Recognizing Interspersed Sketches Quickly

Tracy A. Hammond[*]
Sketch Recognition Lab
Department of Computer Science
Texas A&M University

Randall Davis[†]
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology

Abstract

Sketch recognition is the automated recognition of hand-drawn diagrams. When allowing users to sketch as they would naturally, users may draw shapes in an interspersed manner, starting a second shape before finishing the first. In order to provide freedom to draw interspersed shapes, an exponential combination of subshapes must be considered. Because of this, most sketch recognition systems either choose not to handle interspersing, or handle only a limited pre-defined amount of interspersing. Our goal is to eliminate such interspersing drawing constraints from the sketcher. This paper presents a high-level recognition algorithm that, while still exponential, allows for complete interspersing freedom, running in near real-time through early effective sub-tree pruning. At the core of the algorithm is an indexing technique that takes advantage of geometric sketch recognition techniques to index each shape for efficient access and fast pruning during recognition. We have stress-tested our algorithm to show that the system recognizes shapes in less than a second even with over a hundred candidate subshapes on screen.

Index Terms: H5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces—

1 Introduction

As computers have become an integral part of our lives, it is becoming increasingly important to make working with them easier and more natural. Our long-term vision is to make human-computer interaction as easy and as natural as human-human interaction. Part of this vision is to have computers understand a variety of forms of interaction that are commonly used between people, such as sketching. Computers should, for instance, be able to understand the information encoded in diagrams drawn by and for scientists and engineers.

Ordinary paper offers the freedom to sketch naturally; e.g., drawing objects with any number of strokes in any order. But sketches are of limited utility to the computer unless they can be interpreted. In an attempt to combine the freedom provided in a paper sketch with the processing capabilities of an interpreted diagram, sketch recognition systems that automatically recognize hand-drawn diagrams have been developed for many domains, including Java GUI creation [CGFJ02], UML class diagrams [HD02] [LTC00], and mechanical engineering [Alv00]. Sketch interfaces have a number of advantages. They: 1) interact more naturally than a traditional mouse-and-palette tool by allowing users to hand-sketch the diagram; 2) allows a computer to automatically understand a drawing and connect to a back-end system (such as a CAD tool) to offer real-time design advice directly from the designer's hand-drawn diagrams, thus avoiding a reproduction of work otherwise common in design; 3) recognize the shape as a whole to allow for more powerful editing; 4) beautify diagrams, removing mess and clutter; and 5) notify the sketcher that the shapes have been recognized.

Sketch recognition systems tend to recognize shapes by one of three methods (or some combination): gesture, vision, or geometry. Gesture recognition methods recognize shapes by the path of the stroke [Rub91] [Lon01] [WWL07]. In gesture-based recognition methods, either a user must learn how to draw in the style required by the system (such as in the Graffiti system used by the Palm Pilot), or each user must train the system with his or her particular drawing style. Vision-based methods allow for more drawing freedom, as shapes are recognized by what they look like rather than how they are drawn. However, in vision-based methods, shapes must be mappable to a predefined bitmap-like template. Vision-based recognition methods don't take advantage of stroke-based information that may be necessary to recognize shapes with markedly-different, but perceptually-allowable variations, such as those of an arrow. Thus, vision-based recognition methods constrain users to draw shapes so that they look similar to the template at a pixel level. Geometry-based recognition methods allow users to draw shapes as they would naturally, by recognizing shapes by the perceptually important geometric constraints that hold for a shape [AD04]. Because the authors of this paper believe in a 'walk up and draw' mentality, geometry-based recognition methods are the focus of this paper.

In geometry-based sketch recognition higher-level shapes are recognized as the combination of low-level strokes. These higher-level shapes are defined by the subshapes of which they are composed and constraints specifying how the subshapes fit together. If complete interspersing drawing freedom is allowed, in order to recognize all higher-level shapes, the recognition system must examine every possible combination of subshapes, as well as every permutation of these subshapes. This implies that any straightforward algorithm would take exponential time, which is clearly impractical for any non-trivial sketch.

Mahoney et al. [MF02] have discussed the inherent complexity involved in the structural matching search task, explaining how the problem is theoretically reducible to a constraint satisfaction subgraph problem, and thus requires exponential time to deliver a solution that effectively examines all possible interpretations. Hammond and Davis [HD05] have created a geometric-based sketch recognition system that allows complete interspersing drawing freedom. However, the system described in [HD05] quickly slows down to a halt when several ungrouped strokes (on the order of one hour of computation for 30 strokes) remain on the screen due to the exponential number of computations necessary.

To combat this problem, recognition systems have placed drawing requirements on the user, such as forcing users to draw each shape in a single stroke, such as in gesture-based recognition systems [Rub91] [Lon01] [WWL07], or requiring users to draw each shape in its entirety before starting the next shape. Some systems exist that allow limited amount of interspersing. [HD02] allows interspersing, but requires that shapes be composed of only the last 7 strokes drawn on the screen. [Sez06] allows for interspersing, but requires that examples of all possible interspersings be part of the

[*]e-mail: hammond@cs.tamu.edu
[†]e-mail:davis@csail.mit.edu

Graphics Interface Conference 2009
25-27 May, Kelowna, British Columbia, Canada

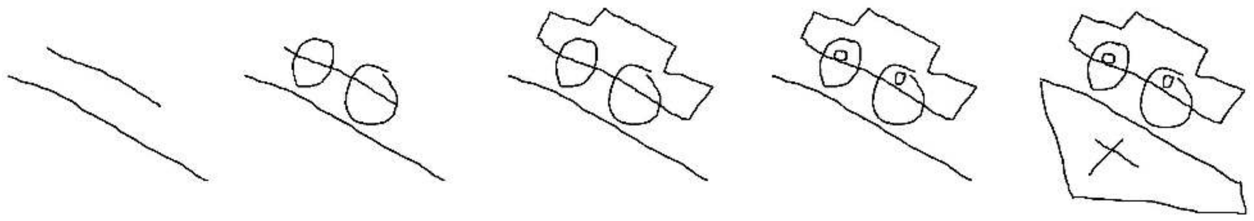


Figure 1: Five snapshots in the design of a car on a hill in mechanical engineering. Note that the top of the ground is drawn first, followed by the car base, to ensure that they are parallel. Then wheels (which act as the connection between the car and the ground) were added. The wheels were attached to the car only after the car was completed. The first started object, the ground, was completed last since its purpose in completion was more in terms of general functioning when attaching it to a CAD system than in design.

training set.

While the systems above allow limited amounts of interspersing, they still restrict the user in a way that seems unnecessary, given the fact that humans don't require such restrictions in order to recognize shapes. We cannot simply ignore the need for interspersing. In many domains, users will intersperse strokes, starting to draw one shape, before finishing another. In our experience observing users sketch in a variety of domains, we have found it not uncommon for someone to draw part of a shape, stop, continue drawing other objects in diagram, and then come back to finish the original shape. For example, In UML class diagrams, users interspersed strokes to improve overall layout. Additionally, the authors have witnessed interspersing in software design, where UML class diagram designers sometimes initially draw connections as simple dependency associations, until most of the classes have been drawn, at which point they will have a better understanding of the emerging design, and make a better decision about the type of association that would be appropriate between two objects. This will cause them to add different kinds of arrowheads to the associations drawn earlier, producing an interspersed sketch. The authors have also witnessed interspersing in electrical engineering; sketchers often add voltage directions only later in the design process after most of the circuit has been drawn. In finite state machines, arrow heads were often added as an afterthought. Figure 1 shows an example of interspersing in mechanical engineering.

