A Lightweight Multistroke Recognizer for User Interface Prototypes

Lisa Anthony[1*], Jacob O. Wobbrock[2]

[1]Lockheed Martin Advanced Technology Laboratories, [2]Information School, DUB Group, University of Washington

ABSTRACT

With the expansion of pen- and touch-based computing, new user interface prototypes may incorporate stroke gestures. Many gestures comprise multiple strokes, but building state-of-the-art multistroke gesture recognizers is nontrivial and time-consuming. Luckily, user interface prototypes often do not require state-of-the-art recognizers that are general and maintainable, due to the simpler nature of most user interface gestures. To enable easy incorporation of multistroke recognition in user interface prototypes, we present $N, a lightweight, concise multistroke recognizer that uses only simple geometry and trigonometry. A full pseudocode listing is given as an appendix.

$N is a significant extension to the $1 unistroke recognizer, which has seen quick uptake in prototypes but has key limitations. $N goes further by (1) recognizing gestures comprising multiple strokes, (2) automatically generalizing from one multistroke to all possible multistrokes using alternative stroke orders and directions, (3) recognizing one-dimensional gestures such as lines, and (4) providing bounded rotation invariance. In addition, $N uses two speed optimizations, one with start angles that saves 79.1% of comparisons and increases accuracy 1.3%. The other, which is optional, compares multistroke templates and candidates only if they have the same number of strokes, reducing comparisons further to 89.5% and increasing accuracy another 1.7%. These results are taken from our study of algebra symbols entered *in situ* by middle and high schoolers using a math tutor prototype, on which $N was 96.6% accurate with 15 templates.

KEYWORDS: Gesture recognition, stroke recognition, symbols, marks, user interfaces, rapid prototyping, unistrokes, multistrokes.

INDEX TERMS: H.5.2. [Information interfaces and presentation]: User interfaces—*input devices and strategies*; I.5.5. [Pattern recognition]: Implementation—*interactive systems.*

1 INTRODUCTION

New technologies comprising pens, wands, touch, tabletops, and surfaces are regularly emerging. A key factor in the success of user interfaces designed with these elements is the speed with which user interface prototypes can be created, tested, and iterated upon [26]. Rapid prototyping tools and techniques are essential to good user interfaces emerging for these new paradigms.

One feature many of these prototypes may desire to employ is *multistroke* recognition (Figure 1). Examples are a two-stroke "X" to delete an object or a two-stroke arrow (→) to trigger a prompt for an annotation. But building or even using a state-of-the-art multistroke recognizer is nontrivial and may slow the rapid prototyping process with extensive training time and integration. Furthermore, gesture recognition libraries that exist on desktop platforms may not exist for new prototyping environments or on new platforms. What is needed is a lightweight, concise, quickly deployable multistroke recognizer capable of allowing user interface creators to define the gestures they want without the burden of implementing a complex recognizer. Other desirable properties include the ability for users to define their own gestures, enabling customization, and for the recognizer to operate at speeds supporting fluid interaction. Also, multistroke gestures can be made in multiple ways owing to stroke order and direction (*cf.* Figure 3), but neither designers nor users should have to define more than *one* version of each multistroke; rather, the system should recognize other variations automatically.

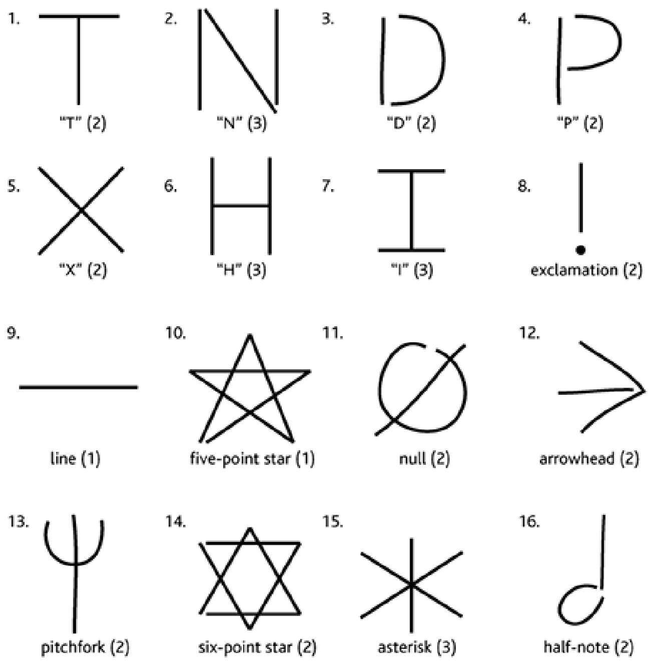

Figure 1. Multistrokes from the HTML/JavaScript version of $N. Such strokes are kept simple for human memorability and performance.

Although a concise recognizer specifically intended for rapid prototyping may not rival state-of-the-art recognizers in terms of power and complexity, it would nevertheless be an advance in human-computer interaction by providing a useful tool to the design process. Later, prototypes that become productized or otherwise formally deployed may choose to invest more heavily in the development or integration of a state-of-the-art recognizer.

1.1 A Lightweight Multistroke Recognizer

We have developed a multistroke recognizer called $N for user interface prototypes that meets the criteria above. $N is accurate, fast, simple, and easy to put into prototypes owing to its approximate 240 lines of code. Simple geometry and trigonometry are used to perform template matching between stored templates and entered candidates, giving $N a deterministic quality whereby candidates that look most like their templates are usually recognized as such. $N is a significant extension of the $1 unistroke recognizer by Wobbrock *et al.* [33], which uses about 100 lines of code and has seen quick uptake in user interface prototypes. Developers and researchers have created versions of $1 in ActionScript, Python, C#, C++, Objective-C, Java, JavaME, and JavaScript. $1 has been used in conjunction with computer

[1]Cherry Hill, NJ, USA, [2]Seattle, WA, USA
[1]lanthony@atl.lmco.com, [2]wobbrock@u.washington.edu
*Work by this author was done while affiliated with the Human-Computer Interaction Institute, Carnegie Mellon University.

vision [7] and has been incorporated into prototyping toolkits [34]. $1 also served as the basis for input to a video game that won Best Windows Game in *The Dobbs Challenge* for 2008 [24].

Unfortunately, $1 has limitations that warrant attention. For one, it is inherently a unistroke recognizer and does not handle gestures comprising multiple strokes (Figure 1). For another, $1 fails to recognize one-dimensional (1D) gestures such as lines. Also, $1 uses full rotation invariance, meaning symbols differing only by orientation (*e.g.*, **A** *vs.* ∀) cannot be distinguished. Prior work [17] shows that choosing good gestures is inherently difficult; ideally, a prototyper should not have to worry about the limits of the recognizer as well.

