From: AAAI Technical Report SS-02-08. Compilation copyright © 2002, AAAI (www.aaai.org). All rights reserved.

Recognizing Multi-Stroke Symbols

Chris Calhoun, Thomas F. Stahovich, Tolga Kurtoglu, Levent Burak Kara
Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
stahov@andrew.cmu.edu

Abstract

We describe a trainable recognizer for multi-stroke symbols. The learned definitions are described in terms of the constituent geometric primitives (lines and arcs), the properties of individual primitives, and the geometric relationships between them. A definition is learned by examining a few examples of a symbol and identifying which properties and relationships occur frequently. During both training and recognition, multiple primitives can be drawn in the same pen stroke. Pen speed and curvature are used to segment a stroke into its constituent primitives. During recognition, an unknown symbol is identified by determining which definition matches it with the least error. There are two recognition methods. One assumes that the primitives of a symbol are always drawn in the same order. This method is fast, but requires some care from the drawer. The other method uses a form of best-first search, with a speculative quality metric and pruning, to recognize symbols when the drawing order is varied.

Introduction

In our research, we are developing sketch understanding techniques that will enable sketched-based user interfaces. Our goal is for people to be able to operate software by drawing the kinds of sketches they ordinarily draw. For example, we would like an engineer to be able to create a dynamic simulation of a mechanism by sketching a simple schematic of it, using familiar symbols and drawing conventions. Similarly, we would like to be able to create viewgraphs by sketching the desired graphics, such as arrows, boxes, quote bubbles, etc.

There are a number of processes underlying a sketch-based user interface. These include the low-level processing of pen strokes, recognition of symbols, reasoning about shapes, and high-level interpretation. The focus of the work described here is the first two of these processes. We have developed techniques for segmenting pen strokes into their constituent lines and arcs. We have also developed a trainable symbol recognizer that learns to recognize a symbol by examining a few examples of it. In (Kurtoglu & Stahovich 2002), we combine this work with high-level reasoning techniques to produce a system that can interpret schematic sketches of physical systems.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Our recognizer operates on the output of the stroke segmenter. This provides for a more natural drawing environment by allowing the user to vary the number of pen strokes used to create a symbol. For example, a square can be drawn as a single pen stroke, or as four separate strokes, or even as two or three strokes. Much of the previous work has relied either on single stroke methods in which an entire symbol must be drawn as single stroke (e.g., (Rubine 1991), (Kimura, Apte, & Sengupta 1994), (Cohen, Huang, & Yang 1995) or single primitive methods in which each stroke must be a single line, arc, or curve (e.g., (Zhao 1993), (Igarashi *et al.* 1997), (Weisman 1999)).

The key challenge in segmenting is determining which bumps and bends in a pen stroke are intended and which are accidents. Our approach to segmenting considers both the shape of the stroke and the motion of the pen tip as the stroke is created. We have found that it is natural to slow the pen when making intentional discontinuities in the pen stroke, thus we can identify discontinuities by examining the speed profile of the pen stroke. This speed-based approach finds many segment points, but not all. We use a smoothed curvature metric to identify other segment points.

Our symbol recognizer employs an approach similar to near miss learning. To train the recognizer, the user provides several examples of a symbol. The strokes are segmented and each example is characterized by a semantic network description. The semantic networks are compared, and any sketch properties (network links and node attributes) that occur frequently are assembled to form a definition of the symbol. We have found that three or four examples are often adequate for learning engineering symbols such as pivots, beams, springs, and pulleys (Figure 1). To recognize an unknown symbol, the strokes are segmented and a semantic network is constructed. The network is matched against each known definition, and an error is calculated describing the difference between the symbol and that definition. The symbol is identified by the definition that fits with the least error.

Pen Stroke Segmenting

The first step in interpreting a sketch is processing the individual pen strokes to determine what shapes they repre-

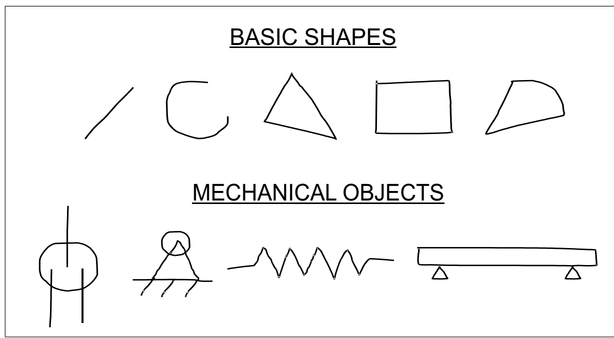

Figure 1: Typical symbols. Basic shapes include a line, arc, triangle, square, and pie slice. Mechanical objects include a pulley and ropes, pivot, spring, and beam. Reliable definitions can be learned from 3 or 4 training examples.

