ones; and sort matching, which relies on geometric information to accelerate the matching. Finally, we present the results of a user study, and discuss the tradeoffs between the various matching techniques.
© 2007 Elsevier Ltd. All rights reserved.

Keywords: Pen computing; Sketch understanding; Symbol recognition; Pattern recognition; Graph matching; Graph isomorphism

1. Introduction

Researchers have developed a variety of approaches for recognizing hand-drawn shapes and symbols. However, many of the current approaches have important limitations. For example, some methods are limited to single-stroke shapes drawn in preferred orientations [1]. Others consider only aggregate properties of a shape and can confuse dissimilar shapes that have similar aggregate properties [1,2]. Other approaches require shapes to be drawn with a consistent pen stroke order [3].

Our work is aimed at overcoming some of these limitations. Our goal is to create an efficient, trainable, multi-stroke symbol recognizer that is insensitive to orientation, scaling, and drawing order. This is achieved via a graphical representation. Specifically, a symbol is represented with an attributed relational graph (ARG) describing its geometry and topology. The nodes in the graph represent the geometric primitives, and the edges represent the geometric relationships between them. Representing a symbol in terms of its topology allows us to achieve invariance to rotation, and uniform and non-uniform scaling, including cases in which different parts of the same symbol are scaled differently. Because of the later capability, our approach is particularly tolerant of large variations in the shape of a hand-drawn symbol. This is an important advantage over approaches such as those presented in [1,4,5].

Symbol definitions are statistical models learned from training examples of a symbol. Using statistical descriptions makes the approach robust to variations common in hand-drawn shapes. Symbol recognition involves finding the definition symbol whose ARG best matches that of the unknown symbol. Much of the power of our approach derives from the particular set of attributes encoded in the ARGs, and our metrics for measuring similarity between two ARGs.

*Corresponding author. Tel.: +1 951 827 7719; fax: +1 951 827 2899.
E-mail address: stahov@engr.ucr.edu (T.F. Stahovich).

With our approach, symbol recognition is a graph matching or "graph isomorphism detection" [6] problem. The general problem of subgraph isomorphism detection [7,8] is known to be NP-complete [9]. Here, the problem is made more difficult because of noise that comes from variations in how the symbols are drawn as well as from processing errors. For example, it is not uncommon for a symbol to have extra or missing geometric primitives, and thus extra or missing nodes in its ARG.

There has been considerable research in developing efficient graph-matching techniques for a variety of applications [10]. Here we present and evaluate five techniques specifically designed for recognizing hand-drawn shapes. These techniques are designed to be efficient enough for interactive performance, and to be tolerant of the noise inherent in hand-drawn symbols. Furthermore, some of these techniques are able to take advantage of consistent drawing order to achieve higher efficiency.

Our recognizer assumes that the individual symbols in a sketch have been located prior to recognition. In other work, we have developed sketch parsers for locating symbols [11,12].

This work entails several important technical contributions including: a graph-based representation suitable for describing hand-drawn symbols, a method for measuring similarity between symbols, a modified probability density function (MPDF) for making statistical comparisons that are insensitive to the minor variations inherent in hand-drawn symbols, a training technique that is insensitive to drawing order, and a set of efficient graph-matching techniques for symbol classification.

The next section places this work in context by describing related work. This is followed by the details of our approach. Finally, results of a user study and conclusions are presented.

2. Related work

Symbol recognition is an active area of research. An extensive overview of the literature can be found in [13]. Here, we present a representative sample of the literature.

Lee [14] presents a graph-based recognizer in which the graph represents the precise geometry of the object. The approach is suitable for precisely drawn symbols with uniform scaling. For example, the approach has been used to recognize machine drawn symbols, symbols drawn using templates, and precise hand-drawn symbols. Lee's approach requires manual selection of key vertices during training.

Mahoney and Fromherz [15] present a technique for matching models of curvilinear configurations to hand-drawn sketches. The technique is intended for matching articulated figures rather than sketches with a more fixed shape. The approach has been implemented only for human stick figures and requires hand-coded shape definitions.

Rocha [16] uses graph matching for optical character recognition (OCR). The matching technique assumes that the characters have a fixed orientation and no disconnected parts. The approach relies on hand-coded definitions and has not been applied to hand-drawn shapes.

Lladós et al. [17] present an error-tolerant subgraph-matching technique for matching region adjacency graphs. The technique is intended for recognizing symbols composed of adjacent polygonal regions, rather than general arrangements of low-level primitives.

In previous work, we developed a graph-based approach that considered both topology and geometry. A definition was an average graph, with each attribute represented by a single value. In our current work, attributes are described statistically, making our approach significantly more robust to pen stroke segmentation errors and drawing variations. Likewise, our previous approach was incapable of representing symbols with disconnected parts, while the new approach can. Furthermore, to achieve interactive performance, our previous approach required the user to maintain a consistent drawing order. The system did have a mode that allowed for variable drawing order, but it was considerably less efficient than our current approach. Similarly, our previous approach required the training examples to have a consistent drawing order and pen-stroke direction, and was intolerant of segmentation errors. Our current approach overcomes these limitations.

In addition to symbol recognition, graph-based techniques have been used for a variety of other pattern recognition problems. Conte et al. [10] provides an extensive overview of graph-matching techniques and their applications. According to the taxonomy presented there, our stochastic, error-driven, and greedy matching techniques can be considered approximate matching techniques based on continuous optimization.

Many existing approaches to symbol recognition rely on feature-based representations. Fonseca et al. [2] use features such as the smallest convex hull that can be circumscribed around the shape, the largest triangle that can be inscribed in the hull, and the largest quadrilateral that can be inscribed. Because their classification relies on aggregate features of the pen strokes, it might be difficult to differentiate between similar shapes. Rubine [1] describes a trainable gesture recognizer designed for gesture-based interfaces. The recognizer is applicable only to single-stroke symbols and is sensitive to the drawing direction and orientation. Pereira et al. [18] have extended Rubine's method to multi-stroke symbols. However, such symbols must be drawn with a consistent set of strokes. Additionally, they have developed a graph-based symbol recognizer, but it is not trainable. Matsakis [19] describes a system for converting handwritten mathematical expressions into a machine-interpretable typesetting command language. Each symbol requires a multitude of training examples, where each example must be preprocessed to eliminate variations in drawing directions and stroke orderings.

However, the preprocessing makes their approach sensitive to rotations. Gennari et al. [12] describe a trainable recognizer that uses nine geometric features to construct concise probabilistic models of input symbols. The approach is suitable for multi-stroke symbols with arbitrary drawing orders and orientations. However, the features are an abstraction of the topology, and it is possible for shapes with different topologies to have the same features. Hse and Newton [4] developed a particularly accurate recognizer based on Zernike moments, which provide a rotation invariant representation. However, a preprocessing step in which the image size is normalized makes the approach sensitive to orientation. The preprocessing results in insensitivity to some forms of non-uniform scaling, but not cases in which different parts of a symbol are scaled differently (e.g., the circle in a pivot is drawn large, while the triangle is drawn small).

In addition to graph-based and feature-based methods, researchers have also explored a variety of other representations and approaches. For example, Sezgin and Davis [3] present a technique based on hidden Markov models. The approach requires shapes to be drawn with a consistent pen stroke ordering. Hammond and Davis [20] developed a recognizer that relies on hand-coded shape descriptions. Their representation is similar to ours in that both contain topological information. However, their descriptions are hand-coded while ours are learned from training examples. In later work [21], they extended their approach to use hand-drawn and machine-generated examples to assist the developer in interactively creating shape descriptions. Shilman et al. [22] present a sketch recognition approach that requires a manually encoded visual grammar. A large corpus of training examples is used to learn the statistical distributions of the geometric parameters used in the grammar, resulting in a statistical model. Composite objects are defined hierarchically in terms of lower-level, single-stroke symbols, which are recognized using Rubine's method [1]. Gross' [23] approach relies on a 3×3 grid inscribed in the symbol's bounding box. The sequence of grid cells visited by the pen distinguishes each symbol. Because of the coarse resolution of a 3×3 grid, this approach may not be able to handle symbols with small features. Kara and Stahovich [5] developed a recognizer based on a bitmap representation. One advantage of the approach is that it is tolerant of over-stroking and variations in line styles. However, the approach is sensitive to non-uniform scaling.