$N remedies the above limitations of $1. The contributions of $N include (a) a novel way to represent a multistroke as a set of unistrokes representing all possible stroke orders and directions; (b) the conception of a multistroke as inherently a unistroke, but where part of this unistroke is made away from the sensing surface; (c) the recognition of 1D gestures (*e.g.*, lines); (d) the use of bounded rotation invariance to support recognition of more symbols; and (e) an evaluation of $N on a set of handwritten algebra symbols made *in situ* by middle and high school students working with a math tutor prototype on Tablet PCs [2,3]. Although we could have evaluated $N with user interface gestures made by adults (*e.g.*, with those from Figure 1), the results would be predictable based on those from testing $1 [33], since $N uses $1 "under the hood." By testing $N with school students writing algebra symbols in a math tutor prototype, we see how $N might be used in the real world, and put $N to a more difficult test.

Overall, our study results show that $N achieved 96.6% accuracy with 15 templates for each of 20 different algebra symbols. Also, $N achieved 96.7% accuracy with 9 templates for each of 16 different gestures from the $1 *unistroke* corpus collected in a laboratory and made by adults [33]. Note that $N achieves these recognition rates even when the candidates are made using *different* stroke orders or directions than the predefined templates (*cf.* Figure 3).

We emphasize that $N's purpose, like $1's before it, is to facilitate rapid prototyping on just about any platform with minimal developer effort. $N does not attempt to solve many longstanding problems in sketch and handwriting recognition, *e.g.*, the segmentation problem. Determining where one multistroke ends and another begins is a method-specific consideration, and may be achieved with buttons, taps, pressure, timeouts, spatial layout, and so forth. Addressing it and other classic issues is not the goal of $N, although $N could be used to facilitate such explorations. The literature contains reviews of specific challenges and approaches [23,31,32].

2 RELATED WORK

Gesture, character, shape, sketch, and handwriting recognition are longstanding areas of research, and various approaches have been used, including finite state machines [11], Hidden Markov Models (HMMs) [1,28], neural networks [22], feature-based statistical classifiers [6,27], dynamic programming [19,30], template matching [13,14], and *ad hoc* heuristic recognizers [20]. These techniques have been used for both on-line and off-line recognition. Space precludes an in-depth review of these approaches. A nice review is found in the literature [12].

Many gesture recognition methods are ill-suited for use in rapid prototyping. Sophisticated pattern matching algorithms like neural networks [22] require numerous training examples and are unsuitable when the gesture set may not be defined prior to use. Studies show that machine learning can be a rather opaque topic for many programmers, and pattern matching is no exception [21].

Some recognizers do not represent gestures as strokes, but use atemporal geometric properties of the final drawn shape. Example properties are a gesture's bounding box or convex hull [4,5]. Although geometry-based recognizers can process multistroke gestures, they often do not generalize well to non-shape gestures such as handwriting [4]. Yin and Sun developed a geometric recognizer that can handle multistroke symbols [35], but this recognizer uses complex dynamic programming techniques.

In sum, gesture, character, shape, sketch, and handwriting recognition are active areas of research containing many approaches to the computational challenge of recognition. However, for the rapid prototyping of user interfaces, the power of these techniques, and the complexity that comes with that power, is often unnecessary. After all, user interface gestures must remain simple to facilitate human memorability and speed [17].

Like $N, other work has attempted to make gesture recognition easier for rapid prototyping. Some efforts have incorporated gesture recognizers in user interface toolkits. For example, SATIN [10] combined gesture recognition with other ink-handling support for developing informal pen-based user interfaces. Other toolkits with gesture recognition capabilities include those by Henry *et al.* [9], Landay and Myers [15], and the Amulet toolkit by Myers *et al.* [18]. Although such toolkits can be powerful development aids, they have not been widely adopted. In contrast, $N is concise, simple, and conventional enough to be implemented wherever necessary. For example, we have implemented $N for use on Web pages in JavaScript and HTML.[1]

$N is not the first multistroke extension of $1 to be published; in fact, Field *et al.* [8] also did this, using a very different approach from $N.[2] Their recognizer does not re-sample through the air, but only uses points actually drawn on the sensing surface, which allows them to use the same number of resampling points as $1, rather than the increased number used in $N to account for the possible loss in sensitivity. Also, their recognizer uses a more complex nearest-neighbor computation that minimizes the distance between pairs of matched points, over all possible pairings, rather than using drawing order (and explicitly storing order permutations), as $N does. However, $N is more suited for rapid prototyping of user interfaces because the increase in complexity introduced by Field *et al.*'s use of simulated annealing to find the best matching template. The average accuracy obtained by Field *et al.* for recognizing 3 symbols (AND, OR, and NOT gates) with 5 templates was about 90%—comparable to $N's results with 5 templates for each of 20 algebra symbols.

3 OVERVIEW OF THE $1 UNISTROKE RECOGNIZER

Because the $N multistroke recognizer builds upon the $1 unistroke recognizer [33], a brief overview of $1 is warranted. $1 matches templates; it compares an articulated candidate unistroke C to a set of stored templates T. The template T_i closest to C is the recognition result, where "closeness" is determined by the average Euclidean distance between corresponding points in C and T_i. Multiple T_i's can have the same name (*e.g.*, "arrow"), allowing for increased flexibility. Also, because candidates and templates are both unistrokes, a misrecognized candidate can immediately be added as a new template, allowing users to teach $1 at runtime.

An important step is how corresponding points are established. On both templates and candidates, $1 uses four steps, illustrated in Figure 2. First, it spatially resamples a stroke such that a fixed number of points are spread equidistantly along the stroke's path. Second, it rotates the stroke such that its "indicative angle," defined by the centroid $(\bar{x}, \bar{y})$ to the first point, is at 0°. This serves as an approximation for alignment, but *Golden Section Search* (GSS) [25] (pp. 397-402) is later used to find the optimal

[1] http://depts.washington.edu/aimgroup/proj/dollar/ndollar.html

[2] Field *et al.* [8] mention $N and compare their algorithm to it based on the $N Web page and online JavaScript.

angular alignment. Third, it scales the stroke non-uniformly to match a reference square. Fourth, it translates the stroke so that its centroid is at the origin. These steps normalize all strokes so that each point in a candidate corresponds spatially with one point in a template. The template T_i with the least point-to-point distance from the candidate is the recognition result.

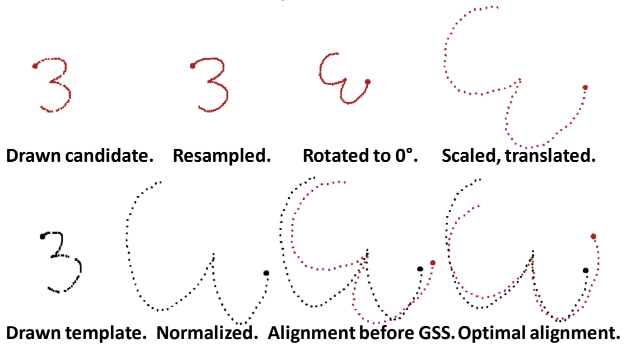

Figure 2. Steps in the $1 matching process and the aligning of a candidate and template. GSS means *Golden Section Search*.