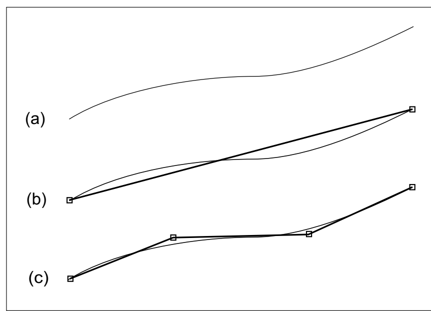

Figure 2: (a) A raw pen stroke. (b) Interpretation as a single line. (c) Interpretation as three lines.

sent. Much of the previous work in this area assumes that each pen stroke represents a single shape, such as a single line segment or arc segment, which ever fits the stroke best. While this kind of approach facilitates shape recognition, it results in a less than natural user interface. For example, one would be forced to draw a square as four individual pen strokes, rather than a single pen stroke with three $90°$ bends.

Our objective is to facilitate a natural sketch interface by allowing pen strokes to represent any number of shape primitives connected together. This requires examining each stroke to identify the *segment points*, the points that divide the stroke into different primitives. The key challenge is determining which bumps and bends are intended and which are accidents. Consider, the pen stroke in Figure 2a, for example. Was this intended to be a single straight line as in Figure 2b, or three straight lines as in Figure 2c? Similarly, was the pen stroke in Figure 3a intended to be two straight lines forming a corner as in Figure 3b, or was it intended to be a segment of an arc as in Figure 3c? We have found it difficult to answer these sorts of question by considering shape alone. The size of the deviation from an ideal line or arc is not a reliable indicator of what was intended: sometimes small deviations are intended while other times large ones are accidents.

Our approach to this problem relies on examining the motion of the pen tip as the strokes are created. We have discovered that it is natural to slow the pen when making many kinds of intentional discontinuities in the shape. For example, if the stroke in Figure 3a was intended to be two lines forming a corner, the drawer would likely have slowed down when making that corner. Similarly, when drawing a rectangle as a single pen stroke, it is natural to slow down at the corners, which are the segment points. Figure 4 shows the speed profile for a typical square. The corners can be easily identified by the low pen speed.

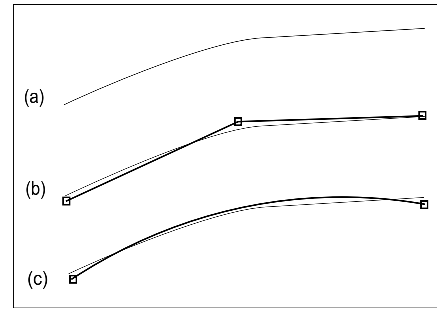

Figure 3: (a) A raw pen stroke. (b) Interpretation as two lines. (c) Interpretation as an arc.

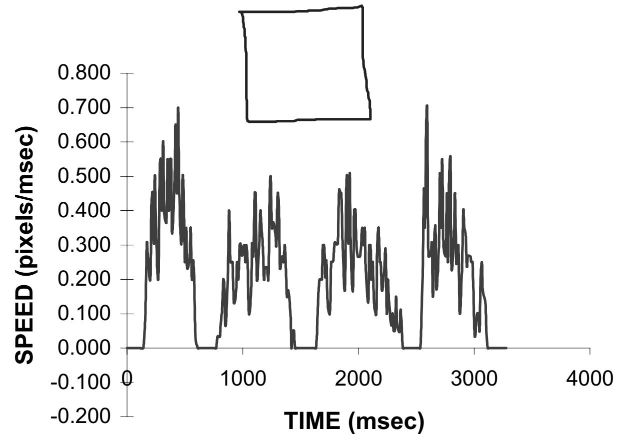

Figure 4: A square and its speed profile. The corners are identifiable by the low speed. A mouse was used for drawing.

We calculate pen speed in the obvious way, as the distance traveled between consecutive pen samples divided by the time elapsed between the samples. Distance is measured in the hardware coordinates of the input device. Because most pen input devices emulate a mouse, we have written our software to use a standard mouse programming interface. This has allowed us to use our software with an electronic whiteboard, a stylus and digitizing pad, and a conventional mouse. We initially used an event-driven software model, but found that the temporal resolution was inadequate on some platforms. Our current approach is to use the event-driven model to handle pen up and pen down events, and to poll for the

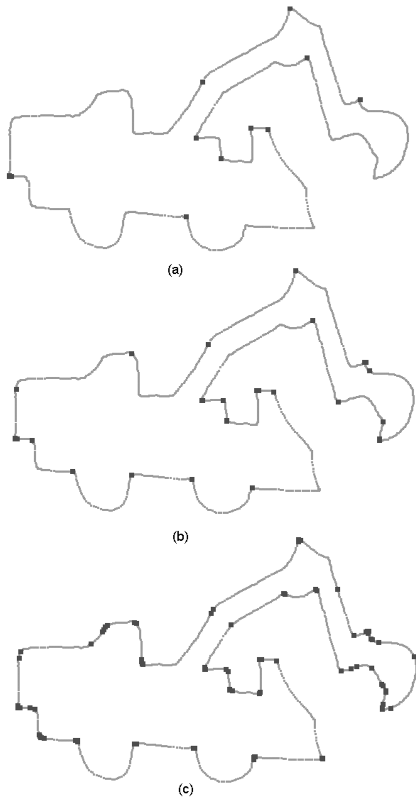

Figure 5: Segment points for thresholds of (a) 20% (b) 25% and (c) 35% of the average pen speed.

mouse position in between. This has allowed us to increase the resolution, but it does result in redundant samples when the mouse is stationary. When the mouse is stationary, there is a sequence of samples that all have zero velocity. We discard all but the first sample in these sequences.