Parametric methods such as polygon, B-spline, and Bezier curve fitting techniques have also been considered in shape representation and classification [24,25]. A benefit of these approaches is that there is no need to segment the pen stroke into geometric primitives such as lines and arcs. Additionally, since only a few parameters are needed for shape description, these methods are computationally efficient. Similar to Rubine's method, however, these methods are primarily applicable to single-stroke symbols or gestural commands.

3. Representation

We represent a symbol with an attributed relational graph (ARG) describing its geometry and topology. The nodes in the graph represent the geometric primitives, and the edges represent the geometric relationships between them.

Each node is characterized by the type of the primitive—line or arc—and its relative length. The primitives are obtained from the raw pen strokes via a speed-based pen stroke segmenter [26]. The relative length of a primitive is defined as the ratio of its length (in pixels) to the total length of the primitives comprising the symbol. For example, each of the four line segments in a perfect square would have a relative length of 0.25. Defining length on a relative basis results in a scale-independent recognizer.

The edges in a graph represent the geometric relationships between the primitives. Each pair of primitives is characterized by the number of intersections between them, the relative locations of the intersections, and for lines, the angle of intersection. When extracting intersections from a sketch, a tolerance of 10% of the length of the primitives is used to allow for cases in which an intersection was intended but one of the primitives was a little too short. Intersection locations are measured relative to the lengths of the two primitives. For example, if the beginning of one line segment intersects the middle of another, the location is described by the coordinates (0%, 50%). The intersection angle is defined as the acute angle between two line segments. It is defined for both intersecting and non-intersecting line segments. Defining an intersection angle for non-intersecting segments allows the program to represent the topology of disconnected symbols, such as the dashpot in Fig. 12. Intersection angle is undefined for an intersection between an arc and another primitive.

Fig. 1 shows an example of an ARG for an ideal square. Each side of the square has a relative length of 0.25 and intersects two other sides with an intersection angle of 90°. Because of the drawing directions used in this example, all intersections are located at the end of one line segment and the beginning of another.

A definition for a symbol is created by constructing an "average" ARG from a set of training examples. (Additional details of the training process are described in Section 6.) The number of nodes in a definition is taken to be the most frequently occurring number of nodes in the training examples. Each node in the definition is assigned the primitive type that occurred most frequently for that node in the training data. The number of intersections assigned to a pair of primitives is determined in an analogous fashion. A pair of primitives is assigned two intersections if at least 70% of the examples had two. If less than 70% had two intersections, but there was at least one intersection 70% of the time, the pair is assigned one. Otherwise, the pair is assigned zero intersections. The remaining properties of the ARG—relative length, intersection angle, and intersection location—are continuous-valued properties.

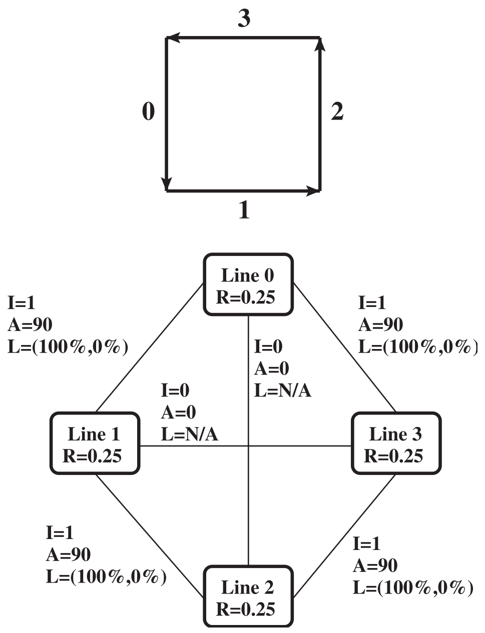


Fig. 1. Top: an ideal square drawn with a single, counterclockwise pen stroke. Arrows show the direction of drawing. Bottom: the corresponding ARG. *I*; number of intersections; *A*, intersection angle; *L*, intersection location; *R*, relative length.

These are characterized by the means and standard deviations of the values from the training examples.

4. Measuring similarity

During recognition, it is necessary to compare the ARG of the unknown symbol to that of each definition symbol to find the best match, and hence the classification of the unknown. The match between an unknown and a definition is quantified in terms of a dissimilarity score, which is computed using an ensemble of error metrics that consider both the intrinsic properties of the geometric primitives and the relationships between them. The former are encoded in the nodes of the ARG, the latter in the edges.

Table 1 lists our six error metrics and the weights applied to them when computing the dissimilarity score. The weights, which are based on empirical studies, reflect the relative importance of the various error metrics for discriminating between symbols. For the purposes of recognition, the dissimilarity score is converted to a *similarity score* in the obvious way:

$$\textit{similarity score} = 1 - \sum_{i=1}^{6} w_i E_i, \tag{1}$$

where the E_i are the error metrics and the w_i are the weights listed in Table 1.

The error metrics for relative length, intersection angle, and intersection location involve comparing properties of the unknown symbol to distributions of those properties encoded in a definition. For example, it is necessary to compare the relative length of each primitive in the unknown to the mean and standard deviation of the relative length of the corresponding primitive in the definition. Ordinarily, this is done with a Gaussian probability density function. As an alternative, we have developed a modified probability density function (MPDF) that is better suited to our recognition task:

$$P(x) = \exp\left[-\frac{1}{50.0}\cdot\frac{(x-\mu)^4}{\sigma^4}\right]. \tag{2}$$

Here, μ and σ are the mean and standard deviation of the features from the training examples. This function was designed empirically such that its top is flatter than the Gaussian probability density function for the same μ and σ. This makes it easier to detect matches that are in the "vicinity" of the definition. Additionally, we have found that the Gaussian distribution dies off too quickly towards its tails, which decreases its usefulness for recognition. For comparison, Fig. 2 shows both the Gaussian probability density function and our MPDF for $\mu = 0$ and $\sigma = 1$.

The six error metrics used for computing the similarity score are described in the following sections. Here we use the term "unknown" to refer to the symbol to be recognized, or equivalently, the ARG of that symbol. Likewise, the term "definition" refers to the ARG of a definition symbol. Note also that each metric is normalized

Table 1
Error metrics and corresponding weights

Error metrics (E_i)	Weight (w_i) (%)
E_1: primitive count error	20
E_2: primitive type error	20
E_3: relative length error	20
E_4: number of intersections error	15
E_5: intersection angle error	15
E_6: intersection location error	10

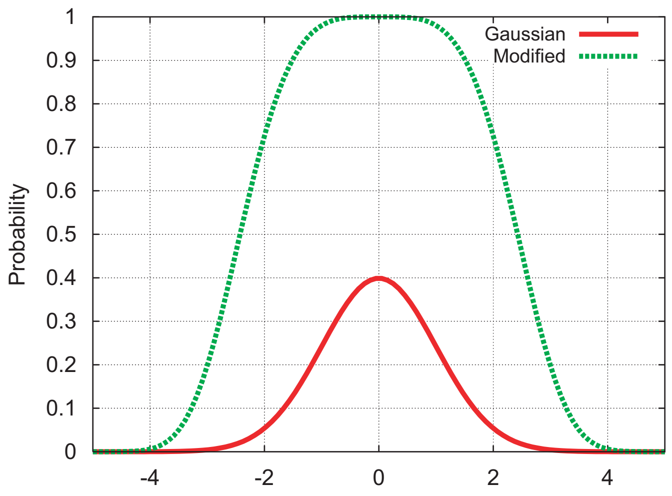


Fig. 2. Gaussian probability density function and modified probability density function for $\mu = 0$ and $\sigma = 1$.

to the range $[0,1]$ so that the weights in Table 1 have predictable influences.

4.1. Primitive count error

The primitive count error is the difference between the number of nodes in the unknown and definition ARGs, normalized by the minimum number of nodes:

$$E_1 = \min\left(1.0, \frac{|N_U - N_D|}{N_{min}}\right). \quad (3)$$

Here, N_U and N_D are the numbers of nodes in the unknown and definition ARGs, respectively, and $N_{min} = \min(N_U, N_D)$. We normalize by N_{min} to quantify the significance of the mismatch in the primitive count. The fewer primitives there are, the more significant a given mismatch is. The error saturates at one so that all errors have the same range of $[0, 1]$.