Limitations of $1 include its restriction to 2D unistrokes, its inability to recognize lines, and its inability to support orientation-dependence. These are all remedied in $N.

4 THE $N MULTISTROKE RECOGNIZER

The goal of $N is to provide a useful, concise, easy-to-incorporate multistroke recognizer deployable on almost any platform to support rapid prototyping. $N's goals are shared with $1, but $N is much more versatile by (1) recognizing gestures comprising multiple strokes, (2) automatically generalizing from one multistroke template to all possible multistrokes with alternative stroke orderings and directions, (3) recognizing 1D gestures such as lines, and (4) providing bounded rotation invariance. Despite these capabilities, $N only employs about twice the code of $1 (mostly handling the additional complexity of generating multistroke permutations), making $N easy to write from scratch using the pseudocode listing in Appendix A. This gives $N an advantage over platform-specific libraries or toolkits, especially in new prototyping environments.

4.1 $N Algorithm

A key challenge of performing online recognition of multistroke gestures is that within any multistroke, the component strokes can be made in any order and in either direction (Figure 3). Even a simple circle can be made clockwise or counterclockwise. In $1, this kind of variation must be handled by defining all desired gesture variations. But it is infeasible to require designers or users to define all permutations of a multistroke gesture. For $N to be effective as a prototyping tool, it must enable designers and users to define *one* multistroke, and ensure that different stroke orders and/or directions will be properly recognized.

To permit this flexibility, $N computes and stores each multistroke's "unistroke permutations." Each permutation represents one possible combination of stroke order and direction, *i.e.*, one of the 8 possibilities in Figure 3, which is then made into a unistroke by simply connecting the endpoints of component strokes. This is, in effect, treating a multistroke as if it were fundamentally a unistroke, but where part of the stroke is made away from the sensing surface, *i.e.*, by following the user's hand. Thus, a two-stroke "x" permutes into the 8 unistrokes of Figure 4.

We use *Heap Permute* [16] (p. 179) to generate all stroke orders in a multistroke gesture. Then, to generate stroke directions for each order, we treat each component stroke as a dichotomous [0,1] variable. There are 2^N combinations for N strokes, so we convert the decimal values 0 to 2^N-1, inclusive, to binary representations and regard each bit as indicating forward (0) or reverse (1). This algorithm is often used to generate truth tables in propositional logic.

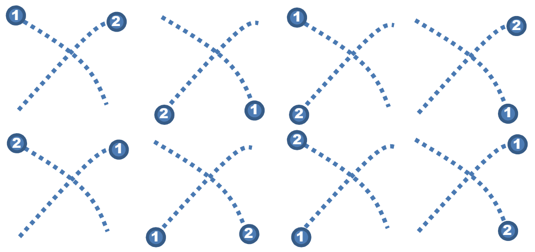

Figure 3. The 8 possibilities for a two-stroke "x". The numbered dots indicate stroke order and beginnings.

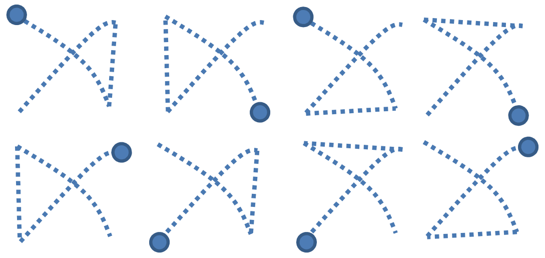

Figure 4. The 8 unistroke permutations for a two-stroke "x" based on the two-stroke gestures in Figure 3.

The permuting of multistrokes takes place only *once* when they are defined or loaded. At runtime, when a candidate multistroke is articulated, its component strokes are simply connected in the order drawn to form a unistroke, which is then preprocessed according to the usual steps (*cf.* Figure 2) and compared to unistroke permutations using Euclidean distance as in $1. The score for a multistroke template is the *best* of its unistroke permutations' scores.

Admittedly, the approach of creating unistroke permutations to represent multistroke templates "under the hood" results in a combinatoric explosion. Three things, however, mitigate this explosion during recognition: (1) in practice, most multistroke gestures have only a few strokes because more elaborate gestures are harder for users to remember and use [17]; (2) the comparison of a candidate gesture to each unistroke is very fast—much faster than many other template matching algorithms like dynamic time warping [19,33]; and (3) we employ two speed optimizations, one mandatory and one optional, which reduce the number of candidate-to-unistroke comparisons by about 80-90%. In our study, each optimization actually *increased* accuracy by about 1-2%, because the optimizations cull potential candidates based on relevant features not otherwise used by the simple template matching process of $1. The number of unistrokes U resulting from a multistroke with S component strokes is:

$$U = \prod_{i=1}^{S} 2i \qquad (1)$$

4.2 Bounded Rotation Invariance

The $1 algorithm operated with full rotation invariance (±180°), meaning the drawn orientation of a template or candidate gesture was ignored in the matching process. However, there may be times when otherwise identical symbols differ only in their orientation. For example, to distinguish "**A**" from "∀", rotation must be bounded by less than ±90°. That is, if "**A**" is rotated clockwise 90° and "∀" is rotated counterclockwise 90°, they will match. For this reason, $N has the option of bounding rotation invariance by an arbitrary amount. This feature could be employed on a per-gesture basis by flagging which templates are orientation-dependent at design-time (*e.g.*, with a checkbox).

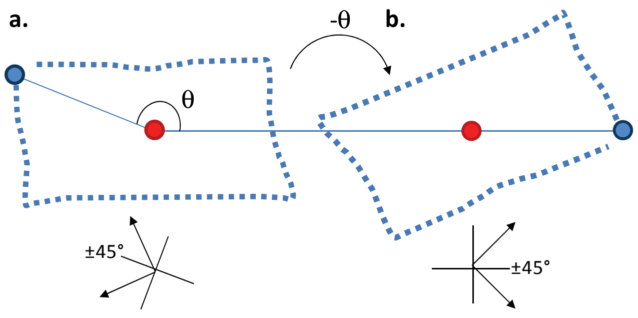

Figure 5. **(a)** A rectangle at its drawn orientation with its indicative angle θ°. For bounded rotation invariance, the gesture will be returned to this orientation and searched over ±45°. **(b)** For full rotation invariance, the gesture is left rotated by -θ° and the search for the best alignment works over ±45° from there.

An algorithmic convenience regarding bounded rotation invariance in $N is that, as noted above, both $1 and $N use Golden Section Search (GSS) [25] to find the best alignment between a template and candidate. Work on $1 showed that it is sufficient to parameterize GSS with ±45° for its search. Thus, if bounded rotation invariance is desired, the template and candidate, after scaling, can be rotated *back* to their drawn orientation, and GSS will search from there (Figure 5a). If full rotation invariance is desired, the gestures will be left rotated such that their indicative angle is at 0°, and GSS will search by the same amount from there (±45°; Figure 5b). Thus, the search procedure itself remains unchanged.

