SEGMENTING HAND-DRAWN STROKES

A Thesis

by

AARON DAVID WOLIN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2010

Major Subject: Computer Science

SEGMENTING HAND-DRAWN STROKES

A Thesis

by

AARON DAVID WOLIN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Tracy Hammond
Committee Members,	Andruid Kerne
	Thomas Ioeger
	Ann McNamara
Head of Department,	Valerie Taylor

May 2010

Major Subject: Computer Science

ABSTRACT

Segmenting Hand-Drawn Strokes. (May 2010)

Aaron David Wolin, B.S., Harvey Mudd College

Chair of Advisory Committee: Dr. Tracy Hammond

Pen-based interfaces utilize sketch recognition so users can create and interact with complex, graphical systems via drawn input. In order for people to freely draw within these systems, users' drawing styles should not be constrained. The low-level techniques involved with sketch recognition must then be perfected, because poor low-level accuracy can impair a user's interaction experience.

Corner finding, also known as stroke segmentation, is one of the first steps to free-form sketch recognition. Corner finding breaks a drawn stroke into a set of primitive symbols such as lines, arcs, and circles, so that the original stroke data can be transformed into a more machine-friendly format. By working with sketched primitives, drawn objects can then be described in a visual language, noting what primitive shapes have been drawn and the shapes' geometric relationships to each other.

We present three new corner finding techniques that improve segmentation accuracy. Our first technique, MergeCF, is a multi-primitive segmenter that splits drawn strokes into primitive lines and arcs. MergeCF eliminates extraneous primitives by merging them with their neighboring segments. Our second technique, ShortStraw, works with polyline-only data. Polyline segments are important since many domains use simple polyline symbols formed with squares, triangles, and arrows. Our ShortStraw algorithm is simple to implement, yet more powerful than previous polyline work in the corner finding literature. Lastly, we demonstrate how a combination technique can be used to pull the best corner finding results from multiple segmen-

tation algorithms. This combination segmenter utilizes the best corners found from other segmentation techniques, eliminating many false negatives (missed primitive segmentations) from the final, low-level results.

We will present the implementation and results from our new segmentation techniques, showing how they perform better than related work in the corner finding field. We will also discuss limitations of each technique, how we have sought to overcome those limitations, and where we believe the sketch recognition subfield of corner finding is headed.

To me

ACKNOWLEDGMENTS

Thanks to all of the members of the Sketch Recognition Lab for their helpful feedback, especially Brandon Paulson, Joshua Johnston, and Paul Corey for early thesis reviews. I'd like to thank my advisor, Dr. Tracy Hammond, for providing valuable input during my tenure in the lab, as well as for her assistance with formulating my thesis. Similarly, thank you to all of my committee members for their advice and guidance during this process.

Finally, I'd like to thank my family for all of their support throughout these years; I couldn't have done this without you.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION . . . 1
- A. Pen-based Interfaces . . . 1
- B. Corner Finding . . . 3
- C. Complexity . . . 4
- D. Contributions . . . 9

II BACKGROUND . . . 10
- A. Sketch Recognition . . . 10
  - 1. Gestures . . . 10
  - 2. Templates . . . 13
  - 3. Geometric Recognizers . . . 16
  - 4. Cognitive Basis . . . 19
- B. Corner Finding . . . 20
  - 1. Polyline Corner Finders . . . 20
  - 2. Multi-primitive Corner Finders . . . 22
  - 3. Direct Corner Finding Applications . . . 30
  - 4. Unsolved Problems . . . 32
- C. Primitive Recognizers . . . 33

III MERGECF . . . 36
- A. Motivation . . . 36
- B. Implementation . . . 37
  - 1. Curvature and Speed Values . . . 37
  - 2. Initial Fit . . . 38
  - 3. Merging Segments . . . 41
  - 4. Algorithm . . . 42
  - 5. Intuition . . . 45
- C. Results . . . 46
  - 1. Accuracy Metrics . . . 50
- D. Discussion . . . 53
  - 1. Algorithm Speed . . . 54
- E. Limitations and Future Work . . . 55
  - 1. Arc Issues . . . 55

CHAPTER Page

2. Polyline Data . . . 58
3. Implementation . . . 59
4. Test Set . . . 59
5. Contributions . . . 62
6. Directions for Future Work . . . 62

IV SHORTSTRAW . . . 63

A. Motivation . . . 63
B. Resampling . . . 64
C. Corner Finding . . . 67
1. Bottom-Up . . . 67
2. Top-Down . . . 68
D. Results . . . 70
E. Discussion . . . 71
1. Simplicity . . . 74
2. Complexity and Time . . . 75
3. Potential Optimizations . . . 76
4. Offline Possibilities . . . 76
5. Relation to Curvature . . . 77
F. Extensions . . . 78
G. Contributions . . . 78
H. Limitations and Future Work . . . 79

V COMBINING CORNER FINDERS . . . 82

A. Motivation . . . 82
B. Implementation . . . 84
1. Step 1: Segmenters Used . . . 85
2. Step 2: Subset Selection . . . 85
3. Step 3: Training and Testing . . . 91
4. Algorithm Summary . . . 95
a. Single-stroke Segmentation . . . 95
b. Training Algorithm . . . 95
C. Results . . . 96
D. Discussion . . . 98
1. Thresholding . . . 99
2. Complexity and Time . . . 100
3. Significance . . . 100

CHAPTER Page

E. Gaining Intuition: Why Do We Need to Run Existing Segmentation Algorithms? 101
F. Future Work 105
G. Contributions 106

VI CONCLUSION 107

REFERENCES 109

APPENDIX A: SHORTSTRAW PSEUDOCODE 118

APPENDIX B: ISTRAW 126

A. Modifications 126
1. Corners From Speed 126
2. Consecutive Collinear Tests 126
3. Hook Removal 127
4. Addition of Curves 128
B. Presented Results 129
C. Discussion 132

APPENDIX C: APPLICATIONS 135

D. Geometric-based Recognizers 135
E. Corners as Features in Arrow Recognition 137

VITA 141

LIST OF TABLES

TABLE Page

I Results for MergeCF and three baseline corner finders. The results are for a set of 501 shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs. The metrics are discussed in Section C.1. 48

II Results comparing MergeCF with lines and arcs and MergeCF with lines, arcs, and curves. 57

III Results for MergeCF and our baseline algorithms on polyline-only data. There are 244 polyline-only strokes in our test set of 501 strokes. 57

IV Results for ShortStraw and our comparison corner finders. The results are for a set of 244 polyline shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs. 72

V Results for our corner subset selection algorithm (CSS) and the six original finders we used. The results are for a set of 244 polyline shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs. 97

VI Results for our corner subset selection algorithm (CSS) without using MergeCF as an original corner finder. We find these results to have comparable performance to CSS with MergeCF, with the added benefit of less segmentation time per stroke. 102

VII Results for modifications of the corner subset selection algorithm. Here we have our original version (CSS), our subset selection technique applied to all points, and our subset selection technique applied to an oversegmented set. 103

TABLE Page

VIII Results comparing ShortStraw to IStraw on the 244 original polyline test strokes, as presented by Xiong and LaViola Jr. [59]. We added a comparison to the ShortStraw algorithm we present in this thesis. IStraw-C is IStraw with curve detection deactivated. The 244 strokes were not tested with IStraw's curve detection turned on. The number of correct corners has been changed to 1841 from 1842 in the original paper; 1842 was a typo in ShortStraw [58]. 131

IX Results comparing ShortStraw (from SBIM 2008 [58]) to IStraw with curve detection. These values are for the 656 polyline strokes in Xiong and LaViola Jr.'s dataset [59]. 131