Once the pen speed has been calculated at each point along the stroke, segment points can be found by thresholding the speed. Any point with a speed below the threshold is a segment point. We specify the threshold as fraction of the average speed along the particular pen stroke. If necessary, the user can adjust the threshold to match his or her particular drawing style. In our informal testing, we have found that with a small amount of tuning, one can achieve good results. Figure 5 shows the segment points that are detected for a typical pen stroke for various values of the threshold. To enhance the performance of this approach, one can slightly exaggerate the slowdown at intended segments points. The drawing experience is still natural because no pen up and pen down events are necessary, and there is no need to stop completely.

While many intentional discontinuities occur at low pen speed, others do not. For example, when drawing an “S” shape, there may be no slowdown at the transition from one lobe to the other. Similarly, when drawing a “J” shape, there may be no slowdown at the transition from the line to the arc. We can locate these kinds of segment points by examining the curvature of the pen stroke. Segment points occur at locations where the curvature changes sign. We consider three distinct signs: positive, negative, and zero. When computing the sign, we examine a window of points on either side of the point in question. We connect the first and last points in the window with a line segment. We then calculate the minimum distance from each point in the window to the line. Distances to the left of the line are positive, while those to the right are negative. Left and right are defined relative to the drawing direction. The signed distances are summed to determine the sign of the curvature. If the absolute value of the sum is less than a threshold, the curvature is considered to be zero. In the example in Figure 6, the curvature is positive because there are more positive distances than negative ones. (In this example, the drawing direction is from left to right.)

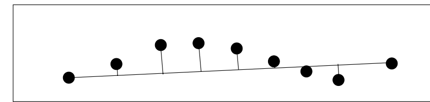

Figure 6: Calculating the curvature sign. The window includes 9 points.

By using a window of points to compute the sign of the curvature, we are able to smooth out noise in the pen signal. Some of the noise comes from minor fluctuations in the drawing, other noise comes from the digitizing error of the input device. The larger the window, the larger the smoothing effect. The size of the window must be tuned to the input device and the user. For mouse input, we have found a window size of between 10 and 30 points to be suitable. Figure 7 shows how the number of segment points varies with the window size.

Once the strokes have been segmented, the next task is to determine which segments represent lines and which represent circular arcs. We compute the least squares best fit line and arc for each segment. The segment is typically classified by the shape that matches with the least error. However, nearly straight lines can always be fit with high accuracy by an arc with a very large radius. In such cases, we use a threshold to determine if a segment should be an arc or a line. To be an arc, the arc length must be at least 15°.

Symbol Recognition: Training

After segmenting the pen strokes, the next step is to recognize individual symbols. We have developed a trainable symbol recognizer for this purpose. Our approach is similar to near miss learning (Winston 1975), except that currently we consider only positive training examples. To train the system, the user provides several examples of a given symbol. Each example is characterized by a semantic network description. The networks for the various examples are compared, and any sketch properties (network links) that occur frequently are assembled to form a definition of the symbol. This definition is a generalization of the examples, and is useful for recognizing other examples of the symbol.

The objects in the semantic network are geometric prim-

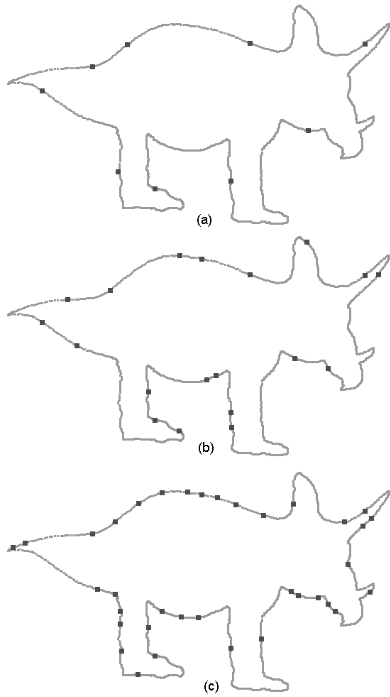

Figure 7: Segment points for curvature window sizes of (a) 30 (b) 15 and (c) 10 points. Note that speed segment points are not shown.

itives: line and arc segments. The links in the network are geometric relationships between the primitives. These include:

- The existence of intersections between primitives.
- The relative location of intersections.
- The angle between intersecting lines.
- The existence of parallel lines.

In addition to the relationships, each primitive is characterized by intrinsic properties, including:

- Type: line or arc.
- Length.
- Relative length.
- Slope (for lines only).
- Radius (for arcs only).