4.2. Primitive type error

The primitive type error is the number of node pairs with mismatched types, normalized by the minimum number of nodes:

$$E_2 = \frac{\sum_{i=1}^{N_{min}}[1 - \delta(Type(U_i), Type(D_i))]}{N_{min}}. \quad (4)$$

Here, U_i is a node from the unknown, D_i is the corresponding node from the definition, $Type(X)$ is a function that returns the primitive type (arc or line) of node X, and $\delta(p,q)$ is one when $p = q$, and zero otherwise.

4.3. Relative length error

Each primitive from the unknown should have a relative length similar to that of the corresponding primitive from the definition. If not, an error is assigned. Here, similarity is measured using the MPDF defined in Eq. (2). The error is computed as

$$E_3 = \frac{\sum_{i=1}^{N_{min}}[1 - P(R(U_i))]}{N_{min}}, \quad (5)$$

where $R(U_i)$ is the relative length encoded in node U_i of the unknown ARG, and $P(x)$ is evaluated using the mean and standard deviation from the corresponding node in the definition. Note that whereas $P(x)$ is the probability of match, $1 - P(x)$ is the probability of mismatch.

4.4. Number of intersections error

A pair of primitives in the unknown should have the same number of intersections as the corresponding pair in the definition. If not, an error is assigned. The total error is computed as

$$E'_4 = \frac{\sum_{i=1}^{N_{min}-1}\sum_{j=i+1}^{N_{min}}|I(U_i, U_j) - I(D_i, D_j)|}{\min(M_U, M_D)}, \quad (6)$$

where $I(X, Y)$ is the number of intersections between the primitives in nodes X and Y, and M_U and M_D are the numbers of edges in the unknown and definition ARGs, respectively. This error is normalized by the number of potentially intersecting pairs of primitives. However, because a pair of primitives can intersect as many as two times, E'_4 has a range of $[0, 2]$. So that all error metrics have the same range of $[0, 1]$, the value of E'_4 is "squashed" with

$$S(x) = \frac{1}{1 + \exp[6(1 - x)]}. \quad (7)$$

This squash function, which is shown in Fig. 3, was chosen such that small differences are attenuated while larger ones are preserved. During our experiments, we found that this choice provides better performance compared to a linear squashing function. As a result, the "number of intersections error" is defined as

$$E_4 = S(E'_4). \quad (8)$$

4.5. Intersection angle error

The intersection angle of a pair of lines in the unknown should be similar to that of the corresponding pair of lines in the definition. (Intersection angle is defined only for pairs of lines.) If not, an error is assigned. Here, similarity is again measured using the MPDF defined in Eq. (2). The error is computed as the sum of the intersection angle errors normalized by the number of line pairs the unknown and definition have in common:

$$E_5 = \frac{\sum_{i=1}^{N_{min}-1}\sum_{j=i+1}^{N_{min}}[1 - P(A_{ij})]}{\sum_{i=1}^{N_{min}-1}\sum_{j=i+1}^{N_{min}} Lines(U_i, U_j, D_i, D_j)}. \quad (9)$$

Here, A_{ij} is the angle at which the primitive from node i of the unknown intersects the primitive from node j of the unknown. $P(A_{ij})$ is evaluated using the mean and standard

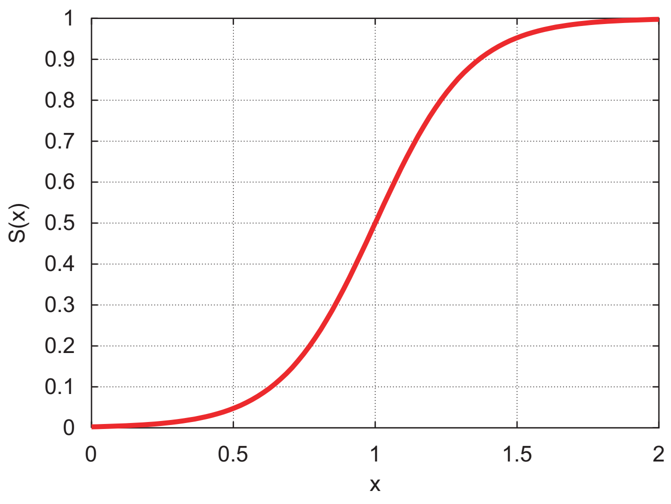


Fig. 3. The squash function from Eq. (7).

deviation from the corresponding pair of primitives from the definition. Note that if the two primitives are not lines, A_{ij} is undefined and $P(A_{ij})$ is taken to be one. $Lines(U_i, U_j, D_i, D_j)$ is one when all of the arguments are nodes representing lines, and zero otherwise.

4.6. Intersection location error

The locations of the intersections between a pair of primitives from the unknown should be similar to those of the corresponding pair of primitives from the definition. If not, an error is assigned. Here, similarity is again measured using the MPDF defined in Eq. (2). Because intersection location is defined by two coordinates, the MPDF is applied twice for each intersection. The total error is computed as

$$E_6 = \frac{\sum_{i=1}^{N_{min}-1}\sum_{j=i+1}^{N_{min}}\sum_{k=1}^{I(D_i,D_j)}([1-P(L_i^k)]+[1-P(L_j^k)])}{\sum_{i=1}^{N_{min}-1}\sum_{j=i+1}^{N_{min}} 2\cdot I(D_i,D_j)}, \tag{10}$$

where (L_i^k, L_j^k) is the coordinates of the kth intersection between the primitives from nodes i and j of the unknown. $I(D_i, D_j)$ is the number of intersections between the primitives from nodes i and j of the definition. In cases where a pair of primitives intersect in the unknown but not in the definition, or vice versa, both $P(L_i^k)$ and $P(L_j^k)$ are set to zero. This error is normalized by twice the number of intersections, as two coordinates can contribute error to each intersection.

5. Graph matching

The previous section described how to compute the similarity between two graphs. This assumed that each node in the unknown ARG was assigned to a specific node in the definition ARG. This section describes how these assignments are obtained. This is a graph matching, or graph isomorphism detection problem. If the user always draws each symbol with a consistent number of primitives and a consistent drawing order, the graph matching problem is trivial. In this case, drawing order would directly provide the correct node-pair assignments. In practice, however, users do not always maintain a consistent drawing order. Furthermore, the problem is made more difficult because of noise. Noise comes from variations in how the symbols are drawn as well as from processing errors. For example, it is not uncommon for there to be extra or missing nodes in the unknown (i.e., extra or missing geometric primitives). Likewise, a primitive that was intended to be a line can be misinterpreted, either through ambiguity or processing errors, as an arc, or vice versa.

We have developed five efficient, approximate matching techniques to find the best match between two ARGs. These are: stochastic matching, error-driven matching, greedy matching, hybrid matching, and sort matching. The first four methods are based on search. The fifth method avoids search by assuming symbols are drawn with a consistent orientation.

The search-based methods make initial node-pair assignments based on drawing order. Assignments are then swapped until the best match is obtained. The quality of the match at each iteration is determined using the similarity score defined in the previous section (Eq. (1)). Our four search-based approaches differ in the way they select the assignments to swap at each iteration.

If the two graphs being matched do not have the same number of nodes, the smaller one is "padded" with empty nodes. This ensures that every node in one graph has a match with a unique node in the other, and hence that every node is considered by the swapping process. When evaluating the error metrics, a pairing with an empty node produces the maximum possible local error. For example, the addition of empty nodes does not reduce the primitive count error, E_1.

Fig. 4 illustrates the typical search process. For ease of explanation, the figure shows hypothetical symbols rather than ARGs. Finding the correct node-pair assignments is equivalent to finding the correct assignment of the primitives of the unknown to the primitives of the definition. Here, the primitives of the definition symbol are numbered according to a typical drawing order. Likewise, the primitives of the unknown are labeled with letters indicating the order in which they were actually drawn. Based on drawing order, primitive **a** of the unknown is initially assigned to primitive **1** of the definition, **b** is assigned to **2**, and so on. It is clear that assignments **b**-**2** and **c**-**3** are correct, while **a**-**1** and **d**-**4** are not. Swapping the latter to produce the assignments **d**-**1** and **a**-**4** is what is needed. The success of this swap can be measured by the resulting increase in the similarity score.

The following sections describe our five matching techniques in detail.