4.3 Recognition of One-Dimensional Gestures

The non-uniform scaling in $1 causes it to mistreat 1D gestures like horizontal and vertical lines. One option for solving this, but not the one adopted here, would be to have designers flag templates explicitly as 1D at design-time (*e.g.*, with a checkbox). Then, when being compared to these flagged templates, candidates would be scaled *uniformly* so as to preserve their aspect ratio. (Recall that 2D gestures are scaled non-uniformly by $1 and $N; *cf.* Figure 2.) The problem with this approach is that it may not be clear at design-time what the proper setting should be. A vertical bar (|) is clearly 1D, but is a left curly brace ({)? It probably depends on how it is drawn. Also, users who are able to define gestures at run-time should not be concerned with such algorithmic issues. Placing the burden on designers or users to make the 1D *vs.* 2D choice is undesirable.

$N solves this problem by *automatically* differentiating between 1D and 2D gestures, and then scaling them uniformly or non-uniformly, respectively. To classify a gesture as either 1D or 2D, $N uses the ratio of the sides of a gesture's oriented bounding box (MIN-SIDE *vs.* MAX-SIDE). If this ratio is less than a threshold, the gesture is considered 1D and is scaled to preserve aspect. This allows lines and other thin gestures to be recognized correctly without having to explicitly flag them.

For the algebra symbols obtained during our study of a math tutor prototype [2,3], we empirically derived a MIN-SIDE-to-MAX-SIDE threshold of 0.30. This threshold was discovered by first computing the oriented bounding box of all symbols, and then comparing the classification of each symbol under varying thresholds to hand-labeled ground truth. We performed an ROC curve analysis to determine the best threshold to use (Figure 6). A threshold of 0.30 gives a true positive rate of 97.0% and a false positive rate of 4.4%. We chose it because 2D gestures are more numerous in our algebra corpus than 1D gestures, and 2D gestures suffer more if scaled improperly. The threshold may need to be adjusted slightly based on performance for use in other symbol sets, and should usually range from 0.20 to 0.35.

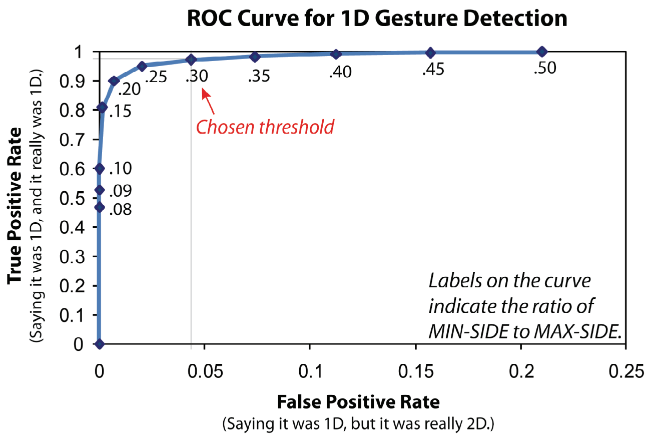

Figure 6. ROC curve of true positives *vs.* false positives. The chosen threshold ratio of MIN-SIDE to MAX-SIDE for our algebra symbols was 0.30.

4.4 Speed Optimization Using Start Angles

$N supports all possible ways of making each multistroke. However, with this flexibility comes a combinatoric explosion of underlying unistrokes to which a candidate must be compared. Although each comparison is fast, having many multistrokes comprising many strokes can result in a slowdown.

To solve this problem, we employ a speed optimization based on gestures' start angles. The idea is to only compare unistrokes whose start directions are "about the same." The start direction is computed only once for every unistroke at the end of the preprocessing steps (*cf.* Figure 2). Then, during recognition, any template that does not begin in the same direction as the candidate is skipped, avoiding the search for the best alignment (Figure 7).

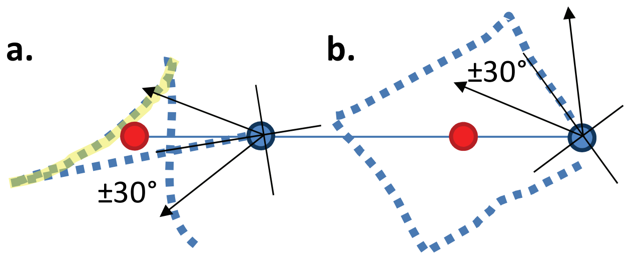

Figure 7. Blue dots are first points; red dots are centroids. **(a)** A two-stroke cross permuted as a unistroke and rotated to 0°. The yellow strip indicates the part of the gesture resampled off the sensing surface. **(b)** A one-stroke rectangle rotated to 0°. These gestures are not compared because they do not begin in the same general direction (±30°).

To determine the start direction of a gesture, we compute the angle formed from the start point through the eighth interval (*i.e.*, point[0] to point[8]). This setting was determined by conducting an experiment using a random subset of our algebra symbols. We compared the six combinations formed by two intervals (fourth, eighth) and three angular windows (±30°, ±45°, and ±60°). The best results were obtained with the eighth interval and ±30°, although the results were close, suggesting there exists tolerance in this choice. $N resamples gestures to 96 points. Anecdotally, it seems that the start-angle interval should be about an eighth of the way through the gesture (*e.g.*, 96/8 = 12).

Using full rotation invariance, the start direction of a unistroke is unaffected by its drawn orientation. Using bounded rotation invariance, the angle is affected by the gesture's drawn orientation, which is good because we want to match candidates drawn at similar orientations.

In the study of our algebra symbols, the start angle optimization reduced unistroke comparisons by 79.1%. With speed savings this large, one might expect accuracy to decline, but the opposite occurred; the accuracy improved 1.3%. This result seems because the optimizations cull potential candidates based on relevant features not otherwise used by the simple template matching process of $1. Testing $N on the $1 unistroke corpus [33] showed that without this optimization, $N took 1.92 times longer than $1 to complete. With this optimization, $N took only 0.67 times as long; that is, $N was *faster* than $1. In general, $N is sufficiently fast for interactive use with 20-30 user interface templates. It is unlikely user interface prototypes would employ more gestures than this due to the limits of human memory.

4.5 Speed Optimization Using Number of Strokes

There may be times when designers or users will expect that their multistrokes will be made with a fixed number of strokes. For example, making a plus sign (+) or equals sign (=) with anything other than two strokes would be unusual. When one's gesture set allows for it, an optional speed optimization can restrict comparisons of candidates to templates that comprise the same number of strokes, skipping all others.

When testing our algebra symbols, we added this option in conjunction with the start angle optimization. Although the middle and high school students were *not* instructed by the math tutor prototype to write symbols using a specific number of strokes, the option to compare only multistrokes containing the same number of strokes resulted in an additional 10.4% reduction in comparisons, for 89.5% total. Like the start angle optimization, this optimization *increased* accuracy, this time by an additional 1.7% for a total of almost 3% improvement, despite being 89.5% faster than the core algorithm using neither optimization.