We describe distance by both an absolute and relative metric. An absolute distance is measured in pixels. Relative distances are measured as a proportion of the total of all of the stroke lengths in the symbol. For example, the relative length of one side of a perfect square is 25%. Using an absolute distance metric allows the program to learn definitions in which size matters, while relative distances ignore uniform scaling. For example, if the training examples are squares of different sizes, the definition will be based on relative length and will recognize squares of all sizes. If, on the other hand, all of the training examples are the same size,

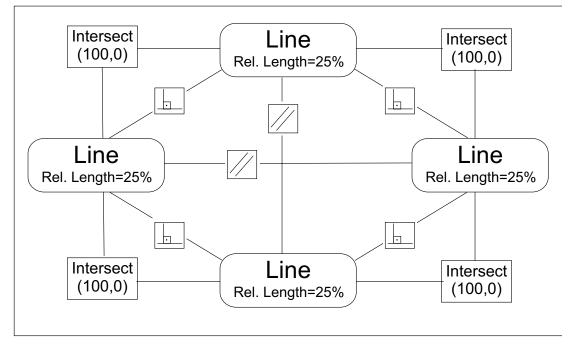

Figure 8: The semantic network definition of squares of varying sizes. The links represent parallel and perpendicular relationships and intersections.

the definition will also include absolute distance, and only squares of that size will be recognized.

The locations of intersections between primitives are measured relative to the lengths of the primitives. For example, if the beginning of one line segment intersects the middle of another, the intersection is described as the point $(0\%, 50\%)$. When extracting intersections from the sketch, a tolerance is used to allow for cases in which an intersection was intended, but one of the primitives was a little too short. The tolerance zone at each end of the primitive is 25% of the length of that primitive. If an intersection occurs in the tolerance zone, it is recorded as being at the end of the primitive: The relative location is described as 0% if the intersection is near the beginning of the segment, or 100% if it is near the end.

If a pair of lines do not intersect, the program checks if they are parallel. Here again, a tolerance is used because of the imprecise nature of a sketch. Two lines are considered to be parallel if their slopes differ by no more than 5 degrees.

To construct the definition of a symbol, the semantic networks for each of the symbols are compared to identify common attributes. If a binary attribute, such as the existence of an intersection, occurs with a frequency greater than a particular threshold, that attribute is included in the definition. Similarly, if an attribute has a continuous numerical value, such as relative length, it will be included in the definition if its standard deviation is less than some threshold. Figure 8 shows a typical example of a definition.

The thresholds are empirically determined, and the values are as follows. The occurrence frequency threshold for intersections is 70%. That is, if at least 70% of the training examples have an intersection between a particular pair of primitives, that intersection is included in the learned definition. An arc can intersect a line, or another arc, in two locations. The occurrence frequency threshold for two intersections is also 70%. The threshold for the existence of parallelism between lines is 50%.

The standard deviation threshold for continuous-valued quantities is 5. The maximum value for a relative length is 100, thus the standard deviation threshold is 5% of the maximum value. Absolute length is measured in pixels and prim-

itives can be a few hundred pixels long. Thus, the threshold for absolute length can be a little more restrictive than for relative length if large symbols are drawn. The maximum value for an intersection angle is 180 degrees. The standard deviation threshold, therefore, is 2.8% of the largest possible intersection angle.

During training, it is assumed that the all of the examples have the same number and types of primitives. Furthermore, it is assumed that the primitives are drawn in the same order and in the same relative orientation. Relative orientation describes which end of a primitive is the start and which is the end. For example, if the four sides of a square are drawn in a clockwise loop with the end of one side connecting to the start of the next, then all examples should be drawn that way. Drawing the square by first drawing one set of parallel sides and then drawing the other set, would constitute a different drawing order. Having the end of one side connect to the end of another (rather than the start) would constitute a different relative orientation. These assumptions make it trivial to determine which primitives in one example match those of another. The advantage is that training costs are negligible.

Symbol Recognition: Matching

After drawing a symbol, the drawer indicates that the symbol is finished by using the stylus to press a button displayed on the drawing surface (CRT or whiteboard). This begins the process of recognizing the symbol, i.e., finding the learned definition that best matches the unknown symbol. We have two methods for performing this task. The first employs the same assumptions used during training. The symbol must have the correct number of primitives, drawn in the correct order, and with the correct relative orientation. This method is computationally inexpensive, and is therefore quite fast. The second method uses a heuristic search technique to relax many of these assumptions. This allows for much more variation in the way a symbol is drawn, but is correspondingly more expensive. We will begin by considering the non-search method, as the other method is an extension of it.

For the non-search method, the order in which one draws the primitives directly indicates correspondence with the primitives in a definition. The error in the match can be directly computed by comparing the semantic networks of the unknown and the definition. This is accomplished by comparing each of the attributes and relationships included in the definition to those of the unknown. The definition that matches with the least error classifies the example. However, a maximum error can be set, such that if the best fit exceeds that maximum, the symbol is not classified.

Error Calculation

Matching errors occur when the number and types of primitives in the unknown symbol, their properties, and their relationships differ from those of the definition. When evaluating the total error, different weights are assigned to different kinds of errors. These weights reflect our experience with which characteristics of a symbol are most important for accurately identifying a symbol.

Quantity	Weight
Primitive count	0.15
Primitive type	1.0
Intersection	1.0
Parallelism	1.0

Table 1: Weights assigned to quantized errors.