X Results comparing ShortStraw (from SBIM 2008 [58]) to IStraw with curve detection. These values are for the 1246 strokes in Xiong and LaViola Jr.'s dataset [59]. The dataset contains both polyline-only data (Fig. 57) and the line and curve data (Fig. 58). . . 132

LIST OF FIGURES

FIGURE Page

1 Examples of sketches in visual domains, such as circuit diagrams 1(a) and mathematics equations 1(b). These domains can be better expressed through sketching than keyboard input. 2

2 Examples of pen-enabled devices. 3

3 The geometric recognizer process from drawn strokes 3(a) to the recognized squares 3(d). Two different squares are first drawn by users, the left square with only one stroke and the right square with two strokes 3(a). The corners of these strokes are then found 3(b), and the strokes are broken down into primitive lines. These lines are then evaluated using geometric constraints, such as if the lines are of equal length or right angles to each other 3(c). When both sketches from 3(a) are examined in this fashion, they appear to be the same shape 3(d). 5

4 Each of these strokes has an additional segment due to noise. Each false positive corner is at an inflection that generates a better segmentation fit to the polygon with less error, but the corners were unintended by the users. Removing these corners from the final segmentation is a difficult task. 7

5 An example of a hook near the endpoint of a stroke. The hook is the result of either pen-up or pen-down noise, which causes a small jitter and rapid change of direction near the start or end of a drawn stroke. 7

6 The same shape is drawn by two different users. The drawing styles and noise levels are not similar, and the last line segment in 6(b) could be difficult to classify as being necessary since it is so small when compared to the other segments. 8

7 This stroke has an ambiguous segmentation. If the user intended to draw a square, then the segmentation in 7(a) is correct. If they tried to draw a pentagon, then the segmentation in 7(b) is correct. . 8

FIGURE Page

8 Palm's Graffiti gesture system allowed users to write on a PDA with a stylus and achieve high recognition results. The gesture language mapped well to Roman characters, and the single-stroke recognition allowed Palm devices to quickly and accurately classify each stroke [14]. 11

9 Pen gestures are used for controlling a virtual reality environment [18]. 12

10 Above is an illustrative example of the Hausdorff distance. Suppose we had a drawn stroke in red, dashed (A) and the template above in blue, solid (B). The directed Hausdorff distance $h(A, B)$ would find the minimum values that a point in A is away from a point in B; the maximum (showed here as a darker arrow) would be the directed Hausdorff distance (Fig. 10(a)). The $h(B, A)$ value would be found in a similar fashion (Fig. 10(b)). Finally, the maximum directed Hausdorff distance is found from the $h(A, B)$ and $h(B, A)$ values (Fig. 10(c)). 14

11 Users could define and search for map markers with an image in Wolin *et al.*'s JavaScript application [22]. In this example, a user could search for the nearest McDonalds using the company's dual arch symbol. 15

12 Two examples of a geometric object and its corresponding shape description [3]. On the left, an arrow is described as a shaft, head, and constraints. The shaft and the head are formed with three primitive lines. On the right, a family tree with a mother and son connection uses the defined arrow description. In this family tree example, more complex shapes like squares have already been formed and recognized from simpler primitives. 17

13 A course of action infantry symbol. The symbol is composed of a rectangle and two lines, where the lines form a cross ('X') in the middle of the rectangle. 17

14 Part of the shape description for the military course of action symbol, Infantry. The shape description specifies the primitives that the shape contains (1 rectangle, 2 lines), as well as the constraint interactions between the primitives. 18

FIGURE Page

15 A walkthrough of Douglas-Peucker's algorithm. 21

16 A walkthrough of PaleoSketch's polyline segmentation algorithm. . . 23

17 A drawn square, with its 4 corners marked with red points. The direction and curvature graphs for this stroke are shown in Fig. 18(a) and 18(b), respectively. 24

18 Direction and curvature and graphs for the stroke in Fig. 17. 25

19 These stroke examples, A, B, and C, show how path length can influence curvature values. Each of these examples are part of some stroke, where the 3 points marked are sequential points used to calculate the curvature of the indicated middle point. In these cases, the window of points we use to calculate curvature is $W = 1$, for simplicity. Without path length, all of the curvature values for the middle point would be equal. But, if we examine these strokes, we see that the spacing of the points matters for curvature. Suppose we sample points at a constant rate. In A, the spacing of the first and second points is so far apart that the third point could be considered an artifact due to the user's hand motions. In B, the first and second point spacing has shrunk, giving more weight to the curvature. Finally, in C, the spacing of the points is relatively even, which indicates that the user's stroke was more steady at this section. In essence, we use the path length to try and capture the user's *intention*. 26

20 Yu and Cai's error metric: feature area [40]. The feature areas is the area of the drawn stroke to some optimal primitive. The feature areas of lines, points, and arcs are shown here as dashed sections. 27

21 The orthogonal distance squared error (ODSQ) is the summed, squared error between every point on a given stroke segment with the optimal representation, such as a perfect line or arc, fit to that segment. The ODSQ is often normalized by either the stroke segment length (Eqn. 2.6), or by the number of points in the stroke segment (Mean-squared Error). 28

FIGURE Page

22 A walkthrough of Sezgin's segmentation algorithm. The walkthrough is for a polyline example so that the error metrics are easily seen for each segmentation. The error values are calculated using the normalized orthogonal distance squared error (Eqn. 2.6). . . 29

23 Sezgin *et al.*'s algorithm first finds a polyline fit for a stroke, and then it fits Bezier curves to the remaining segments. The images here are taken from the original paper [41]. 29

24 Examples of local convexity and local monotonicity, as presented by Kim and Kim [45]. 31

25 A typing gesture in Shark[2] [49]. In this example, the user draws the word "system". 32

26 Pen gestures can be mapped to musical notes based on the number of direction changes [51]. 32

27 Drawn strokes are beautified using Yu and Cai's segmenter and primitive recognizer [40]. 34

28 The same sketched square from Fig. 17, reprinted here. The curvature and speed graphs for this stroke are shown in Fig. 29(a) and 29(b). 39

29 Curvature and speed graphs for the stroke in Fig. 28. 40

30 Initial set of corners found for a stroke, which would split the stroke into 9 primitive lines and arcs. False positives are circled. . . . 42

31 A visual walkthrough of the MergeCF segmentation algorithm. . . . 44

32 Initial set of corners found for a stroke consisting of an arc and a line. Segment 2 is the smallest, unneeded segment and should be merged with Segment 1. 45

FIGURE Page

33 Set of 23 symbols we used for testing. 6 users drew each of these symbols up to 4 times each. 12 of the symbols contained both lines and arcs (a), and 11 of the symbols contained only lines (b). Due to some users quitting the study early and other data collection issues, the total number of symbols collected was 501. Red dots indicate the corners. 47

34 Examples of correctly classified symbols by MergeCF. These symbols come from the set of 501 complex and polyline shapes drawn by six users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper. 49

35 A stroke with examples of true positives, true negatives, false positives, and false negatives highlighted. 50

36 Issues with segmenting arcs. 56

37 Sketched symbols from different, real-world domains. The domains consist mainly consist of shapes formed from lines, polylines, and ellipses. 60

38 Examples of military course of action (COA) symbols. The symbols can be described using primitives and simple shapes such as lines, ellipses, triangles, rectangles, diamonds, and dots. 61

39 The original points in 39(a) are varied in distance away from each other, whereas the resampled points in 39(b) are interspaced evently. 64