4.6 Limitations of $N

With $N's relative simplicity comes inevitable limitations. As a geometric template matcher, $N cannot reason about gesture features. This presents problems when matching gestures whose *gestalt* is their appeal, *e.g.*, a messy scratch-out for erasing. While a benefit of $N using unistroke permutations is that it can recognize multistrokes made with *fewer* component strokes than defined in a template (*e.g.*, a 1-stroke "D" candidate will match a 2-stroke "D" template), candidates made with *more* strokes will not match unless those strokes proceed in the same direction as the template. Although $N has provisions for orientation dependence and aspect ratio, it lacks provisions for scale or position dependence. Also, as with any recognizer, collisions are possible [17]; for example, if the optional number-of-strokes speed optimization is not used, $N may have trouble distinguishing a two-stroke equals sign (=) and a one-stroke "z". Finally, despite its speed optimizations, if $N is used with a large set of multistroke templates (30+), some of which contain many component strokes (5+), slowdowns are likely during the recognition process depending on hardware and other factors.

In practice, although these limitations make $N less powerful than a state-of-the-art sketch or handwriting recognizer, these limitations do not readily affect $N's suitability as a rapid prototyping tool. The goal is to provide an accurate, fast, easily deployable recognizer for small, useful sets of interface gestures. The utility of such sets is limited by human memory and performance, and we know from prior work that gestures should be as distinct, simple, and quick to make as possible [17]. This lessens the algorithmic burden on a recognizer intended for rapid prototyping, but heightens the importance of a quick-to-use approach that avoids extensive coding, training, and fine-tuning like, say, a neural network requires [22]. As our study shows, $N is well-suited to this purpose.

5 Study of a Pen-Based Math Tutor Prototype

Although we could replicate $1's study of user interface gestures [33] like those shown in Figure 1, the results are predictable, given that $N uses $1's algorithm "under the hood" and $1 recognized similar gestures, albeit unistrokes. Instead, we chose to sacrifice experimental control for a more challenging *in situ* study where gestures were not provided by adults, but by youth from middle and high school classrooms. The prototype investigated was a cognitive math tutor with a pen-based interface on Tablet PCs [2,3], and the gestures were not collected for the purpose of evaluating $N, but previously for evaluating the tutor. We reasoned that if $N could perform well on students' algebra symbols, it could be effective on user interface symbols.

5.1 Method

5.1.1 Participants and Apparatus

Forty middle and high school students aged 11-17 provided pen gestures. Most students had not used pen-based input prior to the study. Gestures were collected on Tablet PCs running software that recorded students' strokes. Students copied algebra equations displayed one at a time on the screen. Each student copied 45 equations such as "$2x + 3 = 10$". Students could not erase their strokes once written. After the experiment, the students' strokes were hand-segmented and hand-labeled for ground truth.

5.1.2 Algebra Symbols

Students supplied gestures for 20 distinct algebra symbols as part of their equations. They could write these symbols however they wanted: some almost always were unistrokes (*e.g.*, 3, *c*), while others were multistrokes (*e.g.*, 4, *x*, =, +), and still others varied from student-to-student (e.g., 5, *a*, *b*, *y*).

$$0\text{-}9,\ a\text{-}c,\ x,\ y,\ =,\ +,\ -,\ (,\ ).$$

The final test corpus contained 15,309 gestures. Seventy-percent of these gestures were unistrokes, 30.0% were multistrokes, and 13.4% were 1D. Although our interest in testing $N was primarily in multistroke gestures, user interface prototypes often utilize a mixture of unistrokes and multistrokes, and it is important for $N to succeed on both.

5.1.3 Recognizer Training and Testing

To facilitate comparisons, our procedure for testing $N was based on that of Wobbrock *et al.* for $1 [33]. We tested $N under various configurations on the algebra corpus, and also tested $N on the released $1 unistroke corpus.

Although the term "training" is more suited to feature-based statistical classifiers like Rubine [27], in a sense, we train $N whenever we supply it with a named template. Of a given student's gestures, the number of training examples (*i.e.*, templates) T for each of the 20 gesture types was increased stepwise from T=1 to 15. Because students were not always issued the same algebra equations *in situ*, not every algebra symbol was written the same number of times. Thus, starting at $T = 9$, some symbols were omitted if they were not numerous enough to support training equally across students.

For 100 times per level of T, T training examples were chosen randomly for each algebra symbol for a given student. From the unchosen gestures, one was picked at random and tested as the candidate. Over the 100 tests per symbol per level of T per student, correct outcomes were averaged into a recognition rate.

For a single student, there were about 21,500 recognition attempts. With 40 students, the experiment consisted of about 860,000 recognition attempts. The details of every test were logged, including full N-best lists.

As mentioned, we also tested $N on the released $1 unistroke corpus. This followed the same procedure except that *T* ranged from 1 to 9 for 16 different symbols because fewer examples per subject were available. For more details, readers are directed to the $1 paper [33].

5.2 Results

The recognition results for our various tests are shown in Figure 8. $N reached 96.6% accuracy on the algebra symbols with 15 training examples, and 96.7% on the $1 unistroke corpus with 9 training examples. $N was expectedly poorer than $1 on the $1 corpus of unistrokes ($t(8)=8.12$, $p<.0001$), which reached 99.6%.

For the algebra corpus, we see steady accuracy increases with each additional option. With no options, $N reached 93.6% accuracy and took 569 minutes to complete the entire automated test. $N improved significantly with the start angle optimization ($t(14)=6.47$, $p<.0001$), reaching 95.0% accuracy in 128 minutes. Adding in the same-number-of-strokes optimization also significantly improved accuracy ($t(14)=8.62$, $p<.0001$), reaching 95.7% in just 65 minutes. As an exploration, we tried uniformly scaling *all* symbols, not just 1D ones, and $N improved significantly to 96.6% accuracy also in 65 minutes ($t(14)=7.60$, $p<.0001$). Thus, these options significantly improved both $N speed and accuracy. For all series, there was no significant difference between multistroke and unistroke accuracy.

On the unistroke corpus, $N with no options performed slightly better than with the start angle ($t(8)=3.75$, $p<.01$) and uniform scaling options ($t(8)=3.25$, $p<.05$), which themselves were not significantly different ($t(8)=2.02$, *n.s.*). The start angle optimization reduced recognition time from 37 to 13 minutes.

6 DISCUSSION

$N achieved reasonable recognition rates on both the algebra and unistroke corpora. We should expect that $N would not perform as well as $1 on the unistroke corpus because $N is more complex: it turns each unistroke into two, one for each direction, and automatically differentiates 1D from 2D gestures using a 0.30 MIN-SIDE to MAX-SIDE threshold derived from the algebra corpus. Although there were no 1D gestures in the $1 corpus, some subjects may have made checkmarks (✓), arrows (→), left ([) and right (]) square brackets, and left ({) and right (}) curly braces thinly enough to be classified as 1D. Nevertheless, $N reached 96.7% accuracy. This suggests the possibility of a hybrid design, which, if one's application allows for it, runs $N on multistrokes and $1 on unistrokes.