Property	Range, R	Tolerance, ϵ
Absolute length	Ave. from training	1.0
Relative length	100.0	1.0
Intersection location	100.0	0.33
Intersection Angle	180.0	0.17

Table 2: Constants used for calculating the error for continuous-valued properties.

Some of the errors are quantized, that is an error is assigned based on the number of differences, as described in Table 1. An error is assigned if the unknown symbol and definition have different numbers of primitives. The weight for this is 0.15, that is the error is 0.15 times the absolute value of the difference.[1] For example, if the unknown has 5 primitives, and the definition has 7, the error is 0.3. Similarly, an error is assigned if the type of a primitive in the unknown is different than that of the definition. The weight for this error is 1.0. Likewise an error of 1.0 is assigned for each missing intersection or parallelism between primitives.

The remaining errors are assigned based on the size of the differences, rather than on the number of differences. These proportional errors are used for real valued properties such as relative length or intersection angle. Our error function is a saturating linear function:

$$e(x) = min \left\{ \begin{array}{c} |\frac{x-\overline{x}}{\epsilon R}| \\ \\ 1.0 \end{array} \right\} \qquad (1)$$

where x is the observed value of a property, $\overline{x}$ is the mean value of the property observed in the training examples, ϵ is a tolerance, and R is the maximum expected value for the property. The error saturates at 1.0. ϵ determines how quickly the error saturates as shown in Figure 9. The smaller the value of ϵ, the faster the function saturates. ϵ can be thought of as an error tolerance, because its value determines how much deviation in the property is allowed before the maximum error is assigned. Table 2 shows the error constants used for the various continuous-valued properties.

The more primitives and properties contained in a definition, the more opportunities there are to accumulate error. It may be possible for a definition with many primitives and properties to produce a larger error than a less comprehensive definition, even if the symbol in question is a better

[1] This error constant is smaller than the others, however, when computing the final error, all of the other error terms are normalized while this one is not.

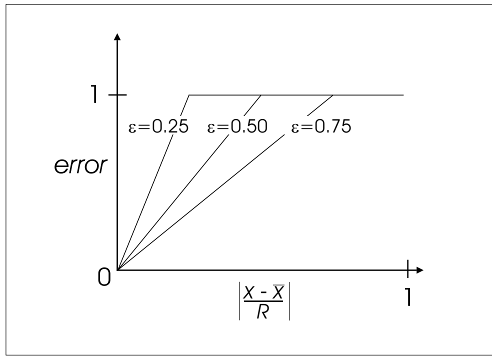

Figure 9: Saturating error function for continuous-valued properties.

match for the former. To avoid this, we normalize the error with the following formula:

$$E^{'} = min \left\{ \begin{array}{c} \frac{E}{n_{prim}+n_{prop}+n_{rel}} + C \\ \\ 1.0 \end{array} \right\} \quad (2)$$

where $E^{'}$ is the normalized error, E is the sum of all errors except the primitive count error, C is the primitive count error, n_{prim} is the number of primitives in the definition, n_{prop} is the number of properties such as relative length, and n_{rel} is the number of relationships such as intersections. With this formula, the primitive count error is weighted much more heavily than the other kinds of errors. This expresses the notion that if the number of primitives in a symbol is significantly different from that of the definition, a match is unlikely.

We often find it useful to consider the accuracy of the match rather than the error. The accuracy is the complement of the error:

$$A = 100.0\%(1.0 - E^{'}) \quad (3)$$

An accuracy of 100 is a perfect match, while an accuracy of 0 is an extremely poor match. The unknown symbol is classified by the definition that matches with the highest accuracy. However, if that accuracy is less than about 65% or 70%, the match is questionable.

Using Search

Thus far, the discussion has concerned matching under the assumptions that the primitives are always drawn in the same order and in the same relative orientation. (Recall that relative orientation describes which end of a primitive is the start and which is the end.) Now we consider a method for relaxing these assumptions in order to allow more variation in the way a symbol can be drawn. With our previous assumptions, the drawing order directly indicated correspondence between the primitives in the symbol and those in the definition, and the direction of the pen stroke directly indicated the relative orientation of a primitive. Now, we will use search to establish correspondence and identify the relative orientations.

Our search technique can be described as best-first search with a speculative quality metric and pruning. A search node contains a partial assignment of the primitives in the unknown symbol to those of the definition. A search node is expanded by assigning an unassigned primitive in the symbol to one in the definition. A search node is terminal if an assignment has been made for each of the primitives in the definition or if there are no remaining unassigned primitives in the unknown symbol.

The search process considers all known definitions at the same time. The process is initialized by generating all possible assignments for the first primitive in each definition. When making the assignments, both choices of orientation are considered. As a consequence, if there are n definitions and m primitives, the search queue will initially contain $2 * n * m$ nodes. It is possible to reduce the search space by postponing consideration of the relative orientation, but our implementation handles drawing order and relative orientation in a uniform way, by using a single search procedure.

Our quality metric is the converse of the matching error. The search queue is sorted in decreasing order of the normalized matching error. The error is computed with Equation 2 except that the primitive count error is excluded. It is excluded because it would most penalize those nodes that are at the shallowest depth in the search tree. If the term were included, the search would become more like depth first search, because the nodes that had the largest number of assignments would have the lowest error, and thus would be expanded first.

