Recognizing Sketched Multistroke Primitives

TRACY HAMMOND and BRANDON PAULSON, Texas A&M University

Sketch recognition attempts to interpret the hand-sketched markings made by users on an electronic medium. Through recognition, sketches and diagrams can be interpreted and sent to simulators or other meaningful analyzers. Primitives are the basic building block shapes used by high-level visual grammars to describe the symbols of a given sketch domain. However, one limitation of these primitive recognizers is that they often only support basic shapes drawn with a single stroke. Furthermore, recognizers that do support multistroke primitives place additional constraints on users, such as temporal timeouts or modal button presses to signal shape completion. The goal of this research is twofold. First, we wanted to determine the drawing habits of most users. Our studies found multistroke primitives to be more prevalent than multiple primitives drawn in a single stroke. Additionally, our studies confirmed that threading is less frequent when there are more sides to a figure. Next, we developed an algorithm that is capable of recognizing multistroke primitives without requiring special drawing constraints. The algorithm uses a graph-building and search technique that takes advantage of Tarjan's linear search algorithm, along with principles to determine the goodness of a fit. Our novel, constraint-free recognizer achieves accuracy rates of 96% on freely-drawn primitives.

Categories and Subject Descriptors: I.5.5 [**Pattern Recognition**]: Implementation—*Interactive systems*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Input devices and strategies*

General Terms: Algorithms, Experimentation, Human Factors, Performance

Additional Key Words and Phrases: Sketch recognition, primitive recognition, neural networks, intelligent user interfaces

ACM Reference Format:
Hammond, T. and Paulson, B. 2011. Recognizing sketched multistroke primitives. ACM Trans. Interact. Intell. Syst. 1, 1, Article 4 (October 2011), 34 pages.
DOI = 10.1145/2030365.2030369 http://doi.acm.org/10.1145/2030365.2030369

1. INTRODUCTION

Sketch recognition encompasses computer algorithms that interpret the hand-drawn markings created by users through electronic, pen-based mediums like Tablet PCs, pen displays, and electronic whiteboards. Using this computer-understanding, researchers can create tools that provide a richer user experience and benefit many fields, including design, engineering, and education. Low-fidelity, hand-drawn designs can now be brought to life with real-time, high-fidelity feedback and simulation.

1.1. Sketch Recognition in Design

Current designers often find themselves sketching out a rough design on paper, then translating that paper sketch into a computer-aided design (CAD) model in order to perform some form of simulation to test the viability of their design. In order to

This work was funded in part by Google, DARPA, and NSF grants 0757557, 0935219 and 0942400.
Contact author's address: T. Hammond, Sketch Recognition Lab, 3112 TAMU, College Station, TX 77845; e-mail: hammond@cse.tamu.edu.

DOI 10.1145/2030365.2030369 http://doi.acm.org/10.1145/2030365.2030369

remove this extra step of translating paper sketches into computer-understandable forms through CAD tools, some designers opt to work directly with the toolbar-based software. However, researchers have discovered that not only do "designers spend too much time working with the tool and not enough time exploring design ideas" [Bailey and Konstan 2003; Goel 1995], but that constrained-input and cleaned diagrams often hinder the creativity of a designer [Black 1990]. Computer-aided design (CAD) systems that accept freely-drawn diagrams for input may encourage greater originality [Wong 1992; Yeung et al. 2008]. Sketch recognition will allow designers to rapidly conceive their ideas using familiar materials like the pen, yet be able to perform computational processes without requiring a shifting of modes from paper to computer. Sketch recognition will allow designers to continue working in a low-fidelity space, yet achieve high-fidelity feedback.

1.2. Sketch Recognition in Engineering and Education

Sketch recognition has already proved itself to be a valuable tool in the field of education. Some domains that use sketch systems for educational purposes include UML class diagrams [Hammond and Davis 2002]; circuit diagrams [Alvarado and Davis 2004]; geography [Paulson et al. 2008]; mechanical engineering [Alvarado and Davis 2001; Kurtoglu and Stahovich 2002; Oltmans and Davis 2001; Stahovich 1996]; mathematics [LaViola and Zeleznik 2004]; chemistry [Tenneson and Becker 2005; Ouyang and Davis 2007]; Kanji [Taele and Hammond 2008, 2009]; biology [Taele et al. 2009]; and civil engineering [Peschel and Hammond 2008]. A future goal of sketch recognition researchers is the development of accurate recognizers which will allow hand-sketched diagrams to be graded for examinations. Because the correction of hand-drawn diagrams takes a significant amount of instructor time, they are typically omitted from traditional exams, and are replaced with easier to grade, multiple-choice questions. This is unfortunate because not only do diagrams test a student's problem-solving ability, but they are also widely used in the industry where he or she may one day be employed. Sketch recognition systems can also be used to enhance higher-level sketch understanding systems, such as that of CogSketch [Forbus et al. 2008], to reduce the amount of manual labeling required to use such a system.

1.3. Sketch Recognition for Domain Practicality

Course-of-action diagrams is one domain where sketch recognition could prove to be useful, due to the practicalities of the domain. Course-of-action diagrams are military planning diagrams which often need to be updated in the field. However, keyboards are impractical in the field, because of the sand and other elements, and keyboards are difficult to operate with military-issue gloves, making pen-input computer a better fit. Thus, most field-issued computers, such as those used by the Blue Force Tracker and the FBCB2 are Tablet PCs—specifically, Toughbook Slate computers. Quickset is a pen and voice interface that recognizes primitive symbols through the use of neural networks and hidden Markov models [Pittman et al. 1996]. Estimates from these two recognition processes are combined to determine the probability for each gesture. A sketch-based solution was developed by Hammond et al. [2010] which recognizes over 900 different COA shapes; a newer version is being developed using the system described in this article.

1.4. Sketch Recognition Overview

Recognizing an entire sketch is a complicated task because sketches are "ambiguous," "dense," and "replete" [Goel 1995]. To facilitate this task, primitives are used to describe the symbols of a given sketch domain. Determining what defines a shape to be a "primitive" or "nonprimitive" shape is an open debate. In this article, we define primitives as those shapes that cannot be divided easily into other shapes (such as the infinity, wave, spiral, and helix shapes) and/or those shapes that are perceptually important, and thus, are drawn with less precision (e.g., arrows, diamonds, and rectangles). It is mathematically impossible for infinity signs, ellipses, spirals, helixes, curves, waves, or arcs to be formed from one another. Each beautified shape is formed from a mathematical equation that is distinct from the other, and thus each shape is its own primitive. To give a few examples, the formula for a beautified ellipse is essentially a circle with a major, a, and minor, b, axis: $[(x^2)/(b^2)] + [(y^2)/(a^2)] = 1$. The formula for a point on a beautified helix is essentially a circle with a moving center: $x = r * cos(k * t)$, $y = r * sin(k * t)$. The formula for a point on a beautified (Archimedean) spiral is essentially a circle with an increasing (or decreasing) radius: $x = rt * cos(t)$, $y = rt * sin(t)$ It is possible to describe an ellipse, spiral, and helix from a new generalized shape that would allow for a major and minor axis, an increasing radius, and a moving center, however, this shape would be difficult for a human to specify or to use to compose higher-level shapes.

While rectangles, diamonds, and arrows *can* easily be formed from other primitives (specifically four lines, or two lines and a path), these shapes are included as "syntactic sugar" because they are perceptually very common and expected. As such, they are drawn more sloppily than other polylines, with the corners hastily drawn such that they often appear as strong curves or arcs rather than a series of polylines. Similarly, a circle is also included as syntactic sugar, due to its perceptual importance, even though it can be formed from an ellipse.

Additionally, most arrows drawn in diagrammatic domains do not consist of three straight lines, but rather may have a curvy path representing the shaft of the arrow. As most domains use arrows to suggest a path from one place to another, that path is rarely drawn straight. In the case of course of action diagrams, the path represents the ideal route to be taken by troops or a plane. In the case of finite state machines and UML class diagrams, that path is often curved, or even circular in the case of self-loops, to improve visualization. This work attempts to include all perceptually recognizable arrows as part of the arrow primitive class.

If a stroke consists of multiple primitives where at least one of the primitives is not a line, we call that shape a *complex shape*. Thus, neither a polyline nor a polygon is a complex shape because both consist only of lines. In this article we make the assumption that either a single stroke consists of a whole number of primitives (e.g., a single stroke containing both a line and a circle) or that a single primitive is composed of a whole number of strokes (e.g., a circle drawn with two arc strokes). For instance, we assume a user would *not* draw a first stroke containing a line and an arc, with the second stroke intending to close the arc into a circle.

Because a sketch typically contains multiple objects and forms of information, it is commonplace to use hierarchical schemes to perform recognition. To give an example, a UML class diagram gives a snapshot of the inner workings of an entire piece of software. This diagram is made up of class objects and interactions, which are represented as rectangles and arrows. These rectangles and arrows can be broken down further into a set of lines. And, taken to the extreme, lines can be broken down even further into points. The term *high-level recognition* refers to the process of finding meaningful, domain-specific symbols within a drawn sketch (e.g., a class object, an anchored body, or an OR gate). *Low-level* or *primitive recognition* refers to the process of interpreting an individual stroke, or group of strokes, as a basic, domain-ignorant shape (e.g., line, ellipse, curve). The differences in labeling between high-level and low-level recognition can be seen in Figure 1.

Many sketch recognition systems have employed a framework similar to the one shown in Figure 2. In this framework, drawn strokes are interpreted as one or more supported shape primitive. These recognized primitives are then given to the

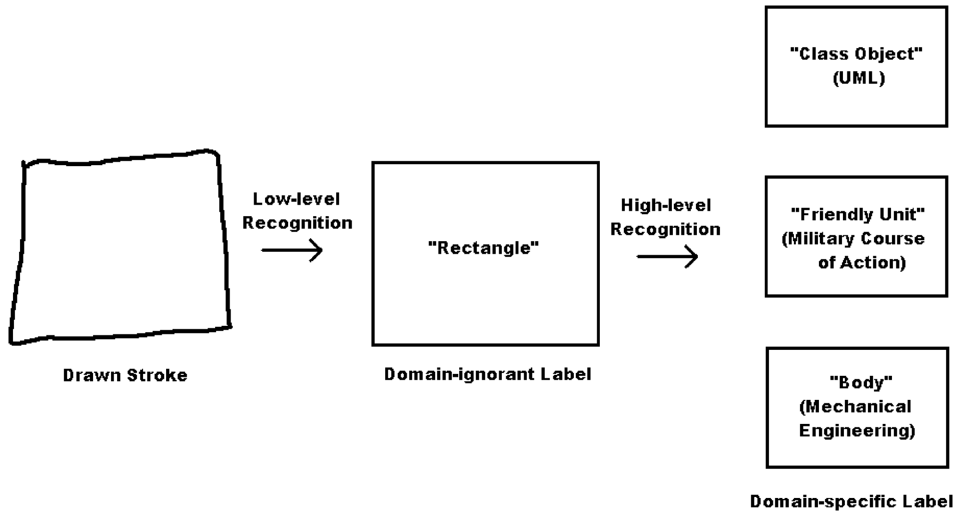


Fig. 1. Example of the distinction in labeling between low-level and high-level recognizers. Low-level recognizers assign domain-ignorant labels while high-level recognizers assign domain-specific labels.

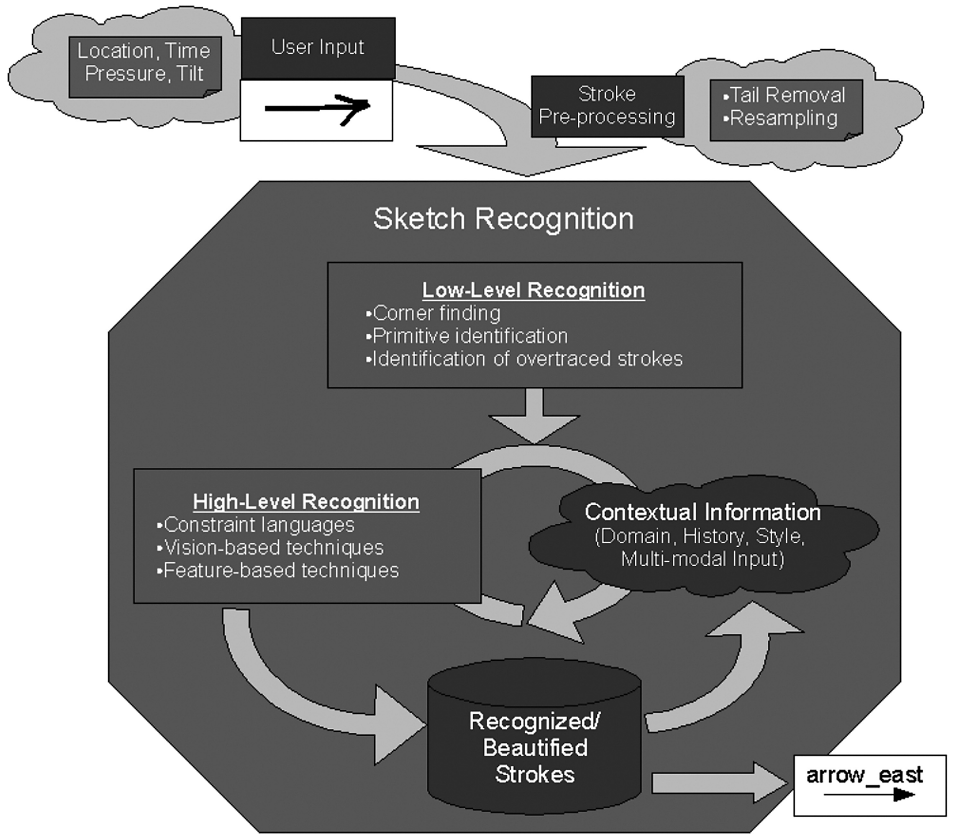

Location, Time Pressure, Tilt
User Input
Stroke Pre-processing
•Tail Removal •Resampling
Sketch Recognition
Low-Level Recognition
•Corner finding
•Primitive identification
•Identification of overtraced strokes
High-Level Recognition
•Constraint languages
•Vision-based techniques
•Feature-based techniques
Contextual Information (Domain, History, Style, Multi-modal Input)
Recognized/ Beautified Strokes
arrow_east

Fig. 2. Example sketch recognition framework.