$N did not perform quite as well on the algebra corpus as it did on the unistroke corpus. $N's accuracy on the unistroke corpus with only 9 templates (96.7%) almost matched its accuracy on the algebra corpus with 15 templates (96.6%). Six factors may explain this difference between corpora. First, $N was used in part on multistroke gestures, which may have more inherent variability than unistrokes. Second, $N was tested on 1D gestures, which had to be automatically distinguished from 2D gestures. Third, the algebra corpus was made *in situ* by middle and high school students, whereas $1's corpus was made in a lab by adults. Fourth, some algebra symbols were quite similar, like "*x*" and "+", or "*a*" and "9". (The most challenging algebra gestures for $N to recognize with 1 template loaded were "c", "2", "y", "9", and "3". The most confusable pairs were "x" and "y", "4" and "y", "(" and "c", "x" and "+", and "9" and "a".) Fifth, $N was tested on 20 symbols, whereas $1 was tested on 16. Sixth, $N used only partial rotation invariance, rather than full rotation invariance like $1, making recognition less accurate if students wrote symbols at unusual angles.

In light of these challenges, the results are quite satisfying. They become more so when $N is compared to more complex recognizers that require more training examples, such as the *Freehand Formula Entry System* (FFES) [29], which suggests 20-40 examples per symbol per user. With 15 training examples, FFES was 91.5% accurate on our algebra corpus [3], lower than the 96.6% accuracy achieved by $N.

It is interesting that both speed optimizations improved recognition accuracy on the algebra corpus but not on the unistroke corpus. The unistroke corpus was tested with full rotation invariance, while the algebra corpus was tested with ±45° from the drawn orientation. It is likely that the start angle optimization is more effective when bounded rotation invariance is used because a matching start angle is more discriminating when drawing orientation matters. Also, the fact that the same-number-of-strokes optimization improved accuracy shows that most algebra symbols were made with the same stroke count.

7 FUTURE WORK

We chose to evaluate $N on a mixture of gestures made *in situ* by middle and high school students on a pen-based math tutor prototype [2,3], reasoning that if $N can perform well, it should be suitable for recognizing user interface gestures just as $1 is. Nevertheless, $N should ultimately be evaluated for its usefulness to rapid prototypers. If the uptake of $1 is indicative, then $N may also be quickly adopted. $1's popularity has led to others working on multistroke extensions to it, such as Field *et al*. [8]; an obvious important test would be to compare such competitors to $N in terms of accuracy and ease of use.

$N was tested in a writer-dependent fashion, which is appropriate for prototypers testing and demonstrating their own systems. However, in cases where training data for new users of a prototype are unavailable, or where the prototype is collaborative and multi-user, $N will be used in a writer-independent fashion, which should be studied.

Additional features could be added to $N to support scale or position dependence. These options, along with others, could be exposed by a gesture design tool (*e.g.*, [17]) that could also automatically determine optimal MIN-SIDE to MAX-SIDE ratios based on drawn or loaded examples (*cf.* Figure 6).

We could also prevent the collisions that may occur due to resampling gestures "through the air," *e.g.*, equals sign (=) and "z". $N could tag each resampled point as occurring on "land" or "air," allowing the recognizer to match only the same type of points to each other.

8 CONCLUSION

The popularization of pen, wand, touch, tabletop, and surface computing raises the need for easy creation of stroke-based gesture recognition to facilitate rapid prototyping. We have shown that $N is capable of producing good recognition rates on algebra symbols from a math tutor prototype, indicating that $N should be effective for smaller user interface symbol-sets. $N generalizes from single multistroke templates to all possible stroke orders and directions, enabling each type of multistroke to be defined once. $N also automatically distinguishes between 1D and 2D gestures, provides for bounded rotation invariance, and employs two highly effective speed optimizations that also improve accuracy. We expect that $N should be useful for quickly adding multistroke gestures to user interface prototypes, just as $1 has been for unistrokes. An application developer can take the pseudocode listing provided in this paper, or the reference JavaScript or C# implementations on the $N website, and translate them into the platform-specific language of his choice for the prototype. This scenario has already been realized by the hobbyist developer of *AlphaCount*, an iPhone application that teaches kids to recognize and write numbers. It uses the $N recognizer and is available at http://itunes.apple.com/us/app/alphacount/id359046783.

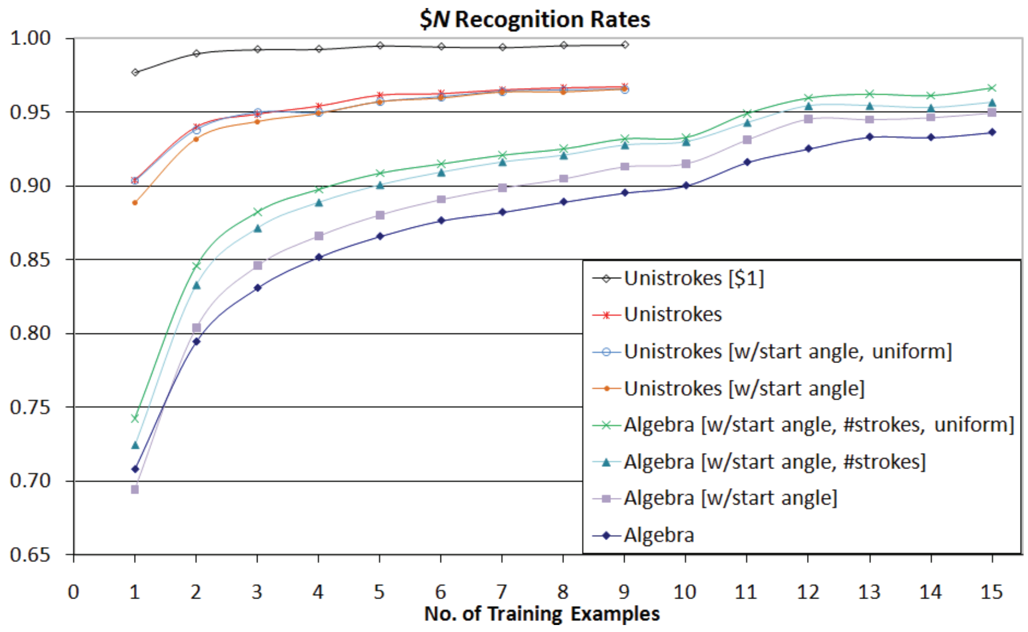

Figure 8. $N recognition rates as a function of training examples. The legend's order from top to bottom matches the order of series from top to bottom. $N was tested on two corpora, the unistrokes from the original $1 study [33] and the algebra symbols made by middle and high schoolers. The top series is the original $1 results. The next three series are $N on the $1 unistroke corpus. The final four series are $N on the algebra corpus. Configuration options include *start angle*, which refers to the start angle optimization; *#strokes*, which refers to the optimization for only comparing multistrokes made with the same number of strokes; and *uniform*, which refers to all gestures being uniformly scaled.