Our goal is to make sketch systems as natural as possible by placing as few requirements on the user as possible. Recognition errors due to interspersing tend to be more disturbing to users than those due to messy drawing since, from the human's perspective, the shape is clearly drawn and obvious. This paper presents an indexing technique for sketch recognition that examines all *possible* subsets within the $2^{|S|}$ powerset space (where $|S|$ is the number of shapes on the screen) when attempting to recognize new shapes, but uses efficient indexing to keep recognition performance close to real-time through fast pruning. This paper also reports timing data that supports the claim that the recognition of new shapes can be kept close to real-time even when all possible shape subsets are considered.

2 Recognition

We have chosen to perform recognition using a geometric method that composes shapes hierarchically from their subshapes, testing the geometric constraints between them. For the most part, the recognition paradigm is similar to those used in [HD05]. However, our approach in that our recognition algorithm does not place interspersing requirements on the user, but still performs recognition is real-time despite an exponential query space.

2.1 Geometric Constraints

Our indexing technique works by indexing the geometric properties of a shape for use in recognition. We have created a vocabulary of geometric constraints (Table 2.1), somewhat similar to those used by [HD05], with which to describe shapes, such as

Orientation-Dependent Constraints:	HORIZONTAL, VERTICAL, NEGSLOPE, POSSLOPE, ABOVE, LEFTOF, HORIZALIGN, VERTALIGN, POINTSDOWN, POINTSLEFT, POINTSRIGHT, and POINTSUP
Orientation-Independent constraints:	ACUTE, ACUTEDIR, ACUTEMEET, BISECTS, COINCIDENT, COLLINEAR, CONCENTRIC, CONNECTS, CONTAINS, DRAWORDER, EQUALANGLE, EQUALAREA, EQUALLENGTH, INTERSECTS, LARGER, LONGER, MEETS, NEAR, OBTUSE, OBTUSEDIR, OBTUSEMEET, ONONESIDE, OPPOSITESIDE, PARALLEL, PERPENDICULAR, and SAMESIDE.
Composite Constraints (subset):	SMALLER, BELOW, RIGHTOF, ABOVELEFT, ABOVERIGHT, BELOWLEFT, BELOWRIGHT, CENTEREDABOVE, CENTEREDBELOW, CENTEREDLEFT, CENTEREDRIGHT, CENTEREDIN, LESSTHAN, and LESSTHANEQUAL
Other Constraints:	EQUAL, GREATERTHAN, GREATERTHANEQUAL, OR and NOT.

Table 1: Geometric Constraints

PERPENDICULAR. Constraints may be either orientation-dependent or orientation-independent. The vocabulary also includes EQUAL, GREATERTHAN, and GREATERTHANEQUAL, allowing comparison of any two numeric properties of a shape (e.g., stating that the height is greater than the width). The vocabulary also contains a number of constraints that can be composed from other constraints. We include these to simplify descriptions and to make them more readable. The constraint modifiers OR and NOT are also present to allow description of more complicated constraints.

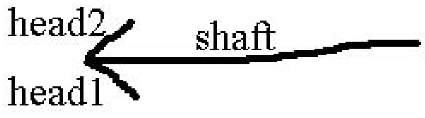


Figure 2: An arrow with an open head.

Figure 3 shows the geometric components (three lines) and the constraints that make up an OPENARROW. New shapes are defined hierarchically in terms of previously-defined shapes and the constraints between them. For example, Figure 5 shows a shape description for a TRIANGLEARROW created from an OPENARROW and a line[1]. Geometric constraints define the relationships between

[1]An *aliases* section in a shape definition is available to simplify hierarchically-defined descriptions, allowing components to be renamed for ease of use later. For example, the OPENARROW definition has aliases for

```
define shape OpenArrow
  description
    "An arrow with an open head"
  components
    Line head1
    Line shaft
    Line head2
  constraints
    coincident shaft.p1 head1.p1
    coincident shaft.p1 head2.p1
    coincident head1.p1 head2.p1
    equalLength head1 head2
    acuteMeet head1 shaft
    acuteMeet shaft head2
  aliases
    Point head shaft.p1
    Point tail shaft.p2
```

Figure 3: The geometric description to recognize an arrow with an open head, as shown in Figure 2.

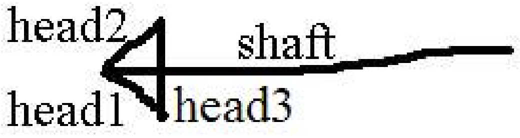


Figure 4: An arrow with a triangle head.

```
define shape TriangleArrow
  description
    "An arrow with a triangle-shaped head"
  components
    OpenArrow oa
    Line head3
  constraints
    coincident head3.p1 head1.p2
    coincident head3.p2 head2.p2
  aliases
    Line shaft oa.shaft
    Line head1 oa.head1
    Line head2 oa.head2
    Point head oa.head
    Point tail oa.tail
```

Figure 5: The geometric description to recognize an arrow with a triangle-shaped head, as shown in Figure 4.

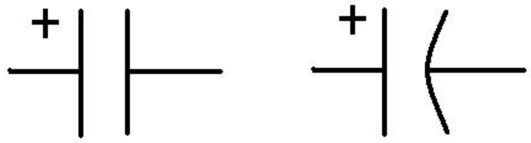


Figure 6: A capacitor can be drawn with two lines or a line and a curve.

those components. For example, the OPENARROW shape definition requires that the HEAD1 and SHAFT meet at a single point and form an acute angle from line HEAD1 to line SHAFT when traveling in a counter-clockwise direction.

2.2 Signal Noise versus Conceptual Variations

Shapes are automatically recognized based on the geometric description of the shape. The recognition handles signal noise, but conceptual variations must be included in the shape descriptions.

By signal noise we mean the unintentional deviations introduced into a shape by the imprecision of hand control. For instance, when drawing a square, all four sides may turn out to be of different lengths even though the sketcher meant for them to be the same length. By conceptual variations, we mean the allowable variations in a symbol that are drawn intentionally. For example, a capacitor in an electronic circuit may be drawn as two parallel lines, or as one straight and one curved line (see Figure 6).

In our system, signal noise is handled by the recognition system. For example, the system can successfully recognize a quadrilateral with uneven sides as a square because the EQUALLENGTH constraint has some built-in tolerance (discussed below). Thus, shapes should be described to the system without accounting for signal noise, i.e., as if drawn perfectly (e.g., a square should be described as having equal length sides). As the system does not automatically take into account the possible conceptual variations, they must be provided for in the shape descriptions (e.g., the two different ways a capacitor can be drawn must be specified in the shape descriptions).

Other signal noise includes a sketcher intending to draw a single line, but using several strokes to do so. In order for the system to deal with these phenomena, it first joins lines by merging overlapping and connecting lines. Figure 7 shows the steps that go into recognizing a square. Figure 7a shows the original strokes. Figure 7b shows the original strokes broken down into primitives. The system has recognized the strokes as lines or polylines; the figure shows the straightened lines that were recognized by the recognition system. (The dots represent their endpoints.) Figure 7c shows the primitives (line segments, in this example) joined together to form larger primitives (again lines, in this example) using the merging techniques described above. Figure 7d shows the higher-level recognition performed on the recognized shapes; the method for this is described int the next section. A higher-level shape can then use the square as one of its components.

the HEAD and TAIL to simplify the TRIANGLEARROW description.