40 An example demonstrating how the interspacing distance for the resampled points is calculated. Note that we fit 80 points to the diagonal in our implementation, but, for image clarity, we only fit 36 points to the diagonal in 40(d). 66

41 An example of "straws" in a stroke. The points (a-e) all have a window of ± 3 points. the distance at endpoints at these windows forms a straw, with the shortest straws being at points (a), (c), and (e). These points are considered corners. Points (b) and (d) have straws that are close to the median straw length, so these points are not initial corner candidates. 68

FIGURE Page

42 The 11 polyline symbols used during corner finder testing. These symbols were drawn up to 4 times each by 6 different users, resulting in 244 polyline strokes. 70

43 Examples of correctly classified symbols by ShortStraw. These symbols come from the set of 244 polyline shapes drawn by six test users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper. 73

44 The resampled points in this stroke are too far apart to accurately find the correct corners in the small horizontal segments at the bottom of this stroke. 80

45 Corner Subset Selection Process: (1) Take an input stroke, (2) segment the stroke using six different techniques, (3) combine the corners from all the techniques into one set, (4) pass the combined corner pool to our subset selection algorithm, and (5) output the best subset found. None of the original segmentations are correct, but the final subset has the correct 6 corners. 84

46 This figure shows an example of the error between the original stroke (gray stroke, black points), and a representation based on a stroke's corners (red lines). To calculate the mean-squared error, the distances (black lines) between the original points and the optimal polyline are squared, summed, and then averaged. 88

47 Mean-squared error (MSE) of the stroke in Fig. 45. As corners are removed, the MSE has little change until critical corners are removed. In this example, the correct number of corners is 6, so critical corners are removed starting at $i = 5$. The segmentations at $i = 14, 10, 6$, and 4 are shown here to illustrate how the subsets change as the number of corners in a subset decreases. 89

48 This is the same data from Fig. 47, but with a log scale for the MSE. 89

FIGURE Page

49 ΔMSE described in Eqn. 5.2. This chart is for the stroke in Fig. 45, whose MSE plot is shown in Fig. 47. ΔMSE is essentially a derivative of the mean-squared error, which deviates only slightly until a critical corner is removed at $i = 5$. The ΔMSE from $i = 6$ to $i = 5$ is calculated to be 28.8. 90

50 An example of how R_{below} and R_{above} are generated during training. . 92

51 Two Gaussian distributions are created from the R_{below} and R_{above} ratios for each set of training data. The optimal threshold is then found to be at the intersection of these two Gaussians; in this case, the threshold would be $t_{\Delta MSE} = 2.102$. Note that R_{below} is a much narrower Gaussian distribution than R_{above}'s, and the probability density for R_{below} goes to approximately 1.6. We chose a smaller y-axis in order to highlight the intersection of R_{below} and R_{above}. 93

52 A subset of the 216 random polyline shapes used for training. The polylines ranged from 2-line to 10-line shapes. The only drawing constraints were the number of line segments in each polyline and that the shape must be drawn with one stroke. Some users drew common symbols (‘M’ and square), others drew common patterns (zigzag), and a few drew random patterns of lines. 94

53 The 11 polyline symbols used during corner finder testing. These symbols were drawn up to 4 times each by 6 different users, resulting in 244 polyline strokes. 98

54 Issues with thresholding in our corner subset selection algorithm. In both of these cases, the mean-squared error of the system would rise considerably (i.e., above our found $t_{\Delta MSE}$) if any corner was removed. 99

55 An example of collinear line test issues in ShortStraw. In ShortStraw, the $A-B-F$ collinear test will eliminate a correct corner, B, before the $B-F-C$ tests remove the false positive, F. This figure was created by Xiong and LaViola Jr. [59]. 127

FIGURE Page

56 At each corner, c_i, IStraw evaluates two angles, α and β, around a window of resampled points. If c_i is a correct corner, such as in the figure on the left, $\beta - \alpha$ is close to 0. If c_i is part of a curve, then $\beta - \alpha$ is greater than 0. This figure was created by Xiong and LaViola Jr. [59]. 128

57 The 11 polyline symbols used for testing in our ShortStraw evaluation. 129

58 The 10 line and curve symbols Xiong and LaViola Jr. collected. This figure was presented in their SBIM 2009 paper [59]. 130

59 The 'R' symbol from Fig. 58 should have another corner where the left vertical line and arc meet (circled here). This corner is missing from the IStraw symbols due to Xiong and LaViola Jr.'s recognition of "curvy" data, rather than curves. 134

60 Part of the shape description for the military course of action symbol, Infantry. The shape description specifies the primitives that the shape contains (1 rectangle, 2 lines), as well as the constraint interactions between the primitives. 136

61 Two arrows can have different, arbitrary paths that indicate the attack direction of units in course of action diagrams. 137

62 The three types of arrow heads we use segmentation to help recognize. 138

63 These two arrows, Task, Fix (63(a)) and Ground Supporting Attack (63(b)), differ only in the number of segments in the arrow's path. 139

64 These two arrows, Task, Follow and Assume (64(a)) and Task, Follow and Support (64(b)), have a different number of segmentations in their tail. The number of segments, 5 and 6, respectively, is one feature that helps classify the arrows. 139

CHAPTER I

INTRODUCTION

Computers today typically enforce interaction through direct-manipulation, WIMP-based, and command-line interfaces. These interfaces require the user to explicitly control objects on a screen through keyboard input and button presses. Although these types of interactions work well in many environments, they have a poor mapping to visual mediums.

Pen-based interaction, on the other hand, is a natural way for humans to communicate with computers within visual domains. For instance, UML diagrams [1], circuit diagrams [2, 3], molecular structures [4], and even mathematical equations [5, 6] can all be represented through pen-based input. By recognizing a user's drawings, the graphical structures and symbols can be inferred with little effort from the user. The user simply has to sketch any symbols within the domain they are working in (Figure 1).

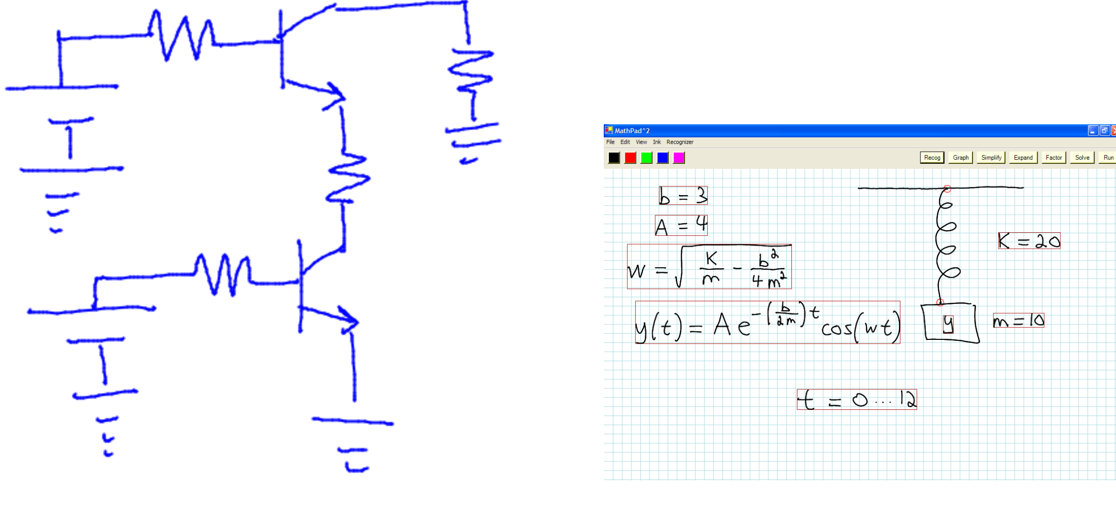
Unfortunately, as sketch recognition interfaces become more intelligent, the burden of interaction and learning is placed on the algorithm developer. Instead of users learning and adapting to a system, O'Connell *et al.* found that any errors are attributed to the sketch recognition [7]. Therefore, in order for sketch recognition to become widely accepted, the recognition itself must be close to perfect.