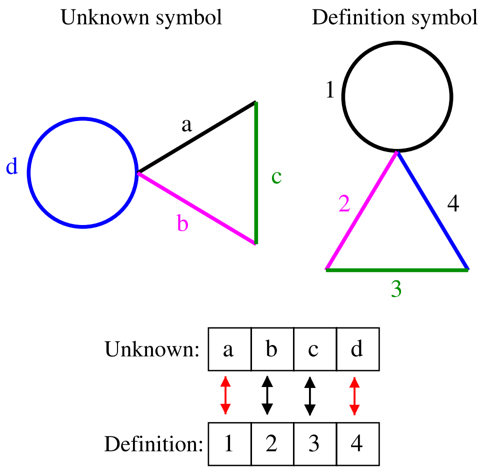


Fig. 4. Graph matching: assignments **b**-**2** and **c**-**3** are correct, while **a**-**1** and **d**-**4** are not.

5.1. Stochastic matching

This approach is based on stochastic search. To begin, the initial node-pair assignments are saved as the current best. Then, three node-pair assignments, which we will call A, B, and C, are randomly selected. A and B are swapped producing assignments A' and B'. B' is then swapped with C. If the new similarity score is better than the current best score, the new assignments are saved as the new current best. This process is repeated a fixed number of times, and the current best node-pair assignments are returned as the best match. In practice, we use a limit of between 100 and 300 iterations. As the number of iterations is fixed (but adjustable), the only cost that varies with problem size is the cost of evaluating the similarity score. This cost is $O(n^2)$, where n is the number of nodes. Pseudo code for this matcher is shown in Fig. 5.

5.2. Error-driven matching

With this approach, a local matching error determines the probability that a node-pair assignment will be selected to be swapped. For example, if a node from the unknown was a line, and the corresponding node from the definition was an arc, there would be a relatively high local matching error, and correspondingly high probability that the node-pair would be selected for swapping. The local matching error of a node-pair is defined as the portion of the dissimilarity score related to that node-pair. This includes all intersection angle, intersection number, and intersection location errors involving the primitives from that node-pair. Likewise, the local matching error also includes primitive type and relative length errors.

At each iteration, the local matching error of each node-pair is computed and selection probabilities are assigned. Based on these probabilities, two node-pairs are selected and swapped. If the similarity score improves, the new assignments are kept. Otherwise, the swap is rejected. This continues until there is a certain number (I_0) of consecutive iterations with no improvement, or until the total number of iterations reaches a limit (I_{MAX}). In practice, we use a value of 300 for I_{MAX}, and a value of between 50 and 200 for I_0. The computational complexity of this approach is similar to that of the stochastic matching approach. Pseudo code for this matcher is shown in Fig. 6.

```
Stochastic_Matcher(unknown, def)
 current_assignment = assign_using_drawing_order(unknown, def)
 best_assignment = current_assignment
 best_score = similarity_score(best_assignment)
 for i = 1 to max_iterations
     (A, B, C) = randomly_pick_node_pairs(current_assignment)
     swap(A, B)
     swap(B, C)
     new_score = similarity_score(current_assignment)
     if (new_score > best_score)
         best_score = new_score
         best_assignment = current_assignment
     endif
 endfor
 return (best_assignment, best_score)
```

Fig. 5. Pseudo code for stochastic matcher.

```
Error_Driven_Matcher(unknown, def)
 current_assignment = assign_using_drawing_order(unknown, def)
 best_assignment = current_assignment
 best_score = similarity_score(current_assignment)
 iterations = 0
 no_improvement = 0
 while (true)
     iterations = iterations + 1
     compute_local_matching_error(current_assignment)
     (A, B) = pick_node_pairs_using_local_error(current_assignment)
     swap(A, B)
     new_score = similarity_score(current_assignment)
     if (new_score > best_score)
         best_score = new_score
         best_assignment = current_assignment
         no_improvement = 0
     else
         swap(A, B) // undo swap
         no_improvement = no_improvement + 1
     endif
     if iterations == iteration_limit
         or no_improvement == no_improvement_limit
         return (best_assignment, best_score)
     endif
 endwhile
```

Fig. 6. Pseudo code for error-driven matcher.

5.3. Greedy matching

This approach uses greedy search to find good node-pair assignments. The program first considers the best assignment for the first node of the unknown. If there are n nodes, the program considers all $n-1$ cases in which the first node-pair is swapped with another. Whichever assignment produces the best similarity score is selected for the first node, and this node-pair is removed from further consideration. This is repeated for the second node-pair and so on. In all, $O(n^2)$ sets of node-pair assignments are considered. The entire search process can be repeated for increased accuracy. We have found that one repetition produces a significant improvement in accuracy, but additional repetitions produce minimal improvement. Pseudo code for this matcher is shown in Fig. 7.

5.4. Hybrid matching

For symbols with a small number of primitives, it is practical to use exhaustive search to find the optimal node-pair assignments. Our hybrid approach uses exhaustive search when there are six node-pairs or less. Otherwise it uses stochastic matching with a limit of 720 iterations. Thus, regardless of the size of the problem, a maximum of 720 search states are explored. Pseudo code for this matcher is shown in Fig. 8.

```
Greedy_Matcher(unknown, def, number_of_rounds)
 current_assignment = assign_using_drawing_order(unknown, def)
 best_score = similarity_score(current_assignment)
 for repeat = 1 to number_of_rounds
    for i = 0 to number_of_node_pairs - 2
       best_swap = i
       for j = i+1 to number_of_node_pairs - 1
          swap(node_pair[i], node_pair[j])
          new_score = similarity_score(current_assignment)
          if (new_score > best_score)
             best_score = new_score
             best_swap = j
          endif
          swap(node_pair[i], node_pair[j]) // undo swap
       endfor
       swap(node_pair[i], node_pair[best_swap])
    endfor
 endfor
 return (current_assignment, best_score)
```

Fig. 7. Pseudo code for greedy matcher.

```
Hybrid_Matcher(unknown, def)
 if(number_of_node_pairs <= 6)
    return(Exhaustive_Matcher(unknown, def))
 else
    return(Stochastic_Matcher(unknown, def))
 endif
```

Fig. 8. Pseudo code for hybrid matcher.

```
Sort_Matcher(unknown, def)
 sort_primitives_in_x_and_y(unknown)
 current_assignment = assign_using_sorted_order(unknown, def)
 best_score = similarity_score(current_assignment)
 return (current_assignment, best_score)
```

Fig. 9. Pseudo code for sort matcher.

5.5. *Sort matching*

This approach does not rely on search. Instead, the nodes are sorted based on the locations of their primitives. Each line segment is characterized by its minimum x and y-coordinates. Each arc is characterized by the coordinates of its center. The primitives are then sorted in ascending order of their x-values. Ties are broken using the y-values sorted in ascending order. The sorted order of the nodes determines the node-pair assignments. The definitions for the sort matcher are learned with a special training procedure (see Section 6) and have pre-sorted nodes. Pseudo code for this matcher is shown in Fig. 9.

This approach is useful only when the drawing orientation is fixed. However, even with a fixed orientation, variations in drawing can result in different sorted orders. For example, if the top edge of a horizontal square were drawn too long, it could be the leftmost primitive rather than the left edge of the square. Nevertheless, as Section 7 describes, the approach often works reasonably well in practice. Additionally, because this approach is particularly efficient, it is suitable for devices with little computational power, such as PDAs.

6. Training

The recognizer is trained by providing a set of training examples for each symbol class. As described in Section 3, the program constructs an "average" ARG for each class. This entails another graph matching problem. To learn a definition, the program must match the ARGs of the various training examples to one another. This task is different from the previous matching problem because a similarity score cannot be computed until after a definition has been learned. For example, during training, the primitive type error cannot yet be determined because the expected primitive type of each node is yet to be determined.

We have explored two solutions to this problem. The first is to require the training examples to be drawn with a consistent drawing order. In this case, the matching problem is avoided as the drawing order uniquely identifies the nodes in the ARG. The second approach, which we call "proximity matching," requires the user to draw symbols with a consistent orientation. In this case, geometric information is used for the matching.

With the proximity matching approach, the training examples are first scaled to have unit bounding boxes and are then translated to the origin. One of the symbols with the most frequently occurring number of primitives is selected as a reference symbol. For example, if five of the examples have six primitives, and one example has seven, an example with six primitives will be selected as the reference. Each of the remaining symbols is then matched to the reference symbol.

To match a symbol, U, to the reference symbol, R, the scaled symbols are first overlayed on top of each other as shown in Fig. 10. Then the directed distance from each primitive in U to each primitive in R is computed.