2.3 Constraint Tolerances

In our approach, signal noise is handled by the shape recognizer by giving each constraint its own error tolerance, chosen to be as close as possible to perceptual tolerance, i.e., the tolerance that humans use. Human perceptual tolerance is context-dependent, depending on both the shape in question and other shapes on the screen. Table 2.3 shows constraints and the error tolerances we use. Note that some constraints have an absolute tolerance, while others are relative. Some constraints have a negative tolerance, which means the constraint has to be not only *geometrically satisfied*, but also *perceptually satisfied*, meaning that humans have to be able to perceive that the constraint is true. To give an example, a shape that is left of another shape by one pixel is geometrically left of another shape, but is not perceptually left of another shape, as it difficult for a human to perceive such a small distance. To ensure that a constraint is *perceptually satisfied*, we add a buffer zone to the tolerance.

Perceptual error tolerances were determined empirically and based from grouping rules and singularities from gestalt principles [Gol72]. Grouping rules attempt to mimic how people perceptually group objects, using concepts such as connectedness, nearness, and other principles. Singularities describe which geometrical shape properties are most noticeable. For instance, humans are particularly sensitive to horizontal and vertical lines and can quickly label a line as horizontal or not horizontal, and identify if a line deviates from horizontal or vertical by as little as five degrees. Humans have considerably more difficulty identifying lines at other orientations, such as 35 degrees, and would have a much more difficult time determining if a particular line was 30, 35, or 40 degrees. Humans tend to group together the angles between 15 and 75 degrees as positively-sloped lines [Gol72].

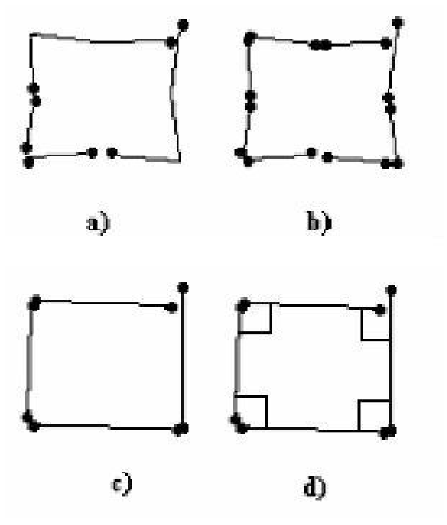


Figure 7: Stages of square recognition; a) original strokes, b) primitive shapes, c) joined/cleaned primitives, and d) higher-level recognition. [HD02]

To calibrate how sensitive people are, we showed nine people a total of 116 lines in a random orientation between 0 and 180 degrees. Users were asked to report the orientation of the line as accurately as possible. We grouped the lines into two groups 1) angles with orientations within 10 degrees of horizontal or vertical (0-10, 80-100, and 170-180 degrees) and 2) angles with orientations not within 10 degrees of horizontal or vertical (10-80, 100-170). When labeling lines from group 1 (near horizontal or vertical), users had a mean error (absolute value of the reported angle minus the actual angle) of 2.8 degrees and a variance of 4.95. When labeling lines from group 2 (far from horizontal or vertical), the users had a mean error of 7.64 and a variance of 25.77. As shown by the variance, large errors between the actual orientation and the correct orientation were common: 24 lines had an error greater than 10; 8 lines had an error greater than 15; and 2 lines had an error greater than 20. The two groups were significantly different with a p value of less than .001. Our user group contained both women and men, as well as a variety of computer scientists, other engineers, and non-mathematically concentrated professionals in the study. The groups were not statistically different in our small sample.

When drawing an ideal positively-sloped line, users tend to draw an angle close to 45 degrees and when drawing a ideal horizontal line they tend to draw an angle close to 0 degrees. However, since differences are much more perceptually important close to horizontal, we have mad the tolerance less for near singularities (such as HORIZONTAL, VERTICAL, PARALLEL, and PERPENDICULAR). Based on the mean error of only 2.8 degrees for angle singularities shown in the footnote, one may expect the singularity tolerances to be even smaller. However, because the perceptual error tolerances are used to remove possible shapes from our recognition results, we choose error tolerances that are slightly smaller.

3 Indexing Algorithm

Our current implementation is done in three stages: 1) domain-independent primitive finding, 2) domain-independent constraint indexing, and 3) domain-dependent shape formation.

3.1 Domain-Independent Primitive Finding

Low-level recognition is performed on each stroke when it is drawn. During processing, each stroke is broken down into a collection of primitive shapes, including line, arc, circle, ellipse, curve, point, and spiral, using techniques from Sezgin [SSD01]. Corners used for segmentation are found using a combination of speed and curvature data (as in Sezgin [SSD01]). By breaking strokes down into these primitives, and performing recognition with primitives, we can recognize shapes drawn using multiple strokes, and handle situations in which a single stroke was used to draw multiple shapes.

horizontal	angle	10 degrees
vertical	angle	10 degrees
posSlope	angle	35 degrees
negSlope	angle	35 degrees
coincident	x location	10 pixels
coincident	y location	10 pixels
bisects	x location	(length / 4) pixels
bisects	y location	(length / 4) pixels
near	x location	50 pixels
near	y location	50 pixels
concentric	x location	(width / 5) + 10 pixels
concentric	y location	width / 5 + 10
sameX	x location	20 pixels
sameX	width	20 pixels
sameY	y location	20 pixels
sameY	height	20 pixels
equalSize	size	(size / 4) + 20 pixels
parallel	angle	15 degrees
perpendicular	angle + 90	15 degrees
acute	angle + 40	30 degrees
obtuse	angle + 130	30 degrees
contains	min X,Y & max X,Y	-5 pixels
above	y location	-5 pixels
leftOf	x location	-5 pixels
larger	size	-5 pixels

Table 2: Constraints and their error tolerances. (Note: size = length of diagonal of bounding box. We use this formula instead of the area to enable us to compare lines to two-dimensional shapes. Rubine uses the same formula for size when calculating feature number five. [Rub91])

If a stroke has multiple primitive interpretations, all interpretations are added to a pool of interpretations, but a single interpretation is chosen for display. For example, both the LINE and ARC interpretation of the STROKE in Figure 8A will be added to the pool for recognition using any of the interpretations. (Appropriate bookkeeping is performed to ensure that multiple interpretations are effectively managed.)

In cases when a stroke is determined to be composed of several primitives, (e.g., the POLYLINE interpretation in Figure 8B), the STROKE is segmented, the segmented substrokes added as components of the original full STROKE. Further interpretations can use either the full stroke, as the CURVE does in Figure 8B, or one or more of the segmented substrokes. This allows several shapes to be drawn with a single stroke.

3.2 Domain-Independent Constraint Indexing

We would prefer to place as few drawing requirements as possible on the sketcher, and must as a consequence find a way to deal with the exponential. While our solution does not eliminate the exponential, we can use indexing to help prune a significant number of shape possibilities quickly.