ACKNOWLEDGMENTS

The authors thank Isaac Simmons for advice and consultation.

REFERENCES

[1] Anderson, D., Bailey, C. and Skubic, M. (2004) Hidden Markov Model symbol recognition for sketch-based interfaces. *AAAI Fall Symposium*. Menlo Park, CA: AAAI Press, 15-21.

[2] Anthony, L., Yang, J. and Koedinger, K.R. (2007) Benefits of handwritten input for students learning algebra equation solving. *Proc. Artificial Intelligence in Education (AIEd '07)*. Amsterdam, The Netherlands: IOS Press, 521-523.

[3] Anthony, L., Yang, J. and Koedinger, K.R. (2008) Toward next-generation, intelligent tutors: Adding natural handwriting input. *IEEE Multimedia 15* (3), 64-68.

[4] Apte, A., Vo, V. and Kimura, T.D. (1993) Recognizing multistroke geometric shapes: An experimental evaluation. *Proc. UIST '93*. New York: ACM Press, 121-128.

[5] Calhoun, C., Stahovich, T.F., Kurtoglu, T. and Kara, L.B. (2002) Recognizing multi-stroke symbols. *AAAI Spring Symposium*. Menlo Park, CA: AAAI Press, 15-23.

[6] Cho, M.G. (2006) A new gesture recognition algorithm and segmentation method of Korean scripts for gesture-allowed ink editor. *Information Sciences 176* (9), 1290-1303.

[7] deadpocky. (2008) Object tracker with $1 gesture recognizer. *YouTube*. http://www.youtube.com/watch?v=wl7fLUs7QX4.

[8] Field, M., Gordon, S., Peterson, E., Robinson, R., Stahovich, T. and Alvarado, C. (2009) The effect of task on classification accuracy: Using gesture recognition techniques in free-sketch recognition. *Proc. Eurographics SBIM '09*. New York: ACM Press, 109-116.

[9] Henry, T.R., Hudson, S.E. and Newell, G.L. (1990) Integrating gesture and snapping into a user interface toolkit. *Proc. UIST '90*. New York: ACM Press, 112-122.

[10]Hong, J.I. and Landay, J.A. (2000) SATIN: A toolkit for informal ink-based applications. *Proc. UIST '00*. New York: ACM Press, 63-72.

[11]Hong, P., Turk, M. and Huang, T.S. (2000) Constructing finite state machines for fast gesture recognition. *Proc. ICPR '00*. Los Alamitos, CA: IEEE Press, 691-694.

[12]Johnson, G., Gross, M.D., Hong, J. and Do, E.Y.-L. (2009) Computational support for sketching in design: A review. *Foundations and Trends in Human-Computer Interaction 2* (1), 1-93.

[13]Kara, L.B. and Stahovich, T.F. (2004) An image-based trainable symbol recognizer for sketch-based interfaces. *AAAI Fall Symposium*. Menlo Park, CA: AAAI Press, 99-105.

[14]Kristensson, P.-O. and Zhai, S. (2004) SHARK^2: A large vocabulary shorthand writing system for pen-based computers. *Proc. UIST '04*. New York: ACM Press, 43-52.

[15]Landay, J. and Myers, B.A. (1993) Extending an existing user interface toolkit to support gesture recognition. *Companion to INTERCHI '93*. New York: ACM Press, 91-92.

[16]Levitin, A.V. (2003) *Introduction to the Design and Analysis of Algorithms*, 1st ed. Reading, MA: Addison-Wesley.

[17]Long, A.C., Landay, J.A. and Rowe, L.A. (1999) Implications for a gesture design tool. *Proc. CHI '99*. New York: ACM Press, 40-47.

[18]Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A. and Doane, P. (1997) The Amulet environment: New models for effective user interface software development. *IEEE Trans. Software Engineering 23* (6), 347-365.

[19]Myers, C.S. and Rabiner, L.R. (1981) A comparative study of several dynamic time-warping algorithms for connected word recognition. *The Bell System Technical Journal 60* (7), 1389-1409.

[20]Notowidigdo, M. and Miller, R.C. (2004) Off-line sketch interpretation. *AAAI Fall Symposium*. Menlo Park, CA: AAAI Press, 120-126.

[21]Patel, K., Fogarty, J., Landay, J.A. and Harrison, B. (2008) Examining difficulties software developers encounter in the adoption of statistical machine learning. *Proc. AAAI '08*. Menlo Park, CA: AAAI Press, 1563-1566.

[22]Pittman, J.A. (1991) Recognizing handwritten text. *Proc. CHI '91*. New York: ACM Press, 271-275.

[23]Plamondon, R. and Srihari, S.N. (2000) On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Trans. Pattern Analysis & Machine Intelligence 22* (1), 63-84.

[24]POW Studios. (2008) *Mr. Spiff's Revenge*. http://dobbschallenge.com/index.php/View-document-details/72-Mr.-Spiff-s-Revenge-POW-Studios.html.

[25]Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (1992) *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press.

[26]Rettig, M. (1994) Prototyping for tiny fingers. *Communications of the ACM 37* (4), 21-27.

[27]Rubine, D. (1991) Specifying gestures by example. *Proc. SIGGRAPH '91*. New York: ACM Press, 329-337.

[28]Sezgin, T.M. and Davis, R. (2005) HMM-based efficient sketch recognition. *Proc. IUI '05*. New York: ACM Press, 281-283.

[29]Smithies, S., Novins, K. and Arvo, J. (2001) Equation entry and editing via handwriting and gesture recognition. *Behaviour & Information Technology 20* (1), 53-67.

[30]Tappert, C.C. (1982) Cursive script recognition by elastic matching. *IBM Journal of Research and Development 26* (6), 765-771.

[31]Tappert, C.C., Suen, C.Y. and Wakahara, T. (1990) The state of the art in online handwriting recognition. *IEEE Trans. Pattern Analysis & Machine Intelligence 12* (8), 787-808.

[32]Tappert, C.C. and Cha, S.-H. (2007) English language handwriting recognition interfaces. In *Text Entry Systems*, I. S. MacKenzie and K. Tanaka-Ishii (eds.). San Francisco: Morgan Kaufmann, 123-138.

[33]Wobbrock, J.O., Wilson, A.D. and Li, Y. (2007) Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. *Proc. UIST '07*. New York: ACM Press, 159-168.

[34]Yeh, R.B., Paepcke, A. and Klemmer, S.R. (2008) Iterative design and evaluation of an event architecture for pen-and-paper interfaces. *Proc. UIST '08*. New York: ACM Press, 111-120.