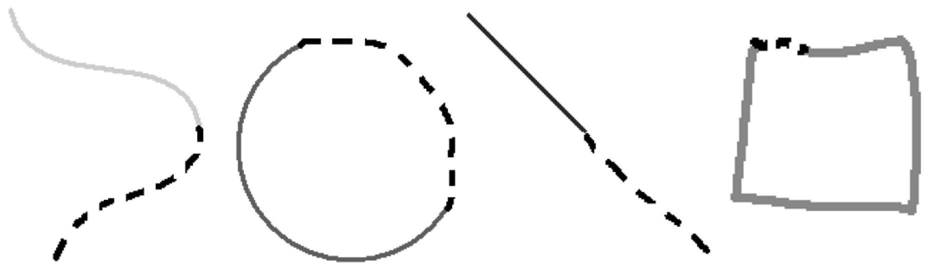


Fig. 3. Examples of continuation strokes.

high-level recognizer which attempts to identify domain-specific constructs. In order to support a large variety of sketch domains, several high-level algorithms have utilized visual grammars to define the symbols of a given domain [Hammond and Davis 2005; Pasternak and Neumann 1993; Futrelle and Nikolakis 1995; Calhoun et al. 2002; Costagliola et al. 2004, 2005]. These grammars define symbols in terms of a set of shapes that meet particular spatial and geometric constraints.

1.5. Why Multistroke Primitives?

Mostly all recognizers assume that any primitive shape will be drawn with a single stroke. Although this assumption typically holds true, there are many instances in which a user may choose to draw a primitive shape with more than one stroke. We call these additional strokes *continuation strokes* (Figure 3).

The reason continuation strokes are problematic is twofold. First and foremost is that continuation strokes may alter the meaning of the original stroke (e.g., changing it from an arc to a circle; or extending the length of a line significantly), and may alter high-level recognition. Second, continuation strokes may cause additional complications because high-level recognizers assume that every primitive has meaning. For example, if a user draws a line to the screen and decides later in the sketching process to make it longer, the low-level recognizer will usually find and return two individual lines to the high-level recognizer. At this point, the high-level recognizer will attempt to recognize high-level shapes that consist of two lines rather than one.

As another example, imagine that a user draws a rectangle to the screen, but does not fully close the space between the endpoints (as seen in Figure 3). The user then mistakenly attempts to help the recognizer by adding an additional stroke to close the gap. We also call these continuation strokes *touch-up strokes*. Most likely, the primitive recognizer will still be able to classify the first stroke as a rectangle, but will classify the second stroke as a line. Now, the high-level recognizer has an additional line that it is incorrectly attempting to recognize as part of a high-level shape.

The ultimate goal of our work is to place no constraints on how users must draw primitive shapes—allowing primitives to be drawn with multiple interspersed strokes, without direction or order requirements. We believe it is the job of the low-level recognizer to handle multistroke primitive recognition because it is the part of the multi-tiered sketch process that has knowledge of basic shapes. The high-level recognizer only has knowledge of spatial and geometric relationships between recognized primitives.

1.6. Algorithmic Assumptions

The algorithm described in this article relies on several assumptions about the context of the domain in which it is to be used, the shapes in the domain, and the allowable variations with which they are to be drawn, the types of high-level sketch recognition algorithms to be employed, the ink collection methods, and other user interface issues regarding how the algorithm is to be used.

(1) This algorithm is only appropriate when attempting to recognize diagrams built up from primitives drawn from traditional pen down to pen up strokes. It is not meant for use in free-form diagrams with shading. However, this technique has been shown to work well in less traditional domains such as facial drawing [Dixon 2009], provided it is used appropriately.
(2) This algorithm is also meant to work in a hierarchical sketch recognition system that combines low-level primitives into high-level shapes using geometrical rules. This algorithm would not be necessary when using pixel-based, or other non-stroke-based methods, as described in the Previous Work section, under Appearance-based Methods.
(3) This algorithm expects that only one of three situations will occur when drawing a primitives: (a) multiple primitives are drawn completely with a single stroke; (b) a single primitive is drawn with a single stroke; (c) multiple strokes are drawn to draw a single primitive. While this algorithm could be expanded to include a case where a single stroke draws one and a half primitives, it would cause an increase in running time, which could prove to be impractical for use with complicated diagrams. In practice such strokes are rare, and did not occur in our user data. In the small instances where they do occur, users will instead have to redraw their primitive for it to be recognized using our algorithm in order to keep the algorithm operating in real-time.
(4) This algorithm assumes that strokes to be merged will have endpoints somewhat near each other. This may cause some difficulty in cases where strokes to be merged have significant overlap, causing the endpoints to no longer be near to one another.
(5) The authors of this article are attempting to solve a subproblem of the larger problem of free-sketch recognition, which allows people to walk up to a system, draw as they would naturally, and have the system recognize it just as a human could. Other valid scenarios for pen-input systems may allow for one or more of the following constraints, and may instead choose to (a) train the user; (b) have the user train the system; (c) require the user to shapes in the same order; (d) require the user to finish drawing one shape before drawing another; (e) wait a predetermined amount of time after drawing each shape; (f) draw each shape on a separate panel; (g) press a button when finished; (h) press a symbol button on a palette before drawing the shape. This article attempts to solve the problem of multistroke primitive recognition without requiring the user to follow any of these constraints.

1.7. Contributions of this Article

The primary contributions in this article include the following.

(1) Results from a data collection scenario show that multistroke primitives are a significant part of freely-drawn sketches and that the recognition of multistroke primitives is an important problem to be addressed. In fact, results show that multi-stroke primitives are more probable than complex strokes (a single stroke used to draw multiple connected primitives), which is a highly-researched problem (in contrast to the dearth of research surrounding multistroke primitives).
(2) The article includes the description of a novel algorithm to identify and recognize multistroke primitives via a unique graph building and searching technique that takes advantage of Tarjan's linear search algorithm.
(3) A list of principles is provided to help determine the goodness of fit.

(4) An evaluation of this algorithm shows that it performs significantly better than existing algorithms in either number of primitives recognized, speed of recognition, or both.
(5) Results from the data collection show that certain shapes are recognized with higher accuracy when drawn with multiple strokes and vice-versa. This data, along with author-provided insight as to where the current algorithm fails, could provide insight into how to create better recognizers in the future.
(6) Results from the data collection scenario confirm the results of van Sommers [1984], who stated that when there are more sides to a figure, less threading (drawing multiple lines with a single stroke) occurs.

2. PREVIOUS WORK

Low-level (primitive) recognition is the process of assigning a domain-ignorant label to a stroke or group of strokes. This label represents the name of one of many primitive shapes which are used as part of the building block vocabulary for high-level visual grammars. As the number of supported primitive shapes increases, so too does the expressiveness of the high-level shape grammar. Thus, it is important to not only have accurate primitive recognizers, but to also have primitive recognizers that support many different shapes. Low-level recognizers have traditionally been broken down into three categories: *motion-based*, *appearance-based*, and *geometric-based*, each of which has advantages and disadvantages.

2.1. Motion-based (Gesture-Based Recognition)

Gesture recognition concerns itself primarily with the path of the stroke. Gesture-based recognizers got their start from algorithms meant to interpret gestures that represented either editing commands [Rubine 1991; Long Jr. et al. 2000; Zeleznik and Miller 2006] or alphanumeric characters [Goldberg and Richardson 1993; Newman and Sproull 1973; Blickenstorfer 1995; Li and Yeung 1997], but have also been used as primitive recognizers in some sketch systems [Plimmer and Freeman 2007]. For example, Gross and Do [1996] recognize primitives using a 3×3 grid structure. Because these gesture sets are typically large and may not be obvious initially, there is often a learning curve associated with new users. Gesture-based recognizers concern themselves with classifying shapes based on how the individual strokes were drawn, and not on what the stroke actually looks like (although there is usually a correlation). Therefore, these types of algorithms often place constraints on how users must draw particular shapes in order for the shapes to be recognized. For example, a user may naturally draw circles using a clockwise motion, but may be forced to draw circles in a counter-clockwise direction in order for the circles to be recognized. Drawing habits like these are hard for new users to break. Users are often also required to pretrain the system, and must provide numerous example sketches before using the application. The advantage of trained gesture-based recognizers is that they can be quite accurate when shapes of the same class are always drawn with the same, predefined motion. Further improvements include Wobbrock et al.'s [2007] template-matching algorithm which can handle differences in orientation by rotating symbols along their "indicative angle," that is the angle between the centroid of the stroke and its starting point [Wobbrock et al. 2007], thus allowing for more variation than traditional gesture-based recognizers while maintaining high accuracy rates.

2.2. Appearance-Based Algorithms

Appearance-based algorithms focus primarily on what a sketched shape looks like; the timing and ordering of points are usually ignored, with the strokes translated into their bitmap counterparts before performing recognition. Appearance-based algorithms

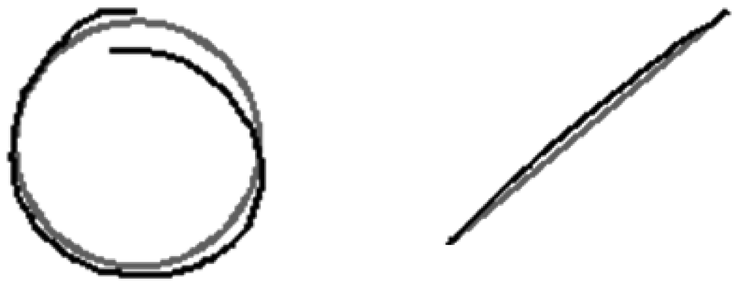


Fig. 4. Geometric-based methods estimate an ideal shape representation (red) of the stroke (black) and then compute shape approximation errors to perform recognition. For example, a circle can be estimated by finding an approximate center (e.g., the center of the bounding box) and an approximate radius (e.g., by taking the average distance of each stroke point to the center). Lines can be estimated by simply connecting the endpoints of the stroke.

often use some form of template-matching to compare a candidate symbol to others in a learned catalog. When performing template matching, many gesture algorithms will use a simple path-based Euclidean distance metric, while the appearance-based may use point-based distance metrics like the Hausdorff distance [Kara and Stahovich 2004; Wolin et al. 2009]. Hse and Newton performed recognition using Zernike moments [Hse and Newton 2005]. Oltmans used shape context to match shapes against a bullseye-shaped template [Oltmans 2007]. Rather than perform template matching directly on a particular sketch, Ouyang and Davis gained higher accuracy by performing template matching on feature images that are generated based on stroke orientation along a set of reference angles [Ouyang and Davis 2009].

Although these appearance-based systems have the advantage of increased drawing flexibility and extensibility, the main disadvantage of these appearance-based recognizers has been their inability to handle shapes that can be drawn in an arbitrary number of configurations; thus, they require a separate template for each variation of a shape. Imagine an arrow that can have an unlimited number of acceptable paths (e.g., straight line, curved, or squiggled). As mentioned earlier, we define a primitive arrow to allow the shaft to specify any path. As such, it would be impractical (and impossible) to have all possible path trajectories in our sample data in advance. Essentially, the training space for these arbitrary symbols would be infinite, requiring the user to provide explicit examples of every possible configuration of the shape. In most domains, this would be boundless and unreasonable.

2.3. Geometric-Based Recognizers

Geometric-based recognizers attempt use geometric formulas to describe primitives (see Figure 4). A benefit of this approach is that primitive shapes can be automatically beautified because the parameters needed to perform beautification are also used for recognition. Sezgin et al. [2001] presented a single-stroke algorithm that estimated beautified versions of primitive shapes and compared them to the originally-drawn strokes using an orthogonal-distance squared-error metric. The system was limited to only a few primitive shapes: lines, curves, ellipses, and complex fits (those composed of a combination of lines and curves). Yu and Cai presented an alternative single-stroke primitive recognizer that utilized their novel feature area error metrics [Yu and Cai 2003]. The shape set of the Yu and Cai recognizer included lines, polylines, circles, ellipses, arcs, and helixes. PaleoSketch further expanded upon the work mentioned above to recognize upwards of nine primitive shapes with 98% accuracy [Paulson and Hammond 2008b].

	CALI	HHR	Rub	Long	$1	Comb	Pal	All	CCM
Arc	0.99	0.95	0.48	0.65	0.98	0.89	0.99	0.95	0.95
Circle	0.93	0.71	0.76	0.74	0.97	0.94	0.90	0.95	0.95
Complex	0.80	0.81	0.47	0.38	0.93	0.73	0.84	0.90	0.97
Curve	0.91	0.83	0.69	0.78	0.85	0.98	0.94	0.97	0.95
Ellipse	0.99	0.46	0.77	0.94	0.95	1.0	0.99	1.0	1.0
Helix	0.97	0.93	0.90	0.95	0.96	0.97	1.0	0.99	0.99
Line	1.0	0.99	0.95	0.94	0.69	1.0	1.0	1.0	1.0
Polyline	0.85	0.54	0.52	0.62	0.56	0.75	0.97	0.96	0.99
Spiral	1.0	0.98	0.94	1.0	1.0	1.0	1.0	1.0	1.0
Avg.	93.8%	80.0%	72.0%	77.8%	87.7%	91.8%	95.9%	96.9%	97.8%

Fig. 5. Accuracy results for the different feature sets using a multilayer perceptron classifier. The first four columns represent the accuracies of the five individual feature sets CALI [Fonseca et al. 2002]; HHReco [Hse and Newton 2005]; Rubine [Rubine 1991]; and Long [Long Jr. et al. 2000]. Column *$1* shows the accuracy using the $1 [Wobbrock et al. 2007] algorithm. Column *Comb* refers to the combined feature set of CALI, HHReco, and Long. Column *Pal* refers to the recognition accuracy using the PaleoSketch feature set [Paulson and Hammond 2008b]. All refers to the combination of the Combined feature set, plus Paleo features. Column *CCM* uses the same feature set as All, but also utilizes the complex confidence modification algorithm [Paulson 2010].