For non-terminal nodes, the error in some of the properties cannot be evaluated because the associated primitives have not yet been assigned. For example, if one (or both) of a pair of intersecting lines has not been assigned, it is not possible to determine if the intersection actually exists or what the error in the location of the intersection would be if it did. In such cases, we use a speculative error estimate. If an error cannot be measured because some of the primitives have not been assigned, we assign a small default error. Currently, we assign a value of 0.05 for each such uncomputable error. Doing this makes sense because sketches, due to their imprecise nature, always differ to some extent from the learned definitions.

Our speculative error calculation helps to prevent poor partial assignments from being expanded further. If the initial few assignments produce a large error, and there are many properties that cannot yet be evaluated, the search node will be assigned a relatively large error value. When the queue is sorted, such nodes will effectively be eliminated from consideration. In this sense, the speculative error calculation helps the search to be efficient.

To limit the search, we set a maximum error threshold. If the error of any (non-terminal) node exceeds the threshold, it is pruned from the search. This, again, helps to make the search efficient. We typically use an error threshold of 0.2 to 0.3. Adjusting the threshold and the speculative error constant allow one to tune the search method. For example, by increasing the speculative error constant and decreasing the threshold, the search can be accelerated but there is an increased chance that the correct definition will not be found.

Conversely, if the speculative error constant is set to zero and the threshold is made large, the search will become exhaustive, ensuring that the correct definition will always be found.

Discussion and Future work

The current implementation of our trainable recognizer uses an earlier approach that we developed for segmenting pen strokes. The segmenting approach described here is implemented, but is not yet integrated with the recognizer. Thus, we cannot yet quantitatively evaluate the overall performance of our system.

In informal tests, we have found that if the segmentation is accurate, the recognition rate is high. Our current system, which employs the old segmenter, provides the user with the option to redraw incorrectly segmented strokes. When segmenting errors are corrected in this fashion, we achieve recognition rates that could be estimated at 95% or better for symbols like those in Figure 1.

We have found that often three or four training examples are adequate. Furthermore, our definitions have the ability to discriminate between similar shapes. For example, the system can distinguish between squares and non-square rectangles. Similarly it can distinguish between three lines forming a triangle and three lines forming a "U" shape.

Our search-based matching method has demonstrated that it is possible to accurately match symbols when the drawing order is varied. However, the method is expensive if there is a large number of definitions or a large number of primitives in the unknown symbol. There are simple things that can be done to make the approach more efficient. For example, the relative orientation property can be handled as post-processing step. A default orientation can be assumed, and if this results in appreciable errors in intersection locations, the orientation can be flipped.

We are also working to develop more fundamental improvements in our matching technique. We are exploring methods for reducing the size of the search space and for exploring that reduced space more efficiently. In addition, we currently handle segmenting separately from matching. However, during matching, it may be desirable to revisit the segmenting process to correct errors in the segmentation. We are exploring methods to accomplish this.

Related Work

The problem of polygon fitting and corner point (segment point) detection from digital curves has attracted numerous researchers ((Witkin 1984), (Rattarangsi & Chin 1992), (Bentsson & Eklundh 1992), and (Dudek & Tsotsos 1997)). For corner identification, most algorithms search for abrupt changes in direction by maximizing the curvature function. To suppress noise and false corners, the input data is usually smoothed with a filter. The main challenge here is to determine a reliable "observation scale" or amount of smoothing. Single-scale representations often lead to non-optimal results. Too little smoothing leads to superfluous corners whereas excessive smoothing causes the disappearance of true corners.

(Witkin 1984) and (Rattarangsi & Chin 1992) describe methods based on a multiple-scale representation in which different levels of smoothing can be used for different regions along a curve. Corners are detected by monitoring points that maintain high values of curvature as the amount of smoothing is successively increased. These approaches are computationally intensive and thus may not be suitable for interactive sketching systems. Furthermore, these methods consider only the shape of a curve. Our approach considers pen speed to help determine which discontinuities were intended.

(Sezgin 2001) has combined our pen speed approach with the scale-space method. A single scale is used for smoothing both the speed and curvature data. The corner points obtained from the smoothed speed data are selectively combined with those of the smoothed curvature data to produce the set of true corners.

(Igarashi *et al.* 1997) created an interactive beautification system. Their task is to transform the user's pen strokes into cleaned-up line segments and infer any intended connectedness, perpendicularity, congruence, and symmetry. The resulting image resembles a carefully drafted diagram despite the imprecision associated with the user's original sketch. Our task differs from theirs in that we are trying to understand the semantic content of the strokes while they focus on improving the visual structure. Also, they consider only lines (no arcs) and they require each line to be drawn with a separate pen stroke. We consider both lines and arcs and allow multiple primitives in a pen stroke.