A. Pen-based Interfaces

People can communicate with computers through a multitude of pen-based input devices, such as Palm PDAs [8], Wacom tablets and monitors [9], and Tablet PCs

---

The journal model is *IEEE Transactions on Automatic Control.*



(a) Circuit diagrams from [3]. (b) Mathematics equations from [5].

Fig. 1. Examples of sketches in visual domains, such as circuit diagrams 1(a) and mathematics equations 1(b). These domains can be better expressed through sketching than keyboard input.

(Fig. 2). These devices collect pen-data as a series of time-stamped coordinate points in (x, y, t) tuples. Points are collected as soon as the pen is pressed down onto the digitizing or touch-sensitive screen, and the recording stops once the user lifts their pen up.

A single series of points, from pen-down to pen-up, is called a *stroke*. Strokes are the building blocks for pen-based applications. At the lowest end, they can be used to communicate information through invariable pen gestures that map to commands. At a higher level, individual strokes can be combined together to form complex *symbols* and *shapes*.

Sketch recognition is the study of how to classify gestures, symbols, and shapes, as well as the interactions between the sketched components.

(a) A Cintiq 21UX Wacom interactive display [9].

(b) A X200t tablet PC from Lenovo [10].

Fig. 2. Examples of pen-enabled devices.

B. Corner Finding

Corner finding, also known as *segmentation*, *fragmentation* and *cusp detection*, is one of the first steps to sketch recognition and is the focus of this thesis.

To fully support the claim that sketch recognition allows natural and intuitive interaction (i.e., free-form), users should be able to draw in any style they choose. Yet, there are an infinite number of ways to draw even a simple symbol, and, as system developers, we do not want to create a template for every drawing approach. Instead, we want to find the basic components of a symbol and transform the user's drawing into these components.

Corner finding is a critical step in free-form sketch recognition because the technique breaks strokes into their simplest building blocks called *primitives*. These primitives can then be recombined to form more complex shapes, much like how steel beams

can form a complex structure for a building. The most common primitives are lines, arcs, and curves, but primitives can also be more complex shapes like ellipses, spirals, or helixes [11].

The technique is called corner finding because it divides the stroke at the *corners* between primitives; similarly, it is also called segmentation because it splits a stroke into primitive *segments*.

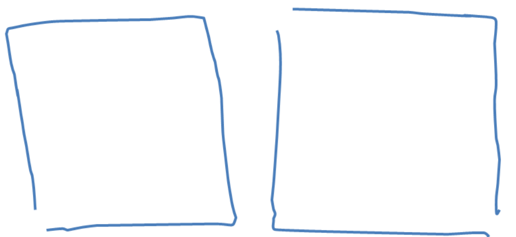
To illustrate why corner finding is useful, suppose a user draws the two symbols in Fig. 3(a). Both symbols are squares, yet they are drawn using a different number of strokes. Finding the corners of each stroke allows us to describe each symbol in terms of four primitive lines (Fig. 3(b)). More complex shape descriptions are then avoided, such as trying to account for a square drawn with two or three strokes.

Sketch recognizers can then apply sets of rules to the four lines in order to determine a shape. The recognizer can find that the four lines are perpendicular and of equal length (Fig. 3(c)), and the recognizer then classifies each symbol as a square based on the primitives within the symbol and the geometric constraints satisfied. More information on how these recognizers work is discussed in Background, Chapter II, Section A.3.

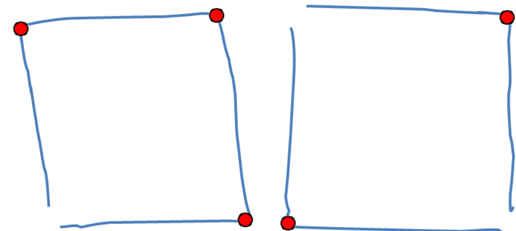
The main point to take away from this example is that corner finding is necessary to have large-scale, free-sketch recognition systems. As the number of strokes in a symbol grows, such as in the Figure 1 sketches, corner finding becomes a necessity since there are too many symbol variations to account for alone.

C. Complexity

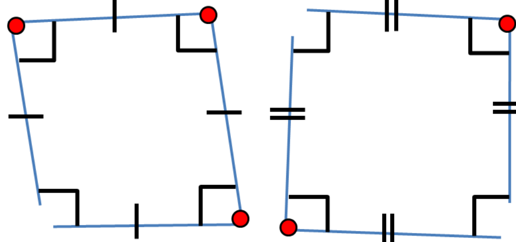
Corner finding is a difficult problem and worthy of research. People draw in many different styles and speeds, which can introduce varying amounts of noise into stroke



(a) Two squares drawn by a user; each square is drawn differently.



(b) The corners are found within all of the strokes.



(c) The primitive lines are examined under square constraints.

(d) The two shapes are both recognized as squares.

Fig. 3. The geometric recognizer process from drawn strokes 3(a) to the recognized squares 3(d). Two different squares are first drawn by users, the left square with only one stroke and the right square with two strokes 3(a). The corners of these strokes are then found 3(b), and the strokes are broken down into primitive lines. These lines are then evaluated using geometric constraints, such as if the lines are of equal length or right angles to each other 3(c). When both sketches from 3(a) are examined in this fashion, they appear to be the same shape 3(d).

data. In order to create a corner finder that can work for a variety of users, we need to:

- Reliably sample point data for strokes, ensuring that the user's intended stroke inflections are captured.
- Filter noise from strokes, removing any unwanted pen fluctuations while not eliminating any important information.
- Find the perceived corners, picking only the points that are correct corners while avoiding noise-induced segmentations.
- Ensure that corner finders run in real-time.

Each of these items is problematic when creating segmenters. Stroke-capturing devices have different resolutions and sampling rates, so a corner finder designed using data from one digitizing pad could be overtrained to that particular sampling rate and have thresholds that do not work well for slower or faster sampled strokes. Likewise, a higher sampling resolution could provide more reliable pen data, but it can also introduce more noise from unintended pen movements. Any small jiggle or shake of a pen can produce sharp inflections that appear to be corners (Fig. 4). A common problem with stroke sampling is the introduction of "hooks" at the beginning and end of stroke capturing, where the placing-down or lifting-up of a stylus will cause sharp inflections to be captured (Fig. 5).

We also cannot simply assume that some segments, such as small sections near the endpoints, are unintended. For instance, in the shapes in Fig. 6, the small segment near the endpoint was deliberately drawn. In many of these cases, a stroke could be segmented differently even by human recognizers (Fig. 7). Distinguishing between these intended strokes and unwanted noise is a very difficult problem.