2.4. PaleoSketch

PaleoSketch is a precursor to the work described in this article which expanded on the prior work described above to recognize upwards of nine primitive shapes with 98% accuracy [Paulson and Hammond 2008b]. That work, along with that of Paulson et al. [2008], showed the difficult in recognizing primitives using only gesture-based or appearance-based methods, as gesture-based algorithms placed too much restriction on the way that things were drawn, and appearance-based algorithms placed too much emphasis on what the shapes looked like. Grandma, the Rubine recognizer, is a classic gesture-based recognizer and performed quite poorly on primitive shapes due to the various ways that people draw shapes (see Figure 5). Paulson et al. [2008] tested all known features and selected the optimal features for recognition using feature subset selection. Notice that geometric features fare better than traditional gesture or vision features.

2.5. Perception-based Groupers

Due to the fact that the pen is used both to create elements and to modify elements, many HCI studies have focused on how to edit sketch elements once they have been drawn (e.g., scale, translate, rotate, cut, copy, delete). Saund and Moran's PerSketch system focused on utilizing the principles of perceptual organization in order to perform complex editing tasks [Saund and Moran 1995]. PerSketch followed a draw/select/modify paradigm, and used a modal button press to switch between drawing and editing modes. Saund and Lank would later show that a modeless switch was possible using pen trajectory and context [Saund and Lank 2003]. The principles of perceptual organization were also used in Scan-Scribe to form composite groupings of primitive sketch elements, and also shown to be promising for grouping text strokes [Saund et al. 2002].

2.6. Multistroke Primitive Recognizers

The biggest obstacle to achieving multistroke primitive recognition at the low-level stage of recognition is developing adequate grouping algorithms for detecting strokes that make up a single primitive while remaining domain-ignorant. By itself, grouping strokes is an exponentially hard problem because every stroke must be compared with every other stroke on the screen. Some researchers have looked at ways of speeding

1. Endpoint to stroke length ratio (100%)	**12. Curve least squares error (90%)**	**23. Spiral fit: avg. radius/bounding box radius ratio (60%)**	34. Length of bounding box diagonal (20%)
2. Normalized distance between direction extremes - NDDE (90%)	**13. Polyline fit: # of sub-strokes (70%)**	**24. Spiral fit: center closeness error (70%)**	35. Angle of the bounding box diagonal (40%)
3. Direction change ratio - DCR (90%)	**14. Polyline fit: percent of sub-strokes pass line test (50%)**	25. Spiral fit: max distance between consecutive centers (20%)	36. Distance between endpoints (10%)
4. Slope of the direction graph (20%)	**15. Polyline feature area error (80%)**	26. Spiral fit: average radius estimate (10%)	37. Cosine of angle between endpoints (0%)
5. Maximum curvature (40%)	16. Polyline least squares error (30%)	27. Spiral fit: radius test passed (1.0 or 0.0) (40%)	38. Sine of angle between endpoints (10%)
6. Average curvature (30%)	17. Ellipse fit: major axis length estimate (20%)	**28. Complex fit: # of sub-fits (60%)**	39. Total stroke length (20%)
7. # of corners (30%)	18. Ellipse fit: minor axis length estimate (30%)	**29. Complex fit: # of non-polyline primitives (50%)**	**40. Total rotation (100%)**
8. Line least squares error (0%)	19. Ellipse feature area error (10%)	**30. Complex fit: percent of sub-fits that are lines (90%)**	41. Absolute rotation (10%)
9. Line feature area error (40%)	20. Circle fit: radius estimate (30%)	**31. Complex score / rank (50%)**	42. Rotation squared (10%)
10. Arc fit: radius estimate (0%)	**21. Circle fit: major axis to minor axis ratio (80%)**	32. Cosine of the starting angle (30%)	43. Maximum speed (20%)
11. Arc feature area error (20%)	22. Circle feature area error (0%)	33. Sine of the starting angle (10%)	44. Total time (30%)

Fig. 6. This table above lists the features tested in Paulson et al. [2008]. The implementation details for features 1-31 and 32-44 can be found in Paulson and Hammond [2008b] and Rubine [1991], respectively. The bold features were chosen as the optimal subset for recognition through subset selection, with the percentage values indicating how often a feature was chosen as optimal via various folds of subset selection. Table data from Paulson et al. [2008].

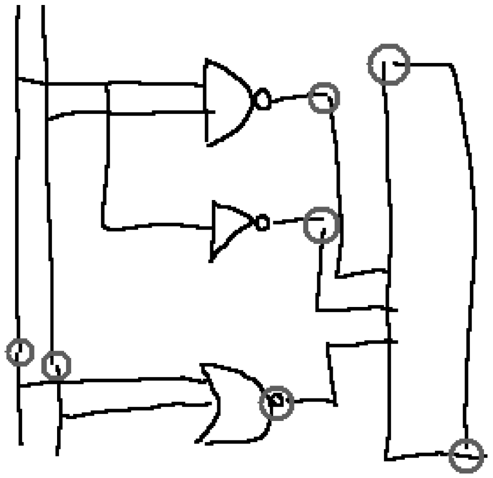


Fig. 7. Examples of continuation strokes in a hand-sketched circuit diagram.

the process up through effective pruning, caching, and data structures, but the process is still exponential in the worst case [Hammond and Davis 2009]. The graph-building method used in our multistroke algorithm is an approach often used to identify the configuration of strokes within a sketch. Mahoney and Fromherz implemented a similar method to interpret linearly-composed configurations; however, in this case graphs were used to represent line segments with two nodes representing the endpoints and the edge representing the connection between segments [Mahoney and Fromherz 2002].

When dealing with sketched input, building a data graph can be a challenge; gaps in strokes can lead to an insufficiently connected data graph. One way that researchers have attempted to solve the multistroke problem is by placing constraints on how users draw shapes with multiple strokes. For example, some algorithms use temporal constraints, or timeouts, to denote shape completion [Apte et al. 1993; Fonseca and Jorge 2000]. With these systems, users must either draw an entire shape within a specified time, or make continuation strokes within a given time frame to be recognized. Other systems utilize simple button clicks to designate when a shape is finished and ready to be recognized [Calhoun et al. 2002]. Another constraint used by some systems is to disallow stroke interspersal [Zhao 1993]. That is, if the user draws shape A and then draws shape B, she is not allowed to make additional changes to shape A. Shape A must be fully completed before the user can continue drawing any other shape. In domains like circuit diagrams, lines (wires) are often extended after other shapes have already been drawn (Figure 7).

Primitives are usually expected and intended to be drawn with a single stroke, and their accidental nature is what causes some of the difficulty in recognizing multistroke primitives. Technically, the $N gesture recognizer [Anthony and Wobbrock 2010] could be used to recognize multistroke primitives. However, the $N algorithm has the disadvantage that (1) $N can't recognize stroke order variations when a shape is drawn with more than the expected number of strokes; (2) $N will not be able to handle arrows with varied paths unless every possibly path shape is included and in an appropriate ratio to the head size, and $N will have similar difficulties with polygons and dots; (3) $N will not be able to handle curves, waves, helixes, and spirals that have more waves or cycles than the template; and (4) each shape has to be drawn in isolation. While we can make fixes for (1) and (4) by trying all possible directions and by using the graph-building techniques described in this article, (2) and (3) would require a different core method of recognition.

Finally, some algorithms compare the slopes and spatial distances between strokes, but only work for shapes composed of lines. For example, the Tahuti recognizer supports interspersing drawn primitives, but only supports multistroke rectangles and arrows made from lines [Hammond and Davis 2002]. The goal of this article is to explore a potential means for recognizing multistroke primitives, linear and curvilinear, without requiring these special constraints.

2.7. Recognizing Multistroke Primitives with a High-Level Recognition System

Many high-level recognition systems exist that recognize symbols by a combination of geometric rules to combine shapes [Hammond and Davis 2005b; Mahoney and Fromherz 2001; Gross and Do 2000], along with methods to rectify the natural ambiguity present in sketched diagrams [Johnston and Hammond 2010; Alvarado and Davis 2004; Shilman et al. 2002]. The authors originally attempted to recognize primitives recursively using the LADDER sketch language. An object model was shown to work for simple shapes such as continuation lines [Hammond 2007]. In LADDER, continuation lines can be defined using a recursive (or nonrecursive if only one continuation is allowed) context, as follows.

```
(Line l
  (components
    (Line l1)
    (Line l2)
  )
  (constraints
    (coincident l1.p1 l2.p1)
    (parallel l1 l2)
    (closer l1.p1 l2.p1 l2.p2)
    (closer l1.p2 l2.p1 l2.p2)
  )
  (aliases
    (Point p1 l1.p2)
    (point p2 l2.p2)
  )
)
```

Note that the constraints ensure that the endpoints meet, the opposite endpoints extend away from each other, and that the lines are parallel. However straightforward this case may seem, the situation becomes significantly more difficult when joining two curves, arcs, or other types of objects. It is almost impossible with the current set of constraints to determine if two curves should be joined without calculating their *isCurve* error after joining them together. Calculating the composed curve, including calculating their joining point and dealing with overlapping points, is also significantly more difficult than in the case of a line. A line by definition is straight, so defining the composed line is as simple as connecting the new endpoints. This is not the case for a curve, for which additional points must be specified. Also, while multistroke rectangles, diamonds, and arrows are definable using a shape grammar, the high-level recognizer may have difficulty recognizing the composed shapes, as the primitive “lines” may instead be recognized as curves or arcs due to the inherent sloppiness that generally accompanies these perceptually prominent shapes.

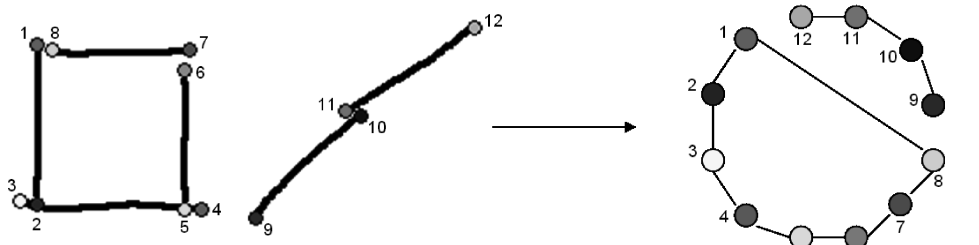


Fig. 8. Example graph built from a set of six strokes representing two primitive shapes. Note that the nodes on the left are numbered and colored to match one-to-one the nodes on the left. An edge is constructed between two endpoints of a single line (such as in 1-2, 3-4, 5-6, 7-8, 9-10, 11-12) or if the two endpoints are close together (as in 2-3, 4-5, 6-7, 8-1, 10-11).

3. IMPLEMENTATION

Our multistroke algorithm first requires a set of strokes as input to the low-level recognizer. These can be all of the strokes on the screen (as in this article), or just a subset of the total strokes if the high-level recognizer employs its own form of stroke grouping in Hammond and Davis [2009]. The low-level algorithm returns, as output, an optimal set of recognized primitives. The low-level recognizer maintains a hash table history, keyed to the unique ID of each stroke, to avoid redundant recognition of individual strokes.

Before starting multistroke primitive recognition, the multistroke primitive recognizer first classifies each individual stroke using a traditional single-stroke primitive recognizer. For this work, we have chosen to use a neural network version of our PaleoSketch algorithm [Paulson 2010]. The original version of PaleoSketch recognized nine primitive shapes, including lines, arcs, curves, ellipses, circle, polylines, spirals, helixes, and complex shapes (those that contain multiple primitives rather than all lines in a single stroke) [Paulson and Hammond 2008b] via a set of heuristic rules to rank shape interpretations. The neural network version of PaleoSketch produces confidence values associated with each shape interpretation. This version also recognizes additional primitive shapes such as rectangles, diamonds, polygons, dots, arrows, waves and infinity signs [Paulson 2010]. Throughout the rest of this article, when we refer to "PaleoSketch," we are talking about the neural network version.

At this point, the multistroke algorithm has five parts: graph building, graph searching, stroke merging, false positive removal, and arrow detection.

3.1. Graph Building

The first step of our multistroke algorithm involves constructing a graph of spatially close strokes (Figure 8). Note that strokes with complex interpretations are not added to the graph because of our assumption that a single primitive will be drawn with multiple, shapes or vice versa. We make the assumption that candidate strokes for multistroke primitives will likely have endpoints that are near one another. In our graph, nodes represent the endpoints of strokes and edges represent endpoints that are spatially near or lie on the same stroke. As each stroke is drawn, the endpoints of the stroke are added as nodes to the graph, and an edge is added between the two nodes of that stroke because they are connected by that stroke. Additionally, for each endpoint node of the recently drawn stroke, an edge is added between it and any nodes representing nearby endpoints from other strokes. To step through the example in Figure 8: first, *stroke1* is drawn with endpoints 1 and 2. Endpoint 1 is added as *node1*. Endpoint 1 is not close to any existing endpoints, so no edges are added. Endpoint 2 is added as *node2*. An edge is added between *node1* and *node2* because they are part of the same stroke. No endpoints are close to endpoint 2, so no further edges are added. *Stroke2* is drawn with endpoints 3 and 4. *Node3* is added, representing endpoint 3, along with an edge connecting it to *node2* because endpoint 3 is close to endpoint 2. *Node4* is added, along with an edge connecting it to *node3* because they are part of the same stroke, and so on. Figure 8 shows the final graph built from the set of six strokes forming two primitive shapes (a square and a line). The graph edges constructed between two endpoints of a single line include 1-2, 3-4, 5-6, 7-8, 9-10, and 11-12, while the graph edges constructed because two endpoints are close together include 2-3, 4-5, 6-7, 8-1, and 10-11.

To determine if the endpoints of two different strokes are near, the following conditions must be met.

(1) The Euclidean distance between the endpoints, divided by the average stroke length of the two strokes, must be within some threshold (we used 0.1[1] of the average stroke length). Thus, the gap between the two strokes must be less than 10% of the average stroke length of the two strokes. A similar metric was used in the original version of PaleoSketch to determine if a stroke represented a closed shape [Paulson 2010], where this threshold was tested and evaluated on thousands of additional shapes.

(2) The Euclidean distance between the endpoints, divided by the average width (or height, whichever is largest) of the bounding boxes of the two strokes, must be within some threshold (we used 0.15[2]). This confirms that the stroke endpoints are perceptually near each other, even in the case where their bounding boxes are small but their stroke length is large. To give an example, two filled-in dots may have very long stroke lengths but are not perceptually close, given their bounding boxes.