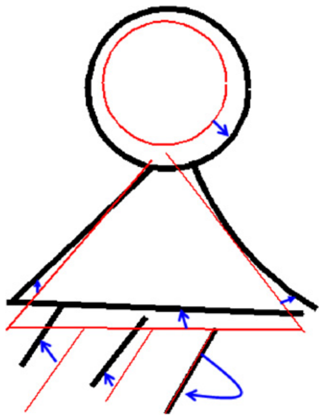


Fig. 10. Proximity matching of two pivot symbols. The reference symbol is shown with bold primitives. The arrows indicate the assignments of primitives.

To facilitate this, the primitives are resampled to have a uniform point spacing of 50 pixels. (We have found that resampling at 50 pixel intervals produces good accuracy with reasonable cost.) The distance from primitive A in symbol U to primitive B in symbol R is computed by finding, for each point a in A, the closest point b in B:

$$d(A,B) = \frac{1}{N_a}\sum_{a\in A}\min_{b\in B}\|a-b\|, \tag{11}$$

where N_a is the number of points in A. Multiple points in A may be closest to the same point in B, and there may be some points in B that are not the closest point of any point in A. Fig. 11 show an example of the directed distance from a line to an arc.

The best match between U and R is defined as an assignment of each primitive in U to a unique primitive in R such that the sum of the directed distances is minimized. This can be expressed mathematically as

$$BestMatch = \underset{m\in M}{\text{argmin}} \sum_{A\in Segs(U)} d(A, m(A)), \tag{12}$$

where M is the set of all one-to-one mappings of the primitives of U to the primitives of R. If R has fewer primitives than U, dummy primitives are added to R so that the number of primitives is the same for both.

To find the best match defined by Eq. (12), we use depth first search with branch and bound. During the search, a partial match is pruned if the sum of the directed distances thus far exceeds that of the current best solution. Efficiency can be further improved through the use of a heuristic under-estimate. The heuristic distance for a given unmatched primitive from U is the minimum directed distance from that primitive to a primitive in R. If the sum of the directed distances for the matched primitives, plus the sum of the heuristic distances for the unmatched primitives exceeds the current best match, the partial match can be pruned. Because training is done off-line, efficiency has not been an issue and we have not implemented this heuristic approach.

Once all of the training symbols have been matched to the reference training symbol, the average ARG is constructed, thus forming the definition of the symbol.

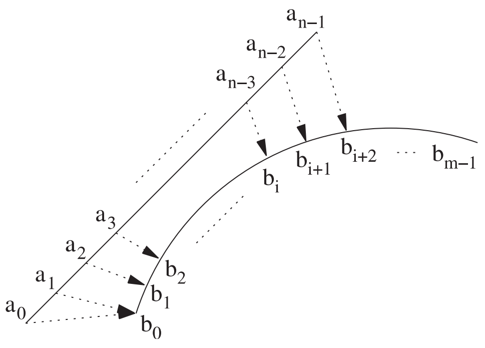


Fig. 11. Computing the directed distance from primitive A to primitive B. Each point on A is mapped to the closest point on B.

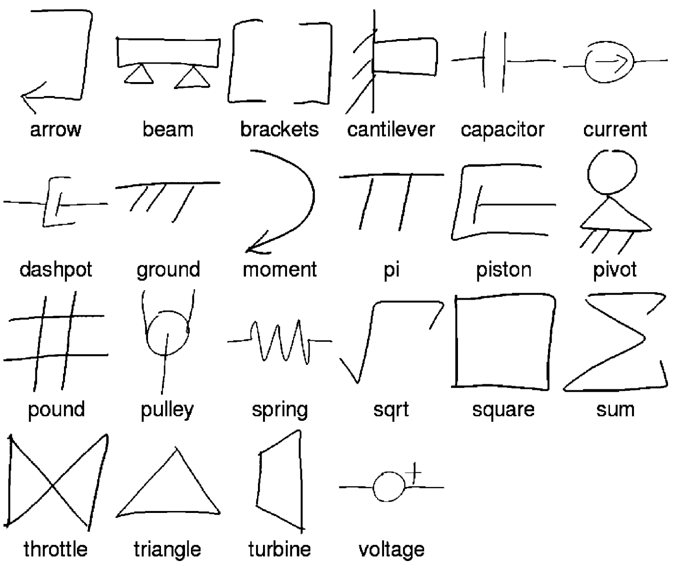


Fig. 12. Symbols from the user study drawn by one of the participants.

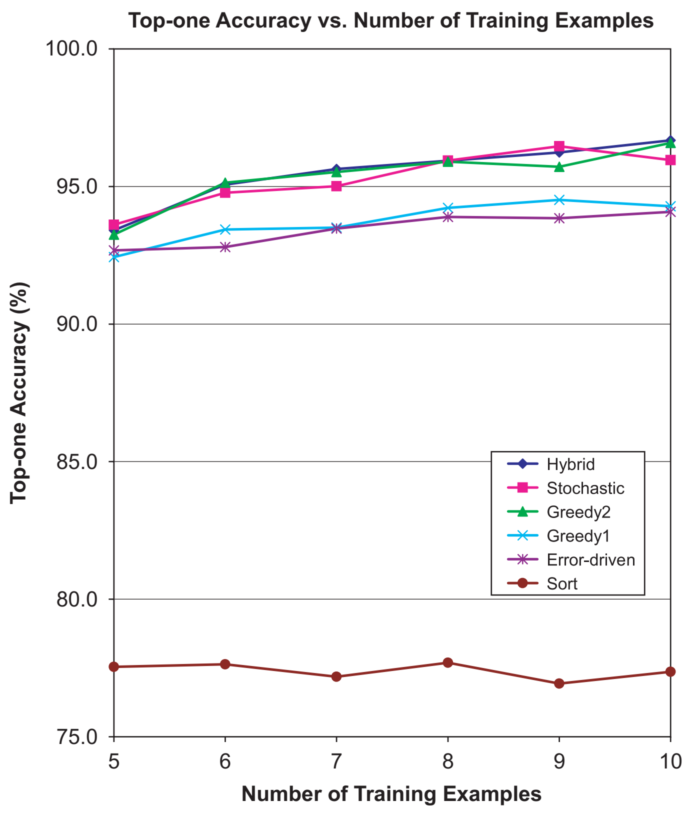
To be consistent with the assumptions underlying the sort matcher, definitions for it are learned using a special procedure. Rather than using geometric proximity to match the training examples to one another, they are matched based on the sorted locations of their primitives. As before, lines are characterized by their minimum x and y-coordinates, and arcs are characterized by their centers. The primitives are sorted in ascending order of their x-values, and ties are broken using the y-values sorted in ascending order.

7. Results

We conducted a user study to evaluate the performance of our five matching techniques. The study involved nine participants, consisting primarily of engineering and computer science graduate students. Two had minimal prior experience with pen-based systems and the rest had essentially none. Because the participants were novices, this is a worst-case evaluation of our system. We expect that even better results would have been obtained if the participants had prior experience using our system.

Each participant was asked to provide 15 sets of the 22 symbols[1] shown in Fig. 12. Participants were instructed to draw naturally but reasonably carefully, and to not intentionally try to trick or break the system. They were also instructed to avoid over-stroking, and to instead redraw a symbol if necessary. (This was rarely done.) Data were collected using a tablet computer, which displayed only the raw ink rather than the processed (segmented) ink.

[1]Data were collected for another symbol class, but it was not used. An anomaly with some examples of this symbol from one particular participant caused slow training. Ordinarily, this would not be a problem, but the experiments reported below required repeating the training process 1800 times.