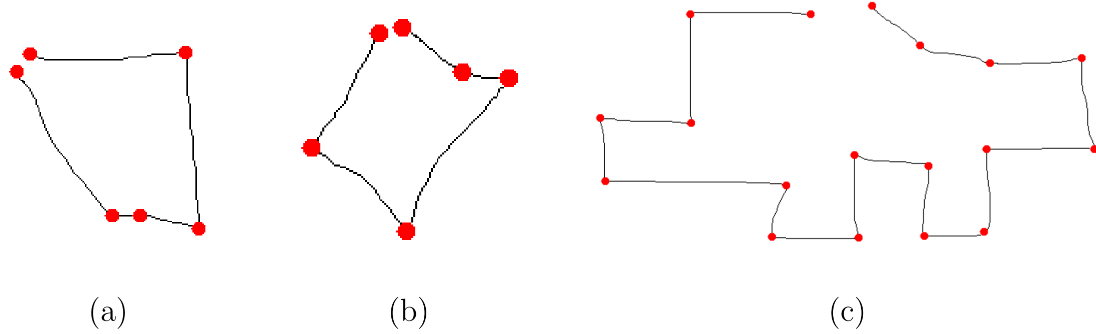
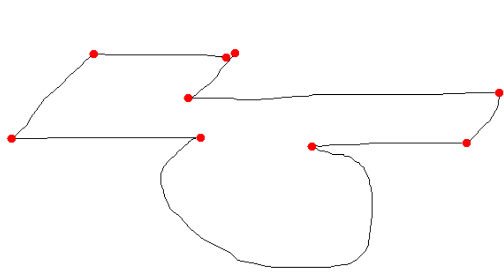


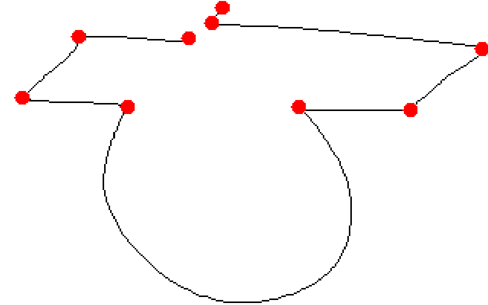
Fig. 4. Each of these strokes has an additional segment due to noise. Each false positive corner is at an inflection that generates a better segmentation fit to the polygon with less error, but the corners were unintended by the users. Removing these corners from the final segmentation is a difficult task.

Fig. 5. An example of a hook near the endpoint of a stroke. The hook is the result of either pen-up or pen-down noise, which causes a small jitter and rapid change of direction near the start or end of a drawn stroke.

Lastly, segmentation algorithms must run in real-time since they are only one component (and often the first step) in sketch recognition systems. The algorithms must be able to efficiently segment strokes composed of hundreds of points, which eliminates the usage of highly-accurate, but slow, dynamic programming approaches.

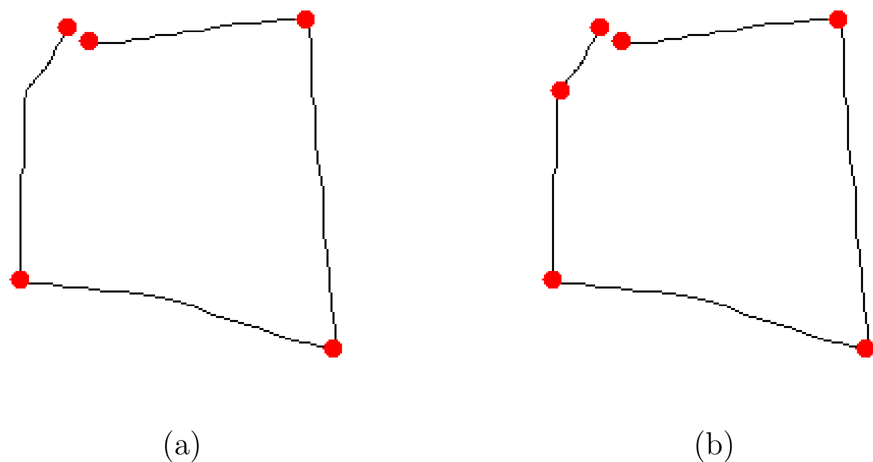


(a) A shape with 9 segmentations: 3 line segments, 1 arc segment, and 4 more line segments.



(b) The same shape as in 6(a), but with a more ambiguous segment near the endpoint. In this case, the last, small line segment could easily have been classified as an unintended hook.

Fig. 6. The same shape is drawn by two different users. The drawing styles and noise levels are not similar, and the last line segment in 6(b) could be difficult to classify as being necessary since it is so small when compared to the other segments.



(a) (b)

Fig. 7. This stroke has an ambiguous segmentation. If the user intended to draw a square, then the segmentation in 7(a) is correct. If they tried to draw a pentagon, then the segmentation in 7(b) is correct.

D. Contributions

This thesis benefits the sketch recognition community by addressing many corner finding problems:

- We will demonstrate that smaller segments are more likely to be a result of noisy data, and we eliminate them by merging smaller segments with their larger neighbors. (Chapter III)
- We introduce the notion that segmentating strokes into polylines (as opposed to multiple primitives such as lines and arcs) is sufficient for sketch recognition. (Chapter III)
- We implement a fast and efficient corner finder, ShortStraw, that is more accurate that other polyline corner finders. (Chapter IV)
- We provide simple, elegant, and easily understandable pseudocode for our ShortStraw corner finder, allowing sketch application developers to quickly incorporate segmentation techniques. (Appendix A)
- We address the problem that different corner finder techniques often find different segmentations. By using an ensemble learning approach to merge results from multiple segmentation algorithms, we are able to choose a better overall segmentation for a stroke. (Chapter V)
- We analyze both the benefits and drawbacks of each segmentation technique we introduce, and we mention how the work can be extended in future segmentation research.

CHAPTER II

BACKGROUND

A. Sketch Recognition

There are many techniques and algorithms used for recognizing sketches. The techniques range from being inflexible to allowing unconstrained drawing, and they accommodate single-stroke to multi-stroke input.

1. Gestures

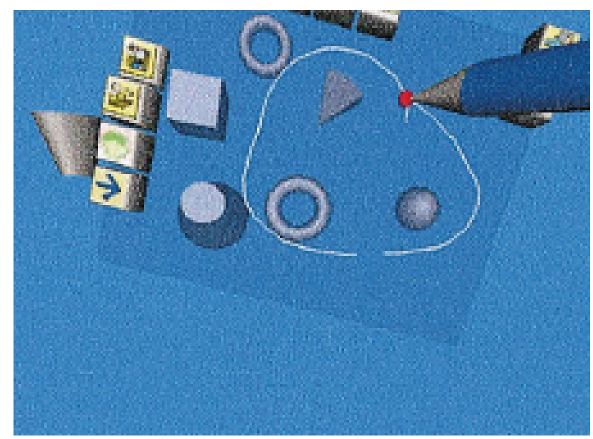
The simplest sketch recognition solutions work with constrained, single-stroke gestures. Gestures are highly-constrained pen drawings that correspond to basic symbols or commands. Rubine introduced single-stroke pen gestures as an alternative to direct manipulation interfaces [12]. Rubine analyzed stroke data and extracted a set of features from the series of x, y, and *time* values recorded. His small, but robust, feature set included 13 features such as the total gesture length, the total curvature of the gesture, and the maximum drawing speed of the gesture. Rubine's technique then trains a linear classifier on a set of gestures to classify, creating a feature key for each gesture. When a user creates a new stroke, the 13 features of the stroke are first extracted, and the new stroke's feature vector is compared against all of the features keys. The new input stroke is then classified according to the gesture the stroke's feature vector is closest to in the 13-dimension feature space. Newer research by Long *et al.* has improved the feature sets used during gesture classification [13].

Gesture recognition is important to the sketch recognition community because it allows strokes to be easily classified using a simple set of features. The ease of implementation of the classifier also allows many application developers to utilize sketched

Fig. 8. Palm's Graffiti gesture system allowed users to write on a PDA with a stylus and achieve high recognition results. The gesture language mapped well to Roman characters, and the single-stroke recognition allowed Palm devices to quickly and accurately classify each stroke [14].

gestures as an alternative to pure direct-manipulation interfaces. Palm Computing's Graffiti handwriting system is one of the more well-known examples of gesture recognition. Graffiti classifies a single-stroke gesture alphabet that has a decent mapping to its Roman character counterpart [14, 15] (Fig. 8).