The indexing process occurs only once for each shape, when it stroke is drawn or new shape is recognized. Each low level shape is indexed according only to its own properties and not in terms of those around it, and thus, the time it takes to index all of the shapes is logarithmic in terms of the number of shapes on the screen. (It is logarithmic because indexing involves the insertion of shapes into a sorted hash map, which take time of $O(logn)$.) For each new shape that is recognized, a significant portion of the recognition time is

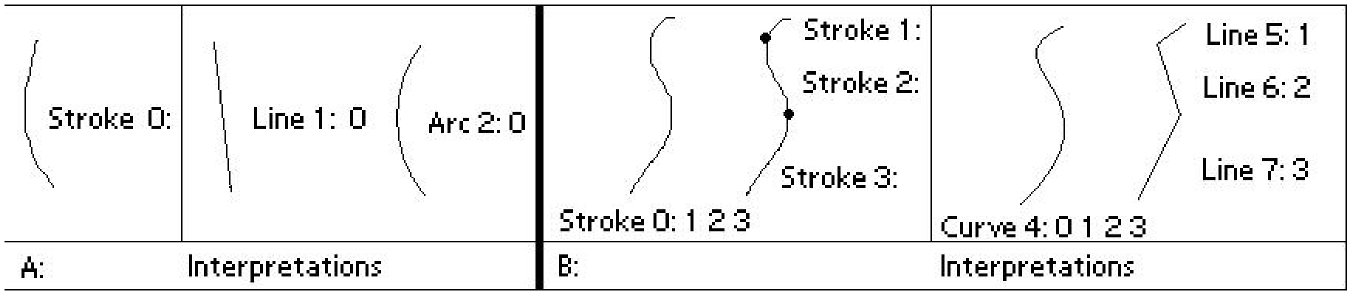


Figure 8: Multiple interpretations and their identifiers are added to the recognition pool. In A, Stroke 0 has two interpretations; Line 1 and Arc 2 are each composed from Stroke 0. In B, Stroke 0 can be broken down into three Strokes (1,2,3). Stroke 0 has two interpretations, Curve 4 composed of Stroke 0 (and thus also Strokes 1,2,3) and three lines: Line 5 composed of Stroke 1, Line 6 composed of Stroke 2, and Line 7 composed of Stroke 3.

spent performing the shape indexing. The recognition process is still exponential, as it must be if we are still to consider all possible subshapes. However, in practice, it is very fast because most of the recognition work is being performed in the indexing stage. As the work that is performed in the indexing stage need not take into account other shapes on the screen, as more shape candidates are present on the screen, recognition time increases more slowly than it would otherwise.

When a new shape is recognized, the system computes its properties, including its orientation, angle, location, and size, and places that shape in a hash map with the key being the value. Each shape property (such as x-location) has its own hash map, permitting quick retrieval of shapes with the desired property value. For instance, the angle hash map is used when searching for horizontal lines or lines of equal angle. When a line is recognized, its angle is measured and categorized as HORIZONTAL, POSSLOPE, VERTICAL, or NEGSLOPE. The category is used as the key for a hash map whose values are a linked list of shapes. This allows constant-time insertion and retrieval of a list of all of the lines in a particular orientation. We also want to find parallel lines, so exact angles are used to add the shape to a sorted map list. This allows for a logarithmic-time insertion and retrieval of the list of lines that are close to a particular angle. Similarly, location and size are also stored in a both a categorized hash map as well as a continuous sorted hash map list since it is often convenient to get a range of values depending on the constraint. Since it is faster to retrieve shapes with angles close to our predefined categories, the system does so for those constraints where it is appropriate.

The complete list of properties are indexed for each shape and for each of its components. The components are shape specific, but to give an example, the components for a line are the two endpoints, the center, and the bounding box. The list of properties for each shape is: shape type, shape name, shape angle, center-x location (note that the center-x location of an endpoint is just the x location of the endpoint itself), center-y location, min-x location, min-y location, max-x location, max-y location, size (actually the length of the diagonal of the bounding box), height, width, and stroke length.

3.3 High Level Shape Recognition

Once properties are computed and indexing has been done, the system tries to see whether a higher-level shape can be made from this new shape and shapes already on the screen. We need to check whether this new shape can be a part of any of the shapes defined in the domain description. For each shape template in the domain, the system assigns the new shape to each possible slot component. If there are n domain shapes, and each shape $\mathcal{S}$ is composed of m components ($\mathcal{C}_1$ to $\mathcal{C}_m$), then the just processed shape is assigned to each slot separately in different shape templates. Figure 9 shows an example. A newly interpreted line is added to the system. The system checks to see whether the newly interpreted line can be used

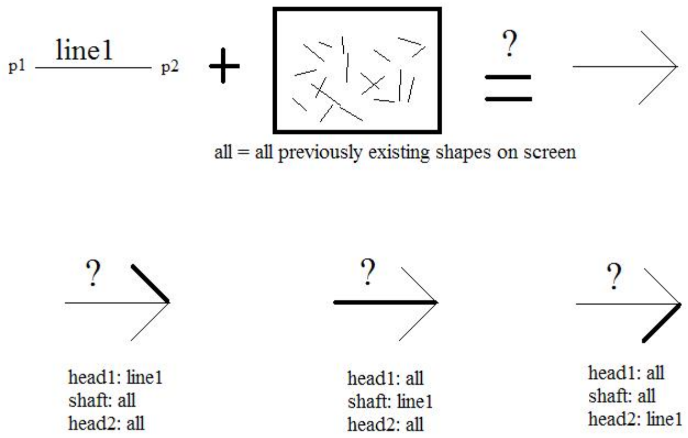


Figure 9: A template is made for each possible slot.

to create any of the shapes in the domain. (In this example, we are checking only the domain shape OPENARROW.) The system creates three templates, one for each component of the OPENARROW of the correct type (in this case a line), assigning the just processed line to a different component to see whether an OPENARROW can be formed with that component. [2]

The system then computes the function $\mathcal{L}_{ij} = f(S_i, C_j)$, which returns a list $\mathcal{L}_{ij}$ of shapes of type $\mathcal{S}_i$ and the components that make up these shapes, which can be formed with the just processed shape assigned to component $\mathcal{C}_j$. For example, if the domain description includes 10 shape descriptions, and OPENARROW is the third description, S_3 = OPENARROW (shown in Figure 3). An arrow has three slots (one for each line). If the system puts the recently drawn shape into slot 2, then C_2 = SHAFT. Thus, $\mathcal{L}_{32}$ returns a list of all of the possible OPENARROW's with the most recently drawn shape acting as the SHAFT of the stroke. The length of $\mathcal{L}_{ij}$ may be 0 (no such interpretations are possible), 1 (one interpretation is possible), or > 1 (multiple interpretations are possible, see Figure 10).

The new shape can be placed in any slot in any template provided it is of the right type. (An arc can't be placed in a line slot.) $\mathbb{P}$ is the

[2] We use three templates here for explanation simplicity, but in actuality, when a single line is drawn, the system creates six arrow templates. The recognition creates two copies of every line, one in each direction. (i.e., the second is equal to the first with the endpoints flipped.) Each directional line is assigned one at a time to each component of three arrow templates. This is actually not a phenomenon applied specifically to lines, but any group of multiple interpretations using a single subshape.

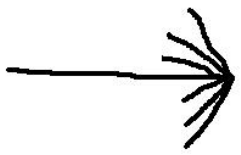


Figure 10: Multiple OPENARROW interpretations are possible using the center stroke as the SHAFT.

union of all the possible shapes formed with the new shape.

$$\mathbb{P} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (\mathcal{L}_{ij} = f(\mathcal{S}_i, \mathcal{C}_j))$$

At this point, each template currently has only one slot filled (with the new shape). To compute $\mathcal{L}_{ij} = f(S_i, C_j)$, the system assigns a list of all of the other shapes on the screen to the other slots on the template, so that each slot on the template holds the possibilities for that slot.