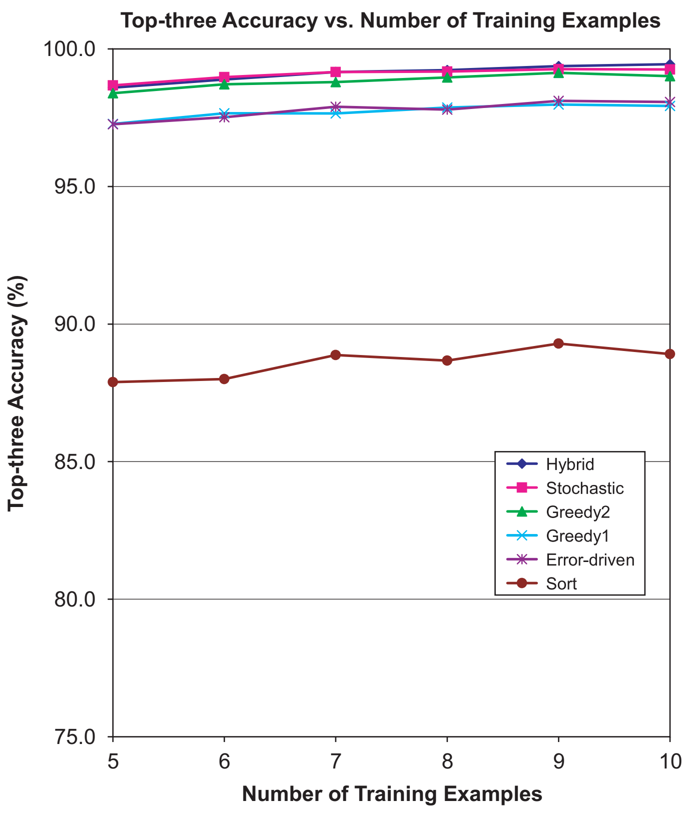


Fig. 13. (Left) Top-one and (Right) top-three accuracy vs. the number of training examples. Stochastic: max-iterations = 300. Greedy1 = one round of greedy search. Greedy2 = two rounds. Error-driven: no-improvement-limit = 150 iterations, max-iterations = 300.

Recognition accuracy was computed after the data were collected so that the participants would receive no feedback that could bias their performance.

7.1. Experiment one: learning rate

As one measure of performance, we evaluated recognition accuracy as a function of the number of training examples, n_t. We computed both the "top-one" accuracy, the rate at which the class ranked highest by the recognizer is indeed the correct class, and the "top-three" accuracy, the rate at which the correct class is one of the three highest ranked classes. The results are shown in Fig. 13. The average recognition times for this experiment are listed in Table 2. All tests were conducted on a Pentium 4 machine with a 3.2 GHz processor and 1 GB of memory. Note that recognition time is independent of the number of training examples.

For this experiment, the maximum number of iterations for the stochastic matcher was set to 300. Likewise, the error-driven matcher was limited to 150 consecutive iterations with no improvement ($I_0 = 150$) or a total of 300 iterations ($I_{MAX} = 300$).

The recognizer was evaluated separately for each user using a cross-validation approach. Each test consisted of randomly selecting n_t of the user's 15 symbol sets for training, and using the remaining $15 - n_t$ sets for testing.

Table 2
Average time to classify a symbol in ms

Hybrid	Stochastic	Greedy2	Greedy1	Error-driven	Sort
41.8	35.9	4.0	2.1	34.9	0.24

Stochastic: max-iterations = 300. Greedy1 = one round of greedy search. Greedy2 = two rounds. Error-driven: no-improvement-limit = 150 iterations, max-iterations = 300.

The test was then repeated nine times, and the results averaged. For each value of n_t, the results were then averaged across all nine participants. Thus, each data point in Fig. 13 represents an average of 90 iterations: 10 cross-validation iterations for each of nine participants.

For this experiment, the hybrid matcher, stochastic matcher, and greedy matcher with two rounds of greedy search all achieved nearly the same performance. With only five training examples, these methods achieved top-one and top-three accuracies of better than 93.3% and 98.4%, respectively. With 10 training examples, they achieved top-one and top-three accuracies of better than 96.0% and 99.0%, respectively. The hybrid approach achieved the highest top-one accuracy of 96.7% with 10 training examples. The hybrid matcher took on average 41.8 ms to classify a symbol, while the stochastic matcher took on average 35.9 ms. The greedy matcher with two rounds of greedy search was much faster, requiring on average only 4.0 ms to classify a symbol.

The error-driven matcher and greedy matcher with one round of greedy search also achieved nearly identical performance. With only five training examples, they achieved top-one and top-three accuracies of about 92.4% and 97.3%, respectively. With 10 training examples, they achieved top-one and top-three accuracies of about 94.1% and 97.9%, respectively. The error-driven matcher took on average 34.9 ms to classify a symbol, while the greedy matcher took on average only 2.1 ms.

The sort matcher was the least accurate method, but it is exceptionally fast, requiring on average only 0.24 ms to classify a symbol. With only five training examples, the sort matcher achieved top-one and top-three accuracies of 77.5% and 87.9%, respectively. With 10 training examples, it achieved top-one and top-three accuracies of 77.4% and 88.9%, respectively. The sort matcher requires a consistent drawing orientation. The participants in our study tended to draw this way. If they had varied the orientation, the performance would have been lower.

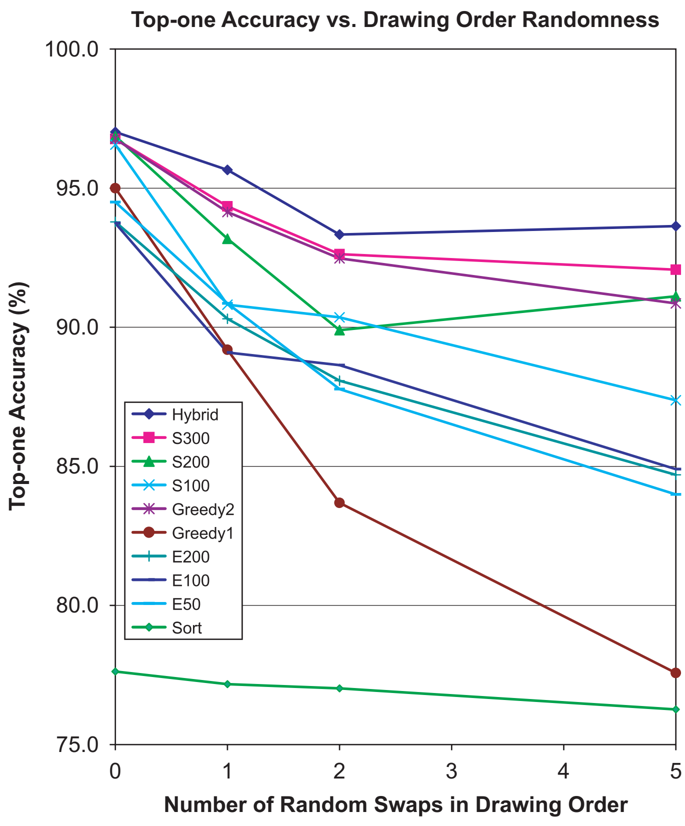
7.2. *Experiment two: drawing order*

As a second test of performance, we measured accuracy as a function of the randomness of the drawing order of the symbols. If the user maintains a consistent drawing order, matching is easier because all of our matching techniques, except sort matching, use the drawing order to construct the initial search state. Thus, randomizing the drawing order provides a good means of evaluating the robustness of our techniques.

In this experiment, the drawing order of the symbols was randomized by selecting pairs of primitives and swapping their drawing orders. The experiment was conducted with one, two, and five random swaps per symbol. Five swaps is a severe test, as it results in as many as 10 primitives having random positions in the drawing order. To provide a baseline for comparison, accuracy was also measured for the original, "un-randomized" drawing order.

Fig. 14 shows the top-one and top-three recognition accuracy as a function of the number of random swaps applied to the drawing order of each symbol. For the stochastic matcher, accuracy is reported for cases in which the number of iterations is limited to 100, 200, and 300. Likewise, for the error-driven matcher, accuracy is reported for cases in which the maximum number of iterations with no improvement (I_0) is limited to 50, 100, and 200. In all cases, the maximum total number of iterations for the error-driven approach is limited to 300 ($I_{MAX} = 300$). The average recognition times for this experiment are listed in Table 3.

In this experiment, the recognizer was again evaluated separately for each user, using a cross-validation approach.