[35]Yin, J. and Sun, Z. (2005) An online multi-stroke sketch recognition method integrated with stroke segmentation. *Proc. ACII '05*. Berlin, Germany: Springer-Verlag, 803-810.

APPENDIX A

Step 1. Take a multistroke gesture *strokes* and generate unistroke permutations. For gestures serving as templates, Step 1, which uses Steps 3-6, should be carried out once on the input points. For candidates, Steps 2-7 should be applied to the input points. For constants we use N=96, *size*=250, ∂=.30, O=(0,0), and I=12.

```
GENERATE-UNISTROKE-PERMUTATIONS(strokes)
    for $i$ from 0 to $|strokes|$ do $order_i \leftarrow i$
    HEAP-PERMUTE($|strokes|$, order, out orders)
    $M \leftarrow$ MAKE-UNISTROKES(strokes, orders)
    foreach unistroke $U$ in $M$ do
        $U_{points} \leftarrow$ RESAMPLE($U_{points}$, $N$)   // step 3
        $\omega \leftarrow$ INDICATIVE-ANGLE($U_{points}$)   // step 4
        $U_{points} \leftarrow$ ROTATE-BY($U_{points}$, $-\omega$)
        $U_{points} \leftarrow$ SCALE-DIM-TO($U_{points}$, size, $\partial$)   // step 5
        $U_{points} \leftarrow$ CHECK-RESTORE-ORIENTATION($U_{points}$, $+\omega$)
        $U_{points} \leftarrow$ TRANSLATE-TO($U_{points}$, $O$)
        $U_{vector} \leftarrow$ CALC-START-UNIT-VECTOR($U_{points}$, $I$)   // step 6
HEAP-PERMUTE(n, order, out orders)
    if $n = 1$ then APPEND(orders, order)
    else
        for $i$ from 0 to $n$ do
            HEAP-PERMUTE($n$-1, order, out orders)
            if IS-ODD($n$) then SWAP($order_0$, $order_{n-1}$)
            else SWAP($order_i$, $order_{n-1}$)
MAKE-UNISTROKES(strokes, orders)
    foreach order $R$ in orders do
        for $b$ from 0 to $2^{|R|}$ do
            for $i$ from 0 to $|R|$ do
                if BIT-AT($b$, $i$) = 1 then   // $b$'s bit at index $i$
                    APPEND(unistroke, REVERSE($strokes_{R_i}$))
                else APPEND(unistroke, $strokes_{R_i}$)
            APPEND(unistrokes, unistroke)
    return unistrokes
```

Step 2. Combine candidate *strokes* into one unistroke *points* path.

```
COMBINE-STROKES(strokes)
    for $i$ from 0 to $|strokes|$ do
        for $j$ from 0 to $|strokes_i|$ do
            APPEND(points, $strokes_{i_j}$)   // append each point
    return points
```

Step 3. Resample a *points* path into *n* evenly spaced points. RESAMPLE remains unchanged from Step 1 in [33]. The reader is directed there.

Step 4. Find and save the indicative angle ω from the *points*' centroid to first point. Then rotate by $-\omega$ to set this angle to 0°.

```
INDICATIVE-ANGLE(points)
    $c \leftarrow$ CENTROID(points)   // computes $(\bar{x}, \bar{y})$
    return ATAN($c_y - points_{0_y}$, $c_x - points_{0_x}$)   // for $-\pi \le \omega \le \pi$
ROTATE-BY(points, $\omega$)
    $c \leftarrow$ CENTROID(points)
    foreach point $p$ in points do
        $q_x \leftarrow (p_x - c_x)$ COS $\omega - (p_y - c_y)$ SIN $\omega + c_x$
        $q_y \leftarrow (p_x - c_x)$ SIN $\omega + (p_y - c_y)$ COS $\omega + c_y$
        APPEND(newPoints, $q$)
    return newPoints
```

Step 5. Scale dimensionally-sensitive based on threshold ∂=.30. Next, *if* using bounded rotation invariance, restore drawn orientation by rotating $+\omega$. Then translate to the origin O=(0,0).

```
SCALE-DIM-TO(points, size, $\partial$)
    $B \leftarrow$ BOUNDING-BOX(points)
    foreach point $p$ in points do
        if MIN($B_{width} / B_{height}$, $B_{height} / B_{width}$) $\le \partial$ then   // uniform
            $q_x \leftarrow p_x \times size$ / MAX($B_{width}$, $B_{height}$)
            $q_y \leftarrow p_y \times size$ / MAX($B_{width}$, $B_{height}$)
        else   // non-uniform
            $q_x \leftarrow p_x \times size / B_{width}$
            $q_y \leftarrow p_y \times size / B_{height}$
        APPEND(newPoints, $q$)
    return newPoints
CHECK-RESTORE-ORIENTATION(points, $\omega$)
    if using bounded rotation invariance then
        points $\leftarrow$ ROTATE-BY(points, $\omega$)
    return points
TRANSLATE-TO(points, k)
    $c \leftarrow$ CENTROID(points)
    foreach point $p$ in points do
        $q_x \leftarrow p_x + k_x - c_x$
        $q_y \leftarrow p_y + k_y - c_y$
        APPEND(newPoints, $q$)
    return newPoints
```

Step 6. Calculate the start unit vector *v* for *points* using index I=12.

```
CALC-START-UNIT-VECTOR(points, I)
    $q_x \leftarrow points_{I_x} - points_{0_x}$
    $q_y \leftarrow points_{I_y} - points_{0_y}$
    $v_x \leftarrow q_x / \sqrt{(q_x^2 + q_y^2)}$
    $v_y \leftarrow q_y / \sqrt{(q_x^2 + q_y^2)}$
    return $v$
```

Step 7. Match candidate *points* having start unit vector *v*, processed from the raw *strokes* in Step 2, where now $S = |strokes|$, against unistroke permutations *U* within each multistroke *M*. We use Φ = 30° for the start angle similarity threshold. DISTANCE-AT-BEST-ANGLE remains unchanged from Step 4 in [33]. The reader is directed there. We pass it θ=±45° and θ_Δ=2°.

```
RECOGNIZE(points, v, S, multistrokes)
    $b \leftarrow +\infty$
    foreach multistroke $M$ in multistrokes do
        if $S = |M_{strokes}|$ then   // optional: require same # strokes
            foreach unistroke $U$ in $M$ do
                if ANGLE-BETWEEN-VECTORS($v$, $U_{vector}$) $\le \Phi$ then
                    $d \leftarrow$ DISTANCE-AT-BEST-ANGLE(points, $U$, $-\theta$, $\theta$, $\theta_\Delta$)
                    if $d < b$ then $b \leftarrow d$, $M' \leftarrow M$
    $score \leftarrow 1 - b / [\frac{1}{2}\sqrt{(size^2 + size^2)}]$
    return $\langle M', score \rangle$
ANGLE-BETWEEN-VECTORS(A, B)
    return ACOS($A_x \times B_x + A_y \times B_y$)
```