In both of these distance metrics, we divided by measures of stroke length or area. We did this so that larger strokes (by length and area) are allowed to have larger gaps between them, while smaller strokes should have smaller gaps. In the case of very small strokes, which are common in continuation lines, there is a minimum endpoint threshold of 10 pixels, else small lines would never be combined with larger lines. If the distance between the two endpoints is smaller than 10 pixels, then the endpoints are automatically considered near. In these cases, we use the static pixel threshold, rather than the tests based on stroke size, to allow for small strokes to be considered connected as well. We found multiple instances of small strokes at the beginning of much larger strokes. This typically occurs by accident when the user begins to sketch. With this static threshold, however, such errant strokes can be absorbed into the larger strokes.

3.2. Graph Searching

Once our graph has been generated, we search it for *strongly connected components* (Figure 9). Strongly connected components are the maximal strongly connected subgraphs within a graph. By "strongly connected," we mean that from every node in the graph there exists some path to every other node in the graph. Typically, these strongly connected subgraphs will indicate candidate strokes that need to be merged into multistroke primitives.

To search for strongly connected components, we utilize Tarjan's algorithm [Tarjan 1972]. Tarjan's algorithm is a depth-first search algorithm and runs in linear time in

[1]Thresholds were determined empirically with an original set of author-drawn data, and then confirmed on user-drawn data of 10 outside users and 2,471 primitive shapes.
[2]Tested similarly to the threshold above.

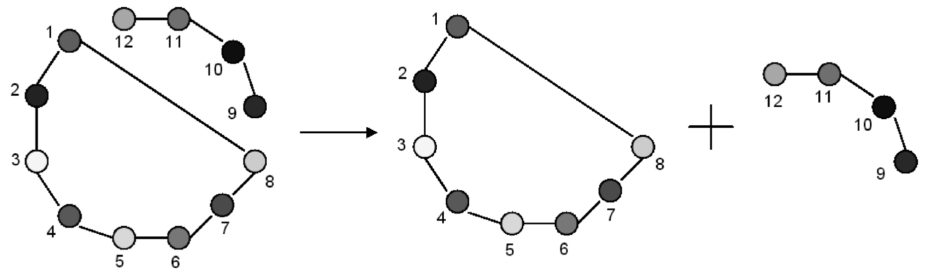


Fig. 9. We use a searching algorithm to find the strongly connected components of the graph. They correspond to the strokes that are likely candidates to be merged.

the number of edges present in the graph. The solution shown in Figure 9 is the sole solution produced by Tarjan's algorithm.

The sample images in Figure 8 are somewhat isolated shapes. However,the algorithm will still work on larger connected shapes. Tarjan's algorithm runs in linear time based on the number of edges in the graph. The largest number of edges exist when all of the strokes form a clique, with every stroke endpoint connected to every other endpoint. The maximum number of edges in a clique is $n(n-1)/2$ or $O(n^2)$ edges, where n is the number of original strokes. Note that even a highly connected diagram such as a circuit diagram like that in Figure 7 has significantly fewer edges than a clique, more on the order of $O(n)$ edges. A diagram congested near the point of being a clique is very unlikely, and if one were to exist, it would be close to unparsable by a human eye. Even so, the maximum time for the Tarjan algorithm to run would still be $O(n^2)$, where n is the number of strokes.

We then compute all of the possible subgraphs found by Tarjan's algorithm. In the case that the drawing is a clique, searching all possible subgraphs would be the powerset of the existing strokes, which is exponential. Note, however, that in practice this algorithm runs in real time due to (a) the algorithm is run on a stroke-by-stroke-basis as the strokes come in, so only subgraphs that contain the most recently drawn stroke will be included; (b) the algorithm is greedy, and replaces strokes with merged strokes as matches are found; (c) the actual computation at each step (running PaleoSketch on the merged stroke) is quite fast and can run thousands of such comparisons in under a second; (d) strokes with complex interpretations are not added to the mix, adding sparcity to the graph; and (e) the worst case is very improbable, since it is very rare for more than five strokes to meet at a single point. Additionally, if it happens that the diagram is so large that this becomes a problem, and one wishes to use it solely as a preprocessing step, one could limit the size of subgraphs to six strokes per primitive, which is a more than a reasonable number for almost any domain.

Each of the subgraphs found by Tarjan's algorithm represents possible stroke *merging*. Once all possible subgraphs have been discovered, we rank the subgraphs from those containing the most nodes to those containing the least nodes. Each merge in the list likely has overlapping strokes with the other possible merges in the list. The example in Figure 9 shows only the top two connected components present in the list of possible merges. All possible subgraphs of these two components will also be present in the list, but by ranking the list, more likely merges get processed first.

Next, we greedily iterate through the list of possible merges. The strokes of each possible merge are merged using the algorithm in the next step (*Stroke Merging*). This merged stroke is then classified using PaleoSketch. The stroke is then sent to the *false positive removal* stage, which determines if the stroke should remained merged or not. If the stroke passes this stage, we have successfully recognized a multistroke primitive.

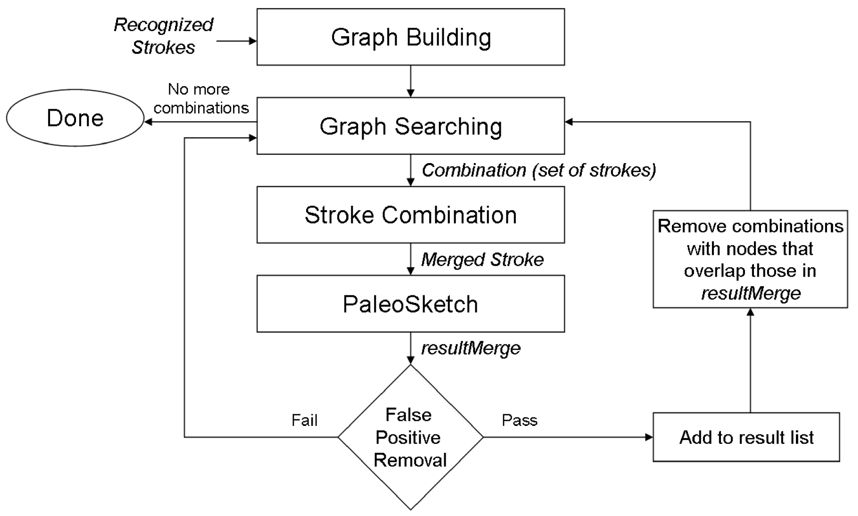


Fig. 10. Multistroke recognition flowchart.

All remaining possible merges in the list that contain any of the nodes that were used for this primitive are removed. If the stroke fails the *false positive removal* stage, then the merge fails and the algorithm continues with the next possible merge in the list. Figure 10 shows a flowchart of this process.

Figure 9 shows the strongly connected components from Figure 8. If we assume the algorithm is run on a stroke-by-stroke basis, then only the subgraphs containing the last drawn stroke need be considered, since all other subgraphs will have been attempted previously. Thus, whether or not the square on the left was merged properly, processing *stroke 11-12* will take the same amount of time. When *stroke 11-12* is drawn on the page, only two subgraphs are processed, the one containing only the last *stroke 11-12*, and the other containing the last two *strokes, 10-11* and *11-12*. In practice, after each time step, the diagram in Figure 8 would contain the following primitives: Time 1: *Line 1-2*; Time 2: *Polyline 1-4*; Time 3: *Polyline 1-6*; Time 4: *Square 1-8*; Time 5: *Square 1-8, Line 9-10*; Time 6: *Square 1-8, Line 9-12*. Thus, in practice, due to the greedy merging of primitives, this algorithm runs quite quickly.

3.3. Stroke Merging

Stroke merging represents a significant step in this recognition process. Without the proper grouping, the low-level recognizer cannot correctly classify the resulting stroke. Initially, stroke merging seems like a trivial problem, but some consideration must be taken before merging strokes. First, we must ensure that the strokes are in a logical order. This order may not necessarily be the one in which they were drawn. For example, imagine that a user draws a rectangle with four lines. The first line drawn is the left side of the rectangle (L), followed by the bottom (B), then the top (T), and finally the right side (R). In this case, it is not logical to order them based on time ($LBTR$), because we would make an incorrect connection between the endpoints of the top and bottom strokes. Instead, we want a logical merge, such as $LBRT$ or $LTRB$; we also need to concern ourselves with the logical direction of the merged strokes.

Another issue we must consider when merging strokes is that connecting the endpoints directly may not be the best option when merging two strokes. Algorithms such as PaleoSketch rely heavily on the consistency of direction and curvature graphs. If we

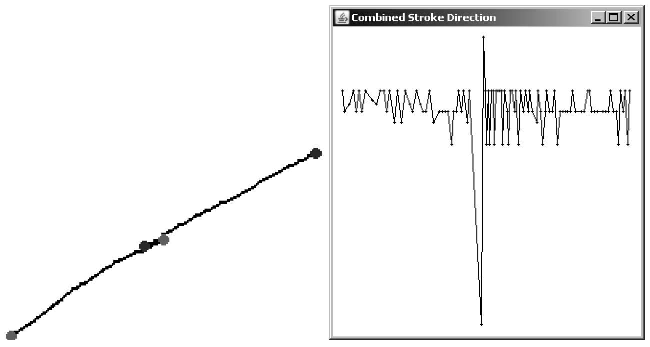


Fig. 11. Example of two lines that are merged by simply connecting the endpoints. In this case, the direction graph contains a discontinuity, even though lines typically have a smooth direction graph.

simply connect the endpoints of two strokes that slightly overlap, then we will have a discontinuity in the merged stroke's direction graph, as shown in Figure 11.

Therefore, if we are connecting the last endpoint of stroke A with the first endpoint of stroke B, we start by searching for the two points (within the last 10% of stroke A and first 10% of stroke B) that are closest to one another. We then connect the strokes at these two points, clipping the trailing/leading portions of the two strokes at the ends. If the strokes overlap, then this clipped portion will coincide with the overlapping sections. If the two strokes have a gap between them, then the two points that are closest to each other would naturally be the endpoints. In this case, it is safe to simply connect them.

3.4. False Positive Removal

Once a stroke has been generated from the merging of other strokes, we want to determine if the resulting classification of that stroke is better than the result of simply leaving the strokes unmerged. Essentially, these are rules for when not to merge strokes. Recall that we not only have the interpretation of the merged stroke, but we also have the original shape interpretations of the strokes that were merged.

We have established the following principles that specify conditions in which a set of strokes should **not** be merged. These principles have been determined empirically through the observation of merging problems with an original set of author-drawn data, and confirmed on the outside user test data.

(1) *Avoid complexity when possible.* For example, if the result of merging two shapes together is a complex primitive, then merging should not be performed. Obviously, merging two strokes, only to divide them later in the recognition process, is counter-intuitive and unnecessary.
(2) *Shapes should maintain constant (curvi)linearity.* If the result of merging is a linear shape (e.g., line, polyline, polygon), then all of the strokes that were merged should originally have been linear as well. For example, two arcs should not form a polyline. Furthermore, when the interpretation of merging strokes is a polyline, the degree of that polyline should equal the sum of the degrees of the interpretations of the substrokes (e.g., four lines should not be combined into a "Polyline (5)" or a "Polyline (3)"). Likewise, if the result of merging is a curvilinear shape (e.g., arc, ellipse, wave), then at least one of the original shapes should have also have been curvilinear. For example, three lines should not be merged into an arc.
(3) *Touch-up strokes should have a minimal effect on existing recognized closed shapes.* Touch-up strokes are short continuation strokes that occur when users are

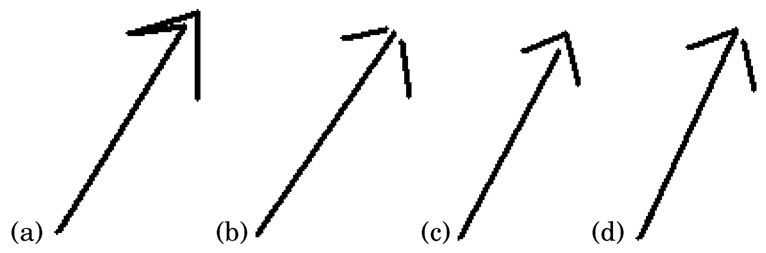


Fig. 12. Examples of the different ways that users draw arrows: with one stroke (a); with three strokes (b); or two different variations of two strokes (c,d).

"cleaning up" existing recognized, completed shapes. Hence they should not change the recognition of the previously recognized stroke. For example, if the previously recognized shape is a closed shape (e.g., rectangle, ellipse), then merging an additional stroke with this shape should not change the recognition result. If it does, then we should not merge. Touch-up strokes are also often small compared to the original shape. Thus, we also prohibit merging if the stroke alters an original closed shape's bounding box by more than 10%.

(4) *Continuation strokes should maintain consistency and complexity.* Continuation strokes may occur when a user attempts to elongate, complete, or enlarge an existing primitive. When continuation strokes are added to an already complex primitive (such as an arrow, helix, wave, spiral, or closed shape), the primitive should remain similar. When extending the shaft of a previously recognized arrow, the result of the merge should remain an arrow. Shapes like waves should also maintain a consistent path. To check this, we verify that the new wave's height does not exceed 50% of the original wave's height.

(5) *Impose confidence constraints on shapes when needed.* In some cases, particular shapes may lend themselves to being the frequently defaulted-to option when existing shapes are errantly merged. In the case of our initial set of author-drawn data, this tended to be complex shapes and arrows. Because complex shapes are avoided (principle 1), we only had to worry about arrows. From our observation, most true positive arrows will have a high confidence, while defaulted-to arrows have a low confidence. By requiring multistroke arrows to have a confidence of at least 75%, we were able to continue to recognize true positive arrows while reducing the number of false positive merges.