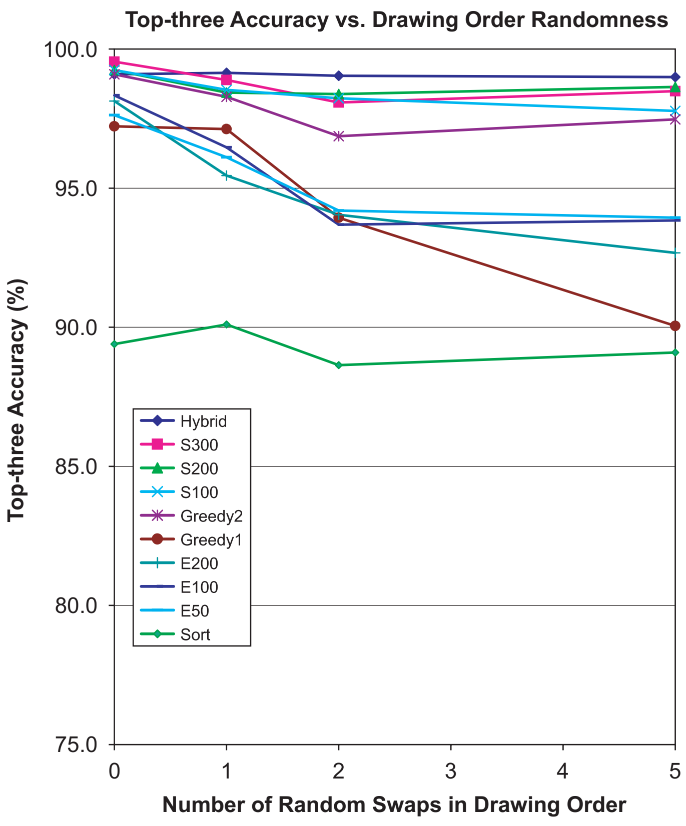


Fig. 14. (Left) Top-one and (Right) top-three accuracy vs. the number of random changes to the drawing order. One random change consists of swapping the drawing order of a randomly selected pair of primitives. S*xxx* = stochastic search with max-iterations = *xxx*. Greedy1 = one round of greedy search. Greedy2 = two rounds. E*xxx* = error-driven search with no-improvement-limit = *xxx* iterations and max-iterations = 300.

Table 3
Average time to classify a symbol in ms

Hybrid	S300	S200	S100	Greedy2	Greedy1	E200	E100	E50	Sort
41.6	35.6	23.9	12.2	4.0	2.1	44.9	24.3	13.5	0.29

S*xxx* = stochastic search with max-iterations = *xxx*. Greedy1 = one round of greedy search. Greedy2 = two rounds. E*xxx* = error-driven search with no-improvement-limit = *xxx* iterations and max-iterations = 300.

Each test consisted of randomly selecting 14 of the user's 15 symbol sets for training, and using the remaining set for testing. The test was then repeated nine times, and the results averaged. The results were then averaged across all nine participants. Thus, each data point in Fig. 14 represents an average of 90 iterations: 10 cross-validation iterations for each of nine participants.

Examination of Fig. 14 reveals that as the drawing order is increasingly randomized, more search is needed to achieve a given level of accuracy. For example, with five random swaps and an iteration limit of 100, stochastic search achieved a top-one accuracy of 87.4% and a top-three accuracy of 97.8%. When the iteration limit was increased to 300, the top-one and top-three accuracies increased to 92.1% and 98.5%, respectively.

The hybrid approach is the best performing method in cases where the drawing order varies. This is because it explores more of the search space than the other methods: it uses exhaustive search for small problems, and 720 iterations of stochastic search for larger ones. With five random swaps, the hybrid approach still achieved a top-one accuracy of 93.6% and a top-three accuracy of 99.0%. Despite exploring more of the search space, the hybrid approach is still fast, requiring on average only about 41.6 ms to classify a symbol.

As expected, the performance of the error-driven approach also increased with increased iteration limits. However, for a given amount of processing time, this approach was not as accurate as the stochastic approach. This is due, in part, to the fact that the error-driven approach must compute both a local error and the dissimilarity score, while the stochastic approach computes only the latter.

The performance of the greedy matcher with one round of greedy search did degrade with increasing randomness in the drawing order. This is to be expected as greedy search methods tend to find only local maxima. However, applying a second round of greedy search substantially improved the performance. For example even with five random swaps in the drawing order, the greedy matcher with two rounds of search achieved top-one and top-three accuracies of 90.9% and 97.5%, respectively. Furthermore, it achieved this high level of performance while requiring on average only 4.0 ms to classify a symbol.

The sort matcher is insensitive to drawing order and thus its performance did not vary significantly in this experiment. The small variations that did occur are likely a result of variations due to the random selection of training data in the cross-validation process.

8. Discussion

We believe that the results of our user study are quite promising when compared to results reported in the literature. For example, Landay and Myers [27] report an accuracy of 89% on a set of five single-stroke editing gestures. In our case, however, there are 22 symbol definitions which can be drawn with multiple strokes. In a study involving seven multi-stroke and five single-stroke shapes, Fonseca and Jorge [2] report an accuracy of 92%. Hse and Newton [4] report an accuracy of 97.3% using 30 training examples on a database of 13 symbols. Our hybrid matcher achieves an accuracy ranging from 93.6% to 97.0%, depending on the amount of randomness in the drawing order. Furthermore, our approach is insensitive to rotation and non-uniform scaling, where their approach may not be (see Section 2). On a database of 20 symbols, Kara and Stahovich [5] report an accuracy of 97.7%, where each symbol was trained with 14 examples. That method is based on image matching techniques, and thus is sensitive to non-uniform scaling. Also, our methods, particularly our greedy matcher, are faster than those in [5].

The experiments described in the previous section reveal various tradeoffs between our five matching techniques. The hybrid matcher is the most accurate, but the most expensive matcher. It achieves the best accuracy because it explores more of the search space than the other methods.

The stochastic matcher is the second most accurate method. One benefit of this approach is that it can take advantage of consistency in the drawing order. If the user maintains a consistent drawing order, relatively little computation is needed. If the drawing order varies greatly, the amount of computation can be directly adjusted to maintain high accuracy.

Like the stochastic-matcher, the error-driven matcher can take advantage of consistency in the drawing order. However, for a given processing time, this approach is not as accurate as the stochastic approach. This is due, in part, to the fact that each iteration of the error-driven approach requires the error metrics to be evaluated two times rather than one. They must be evaluated once to determine the local matching error and once to determine the dissimilarity score. The code could be optimized to eliminate the redundant computation. Also, the error-driven approach uses a form of hill-climbing: node-pair swaps are rejected if the similarity score decreases. It is possible that this hill-climbing strategy causes the match to become stuck in local maxima. It may be desirable to use a simulated annealing approach in which there is some probability that a decrease in the similarity score will be allowed on any given iteration.

The greedy matcher has the best tradeoff between accuracy and cost. Even with the drawing order randomized five times (i.e., five random swaps in the drawing order), it achieved top-one and top-three accuracies of 90.9% and 97.5%, respectively. Furthermore, it took on average only 4.0 ms to classify a symbol.

The sort matcher works by sorting the coordinates of the primitives. Thus, it is useful only when the drawing orientation is fixed. However, even for a fixed orientation, variations in the shape can result in different sorted orders, and consequently recognition errors. One advantage of this method is that it is very fast, requiring on average only 0.24 ms to classify a symbol with a library of 22 definitions. Because the approach is so inexpensive, it would be suitable for devices with little computational power, such as PDAs. Furthermore, the top-three accuracy of 89% is still relatively high. This suggests that it might be possible to use the sort matcher as an inexpensive pre-recognizer to eliminate low ranked definitions. This would then reduce the amount of computation needed for a more accurate but more expensive matcher, such as the hybrid matcher.

The results in Fig. 14 suggest that the participants in our user study tended to draw with a relatively consistent drawing order. This allowed our program to achieve high accuracy with little computation. Artificially randomized the drawing order necessitated more computation to achieve high accuracy. The consistency of the drawing order in our experiment was likely due to the nature of the data collection task. We expect that when pen-based applications are used for real-world tasks, there will indeed be variation in the drawing order. However, we do not expect it to be entirely random. Rather, we expect that users will likely have a few preferred ways of drawing each symbol. For example, when drawing a pivot, one is likely to draw the triangle and then the circle, or vice versa. Furthermore, it is unlikely that one would draw part of the triangle, then the circle, then the rest of the triangle. As a consequence, we plan to explore the possibility of learning the most common drawing orders for each symbol. Combining these with a small amount of search using one of our matching methods may prove to be an effective approach.

Regardless of the amount of randomness in the drawing order, our experiments indicate that the hybrid, stochastic, and greedy matching techniques provide suitable accuracy. There is little variation in their performance when applied to data with two random drawing order swaps vs. five. In the later case, the drawing order is essentially random because the typical symbol in our study had about five primitives. Note that each random swap actually results in two primitives having random positions in the drawing order.