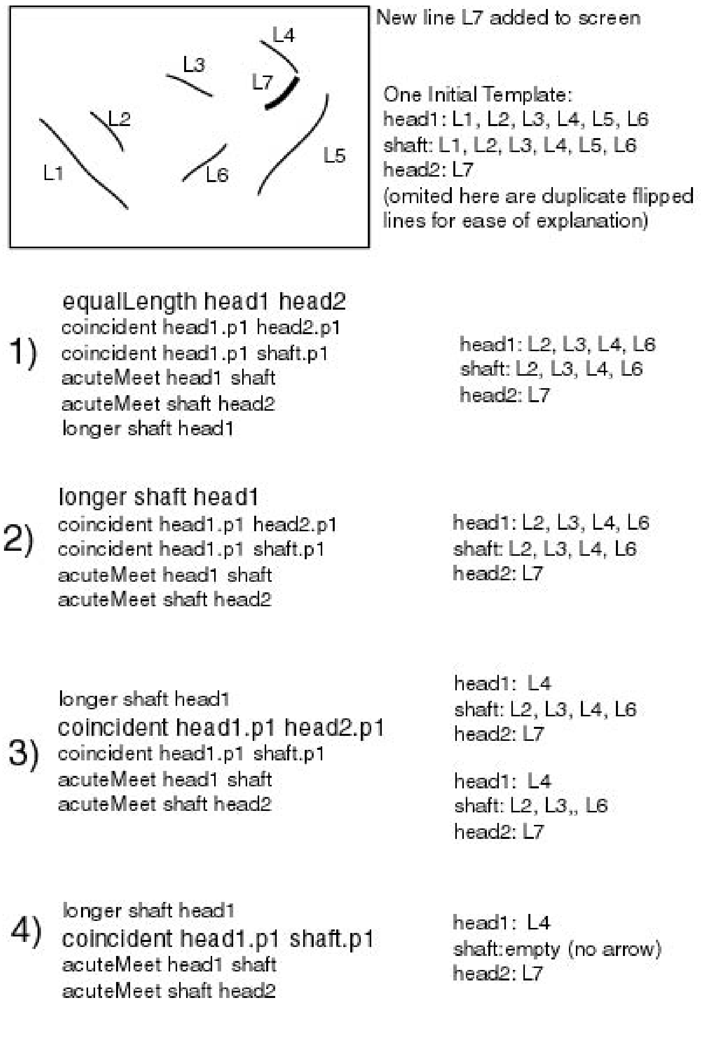
The next stage is to reduce the possibilities for each slot. This is done from the indexing data structures that were created previously. The fundamental idea behind the use of the indexed values is to allow quick pruning to prevent the templates from branching (which helps to limit the inherent exponential).

The system holds a list of unchecked constraints. One main feature of our algorithm is the smart ordering of constraint test based on the number of components in each slot. In order to remove a shape from the list of possible shapes in a slot, we have to be sure a shape cannot be formed using that shape. If only one of the component slots referenced by the constraint contains more than one possible shape, then we can determine for certain which shapes satisfy that constraint, and are allowed in the final shape composition. Let $\mathcal{T}_i$ be the list of constraints from $\mathcal{S}_i$ that are not yet solved. Let $\mathcal{O}_t$ be the number of slots for constraint t with more than one component in the possibilities list. For each constraint with only one slot (or no slots) with more than one component ($\mathcal{O}_t <= 1$), we compute the constraint t for each possible combination (which is a linear number of computations since only one slot of the constraint has multiple possibilities). We then remove all of the shapes from the appropriate slot that would make constraint t false and then remove constraint t from the list of unsolved constraints $\mathcal{T}_i$.

3.4 Pruning Possibilities by Preventing Branching

Figure 11 walks through an example. Line L7 has been added to the screen where previous shapes L1, L2, L3, L4, L5, and L6 already exist. In this example, there are several shape templates, with the smallest shape template that of the OPENARROW, composed of three lines. The reader may be left to wonder what happens before L7 is added to the screen. Initially L1 is added to the screen, but no shape templates exist that contain only a single subshape, so nothing happens. Next, L2 is drawn, and again, there are no shape templates with only two subshapes, so nothing happens. Subsequently, when L3 is drawn, there does exist a shape template with three subshapes (the OPENARROW). Up to this point we have already determined that it is impossible to make a shape with only L1 and L2 (as there weren't enough components to make up a single shape). At this point we know that *if* a shape exists on the screen it *must* contain all three lines since no shape templates exist with fewer than three lines. But more importantly and more generalizably to other cases, since all attempts have failed to make shapes out of the previously existing shapes on the screen, any shape that now exists on the screen must contain shape L3. Thus, the algorithm only attempts to make shape templates with L3. Three different shape templates (x2, see previous footnote) are created for the OPENARROW, where L3 is set as the only possibility for the SHAFT, LINE1, and LINE2 respectively. We then add both L1 and L2 to both of the other remaining two slots, leaving us with:

- Template 1:
  - shaft L3
  - head1 L1 L2
  - head2 L1 L2
- Template 2:
  - shaft L1 L2
  - head1 L3
  - head2 L1 L2
- Template 3:
  - shaft L1 L2
  - head1 L1 L2
  - L3



New line L7 added to screen

One Initial Template:
head1: L1, L2, L3, L4, L5, L6
shaft: L1, L2, L3, L4, L5, L6
head2: L7
(omited here are duplicate flipped lines for ease of explanation)

1)
equalLength head1 head2
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2
longer shaft head1

head1: L2, L3, L4, L6
shaft: L2, L3, L4, L6
head2: L7

2)
longer shaft head1
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L2, L3, L4, L6
shaft: L2, L3, L4, L6
head2: L7

3)
longer shaft head1
coincident head1.p1 head2.p1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L4
shaft: L2, L3, L4, L6
head2: L7

head1: L4
shaft: L2, L3,, L6
head2: L7

4)
longer shaft head1
coincident head1.p1 shaft.p1
acuteMeet head1 shaft
acuteMeet shaft head2

head1: L4
shaft:empty (no arrow)
head2: L7

Figure 11: An example showing how the constraint templates are pruned.

The system then runs the algorithm described in Figure 11, and finds that there are no shapes on the screen. At this point, all of the previously performed work is thrown out, and the process is repeated for L4, and so on. Note that with L4, since we were unable to make a shape with the subshape L1, L2, and L3, we know that any shape that may exist on the screen must contain L4.

Eventually, L7 is added to the screen, and we arrive at the situation in Figure 11. The top right shows the initial template when assigning the new line (with a particular orientation) to the HEAD2 slot of the arrow. Note that all other shapes on the screen are added. The system then attempts to satisfy each of the constraints, removing shapes that do not satisfy a constraint from the template. 1) The system tries to satisfy the constraint EQUALLENGTH and removes all shapes in the HEAD1 slot of the template that are not of equal length to L7. The EQUALLENGTH constraint is now completed and removed from the list of constraints yet to be satisfied. 2) The system attempts to satisfy the LONGER constraint, but since both arguments have more than one shape in the slot, the constraint is postponed. 3) The system tries to satisfy the COINCIDENT constraint and removes L2, L3, and L5 from the HEAD1 slot. Because L4 is now the only possibility for the HEAD1 slot, it is removed from the SHAFT slot, since the same shape cannot be used for both slots. 4) The system tries to satisfy the second COINCIDENT constraint, but since none of the shapes in the SHAFT slot can be coincident with L4, the SHAFT slot is empty, and the system determines that an OPENARROW cannot be formed with the new shape filling in the HEAD1 slot.

More generally, each time a new line is drawn, the system is asking, "could this line be part of any known shape built from it and all previous lines?" It does this by putting the new line is all possible slots in each template, putting all uninterpreted previously drawn lines in all the other slots, and treating the template as a constraint satisfaction problem, with the geometric constraints refining the values (the lines assigned to the slots).