Besides using gestures as symbols and alphabets, gestures allow users to perform commands with a stylus, sans keyboard input. Landay and Myers utilized editing gestures in a prototyping application called SILK [16]. SILK users can sketch an interface layout with a stylus, and gestures for copying, deleting, and grouping sketched components are available. A follow-up sketch design application, DENIM, uses similar editing gestures [17]. Bimber *et al.* use 2D pen gestures to control a virtual reality, 3D environment [18]. Their set of gestures controls object creation, editing, selection, and context menus (Fig. 9).



(a) A 3D virtual reality environment with 2D pen-based interaction. Here, the user controls the environment with a pen selection of objects.

Object control	Gesture					
	Resulting operation	delete	undo	select	deselect	
Mode change	Gesture					
	Resulting mode	Context-sensitive menu	Coloring mode	Window tools	Fish net selection	Object creation

(b) The set of 2D gestures available to the user. Some of the gestures are for content editing, while others are for environment control.

Fig. 9. Pen gestures are used for controlling a virtual reality environment [18].

2. Templates

Image template matching is also an important sketch recognition technique and is more impervious to symbol drawing style than feature-based algorithms, such as Rubine's approach [12]. In template matching, a given set of strokes is converted into a pixelated image, and this image is then compared to a set of template images. The drawn image is classified as the template that it is a closest match to, given some evaluation algorithm and metrics. The actual number of drawn strokes is typically unconstrained, but templates themselves must be very unique, since subtle changes are difficult to detect in templates.

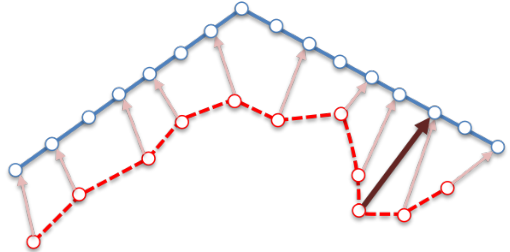
The Hausdorff distance has been widely used to evaluate image-template matches. Eqn. 2.1 defines the directed Hausdorff distance between the sets of points A and B as the maximum of the minimum neighbor distances between a point $a \in A$ and all points in B. In other words, $h(A, B)$ finds the maximum distance bound for a point $a \in A$ to be away from a point $b \in B$. The point distance equation can be any distance function, such as Euclidean distance.

Eqn. 2.2 is referred to as the Hausdorff distance and calculates the maximum of the directed Hausdorff distances between the point sets A and B. Fig. 10 provides a graphical walkthrough of the algorithm.

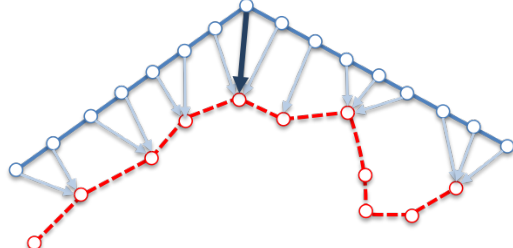
$$h(A, B) = \max_{a \in A} \min_{b \in B} ||a - b|| \tag{2.1}$$

$$H(A, B) = \max(h(A, B), h(B, A)) \tag{2.2}$$

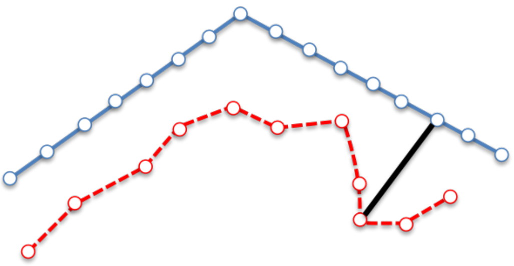
Huttenlocher *et al.* use the Hausdorff distance model to compare complex 2D images, including the ability to match partial images [19]. Dubuisson and Jain propose the use of a modified Hausdorff distance to match templated images while avoiding



(a) Directed Hausdorff distance, $h(A, B)$. A is the dashed, red path and B is the solid, blue path. The maximum of the minimum pair distances is shown as the darker arrow.



(b) Directed Hausdorff distance, $h(B, A)$. The maximum of the minimum pair distances is shown as the darker arrow.



(c) The final Hausdorff distance, taken to be the maximum value from $h(A, B)$ and $h(B, A)$.

Fig. 10. Above is an illustrative example of the Hausdorff distance. Suppose we had a drawn stroke in red, dashed (A) and the template above in blue, solid (B). The directed Hausdorff distance $h(A, B)$ would find the minimum values that a point in A is away from a point in B; the maximum (showed here as a darker arrow) would be the directed Hausdorff distance (Fig. 10(a)). The $h(B, A)$ value would be found in a similar fashion (Fig. 10(b)). Finally, the maximum directed Hausdorff distance is found from the $h(A, B)$ and $h(B, A)$ values (Fig. 10(c)).

Fig. 11. Users could define and search for map markers with an image in Wolin *et al.*'s JavaScript application [22]. In this example, a user could search for the nearest McDonalds using the company's dual arch symbol.

outlier issues presented in a regular Hausdorff distance function [20].

Kara and Stahovich use three separate image comparison techniques, including Dubuisson's Hausdorff distance metric, to match drawn strokes with defined templates [21]. More recently, the Hausdorff distance metric was applied to recognizing symbols drawn in a JavaScript application mimicking an iPhone interface [22]. The applet showed how sketched symbols can be used to mark and search for locations on Google Maps (Fig. 11).

Wobbrock *et al.*'s $1 recognizer is a cross between gesture-based recognition and template matching [23]. $1's core algorithm is template matching: it takes a drawn stroke and compares it to a set of template symbols, choosing the symbol that has the least error between the stroke and the template. $1 is rotation and scale invariant, but it also has some constraints not normally associated with template matching algorithms, chiefly that strokes must be drawn in a gesture-like manner with a given start point and drawing pattern. The main benefit of $1 is that the recognizer is easy to implement, and Wobbrock *et al.* provide pseudocode for the algorithm.

3. Geometric Recognizers

Geometric recognizers provide the most unconstrained multi-stroke recognition. In a geometric recognizer, shapes are defined as a set of primitives and constraints [3, 24, 25]. The constraints can evaluate how primitives are connected, the angle two primitives form, the distance between two primitives, if one primitive contains (encloses) another, etc. Fig. 12 shows how SketchREAD handles geometric recognition, and Fig.13 and 14 show an example of a military course of action shape to recognize and the description of the shape.

In order to have free-form geometric recognition, user-drawn strokes are broken into their primitive components via corner finding algorithms and primitive recognizers. The resulting set of primitives is then evaluated against each shape description available. The drawn shape is scored against a shape description's component and connective constraints, and the resulting shape description values are used to determine a ranking of possible shapes that the drawn primitives satisfy. For example, if a user drew 1 rectangle and 2 lines, the shape might satisfy the shape description for Fig. 13. Yet, if the user drew 1 circle and 2 lines, the shape description in Fig. 14 would not be satisfied.

Due to geometric recognizers' reliance on primitives, improving corner finding provides the greatest benefit to these types of recognition systems.