(Rubine 1991) describes a trainable, single-stroke gesture recognizer for click and drag interfaces. A stroke is characterized by a set of 11 geometric and 2 dynamic attributes. A class of gestures is defined by a linear function of these 13 attributes. Training is accomplished by learning appropriate weights for each attributes in the linear function. The attributes consider aggregate properties of a pen stroke, and it is possible that two different gestures would have the same aggregate properties. Our approach, by contrast considers the attributes of the constituent parts of a symbol, thus providing a greater ability to discriminate between similar symbols. Furthermore, during recognition, our method is able to identify the individual primitives in a symbol. This is useful for applications in which different parts of a symbol convey different information. For example, for a beam symbol (Figure 1), it is often useful to know where the supports are.

(Fonseca & Jorge 2000) describe a method, based on fuzzy logic, for recognizing both multi-stroke and single-stroke shapes. Each shape is characterized by a number of geometric features calculated from three special polygons: 1) the smallest convex hull that can be circumscribed around the shape, 2) the largest triangle that can be inscribed in the hull, and 3) the largest quadrilateral that can be inscribed. Using the areas and perimeters of these polygons, a number of features such as thinness, hollowness and openness are computed. The system is manually trained by identifying the right fuzzy feature sets to characterize a shape and distinguish it from the other shapes. An unknown scribble is recognized by computing its degree of membership in the fuzzy set definitions of the various known shapes. Because

the method relies on aggregate features of the pen strokes, it might be difficult to differentiate between similar shapes. Also, the method is unable to identify the constituent parts of a shape.

(Landay & Myers 2001) presents an interactive sketching tool called SILK that allows designers to quickly sketch out a user interface and transform it into a fully operational system. As the designer sketches, SILK's recognizer, which is adapted from (Rubine 1991), matches the pen strokes to symbols representing various user interface components, and returns the most likely interpretation. Their recognizer is limited to single-stroke shapes drawn in certain preferred orientations. Our method handles multi-stroke shapes drawn in any orientation.

(Cohen, Huang, & Yang 1995; Huang & Cohen 1996) describe a method for matching and classifying curves using B-splines, invariant to affine transformations. This method is particularly suitable for identifying single-stroke sketches such as characters in handwritten text or gestural commands. A reported application involves matching handwritten text to a likely writer for criminal investigations. A benefit of this approach is that there is no need to segment the pen stroke. However, many of the symbols of interest to us cannot be drawn as single strokes.

Gross' Electronic Cocktail Napkin (Gross & Do 1996) employs a trainable recognizer that works for multi-stroke shapes. The recognition process is decomposed into glyph (low-level) and configuration (high-level) recognition. A glyph is described by a state transition model of the pen path, the aspect ratio and size of the bounding box, and the number of corner points. The pen path is described as a sequence of state transitions, where a state is one of the 9 regions obtained by dividing the bounding box into a 3x3 grid. Corners are identified when the change in drawing direction exceeds 45 degrees. Configuration recognition considers the spatial relationships between the glyphs. This method is sensitive to changes in orientation, and the 3x3 grid may be inadequate for symbols containing small features. Our approach is insensitive to changes in orientation and small features pose no difficulties.

Stahovich *et al.* (Stahovich 1996; Stahovich, Davis, & Shrobe 1998) have developed a program called SketchIT that can transform a sketch of a mechanical device into working designs. The program employs a novel behavioral representation called qualitative configuration space (qc-space), that captures the behavior suggested by a sketch while abstracting away the particular geometry used to suggest that behavior. Qc-space allows SketchIT to identify the geometric constraints that must be satisfied for the device to work as desired. The desired behavior is specified by the user via a state transition diagram. Once the program has identified the constraints, it uses them to synthesize new working designs. Each new design is represented as a behavior ensuring parametric model ("BEP-Model"): a parametric model augmented with constraints that ensure the overall device geometry behaves as intended. The constraints of the BEP-Model actually define a family of geometries that all produce the same set of behaviors. SketchIT is concerned only with the high-level processing of the sketch; It assumes that the lines, arcs, and symbols contained in the sketch are extracted by another program. The segmenting and recognition techniques described here could serve this purpose.

(Mankoff, Abowd, & Hudson 2000) have explored methods for modeling and resolving ambiguity in recognition-based interfaces. Drawn from a survey on existing recognizers, they present a set of ambiguity resolution strategies, called mediation techniques, and demonstrate their ideas in a program called Burlap. Their resolution strategies are concerned with how ambiguity should be presented to the user and how the user should indicate his or her intention to the software. These techniques may be useful for improving the robustness of our system.

Symbol recognition has much in common with the general problem of graph-subgraph isomorphism detection. (Ullmann 1976) is considered one of the fastest methods for general problems. For applications that require testing an unknown graph against a database of model graphs, a number of methods based on fast indexing have been proposed ((Horaud & Skordas 1988), (Ikeuchi 1987), (Spirkovska 1993), and (Messmer & Bunke 1995)). All of these methods assume that the nodes and edges in the graphs have unique labels, so that one can unambiguously determine if a particular node (or edge) in one graph is the same as that in another graph. For our application, however, the entities in the semantic networks do not have unique labels. To identify primitives (nodes) it is necessary to consider the geometric relationships (graph edges) with other primitives.