Note that a question may be asked as to why a previously drawn impossible template are not already pruned from the list since these previously computed values could be saved and checked first. All previously computed templates are not saved for several reasons:

1. First off, all possible sub-templates consisting only of earlier strokes are not actually computed, since many branches are cut off very early. This means that we would first have to look up each template combination to see if we have seen it before. This lookup takes about the same amount of time as computing the constraint itself (which is also just a lookup).

2. Second, in a system with many shape templates (>10 or >50), saving all previously computed templates would require a significant amount of space that could otherwise be used more effectively. Any system that did this would either quickly run out of heap space or take a significant time sink to read and write to disk all of the necessary template information. This saving of all templates is in fact what Jess does. Looking at the Results section (Section 4) for timing comparison data, we can see that the Jess method of saving all possible templates along the way causes a time increase rather than a speed up.

3. Third, it is unclear what use, if any, previously computed partially-filled templates would be. While users do sketch in an interspersed manner, many shapes are drawn in a non-interspersed manner. In this case, previously computed templates would be of limited value once a high-level interpretation was found using a subshape in the template. This would imply that either the system takes the space-time hit for saving the templates, or the system must do a significant amount of garbage collection after every high-level interpretation, and it is unclear what the time savings would actually be.

Using our method, all incomplete templates are discarded after each shape template is examined after each stroke. This is a conscious tradeoff of, sacrificing speed for space. The Results section shows that our speed tradeoff still allows our algorithm to perform in real-time. Future work is left to see if a more effective manner (such as only a simple hashcode) of storing impossible subtemplates could be used to improve both speed and space.

Finding the shapes that do not satisfy t is a quick process when using the indexing tables formed above. For each constraint, since $\mathcal{O}_t <= 1$, only one slot is being refined at a time. Thus, the system computes the value(s) that satisfy the constraint for that slot. The system uses the indexing data structures to obtain a list of all of the shapes on the screen with that particular value(s) (e.g., for HORIZONTAL, it would retrieve all lines with an angle near 0). The intersection of this list and the list of shapes in the slot is computed, and shapes that are not in the intersection are removed from the slot.

It is possible that all shapes are removed from the slot, which implies that this shape $\mathcal{S}_i$ cannot be formed with the set of shapes in the slot, and all processing on that template is halted. This cycle is repeated until: 1) the template is determined impossible; 2) all of the constraints are solved, and each slot has only one shape in it; or 3) all of the remaining constraints have more than one component ($\mathcal{O}_t > 1$), and the cycle is stuck (see the next paragraph for what happens). After each cycle, there are some slots that contain only one shape; consistency checking occurs as the system removes these shapes from all of the other slots to make sure the same shape is not used in multiple slots.

It is possible that all of the remaining unsolved constraints have more than one component remaining ($\mathcal{O}_t > 1$). In this case, the system branches the template, choosing the slot with more than one remaining possible assignments that has the fewest such possible assignments. It makes a new copy of the template for each of the possible assignments for that slot, placing only one in each template, then continues trying to solve the remaining unsatisfied constraints on each of the templates.

This branching process can of course cause its own exponential slowdown. The system's near real-time performance results from the fact that 1) branching does not happen often because most of the shapes on the screen do not obey the required constraints, and thus, many shapes are removed from the possibility list at once. (Consider the COINCIDENT constraint. It is uncommon for many shapes to be drawn at the same location, so many possibilities are removed simultaneously from the possibilities list.) And, 2) even in the worst case where every query results in a branching factor, the process of checking the constraints is a small proportion of the overall running time (see the Results section below) This is because the exponential part of the algorithm performs only list retrievals (which has been made fast with the use of sorted hash maps and other data structures) and list intersections.

At the end of this stage, we have a list $\mathbb{P}$ of all of the shape interpretations and their components. All interpretations are added to the recognition system, but a single interpretation is chosen for display. The system chooses to display the interpretation that is composed of the largest number of primitive shapes (i.e., the solution that accounts for more data). Creating shapes with the largest number of primitive shapes also results in fewer more-complicated shapes filling the screen. For example, in Figure 12, we choose the square interpretations rather than the arrow for display purposes, as the square accounts for four primitive, simplifying the diagram to only two high-level shapes, whereas the arrow interpretation accounts for only three lines, simplifying the diagram to three high-level shapes.

The speedup of this algorithm is due to three things:

1. Most of the heavy computation is performed in the (linear-time) indexing stage.

2. Rather than first creating a list of all possible interpretations of a shape, subtrees are pruned as they are generated.

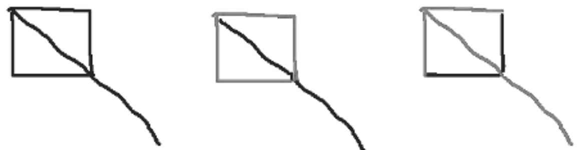


Figure 12: The left shows the originally drawn strokes. The middle interpretation is made up of a line and a square. The right interpretation is made up of an arrow and two lines.

3. Constraint testing uses a smart ordering of constraints; i.e., the constraint that could cause the biggest pruning effect is tested first.

4 Results

4.1 Stress Test

When recognizing new shapes, it is the currently unrecognized strokes that cause difficulties in recognition, as recognized strokes can be pruned away from the recognition system. Thus we tested out system through a stress test which measured the time it took for a new symbol to be recognized when there is a large amount of unrecognized strokes on the screen. All of our results were tested and measured on a tablet PC with 1GB of RAM and a 1.73 GHz processor.

To compare our algorithm against a standard caching system that tries all subset possibilities, the authors implemented an alternate version of the algorithm that instead uses the Rete algorithm in Jess [FH01] to handle the caching and pruning in recognition. The exponential inherent in testing all subsets overwhelmed the system, which slowed unacceptably after about 30 individual strokes (taking one hour to recognize the next shape).

Using the indexing algorithm described above, we stress-tested recognition results in the domain of mechanical engineering, a domain consisting of 15 distinct shapes. The the system recognized a resistor containing six lines in less than a second, with 189 other shapes (besides the six lines creating the resistor) on the screen (see Figure 13(a)), 5 of which were higher-level shapes such as resistors, and the other 186 were random lines on the screen.

Additionally, we wanted to see what percentage of recognition time focuses on each part of recognition. Our goal was to make the exponential part of the algorithm a small part of the running time of recognition. As defined above, recognition consists of 1) Corner detection or stroke fragmentation, where strokes are broken down to their composite parts, which takes $O(n)$ in terms of the number of shapes on the screen. 2) Indexing, which takes $O(logn)$ in terms of the number of shapes on the screen. 3) High-level recognition, or combining strokes to form high level objects, which is exponential in terms of the number of shapes on the screen. The goal of this paper was to minimize this third part of the algorithm.

The authors analyzed the running time of the recognition system on a diagram of over a hundred shapes, using JProbe [QS06], and determined that, with many unrecognized shapes on the screen, approximately 74% of the time was spent on corner finding. Approximately, 25% of the time was spent indexing the lines drawn. Less than 1% of the time was used to perform the high-level (exponential) recognition. Because of the indexing and early checking of constraints, the system was able to perform effective pruning and caching, and prevent the exponential aspect of recognition from causing overwhelming lag in recognition. As a result, an exponential recognition algorithm runs in what can be considered close to real-time, even with 186 shapes on the screen.