Our experiments did not explicitly evaluate our recognizer's insensitivity to orientation and non-uniform scaling. As our representation is entirely insensitive to orientation, we expect that our overall approach is insensitive to orientation. Likewise, because our representation describes the topology of a shape, our approach is tolerant of non-uniform scaling. However, it would be useful to conduct further studies to quantify this.

We do not yet have an automated means of evaluating the performance of the proximity matcher used for training. We do, however, have a tool that allows us to visually inspect the matches it produces. We have informally examined some of the data and have found that the approach is reliable. Although we have not separately evaluated the performance of the proximity matcher in a systematic way, its performance can be inferred from the overall performance of our system. Because the system as a whole performed well, the proximity matcher must have performed well.

In future work, we plan to explore the possibility of using feedback from our recognizer to improve segmentation. The pen stroke segmenter performed accurately, but examination of the data revealed that some of the symbols did contain missing or extra primitives. Likewise, some primitives that were intended to be lines were classified as arcs, and vice versa. Some of these errors were due to sloppy drawing and ambiguity, while others were a result of processing errors. Once a symbol has been classified, the segmenter could be informed of mismatches between the segmentation of the unknown symbol and that of the matching definition so that the segmentation could be improved.

9. Conclusion

We have presented a trainable symbol recognizer for pen-based user interfaces. The approach is suitable for multi-stroke symbols, is insensitive to rotation, and is tolerant of uniform and non-uniform scaling (i.e., different parts of a shape being scaled differently). Furthermore, our user studies have demonstrated that our approach allows a symbol to be drawn with any drawing order. If, however, the user maintains a relatively consistent drawing order, our techniques can take advantage of this and operate more efficiently.

A symbol is represented internally as an attributed relational graph that describes both its geometry and topology. A symbol definition is also represented as an attributed relational graph, but the attributes are learned from training examples and are described statistically. Using a statistical representation makes our approach robust to the types of variations common in hand-drawn shapes.

Symbol recognition involves finding the symbol definition whose attributed relational graph best matches that of the unknown symbol. We have developed a novel set of metrics for comparing graphs in this domain. Much of the power of our approach derives from these metrics, and the particular set of attributes encoded in the graphs.

One challenge addressed in the current work is how to perform graph matching in an efficient fashion so as to achieve both tolerance for drawing variation and interactive performance. We presented five approximate graph-matching techniques: stochastic matching, which is based on stochastic search; error-driven matching, which uses local matching errors to drive the solution to an optimal match; greedy matching, which uses greedy search; hybrid matching, which uses exhaustive search for small problems and stochastic matching for larger ones; and sort matching, which relies on geometric information to accelerate the matching. We have also developed a "proximity" matcher which is used for training purposes.

Our user studies have revealed a number of tradeoffs between these techniques. The hybrid matcher is the most accurate but most expensive approach. Nevertheless, it is still sufficiently fast for interactive use. The stochastic matcher achieves high accuracy and can take advantage of consistency in the drawing order. If the user maintains a consistent drawing order, relatively little computation is needed. If the drawing order varies greatly, the amount of computation can be directly adjusted to maintain high accuracy. The greedy matcher provides the best tradeoff between accuracy and cost: it achieves relatively high accuracy and is fast. The error-driven matcher is accurate, but for a given amount of computation, is not as accurate as the stochastic matcher. The sort matcher is the least accurate approach and requires consistent drawing orientation. However, because this method is particularly efficient, it may be a good solution when computational resources are constrained, such as with a PDA.

Acknowledgments

The authors gratefully acknowledge the National Science Foundation's support for this work through award 0413088.

References

[1] Rubine D. Specifying gestures by example. In: Proceedings of the 18th annual conference on computer graphics and interactive techniques (SIGGRAPH '91); 1991. p. 329–337.

[2] Fonseca MJ, Pimentel C, Jorge JA. CALI—an online scribble recognizer for calligraphic interfaces. In: AAAI spring symposium on sketch understanding; 2002. p. 51–8.

[3] Sezgin TM, Davis R. HMM-based efficient sketch recognition. In: International conference on intelligent user interfaces (IUI'05), New York, 2005.

[4] Hse HH, Newton AR. Recognition and beautification of multi-stroke symbols in digital ink. Computers & Graphics 2005;29(4):533–46.

[5] Kara LB, Stahovich TF. An image-based, trainable symbol recognizer for hand-drawn sketches. Computers & Graphics 2005; 29(4):501–17.

[6] Messmer BT, Bunke H. Efficient subgraph isomorphism detection: a decomposition approach. IEEE Transactions on Knowledge and Data Engineering 2000;12(2):307–23.

[7] Ullmann JR. An algorithm for subgraph isomorphism. Journal of the ACM 1976;23(1):31–42.

[8] Dickinson S, Pelillo M, Zabih R. Introduction to the special section on graph algorithms in computer science. IEEE Transactions on Pattern Analysis and Machine Intelligence 2001;23(10):1049–52.

[9] Garey MR, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness. New York: Freeman and Company; 1979.

[10] Conte D, Foggia P, Sansone C, Vento M. Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence 2004;18(3):265–98.

[11] Kara LB, Stahovich TF. Hierarchical parsing and recognition of hand-sketched diagrams. In: UIST '04: 17th annual ACM symposium on user interface software and technology; 2004, p. 13–22.

[12] Gennari L, Kara LB, Stahovich TF, Shimada K. Combining geometry and domain knowledge to interpret hand-drawn diagrams. Computers & Graphics 2005;29(4):547–62.

[13] Lladós J, Valveny E, Sánchez G, Martí E. Symbol recognition: current advances and perspectives. In: GREC '01: selected papers from the fourth international workshop on graphics recognition algorithms and applications. London, UK: Springer; 2002. p. 104–27.

[14] Lee S-W. Recognizing hand-drawn electrical circuit symbols with attributed graph matching. In: Baird HS, Bunke H, Yamamoto K, editors. Structured document image analysis. Berlin: Springer; 1992. p. 340–58.

[15] Mahoney JV, Fromherz MPJ. Three main concerns in sketch recognition and an approach to addressing them. In: AAAI spring symposium on sketch understanding; 2002.

[16] Rocha J, Pavlidis T. A shape analysis model with applications to a character recognition system. IEEE Transactions on Pattern Analysis and Machine Intelligence 1994;16(4):393–404.

[17] Lladós J, Martí E, Villanueva JJ. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. IEEE Transactions on Pattern Analysis Machine Intelligence 2001; 23(10):1137–43.

[18] Pereira JP, Branco VA, Jorge JA, Silva NF, Cardoso TD, Ferreira FN. Cascading recognizers for ambiguous calligraphic interaction. In: 2004 eurographics workshop on sketch-based interfaces and modeling; 2004. p. 63–72.

[19] Matsakis NE. Recognition of handwritten mathematical expressions, Masters thesis, Massachusetts Institute of Technology, 1999.

[20] Hammond T, Davis R. LADDER, a sketching language for user interface developers. Computers & Graphics 2005;29(4):518–32.

[21] Hammond T, Davis R. Interactive learning of structural shape descriptions from automatically generated near-miss examples. In: IUI '06: proceedings of the 11th international conference on intelligent user interfaces. New York, NY, USA: ACM Press; 2006. p. 210–7.

[22] Shilman M, Pasula H, Russell S, Newton R. Statistical visual language models for ink parsing. In: AAAI spring symposium on sketch understanding; 2002. p. 126–32.

[23] Gross MD. Recognizing and interpreting diagrams in design. In: AVI '94: Proceedings of the workshop on Advanced visual interfaces; 1994. p. 88–94.

[24] Huang Z, Cohen F. Affine-invariant b-spline moments for curve matching. IEEE Transactions on Image Processing 1996;5(10): 1473–80.

[25] Raymaekers C, Vansichem G, Van Reeth F. Improving sketching by utilizing haptic feedback. In: AAAI spring symposium on sketch understanding. AAAI Press; 2002. p. 113–7.

[26] Stahovich TF. Segmentation of pen strokes using pen speed. In: AAAI 2004 fall symposium: making pen-based interaction intelligent and natural; 2004.

[27] Landay JA, Myers BA. Sketching interfaces: toward more human interface design. IEEE Computer 2001;34(3):56–64.