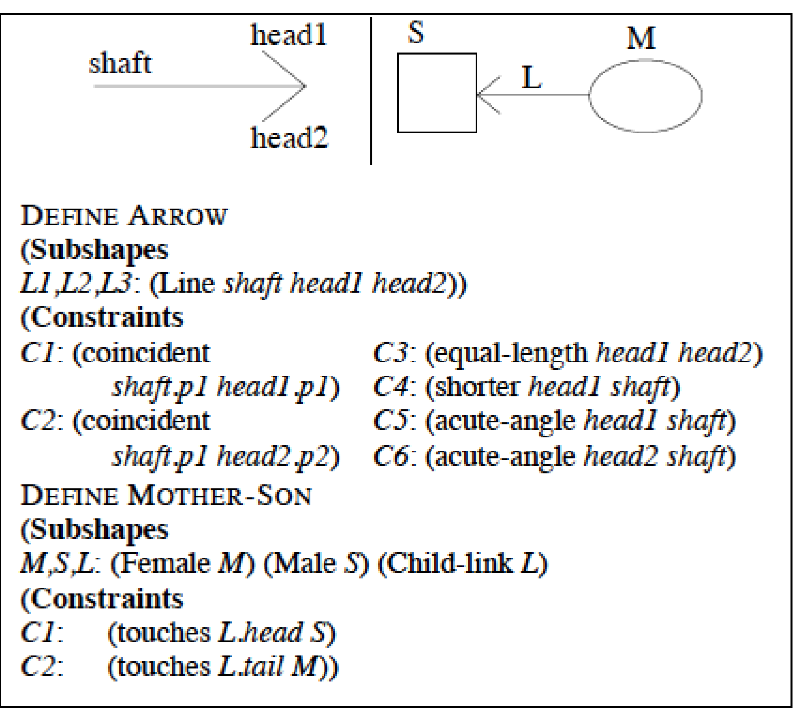


Fig. 12. Two examples of a geometric object and its corresponding shape description [3]. On the left, an arrow is described as a shaft, head, and constraints. The shaft and the head are formed with three primitive lines. On the right, a family tree with a mother and son connection uses the defined arrow description. In this family tree example, more complex shapes like squares have already been formed and recognized from simpler primitives.

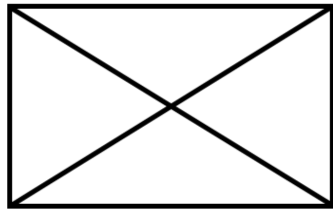


Fig. 13. A course of action infantry symbol. The symbol is composed of a rectangle and two lines, where the lines form a cross ('X') in the middle of the rectangle.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<shapeDefinition name="infantry" description="Infantry">

    <componentList>
        <!-- the frame [rectangle] -->
        <component name="rectangle" type="Rectangle" />

        <!-- the infantry -->
        <component name="posLine" type="Line" />
        <component name="negLine" type="Line" />
    </componentList>

    <constraintList>
        <!-- CONTAINS RELATIONSHIPS -->
        <constraint name="Contains">
            <param component="rectangle" />
            <param component="posLine" />
        </constraint>

        <constraint name="Contains">
            <param component="rectangle" />
            <param component="negLine" />
        </constraint>

        <!-- LINE ORIENTATIONS -->
        <constraint name="PositiveSlope">
            <param component="posLine" />
        </constraint>

        <constraint name="NegativeSlope">
            <param component="negLine" />
        </constraint>