3.5. Arrow Detection

Arrows are the only primitive in which stroke joining often occurs at places other than their endpoints, and thus the correct subgraph produced from Tarjan's algorithm may never exist for an arrow (such is the case in Figure 12c-d). Thus, we have an additional step to check for additional merging that should take place. Arrows are typically drawn in one of four ways, as seen in Figure 12. Some users draw the shaft and the head in a single stroke (Figure 12(a)). This case is handled by the single-stroke primitive recognizer and occurred 65.4% of the time in our data set. Other users draw the shaft in one stroke, and each of the two heads with two additional strokes (Figure 12(b)). In this case, the endpoints of all three strokes should be close, and can be handled by the existing multistroke algorithm using Tarjan's graph building. This case occurred in only 3.7% of the arrows drawn in our data set. However, there are two cases that cannot be handled by the existing multistroke algorithm, because the endpoints of the strokes that need to be merged will not be near each other. To support these cases, we have developed a set of rules to search for these specific types of arrows.

(1) The first case involves the user drawing a shaft with one stroke, and the arrow head with a second stroke (Figure 12(c)). This occurred 20.6% of the time in our data set. In this case, the head will likely be drawn as a two-lined polyline. Therefore, as we loop through the strokes we are processing—if we encounter a “Polyline (2)” interpretation, we perform the following. First, we find the the segmentation point of the polyline. This is the tip of the arrow head. Next, we loop through the remaining nonclosed strokes and determine if any of their endpoints are near the tip of the arrow head (using a similar distance check to that in “Graph Building”). If we find a candidate, then we merge the strokes and classify them by using PaleoSketch. If the interpretation returned is an arrow with greater than 50% confidence, then we return the arrow interpretation.

(2) The second case occurs when the user draws the shaft and one half of the arrow head in a single stroke, and the second half of the arrow head with a single line (Figure 12(d)).[3] This happened 10.3% of the time in our data set. Remember that the shaft does not necessarily have to be a simple line. If we encounter a Polyline or a complex interpretation containing one line and one nonclosed shape, we do the following. First, we loop through the remaining line-only strokes and perform a distance check between its endpoints and the segmentation point of the polyline or complex primtive. If they are near each other, we merge them and reclassify. Again, if an arrow interpretation is returned with greater than 50% confidence, then we keep the strokes merged and return the interpretation.

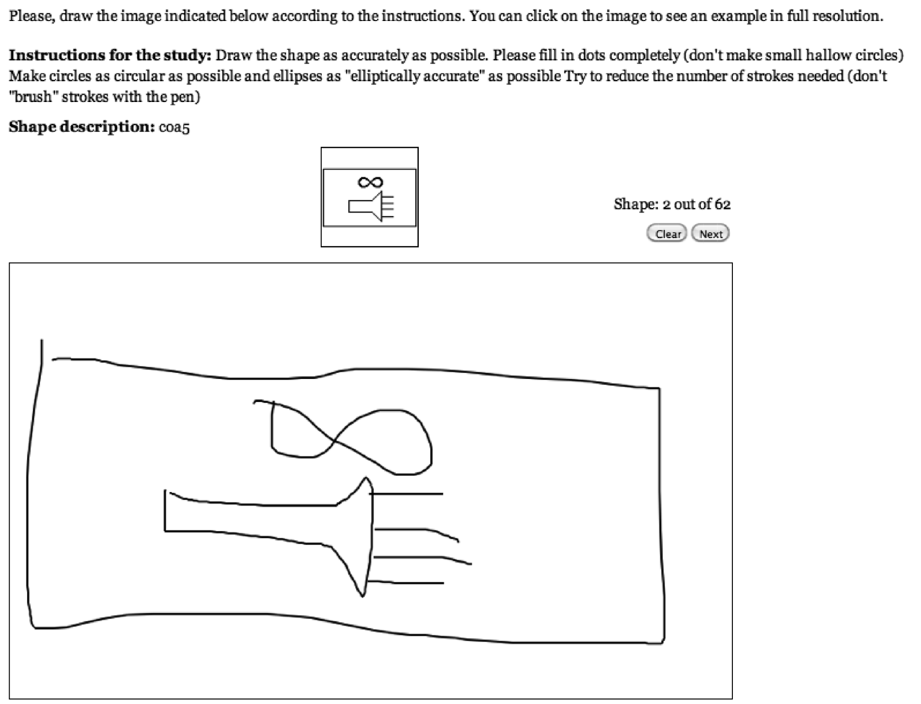
4. EXPERIMENT

We showed a set of 29 symbols composed of multiple primitives to 10 users from various backgrounds, about half of them computer scientists. Participants were shown a small-scale version of the image and were asked to redraw the symbols in Figure 14 one at a time. Symbols were shown in random order, and throughout the study each shape was presented twice. Participants were instructed to draw the shapes legibly, but in a manner natural to them. Because most of the study participants were unfamiliar with drawing with a stylus, participants were first given a prestudy where they were asked to draw five creative diagrams composed of primitives. This data was thrown out, as the purpose was only to orient the participants to the use of the stylus. Data was collected through SOUSA [Paulson et al. 2008], a web-based sketching interface shown in Figure 13. Participants sketched using a stylus on either a Cintiq monitor or Tablet PC. The symbols had an average of 4.3 primitives each, which yielded a total of 2,471 primitive shapes. The symbols were chosen from various sketch domains, including military course of action, circuit diagrams, chemistry, physics, and mechanical engineering, in the hope that by including symbols from various domains our results would be generalizable across domains. The data not only allows us to test our accuracy, but to determine how frequently multistroke primitives occur in naturally sketched data.

At no point in the study were users asked to explicitly state what the intention was of each of their drawn strokes. In order to determine the correctness of our recognizer, we used the interpretation of the shape as it would be defined in a high-level grammar’s shape definition.

To test our algorithm, we performed 10-fold cross validation over the entire set of data. With this, one user’s data is saved for testing while the remaining nine are used for training. This is done for all users and the results are averaged. When training our neural network version of PaleoSketch, we trained only on single strokes (i.e., we did not train on labeled multistroke primitives). We computed accuracies for both

[3]Note that the mirror image of Figure 12(d), with the short line on the left-hand side is handled in exactly the same way, since we are just attempting to figure out where and how we merge the strokes.


Please, draw the image indicated below according to the instructions. You can click on the image to see an example in full resolution.

Instructions for the study: Draw the shape as accurately as possible. Please fill in dots completely (don't make small hallow circles) Make circles as circular as possible and ellipses as "elliptically accurate" as possible Try to reduce the number of strokes needed (don't "brush" strokes with the pen)

Shape description: coa5

Shape: 2 out of 62

Fig. 13. A screenshot of the data collection program.

single-stroke and multistroke primitives. In order for a multistroke primitive to be classified as correct, it had to have been properly merged as well as recognized, with no extra or missing strokes in the interpretation.

5. RESULTS

Figure 16 shows the percentage of the time that we encountered multistroke primitives in our data set. In all, multistroke primitives made up only 9% of all of the primitives in the data set. The most commonly drawn primitives using multiple strokes were polygons (39.3%), arrows (35.2%), diamonds (30%), and rectangles (18.8%). All other primitives were drawn with multiple stroke less than 10% of the time. Interestingly, our results coincide with those of van Sommers, when he explained that the more sides there were in a figure, the less threading (drawing multiple lines within a single stroke) there is per corner [van Sommers 1984].

5.1. Single-Stroke

Figure 17 shows the accuracies of both single-stroke and multistroke primitives. In total, we achieved 96% recognition of all primitives in the data set. With single-stroke primitives, our weighted accuracy was 96.6%. This is slightly more than a full percentage point lower than the results from our previous work [Paulson 2010]. However, much of this error is due to some specific shapes.

For example, single-stroke spirals performed poorly, with only 78.9% recognition. In Paulson [2010], we had a perfect 100% recognition of spirals. We believe the primary reason for this decrease is the lack of spiral training data in the set of symbols we collected. In total, there were only 20 spirals drawn, which is less than 1% of the entire data.

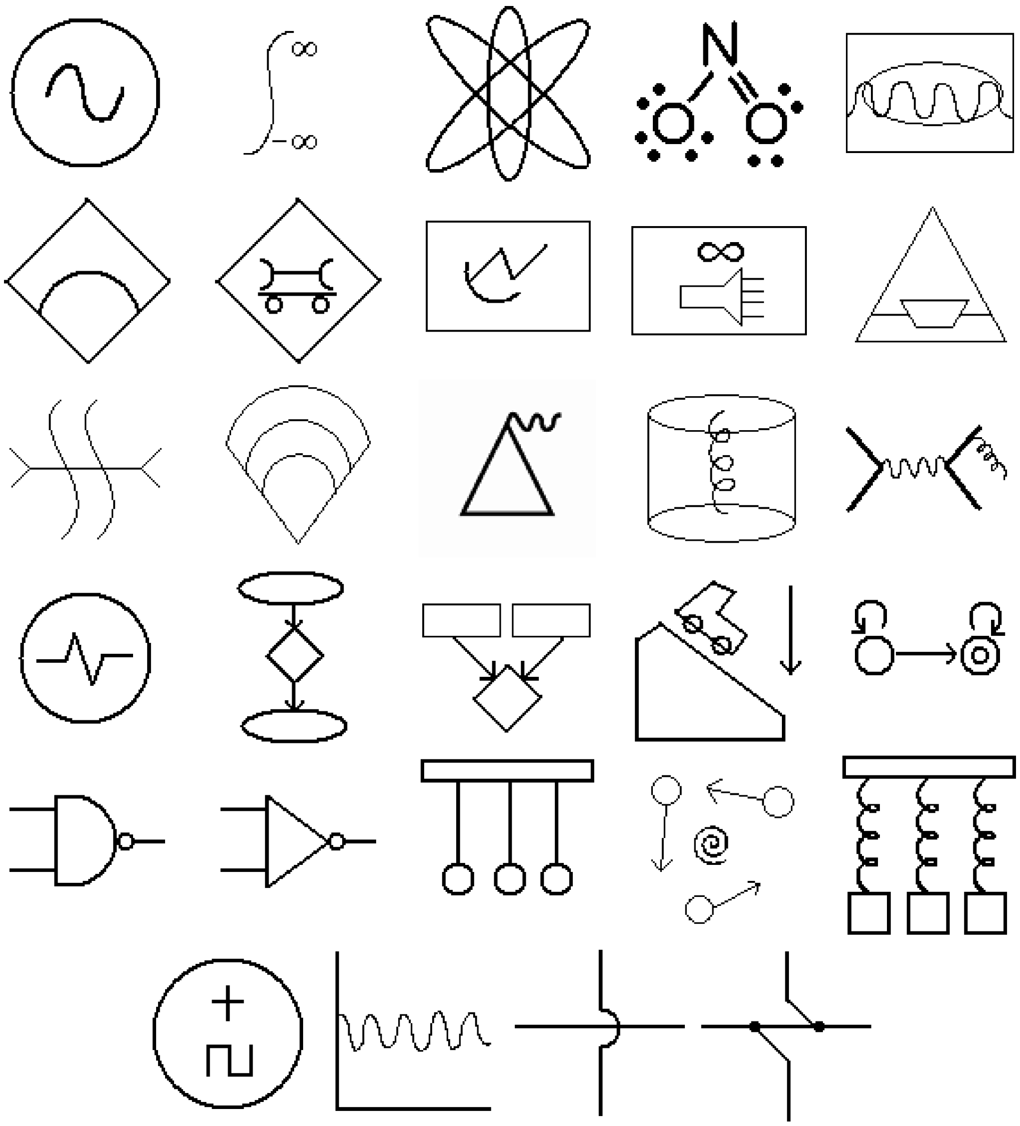


Fig. 14. Symbols from multiple domains used for data collection.

Another troublesome primitive was the dot (88.6%). In this instance, most of the incorrectly recognized dots were all drawn by a single user. Unlike other users who drew dots as small, filled-in circles, this user simply made quick, small tic-marks to the screen to designate dots. In most cases, these were classified as lines (Figure 18). In some systems, any small marking relative to the screen size are automatically classified as dots [Hammond 2007]. If we were to make a similar assumption, then these dots could easily be classified correctly. However, this change would require the addition of a heuristic rule. Note that because our accuracy was computed by leaving one cross validation out, and because no other user included tic-marks as dots, there were no similarly drawn examples in the training set when testing the data for this user.

circle, curve	curve, line, 2 infinity	3 ellipse	polyline (3), 3 line, 10 dot, 2 circle	rectangle, ellipse, wave
diamond, arc	diamond, 2 arc, 2 circle, 2 line	rectangle, polyline (3), arc	rectangle, polygon (6), 4 line, infinity	polyline (3), polygon (4) 2 line
*2 polyline (2), line, 2 curve	polyline (2), 3 arc	polygon (3), wave	2 ellipse, 2 line, helix	helix, wave, 2 polyline (2)
circle, polyline (5)	2 ellipse, diamond, 2 arrow	2 rectangle, 2 arrow, diamond	2 polygon (5), 2 circle, arrow	3 circle, 3 arrow
4 line, arc, circle	3 line, polygon (3), circle	3 circle, 3 line, rectangle	3 circle, 3 arrow, spiral	3 helix, 3 square, rectangle
circle, 2 line, polyline (5)	polyline (2), wave	3 line, arc	line, 2 polyline (2), 2 dot s	

Fig. 15. A list of the primitives in each of the symbols in Figure 14. *This one could also be polyline (3), 2 line, 2 curve.

	Total	Single-stroke	Multi-stroke	% Multi-stroke
Arc	105	101	4	3.8%
Arrow	216	140	76	35.2%
Complex	120	120	-	-
Curve	78	78	0	0%
Diamond	60	42	18	30.0%
Dot	219	202	17	7.8%
Ellipse	478	462	16	3.3%
Helix	84	82	2	2.4%
Infinity	59	59	0	0%
Line	485	475	10	2.1%
Polygon	117	71	46	39.3%
Polyline	214	214	-	-
Rectangle	149	121	28	18.8%
Spiral	20	19	1	5.0%
Wave	67	62	5	7.5%
Overall	2471	2248	223	9.0%

Fig. 16. Number of each type of sketched primitive in the collected data. Complex and polyline shapes cannot be drawn with multiple strokes.

Other shapes that had lower single-stroke accuracies were complex shapes (90.8%) and polygons (93%). The biggest problem for polygons was distinguishing non-rectangular quadrilaterals from rectangles. There was also much confusion between polygons and complex interpretations. Issues with complex shapes will be discussed in more detail in a later section.