4.2 Multiple Domain Test

In additional to our stress test, we test our algorithm on three different additional domains in natural usage. Domain 1 was Japanese Kanji, consisting of 27 shapes of up to 15 primitives per shape, with an average of 10 primitives composing each shape. The total time to recognize the shapes in Figure 13(b) was 3517 milliseconds. Domain 2 was military course of action diagrams, consisting of 23 shapes, of up to 14 primitives per shape, with an average of > 7 primitives composing each shape. The total time to recognize the shapes in Figure 13(c) was 8858 milliseconds. Domain 3 was biology diagrams, consisting of 6 shapes, of up to 10 primitives per shapes, with an average of > 5 primitives composing each shape. The total time to recognize the drawn shapes in the Figure 13(d) was 481 milliseconds. Figure 13(e) shows the overall statistics combined for all three domains, with the total time equaling 15660 milliseconds. Accuracy for all three domains was unfortunately / fortunately 100%.

5 Discussion

5.1 Limitations

This algorithm can be used to describe a wide variety of shapes, but is limited to the following class of shapes.

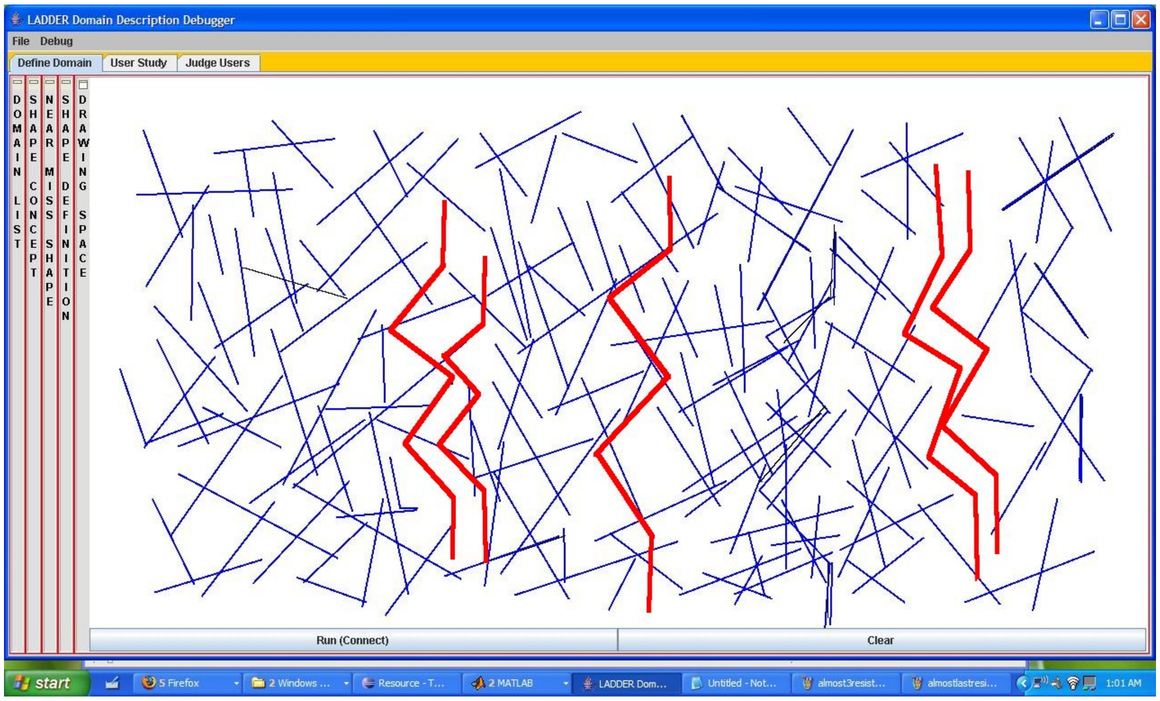
- This algorithm relies on graphical descriptions of shapes. The shapes must be drawn using the same graphical components each time. For instance, we cannot describe abstract shapes, such as people or cats, that would be drawn in an artistic drawing.
- Likewise, since this algorithm requires geometric descriptions, if a shape is highly irregular and complicated, so that it cannot be broken down into subshapes that can be described, it will be cumbersome to define.
- The shapes must be composed solely of the primitive constraints that we define in this paper, and these constraints must be enough to ensure that all shapes are appropriately differentiable from the other shapes in the domain.
- Pragmatically, it is difficult to describe complexly curved shapes. Curves are inherently difficult to describe in detail because of the difficulty in specifying a curve's control points. Future work includes investigating more intuitive ways of describing curves.

5.2 Recognition Limitations

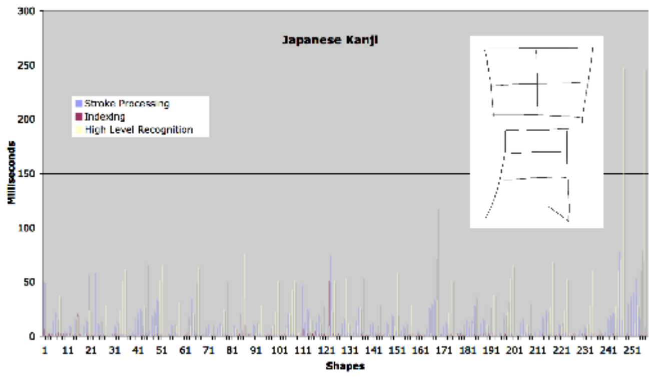
- We are currently limited to recognize only those shapes that are describable using the constraints above.
- It is possible to contrive an example that will cause our approach to branch very often, causing significant slowdown. This will arise if someone draws similar shapes repeatedly on top of themselves. The system would have a very large number of branches, because it would be difficult to eliminate a large number of possibilities in a slot for a single constraint.
- Shapes that are highly symmetric will cause many interpretations that look identical but that have different line assignments. For example, consider a square. The square can be rotated and each of the lines flipped, and thus depending on the form of the description, sixteen interpretations may exist. All of these interpretations must be kept because higher-level shapes formed from that square may require the labeling that occurs in any one of the recognitions. Thus, the number of shapes in the recognition pool can grow exponentially as shapes are recognized. Luckily, in practice, most interpretations are ruled out based on the shape description.

6 Previous Work

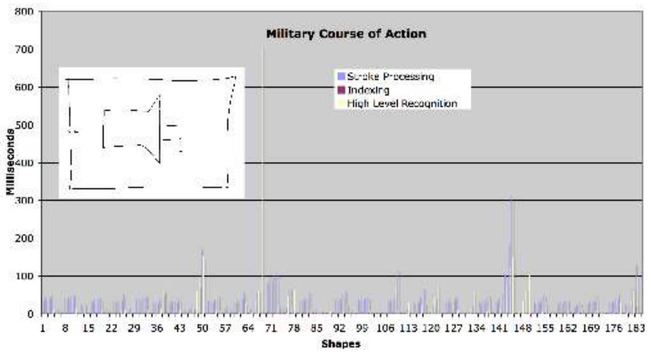
Indexing data to provide speed-up is of course not a new idea. It has been used, for example, in the closely related field of vision object recognition [SAB97] [SM92] [AKJ02] [LVB*93]. Indexing