Given the success of Hidden Markov Models (HMM's) in speech recognition (Lawrence 1989), it is natural to consider if these approaches are suitable for sketch recognition. For speech recognition, sounds are inherently organized in chronological order. One cannot go back in time to change an already uttered sound. This enables a fundamental premise of the state transition dynamics of HMM's, namely recognition of the current sound depends on the previous sound, which is not subject to future alteration. For our application, however, one can, and typically does, go back to a particular spatial location and add new strokes. For this, and other reasons, approaches based on HMM's are not likely to be useful for our task.

Conclusion

We have developed a method that uses both pen speed and shape information to segment a pen stroke into constituent lines and arcs. We have found that pen speed gives insight into which discontinuities were intended by the drawer – it is often natural to slow the pen at such points. In addition, we use a smoothed curvature metric to detect other kinds of segment points, such as transitions between arcs and lines.

We have developed a trainable symbol recognizer that can recognize multi-stroke symbols. Typically only a few training examples are needed to learn an accurate recognizer. Our approach can distinguish between similar shapes and is insensitive to rotations. Depending on the training examples, our approach can learn a definition for a particular size of symbol, or a definition in which scale does not matter. If the primitives in the symbol are always drawn in the same order,

recognition and training are inexpensive. We have demonstrated that search can be used to enable recognition when the drawing order is varied.

References

Bentsson, A., and Eklundh, J. 1992. Shape representation by multiscale contour approximation. *IEEE PAMI* 13:85–93.

Cohen, F.; Huang, Z.; and Yang, Z. 1995. Invariant matching and identification of curves using b-splines curve representation. *IEEE Transactions on Image Processing* 4(1):1–10.

Dudek, G., and Tsotsos, J. 1997. Shape representation and recognition from multiscale curvature. *CVIU* 68(2):170–189.

Fonseca, M. J., and Jorge, J. A. 2000. Using Fuzzy Logic to Recognize Geometric Shapes Interactively. In *Proceedings of the 9th Int. Conference on Fuzzy Systems (FUZZ-IEEE 2000)*.

Gross, M., and Do, E. 1996. Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of UIST 96*, 183–192.

Horaud, R., and Skordas, T. 1988. Structural matching for stereo vision. In *9th ICPR88*, 439–445.

Huang, Z., and Cohen, F. 1996. Affine-invariant b-spline moments for curve matching. *IEEE Transactions on Image Processing* 5(10):1473–1480.

Igarashi, T.; Matsuoka, S.; Kawachiya, S.; and Tanaka, H. 1997. Interactive beautification: A technique for rapid geometric design. In *UIST '97*, 105–114.

Ikeuchi, K. 1987. Generating an interpretation tree from a CAD model for 3-D object recognition in bin-picking tasks. *International Journal of Computer Vision* 1(2):145–65.

Kimura, T. D.; Apte, A.; and Sengupta, S. 1994. A graphic diagram editor for pen computers. *Software Concepts and Tools* 82–95.

Kurtoglu, T., and Stahovich, T. F. 2002. Interpreting schematic sketches using physical reasoning. In *AAAI 2002 Spring Symposium Series, Sketch Understanding*.

Landay, J. A., and Myers, B. A. 2001. Sketching interfaces: Toward more human interface design. *IEEE Computer* 34(3):56–64.

Lawrence, R. 1989. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, 257–286.

Mankoff, J.; Abowd, G. D.; and Hudson, S. E. 2000. Oops: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers and Graphics* 24(6):819–834.

Messmer, B. T., and Bunke, H. 1995. Subgraph isomorphism in polynomial time. Technical Report IAM 95-003, University of Bern, Institute of Computer Science and Applied Mathematics, Bern, Switzerland.

Rattarangsi, A., and Chin, R. T. 1992. Scale-based detection of corners of planar curves. *IEEE Transactionsos Pattern Analysis and Machine Intelligence* 14(4):430–339.

Rubine, D. 1991. Specifying gestures by example. *Computer Graphics* 25:329–337.

Sezgin, T. M. 2001. Feature point detection and curve approximation for early processing of free-hand sketches. Master's thesis, Massachusetts Institute of Technology.

Spirkovska, L. 1993. Three-dimensional object recognition using similar triangles and decision trees. *Pattern Recognition* 26:727–732.

Stahovich, T. F.; Davis, R.; and Shrobe, H. 1998. Generating multiple new designs from a sketch. *Artificial Intelligence* 104(1–2):211–264.

Stahovich, T. F. 1996. Sketchit: a sketch interpretation tool for conceptual mechanism design. Technical report, MIT AI Laboratory.

Ullmann, J. R. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM* 23(1):31–42.

Weisman, L. 1999. A foundation for intelligent multimodal drawing and sketching programs. Master's thesis, MIT.

Winston, P. H. 1975. Learning structural descriptions from examples. In Winston, P. H., ed., *The Psychology of Computer Vision*. McGraw-Hill Book Company, Inc. 157–209.

Witkin, A. P. 1984. Scale space filtering: A new approach to multi-scale description. *Image Understanding* 79–95.

Zhao, R. 1993. Incremental recognition in gesture-based and syntax directed diagram editor. In *Proceedings of InterCHI'93*, 95–100.