	Single-stroke	Multi-stroke	Weighted Avg.
Arc	0.980	0.0	0.943
Arrow	0.979	0.934	0.963
Complex	0.908	-	0.908
Curve	0.962	-	0.962
Diamond	0.976	0.944	0.967
Dot	0.886	1.0	0.895
Ellipse	0.987	0.688	0.977
Helix	0.963	1.0	0.964
Infinity	1.0	-	1.0
Line	0.992	0.900	0.990
Polygon	0.930	0.913	0.923
Polyline	0.958	-	0.958
Rectangle	0.983	0.929	0.973
Spiral	0.789	0.0	0.750
Wave	1.0	1.0	1.0
Average	95.5%	75.5%	94.5%
Weighted Avg.	96.6%	89.7%	96.0%

Fig. 17. Accuracy results after 10-fold cross-validation. “Average” represents flat averages, while “weighted averages” represent averages that are weighted by the number of occurrences of each shape type.

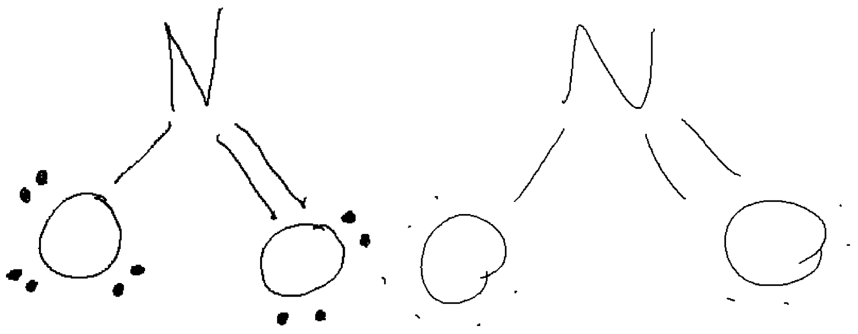


Fig. 18. Examples showing the way that most users drew dots (left) versus the single user who drew them with small strokes (right).

5.2. Multi-Stroke

Overall, we achieved 89.7% weighted accuracy for multistroke primitives. This means that the strokes were both merged and recognized correctly. Examples of correctly classified multistroke primitives can be seen in Figure 19. Strokes that did not merge, but should have, are considered *false negatives*. Primitive strokes that were not meant to be merged, but were, are considered *false positives*.

Most of the errors we encountered were due to strokes not being merged correctly into a single stroke, rather than being misclassified (false negatives). One limitation of our multistroke approach is that it relies heavily on single-stroke interpretations being correct. For example, some strokes that should have merged to form a multistroke primitive were incorrectly classified as a complex shape. Because we disallow strokes originally recognized as complex primitives from being merged, the resulting strokes were never merged into a primitive shape.

Another reason some strokes were not merged is because the distance between their endpoints was too great. Examples of this can be seen in Figure 20. In these examples,

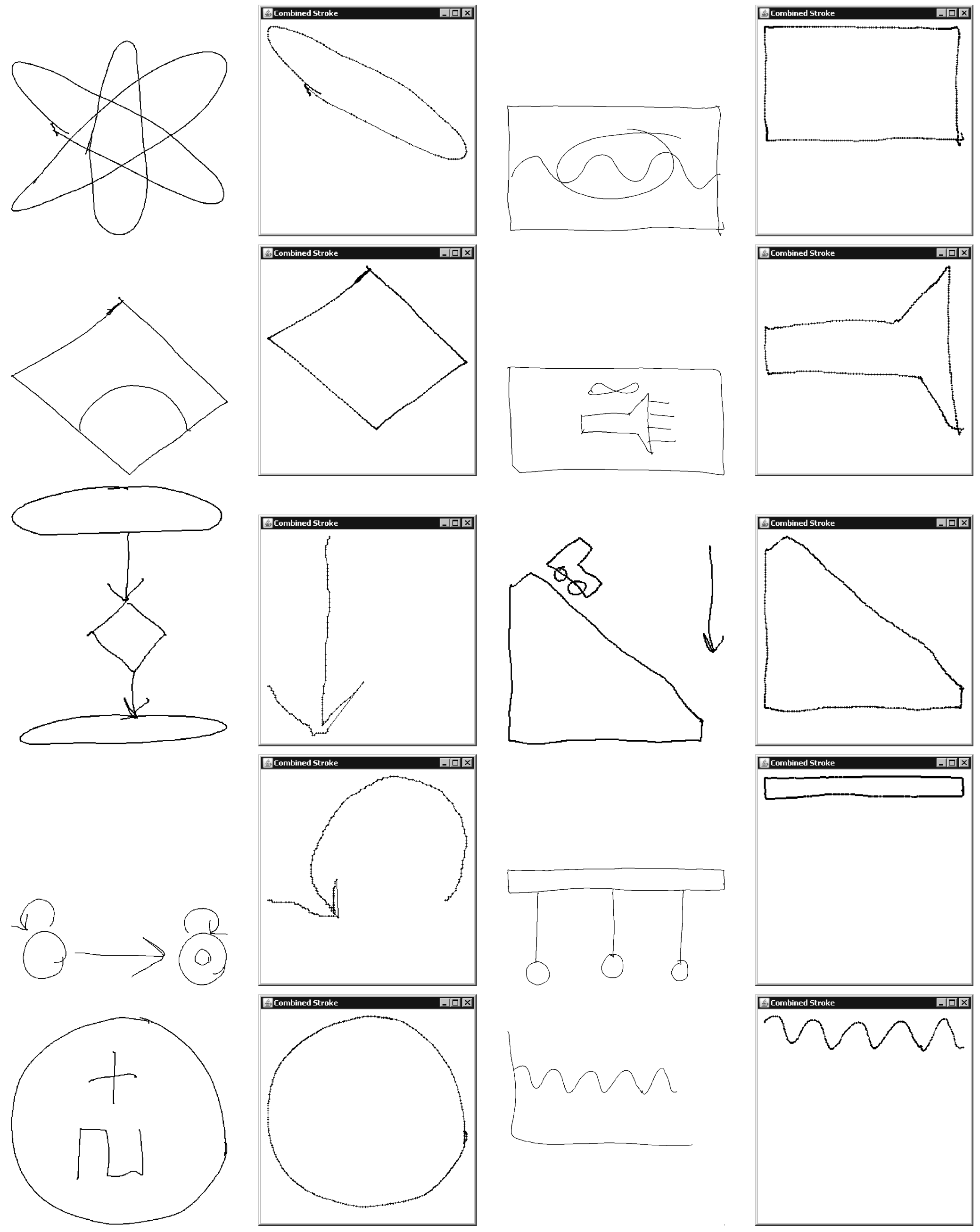


Fig. 19. Examples of correctly merged and classified multistroke primitives (original sketch and merged primitive). Each of the images on the left shows a primitive in context drawn with more than one stroke. For instance the arrows were drawn with separate arrow heads; the circle was made from two arcs; the wave was extended; others have small closure marks. In each of the images on the right the merged stroke is displayed out of context. All of the images on the right were correctly recognized as their intended primitive after merging.

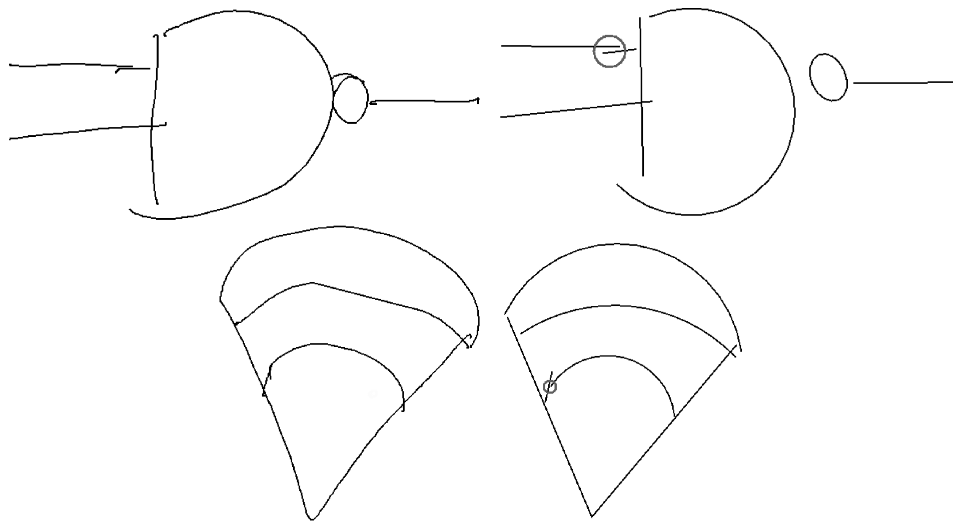


Fig. 20. Examples of multistroke primitives that were unsuccessfully merged due to endpoint distances being too great compared to the stroke length of the smaller stroke.

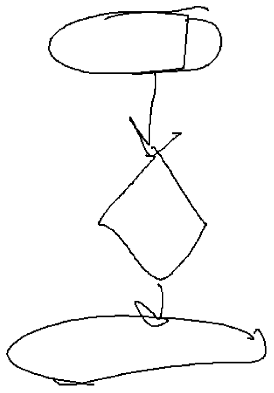


Fig. 21. Example of a user attempting to correct an error in sampling (top ellipse).

the endpoint of the larger stroke is closer to the midpoint of the smaller stroke than it is to its endpoint. Therefore, the distance between the endpoints was not within the thresholds that we specified in the multistroke algorithm, which were based on the size of the strokes to merge.

Additional issues that we encountered were examples of users attempting to correct poor sampling or users overtracing. Figure 21 shows an example of a user who, while drawing an ellipse, encountered a lag during sampling. This resulted in many points being lost, which the user then attempted to supplement with an additional stroke. Likewise, in Figure 22, one user attempted to clean up some of his shapes by adding additional, overtraced strokes. In both of these cases, strokes were never merged because the endpoints were too far apart.

When computing accuracy, we did not account for false positive merges (i.e., produced a primitive that we were not expecting). However, in all 2,471 primitives that we tested, we only encountered 6 false positive merges. Furthermore, most of these false positives

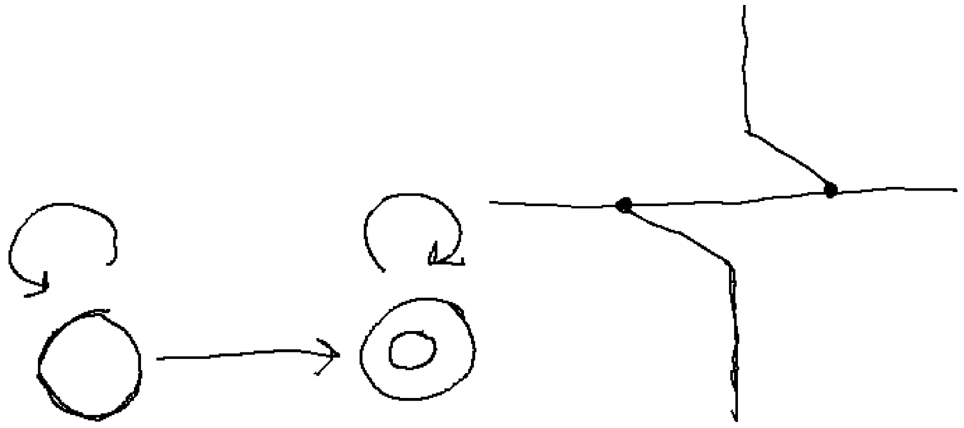


Fig. 22. Examples of multistroke overtracing performed by one user.

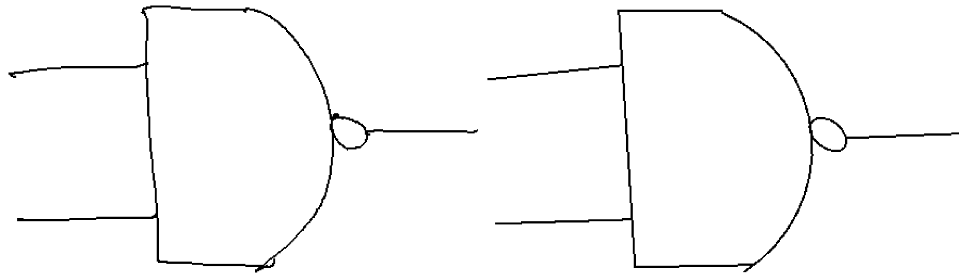


Fig. 23. Example of a NAND gate drawn by user, resulting in both a false positive and a false negative merge.

consequently caused false negatives to occur, since the false positive used up the strokes necessary to compose a primitive.

For example, in Figure 23, one user drew both of his NAND gates using 3 lines and an arc, rather than one line and an arc. Since the shape definition of a NAND gate would only consist of one line and one arc, we consider anything containing more components to be incorrect. In this case, these extra lines were merged with the vertical line to produce a three-line polyline. This was both a false positive because the lines were merged with the vertical line, as well as a false negative because they were not merged with the arc.

As another example, see Figure 24. In this example, a false positive (and negative) was generated due to the greedy nature of our algorithm. The user drew both vertical lines first, followed by two strokes to make up the bottom ellipse. The first stroke that makes up the ellipse was incorrectly merged as a polyline with the two vertical strokes. This, consequently, kept it from correctly merging with the second half of the ellipse, which caused a false negative.

In some few cases, the algorithm did correctly merge the strokes, but they were misclassified. Figure 25 shows an example of two strokes that were correctly merged, but then misclassified as a rectangle (false positive) instead of a polygon (false negative).

5.3. Complex Shapes

As with our experiments in Paulson and Hammond [2008b] and Paulson [2010], we also wanted to determine how well our algorithm performed on complex interpretations.

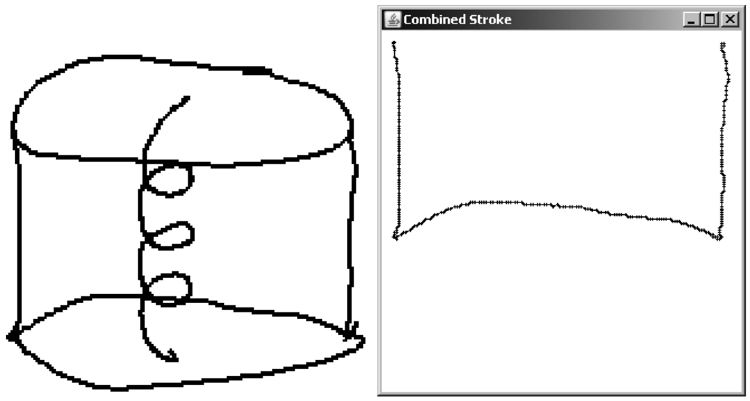


Fig. 24. Example of a false positive generated due to the greedy nature of the algorithm.