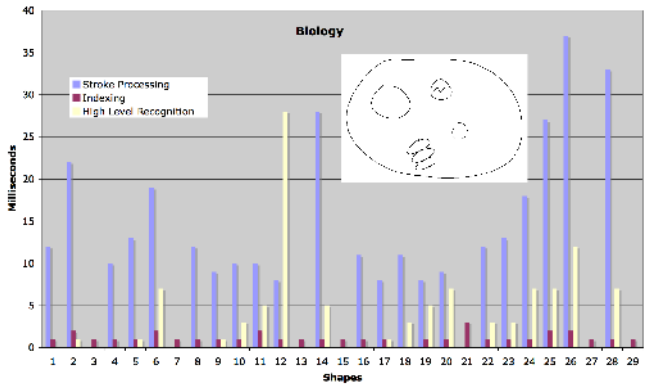
(a) Recognition Stress Test



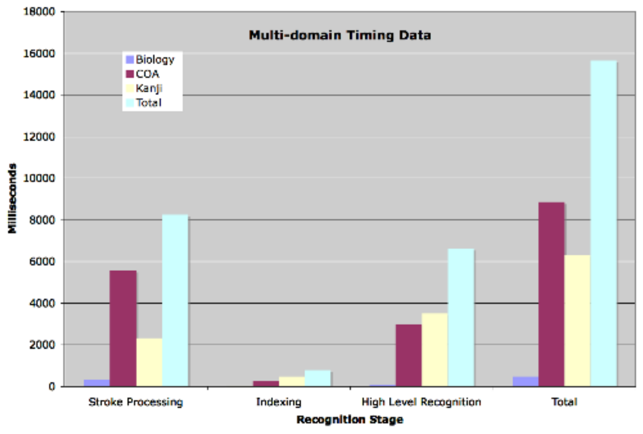
(b) Recognition times for Japanese Kanji



(c) Recognition times for Military Course of Action Diagrams



(d) Recognition Times for Biology



(e) Recognition times for all three domains collated

Figure 13: Recognition Results

sketches to search for photos has been done by [BP97] [KKOH92]. Indexing has also been used in sketch recognition to index software design meetings [HGDS02]. This work appears to be the first use of the idea in support of unconstrained sketch recognition. By breaking the strokes down to line, curve, and ellipse segments, we were able to define shapes in geometric terms, then perform indexing on these terms, a luxury not afforded when indexing photo images or non-geometric sketches (such as an artist's sketch).

7 Future Work

We would like to remove some of the limitations described above by combining the techniques presented in this paper with those that have proved useful in work in vision. By processing and indexing vision features used for recognition, and concurrently indexing on geometric properties as described in this paper, we can quickly access shapes that have the needed visual and geometric features. As a very simple example, vision recognition techniques can easily locate areas of high density ink, or shading, which we are currently not able to recognize using our geometric recognition techniques. We would like to combine vision- and sketch-based features to perform more robust recognition and perhaps recognize a larger class of objects.

8 Conclusion

This paper describes an indexing algorithm for use in sketch recognition. Shapes are indexed as they are being drawn or and when they are recognized, using a vocabulary of geometric properties that permits fast lookup subsequently. Geometric constraint indexing allows us to search through all possible subsets of the shapes on the screen quickly, facilitating recognition of higher-level shapes. Because we can quickly search through all possible subset shapes, we no longer have to limit the search space and can allow sketchers to intersperse the drawing of incomplete shapes with little speed penalty. By enabling a less constrained drawing style, we provide a more natural sketch recognition user interface. Our results show that the system continues to run in near real-time, even when a large number of stroke candidates exist as possible sub-components of a shape.

9 Acknowledgements

This paper is supported, in part, by NSF IIS Creative IT Grant #0757557. Editing comments were appreciated from Jacob Eisenstein, Paul Bogen, Brandon Paulson, Katie Dahmen, Mark Eaton, Henry Choi, and Joshua Peschel. The authors would like to thank the help of Paul Taele and Brian Eoff in the completion of figures for this paper. Additionally, they would like to thank the members of the Sketch Recognition Lab at TAMU and the members and alumni of the Design Rational Group at MIT. Additionally, as this work was a significant part of [Ham07], the authors would like to thank those who helped helped in that endeavor.

References

[AD04] Alvarado C., Davis R.: Sketchread: A multi-domain sketch recognition engine. In *Proceedings of UIST '04* (2004), pp. 23–32.

[AKJ02] Antani S., Kasturi R., Jain R.: A survey on the use of pattern recognition methods for abstraction, indexing, and retrieval of images and video. *Pattern Recognition 35* (2002), 945–965.

[Alv00] Alvarado C.: *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Master's thesis, MIT, 2000.

[BP97] Bimbo A. D., Pala P.: Visual image retrieval by elastic matching of user sketches. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19(2)* (1997), 121–132.

[CGFJ02] Caetano A., Goulart N., Fonseca M., Jorge J.: JavaSketchIt: Issues in sketching the look of user interfaces. *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium* (2002).

[FH01] Friedman-Hill E.: Jess, the java expert system shell. http://herzberg.ca.sandia.gov/jess, 2001.

[Gol72] Goldmeier E.: Similarity in visually perceived forms. In *Psychological Issues* (1972), vol. 8:1.

[Ham07] Hammond T.: Ladder: A perceptually-based language to simplify sketch recognition user interface development. PhD Thesis, Massachusetts Institute of Technology, 2007.

[HD02] Hammond T., Davis R.: Tahuti: A geometrical sketch recognition system for UML class diagrams. *AAAI Spring Symposium on Sketch Understanding* (March 25-27 2002), 59–68.

[HD05] Hammond T., Davis R.: LADDER, a sketching language for user interface developers. *Elsevier, Computers and Graphics 28* (2005), 518–532.

[HGDS02] Hammond T., Gajos K., Davis R., Shrobe H.: An agent-based system for capturing and indexing software design meetings. In *Proceedings of International Workshop on Agents In Design, WAID'02* (2002).

[KKOH92] Kato T., Kurita T., Otsu N., Hirata K.: A sketch retrieval method for full color image databases - query by visual example. *11th IAPA International Conference on Pattern Recognition* (1992), 530–533.

[Lon01] Long A. C.: *Quill: a Gesture Design Tool for Pen-based User Interfaces*. EECS department, computer science division, U. C. Berkeley, Berkeley, California, December 2001.

[LTC00] Lank E., Thorley J. S., Chen S. J.-S.: An interactive system for recognizing hand drawn UML diagrams. In *Proceedings for CASCON 2000* (2000), p. 7.

[LVB*93] Lades M., Vorbruggen J., Buhmann J., Lange J., von der Malsburg C., Wurtz R., Konen W.: Distortion invariant object recognition in the dynamic linkarchitecture. *IEEE Transactions on Computers 42(3)* (March 1993), 300–311.

[MF02] Mahoney J. V., Fromherz M. P. J.: Three main concerns in sketch recognition and an approach to addressing them. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium* (Stanford, California, March 25-27 2002), AAAI Press, pp. 105–112.

[QS06] Quest Software I.: JProbe. website, http://www.quest.com/jprobe, 2006.

[Rub91] Rubine D.: Specifying gestures by example. In *Computer Graphics* (1991), vol. 25(4), pp. 329–337.

[SAB97] Stevens M. R., Anderson C. W., Beveridge J. R.: Efficient indexing for object recognition using large networks. In *Proceedings of IEEE International Conference on Neural Networks* (1997).

[Sez06] Sezgin T. M.: *Sketch Interpretation Using Multiscale Stochastic Models of Temporal Patterns*. PhD thesis, Massachusetts Institute of Technology, May 2006.

[SM92] Stein F., Medioni G.: Structural indexing: Efficient 3-d object recognition. *IEEE Transaction on Pattern Analysis And Machine Intelligence* (1992), 125–125.

[SSD01] Sezgin T. M., Stahovich T., Davis R.: Sketch based interfaces: Early processing for sketch understanding. In *The Proceedings of 2001 Perceptive User Interfaces Workshop (PUI'01)* (Orlando, FL, November 2001).

[WWL07] Wobbrock J., Wilson A., Li Y.: Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of UIST* (2007).