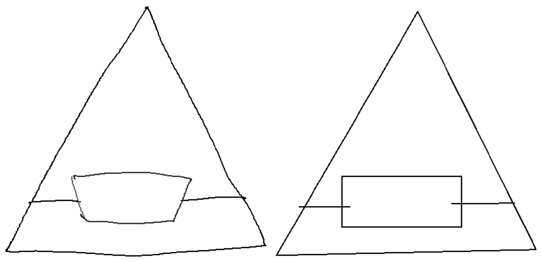


Fig. 25. Example of multistroke primitives that were merged correctly, but classified incorrectly.

Unlike our previous data set, which only contained a complex shape of one line and one arc, this data set contained many more complex inerpretations. Figure 26 shows the multiprimitive shapes from our dataset that were drawn using a single stroke. Underneath each shape is the percentage of time to draw it with a single stroke, rather than with multiple strokes.

Complex shapes made up less than 5% of all of the primitive shapes in the data set. This means there were more instances of multistroke primitives than there were of single-stroke complex primitives. The neural network correctly classified instances of complex shapes 90.8% of the time. We returned the correct interpretation for complex shapes 78% of the time. More specifically, the correct interpretation was returned 88.8% of the time if the complex shape consisted of only two subshapes (73.4% of all complex shapes). However, if the interpretation contained three subshapes the accuracy was only 50% (26.6% of all complex shapes). We did not encounter any complex interpretations that contained more than three shapes.

The majority of misinterpreted complex shapes were due to the most commonly drawn complex shape: the line, arc, line shape in Figure 28. In this case, the stroke was *undersegmented*, that is, the interpretation returned too few shapes. For example, most complex strokes intentionally drawn as line, arc, line were interpreted instead as line, curve. This is because the complex interpreter accepts any subshape that has a confidence level of at least 50% (as described in Paulson [2010]). In this case, the curve had a high enough confidence level that it did not require further segmentation. In fact, most of these curves had confidence levels above 98%. Undersegmented complex shapes made up 83.3% of the total error.

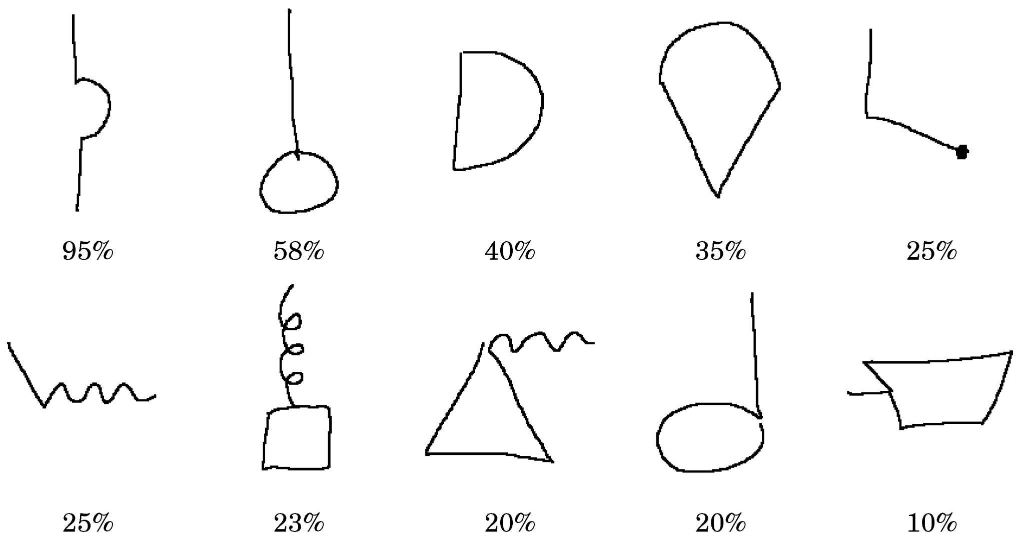


Fig. 26. Examples of the complex shapes encountered in the dataset, along with the percentage of the time that each was drawn with a single-stroke rather than multiple strokes.

The remaining errors came in the form of oversegmented complex interpretations, and a few cases of complex overtracing. Most oversegmentations were a result of the way in which the user drew the shape. Either there was a large tail on the stroke, or the user purposefully added additional shapes to the stroke, as seen in Figure 29, where the user added additional lines to the tops of the helixes.

Occasionally we also encountered examples of overtracing, seen in Figure 30. These examples were due the same user who drew the multistroke, overtraced shapes. In this case, the additional stroke kept the shape from being properly interpreted. Due to the scarcity of overtraced shapes in our data set, we leave this as a problem for future work.

6. DISCUSSION

In this article, we have introduced an algorithm for recognizing multistroke primitives which, unlike previous approaches, places no drawing constraints on the users. Although the algorithm achieved successful results, there is still much room for improvement.

6.1. Comparisons to Other Methods

Comparing our algorithm to other multistroke recognizers is difficult because of the constraints used by many of these approaches. Most have been tested solely on isolated primitive shapes and rely on special button presses [Hse and Newton 2005; Calhoun et al. 2002] or unspecified timeouts [Fonseca and Jorge 2000] to group the strokes that belong to a single primitive. These types of approaches completely disallow stroke interspersing. Furthermore, many of these recognizers do not support the full range of primitives that we have supported with PaleoSketch.

The most comparable recognizer to ours is the Tahuti recognizer [Hammond and Davis 2002]. Unlike the aforementioned recognizers, Tahuti does allow for stroke interspersing, up to a certain number of strokes. However, one notable downside of the Tahuti recognizer is that it only handles multistroke rectangles and linear arrows. It does not handle any form of multistroke, curvilinear shape, such as multistroke

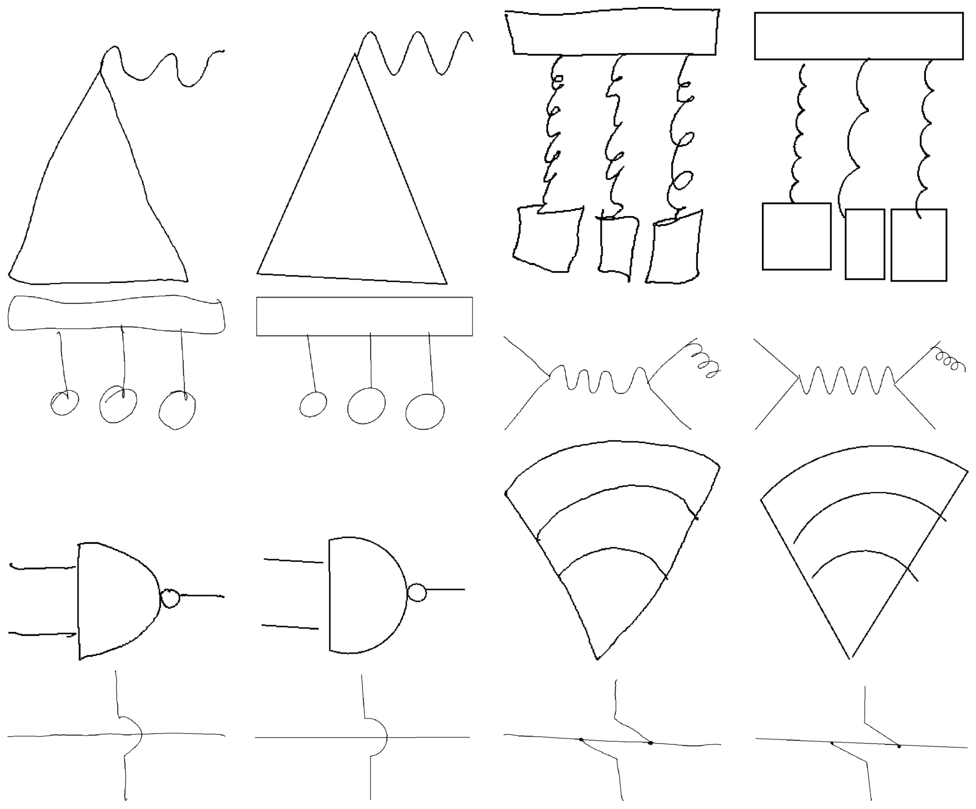


Fig. 27. Examples of correctly interpreted complex shapes (original strokes and recognized shapes).

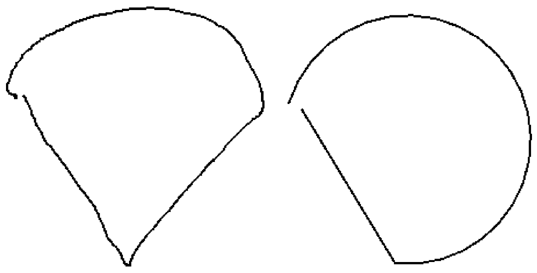


Fig. 28. Example of a complex undersegmentation.

ellipses. We tested our arrow and rectangle data using the Tahuti recognizer and found that it achieved 53.2% accuracy on arrows and only 33.6% accuracy on rectangles, while our recognizer achieved over 96% on both of these shapes. The primary reason for this is because Tahuti does not support arrows with curved shafts or rectangles that have overlapping sides (Figure 31).

Some may also argue that multistroke primitives can be handled by higher-level recognition systems, such as LADDER [Hammond 2007]. For example, a rectangle composed of four strokes can be described as four lines that have perpendicular angles and congruent sides. A multistroke line is simply two lines that have near endpoints

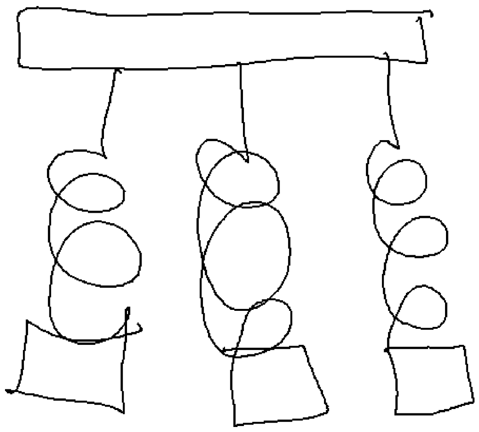


Fig. 29. Example of complex oversegmentation. Additional lines at the tops of the helixes led to oversegmentations. Note that the lines at the top of the helixes did not exist in the original image and caused the helix to be oversegmented.

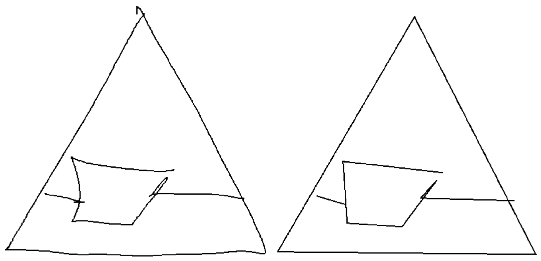


Fig. 30. Overtraced lines within complex shapes kept them from being properly interpreted (left).

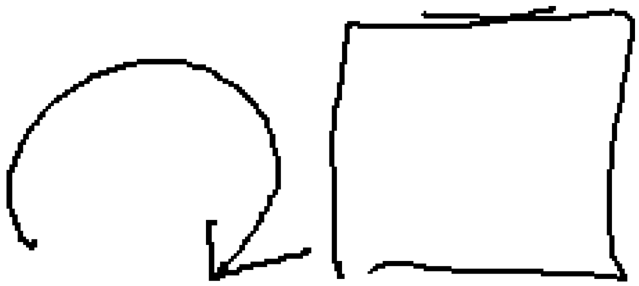


Fig. 31. Examples of curved arrows and rectangles with overlapping sides that Tahuti had trouble recognizing.

and a similar slope. Although this is true, the possible combinations of shapes that can form multistroke primitives would soon become unmanageable. A multistroke circle could be formed from two arcs, or from an arc and a line, or from two arcs and a line, or from three arcs, or from a curve and line, and so on. The possibilities are endless, and each would require a specific shape definition in the high-level system. This is the primary reason why we believe multistroke primitive recognition should be the task of the low-level recognizer.

6.2. Multiple Interpretations

One of the downfalls of our approach is that many aspects of the algorithm are greedy. For instance, when determining if strokes should be merged, the algorithm looks solely at the best primitive interpretation of individual strokes, rather than considering the confidence level of every possible primitive. Furthermore, the algorithm is greedy when it comes to the order in which to merge strokes (recall the problems in Figure 24). One possible improvement to the algorithm would be for it to consider and rank multiple, multistroke interpretations. Each possible merging of strokes could be considered, and confidence could be used to determine the best interpretation possible.

This would be a difficult task, however, because of speed requirements. With our multistroke algorithm, it took, on average, 372 milliseconds to classify each primitive. This is almost two and half times slower than our single-stroke recognizer. Adding the additional complexity of searching through all possible multistroke interpretations for a given set of strokes would slow recognition further.

6.3. Improving Grouping

Finally, another area of our algorithm that could be improved is grouping. Currently, we use a simple spatial distance between endpoints to determine if strokes should be grouped. Although this assumption works for over 93% of the cases,[4] it is still not valid for some examples (e.g., overtraced shapes). Using a combination of temporal and spatial information for clustering may improve the overall accuracy of our approach. We must be careful, though, not to rely solely on temporal information, as this may constrain the manner in which a user has to draw. For overtraced shapes, comparing the overlap of bounding boxes may also be a possible indicator for a good merge.

6.4. Additional Improvements

One thing to remember about our experiments is that we tested on a full set of primitives that represented many different domains. In a real-world application, it is unneccesary to have every primitive turned on for every domain. Specific primitive shapes should be capable of being turned on and off. In most cases, we would expect that real-world accuracies could actually be better than our tested accuracies because fewer primitives would need to be turned on for each domain.

Another thing to keep in mind it that we performed our experiments using cross-fold validation based on the user. This means that each testing user provided no training data to the neural network. As we saw in the example of the user who drew dots with small marks, user-specific styles still exist and may cause problems with recognition. One benefit of our neural network classifier, however, is that it is adaptable, and could be made to learn over time by modifying its weights based on the correct/incorrect classification of user-specific examples. A similar approach was used in our MARQS system, and proved to be beneficial over time [Paulson and Hammond 2008a].

7. CONCLUSION

In this article, we introduced a recognition algorithm that achieves close to 90% weighted accuracy on multistroke primitives and a 96% weighted accuracy overall. The algorithm is capable of recognizing these primitives without requiring special drawing constraints, such as timeouts, button presses, or prohibiting interspersing. Although the approach produces acceptable results, much improvement can still be made. Areas of possible improvement include returning multiple interpretations, improving complex fits, and discovering better methods of stroke clustering.

[4]This was determined by testing multistroke accuracy when we assumed perfect single-stroke accuracy.

ACKNOWLEDGMENTS

Thanks to Danielle Cummings for her help in editing this article. Additionally, the authors would like to thank the members of the Sketch Recognition Lab for their help throughout the development of this work.

REFERENCES

ALVARADO, C. AND DAVIS, R. 2001. Resolving ambiguities to create a natural computer-based sketching environment. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI, Stanfords, CA, 1365–1371.

ALVARADO, C. AND DAVIS, R. 2004. Sketchread: A multi-domain sketch recognition engine. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST'04)*, ACM, New York, 23–32.

ANTHONY, L. AND WOBBROCK, J. 2010. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of the Graphics Interface Conference*. Canadian Information Processing Society, Ottawa, Ontario, 245–252.

APTE, A., VO, V., AND KIMURA, T. D. 1993. Recognizing multistroke geometric shapes: An experimental evaluation. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, 121–128.

BAILEY, B. P. AND KONSTAN, J. A. 2003. Are informal tools better? Comparing DEMAIS, pencil and paper, and authorware for early multimedia design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'03)*. ACM, New York, 313–320.

BLACK, A. 1990. Visible planning on paper and on screen: The impact of working medium on decision-making by novice graphic designers. *Behav. Inf. Technol. 9*, 4, 283–296.

BLICKENSTORFER, C. H. 1995. Graffiti: Wow! *Pen Comput. Mag*. (1/30/95).

CALHOUN, C., STAHOVICH, T. F., KURTOGLU, T., AND KARA, L. B. 2002. Recognizing multi-stroke symbols. In *Proceedings of the AAAI Spring Symposium - Sketch Understanding*. AAAI Press, Stanford, CA, 15–23.

COSTAGLIOLA, G., DEUFEMIA, V., POLESE, G., AND RISI, M. 2004. A parsing technique for sketch recognition systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, Los Alamitos, CA, 19–26.

COSTAGLIOLA, G., DEUFEMIA, V., AND RISI, M. 2005. Sketch grammars: A formalism for describing and recognizing diagrammatic sketch languages. In *Proceedings of the 8th International Conference on Document Analysis and Recognition (ICDAR'05)*. IEEE. Los Alamitos, CA, 1226–1231.

DIXON, D. M. 2009. A methodology using assistive sketch recognition for improving a person's ability to draw. M.S. thesis, Texas A&M University, College Station, TX.

FONSECA, M. J. AND JORGE, J. A. 2000. Using fuzzy logic to recognize geometric shapes interactively. In *Proceedings of the 9th IEEE International Conference on Fuzzy Systems*. Vol. 1. IEEE, Los Alamitos, CA, 291–296.

FONSECA, M. J., PIMENTEL, C., AND JORGE, J. 2002. Cali: An online scribble recognizer for calligraphic interfaces. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, Stanford, CA, 51–58.

FORBUS, K. D., USHER, J., LOVETTE, A., LOCKWOOD, K., AND WETZEL, J. 2008. Cogsketch: Open-domain sketch understanding for cognitive science research and for education. In *Proceedings of the Eurographics Symposium on Sketch-based Interfaces and Modeling (SBIM)*. ACM, New York.

FUTRELLE, R. P. AND NIKOLAKIS, N. 1995. Efficient analysis of complex diagrams using constraint-based parsing. In *Proceedings of the International Conference on Document Analysis and Recognition*. 782.

GOEL, V. 1995. *Sketches of Thought.* MIT Press, Cambridge, MA.

GOLDBERG, D. AND RICHARDSON, C. 1993. Touch-typing with a stylus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, New York, 80–87.

GROSS, M. D. AND DO, E. Y.-L. 1996. Ambiguous intentions: A paper-like interface for creative design. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, 183–192.

GROSS, M. D. AND DO, E. Y.-L. 2000. Drawing on the back of an envelope: A framework for interacting with application programs by freehand drawing. *Comput. Graph. 24*, 6, 835–849.

HAMMOND, T. 2007. Hammond, T. 2007. LADDER: A perceptually-based language to simplify sketch recognition user interfaces development. Ph.D. dissertation, MIT, Cambridge, MA.

HAMMOND, T. AND DAVIS, R. 2002. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *Proceedings of the AAAI Spring Symposium.* AAAI Press, Stanford, CA, 59–66.

HAMMOND, T. AND DAVIS, R. 2005. LADDER, a sketching language for user interface developers. *Comput. Graph. 29*, 4, 518–532.

Hammond, T., Logsdon, D., Peschel, J., Johnston, J., Taele, P., Wolin, A., and Paulson, B. 2010. A sketch recognition interface that recognizes hundreds of shapes in course-of-action diagrams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, New York.

Hammond, T. A. and Davis, R. 2009. Recognizing interspersed sketches quickly. In *Proceedings of the Conference on the Graphics Interface (GI'09)*. ACM, New York, 157–166.

Hse, H. H. and Newton, A. R. 2005. Recognition and beautification of multi-stroke symbols in digital ink. *Comput. Graph. 29*, 4, 533–546.

Johnston, J. and Hammond, T. 2010. Computing confidence values for geometric constraints for use in sketch recognition. In *Proceedings of the Eurographics Symposium on Sketch-based Interfaces and Modeling (SBIM)*. ACM, New York.

Kara, L. B. and Stahovich, T. F. 2004. Hierarchical parsing and recognition of hand-sketched diagrams. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, 13–22.

Kurtoglu, T. and Stahovich, T. F. 2002. Interpreting schematic sketches using physical reasoning. In *Proceedings of the AAAI Spring Symposium.* AAAI Press, Stanford, CA, 78–85.

Laviola, J. J. and Zeleznik, R. C. 2004. Mathpad2: A system for the creation and exploration of mathematical sketches. *ACM Trans. Graph. 23*, 3, 432–440.

Li, X. and Yeung, D.-Y. 1997. On-line handwritten alphanumeric character recognition using dominant points in strokes. *Patt. Recogn. 30*, 1, 31–44.

Long Jr., A. C., Landay, J. A., Rowe, L. A., and Michiels, J. 2000. Visual similarity of pen gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, New York, 360–367.

Mahoney, J. V. and Fromherz, M. P. J. 2001. Interpreting sloppy stick figures by graph rectification and constraint-based matching. In *Proceedings of the 4th IAPR International Workshop on Graphics Recognition*. ACM, New York.

Mahoney, J. V. and Fromherz, M. P. J. 2002. Three main concerns in sketch recognition and an approach to addressing them. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, Stanford, CA, 105–112.

Newman, W. M. and Sproull, R. F. 1973. *Principles of Interactive Computer Graphics* 2nd Ed., McGraw-Hill, New York, 202–209.

Oltmans, M. 2007. Envisioning sketch recognition: A local feature-based approach to recognizing informal sketches. Ph.D. dissertation, MIT, Cambridge, MA,

Oltmans, M. and Davis, R. 2001. Naturally conveyed explanations of device behavior. In *Proceedings of the Workshop on Perceptive User Interfaces (PUI'01)*. ACM, New York, 1–8.

Ouyang, T. Y. and Davis, R. 2007. Recognition of hand drawn chemical diagrams. In *Proceedings of the National Conference on Artificial Intelligence*. AAAI Press, 846–851.

Ouyang, T. Y. and Davis, R. 2009. A visual approach to sketched symbol recognition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, Stanford, CA, 1463–1468.

Pasternak, B. and Neumann, B. 1993. Adaptable drawing interpretation using object-oriented and constraint-based graphic specification. In *Proceedings of the 2nd International Conference on Document Analysis and Recognition*. IEEE, Los Alamitos, CA, 359–364.

Paulson, B. 2010. Rethinking pen input interaction: Enabling freehand sketching through improved primitive recognition. Ph.D. dissertation, Texas A&M University, College Station, TX.

Paulson, B., Eoff, B., Wolin, A., Johnston, J., and Hammond, T. 2008. Sketch-based educational games: Drawing kids away from traditional interfaces. In *Proceedings of the 7th International Conference on Interaction Design and Children (IDC'08)*, ACM, New York, 133–136.

Paulson, B. and Hammond, T. 2008a. MARQS: Retrieving sketches using domain and style-independent features learned from a single example using a dual-classifier. *J. Multi-Modal User Interfaces 2*, 1, 3–11.

Paulson, B. and Hammond, T. 2008b. PaleoSketch: Accurate primitive sketch recognition and beautification. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, New York, 1–10.

Paulson, B., Rajan, P., Davalos, P., Gutierrez-Osuna, R., and Hammond, T. 2008. What!?! No Rubine features?: Using geometric-based features to produce normalized confidence values for sketch recognition. In *Proceedings of the VL/HCC Sketch Tools for Diagramming Workshop*. IEEE, Los Alamitos, CA, 57–63.

Paulson, B., Wolin, A., Johnston, J., and Hammond, T. 2008. SOUSA: Sketch-based online user study applet. In *Proceedings of the Eurographics Symposium on Sketch-based Interfaces and Modeling (SBIM)*. ACM, New York, 81–88.

Peschel, J. and Hammond, T. 2008. STRAT: A sketched-truss recognition and analysis tool. In *Proceedings of the International Conference on Distributed Multimedia Systems*. Knowledge Systems Institute, Boston, MA, 282–287.

PITTMAN, J. A., SMITH, I. A., COHEN, P. R., OVIATT, S. L., AND YANG, T.-C. 1996. Quickset: A multimodal interface for military simulation. In *Proceedings of the 6th Conference on Computer-Generated Forces and Behavioral Representation (CGF-BR'96)*. 217–224.

PLIMMER, B. AND FREEMAN, I. 2007. A toolkit approach to sketched diagram recognition. In *Proceedings of the 21st British CHI Group Annual Conference on HCI (BCS-HCI'07)*. ACM, New York, 205–213.

RUBINE, D. 1991. Specifying gestures by example. In *Proceedings of the International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, New York, 329–337.

SAUND, E. AND LANK, E. 2003. Stylus input and editing without prior selection of mode. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, New York, 213–216.

SAUND, E., MAHONEY, J., FLEET, D., LARNER, D., AND LANK, E. 2002. Perceptual organization as a foundation for intelligent sketch editing. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, Stanford, CA, 118–125.

SAUND, E. AND MORAN, T. P. 1995. Perceptual organization in an interactive sketch editing application. In *Proceedings of the IEEE 5th International Conference on Computer Vision* (*ICCV'95)*. IEEE, Los Alamitos, CA, 597–607.

SEZGIN, T. M., STAHOVICH, T., AND DAVIS, R. 2001. Sketch based interfaces: Early processing for sketch understanding. In *Proceedings of the Workshop on Perceptive User Interfaces (PUI'01)*. ACM, New York, 1–8.

SHILMAN, M., PASULA, H., RUSSELL, S., AND NEWTON, R. 2002. Statistical visual language models for ink parsing. In *Proceedings of the AAAI Spring Symposium.* AAAI Press, Stanford, CA, 126–132.

STAHOVICH, T. F. 1996. Sketchit: A sketch interpretation tool for conceptual mechanical design. Tech. rep. 1573, Artificial Intelligence Laboratory.

TAELE, P. AND HAMMOND, T. 2008. Using a geometric-based sketch recognition approach to sketch Chinese radicals. In *Proceedings of the National Conference on Artificial Intelligence.* AAAI Press, 1832–1833.

TAELE, P. AND HAMMOND, T. 2009. Hashigo: A next-generation sketch interactive system for Japanese Kanji. In *Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference (IAAI'09)*. AAAI Press, 153–158.

TAELE, P., PESCHEL, J., AND HAMMOND, T. 2009. A sketch interactive approach to computer-assisted biology instruction. In *Proceedings of the Workshop on Sketch Recognition at the International Conference of Intelligent User Interfaces (IUI)*. ACM, New York.

TARJAN, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Computing 1*, 2, 146–160.

TENNESON, D. AND BECKER, S. 2005. Chempad: Generating 3D molecules from 2D sketches. In *Proceedings of the International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, New York, 87.

VAN SOMMERS, P. 1984. *Drawing and Cognition: Descriptive and Experimental Studies of Graphic Production Processes*. Cambridge University Press, Cambridge, UK.

WOBBROCK, J. O., WILSON, A. D., AND LI, Y. 2007. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, 159–168.

WOLIN, A., EOFF, B., AND HAMMOND, T. 2009. Search your mobile sketch: Improving the ratio of interaction to information on mobile devices. In *Proceedings of the Workshop on Sketch Recognition at the International Conference of Intelligent User Interfaces (IUI)*, ACM, New York.

WONG, Y. Y. 1992. Rough and ready prototypes: Lessons from graphic design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, New York, 83–84.

YEUNG, L., PLIMMER, B., LOBB, B., AND ELLIFFE, D. 2008. Effect of fidelity in diagram presentation. In *Proceedings of the 22nd British HCI Group Annual Conference on HCI (BCS-HCI'08)*. British Computer Society, 35–44.

YU, B. AND CAI, S. 2003. A domain-independent system for sketch recognition. In *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE'03)*. ACM, New York, 141–146.

ZELEZNIK, R. AND MILLER, T. 2006. Fluid inking: Augmenting the medium of free-form inking with gestures. In *Proceedings of Graphics Interface Conference (GI'06)*. Canadian Information Processing Society, Ottawa, Ontario, 155–162.

ZHAO, R. 1993. Incremental recognition in gesture-based and syntax-directed diagram editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, New York, 95–100.

Received May 2010; revised December 2010; accepted August 2011