        <!-- SIZE RELATIONSHIPS -->
        <constraint name="SameSize">
            <param component="posLine" />
            <param component="negLine" />
        </constraint>
        . . .
    </constraintList>
</shapeDefinition>
```

Fig. 14. Part of the shape description for the military course of action symbol, Infantry. The shape description specifies the primitives that the shape contains (1 rectangle, 2 lines), as well as the constraint interactions between the primitives.

4. Cognitive Basis

Many sketch recognition ideas stem from theories concerning how the brain processes images. One of the more basic vision and sketch recognition techniques is template matching, where an input stroke or sketch is compared to a known pattern for classification [2, 23]. In cognition, this process is referred to as the template matching theory. The templates and input might be rotated, translated, scaled, or reflected to achieve match equivalence [26, 27], but the overall theory does not account for complex visual recognition such as fragmented images or the fact that humans would need hundreds of thousands of templates in order to match patterns [28].

The feature analysis theory of human cognition is more applicable to general sketch recognition processing. In the feature analysis theories, visual objects have distinctive features that make the object unique. For instance, letters are composed mainly of horizontal, vertical, and angled lines with various component interactions and constraints [29]; geometric sketch recognizers work in a similar fashion [25]. By looking at the features of an object, the entire object can be understood. This idea that objects are recognized as sets of features is reinforced through neuroscience. Hubel and Wiesel showed how brain cells in the visual cortex are activated only when presented with certain visual stimulus [30]. A horizontal bar of light might trigger an electronic impulse in one cell, but the same cell would receive no impulse when a vertical bar of light was shown.

B. Corner Finding

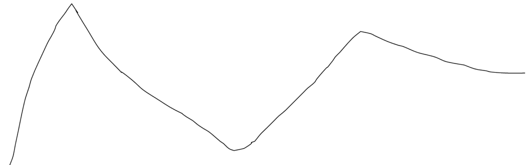
1. Polyline Corner Finders

Polyline corner finders have used a variety of techniques to estimate a polygon representation of stroke data. These techniques seek to model a stroke using the least number of line segments, maximizing each segment's length and minimizing the overall fit error. Some of these techniques include performing a linear search across the line, finding points that deviate heavily from the direction of the line [31, 32], searching for the minimum least-squares error given some metric and constraints [33, 34], and using machine learning algorithms [35].

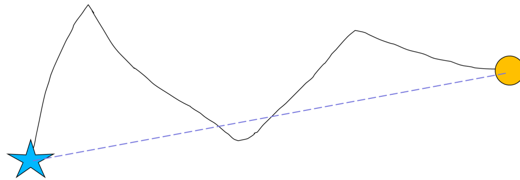
Douglas and Peucker's segmenter is a widely used polyline corner finding algorithm [36]. The algorithm was created in 1973 to simplify cartography map representations. Storing the map contour representations as a series of small lines is preferable to storing every point in the map. Their algorithm starts by creating a line, L, between the first point of the stroke (anchor) and the last point (floater). If the stroke between the anchor and the floater is a line, then the anchor is moved to the current floater. Otherwise, the algorithm finds the point along the stroke between the anchor and floater that is the maximum distance away from L. This new point is added to the list of corners and becomes the new floating point. A new line is found between the anchor and the new floater, and the process is repeated, recursively. When the anchor and floater are the same point (i.e., the last point of the stroke), then the algorithm terminates. The algorithm is shown visually in Fig. 15).

The Douglas-Peucker algorithm has spawned many other polyline corner finders seeking to improve upon its technique [37, 38, 39].

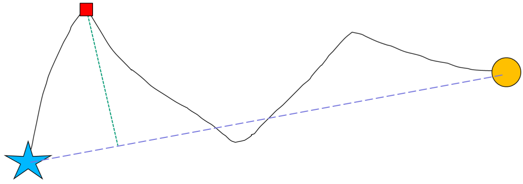
The primitive recognizer PaleoSketch uses a similar line-test algorithm to find corners in polylines [11]. PaleoSketch's polyline segmenter works by sequentially



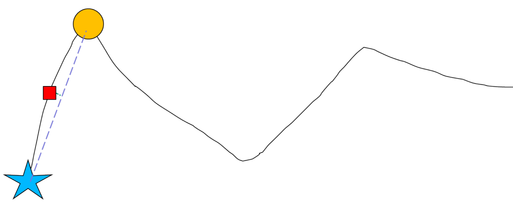
(a) User draws a polyline stroke.



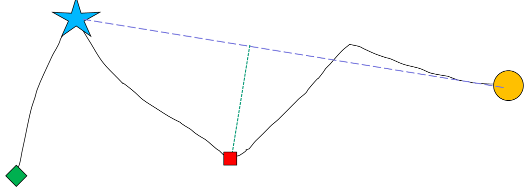
(b) The anchor (star) and floater (circle) are chosen as the endpoints of the stroke. The optimal line between the anchor and floater is shown as a dashed line.



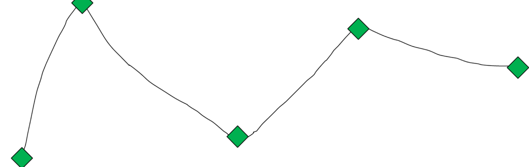
(c) The point on the original stroke that is farthest away from the optimal line is found (square). If this point is far enough away from the optimal line, it is a potential corner.



(d) The furthest point from step 15(c) now becomes the new floater, and a new point furthest from the new optimal lines is found.



(e) The potential corner from 15(d) was not far enough away from the optimal line. The anchor point is moved to the floater's position, and the floater point is reset to the end of the stroke. The previous anchor point is considered a corner (diamond).



(f) The process from 15(c) to 15(e) is repeated until the anchor point reaches the end of the stroke. At this point, all five corners for this stroke have been found.

Fig. 15. A walkthrough of Douglas-Peucker's algorithm.

running line tests (Fig. 16). The algorithm starts by initializing point indexes a and b to be 0 and 1, respectively. The stroke segment between point p_a and p_b is checked to see whether it passes a line test. If so, then $b = b + 1$, and the process is repeated. If the segment between p_a and p_b does not pass a line test, then p_{b-1} is considered a corner and $a = b - 1$ and $b = b$. This process continues until b is equal to the number of points in the stroke (i.e., the last point).

2. Multi-primitive Corner Finders

Sketch recognition often tries to find the corners of complex strokes that contain multiple primitives, not just lines. The most common technique to segment complex strokes involves finding the curvature at each point in the stroke and then choosing the points that satisfy some curvature constraints [40, 41, 42].

Taking the points of a stroke as a data series, we can find the curvature at a point by calculating the second derivative of the data. The first derivative of a stroke is the direction at each point, which is the angle of the stroke between each consecutive pair of points (Eqn. 2.3, Fig. 17 and 18(a)) [40]. The variable i is the index of a point in a stroke, such that $p_0 \leq p_i \leq p_N \in stroke$.

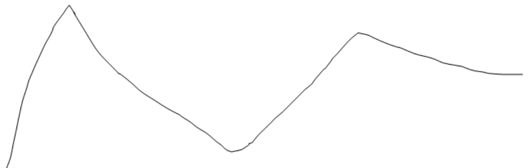
$$direction_i = \arctan\left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) \tag{2.3}$$

The second derivative is the change in direction across a window of points, also known as curvature (Eqn. 2.4, Fig. 18(b)).

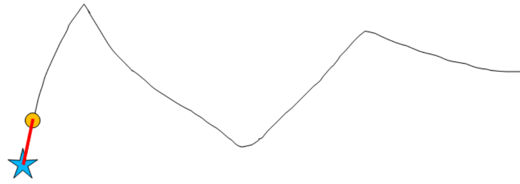
$$curvature_i = \frac{\sum_{k=i-W}^{k+W-1} |direction_{k+1} - direction_k|}{pathLength(i - W, i + W)} \tag{2.4}$$

Fig. 17 and 18 show an example shape and its direction and curvature graphs.

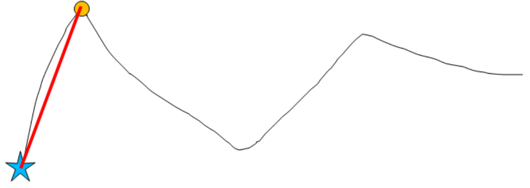
An equation to calculate the path length of a stroke (i.e., the distance of the



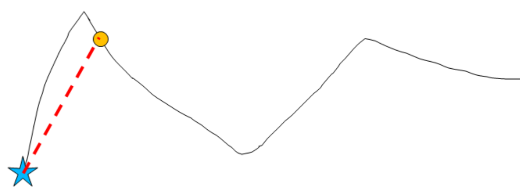
(a) User draws a polyline stroke.



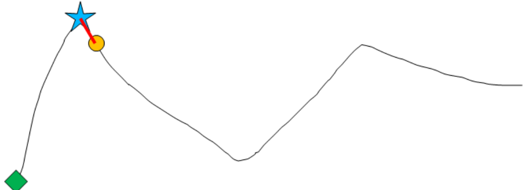
(b) The segment between the points at index a (star) and b (circle) is sent to a line test. The test passes, and b is increased by 1.



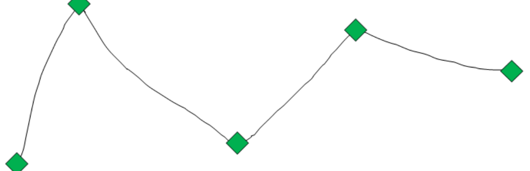
(c) After a few loops, the line test between a and b continues to pass, and b continues to move through the stroke.



(d) The line test between p_a and p_b fails



(e) Add p_{b-1} to a set of corners and set $a = b - 1$ and $b = b$. Previously found corners are labeled with diamonds.



(f) The process from 16(b) to 16(e) is repeated until b reaches the end of the stroke. At this point, all five corners for this stroke have been found.

Fig. 16. A walkthrough of PaleoSketch's polyline segmentation algorithm.

stroke between points at indices a and b) is also provided in Eqn. 2.5. The path length is used in the calculation of curvature to reduce the influence of stroke scale on curvature values; a large direction change over a small distance is more indicative of a curvature change than the same direction changes over a larger path length (Fig. 19).

$$pathLength(a,b) = \sum_{i=a}^{b-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{2.5}$$

It is important to note that curvature and speed data can also be applied to polyline corner detection [43].

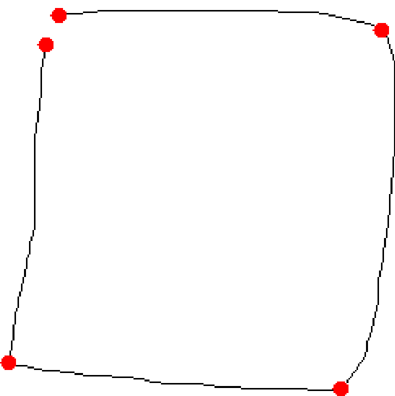
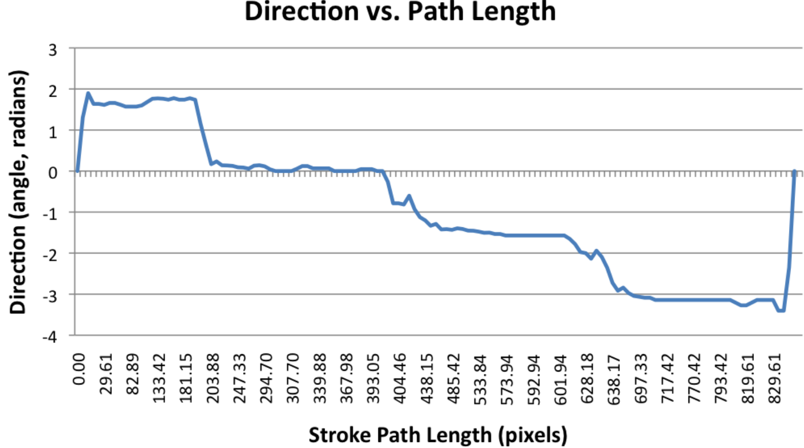
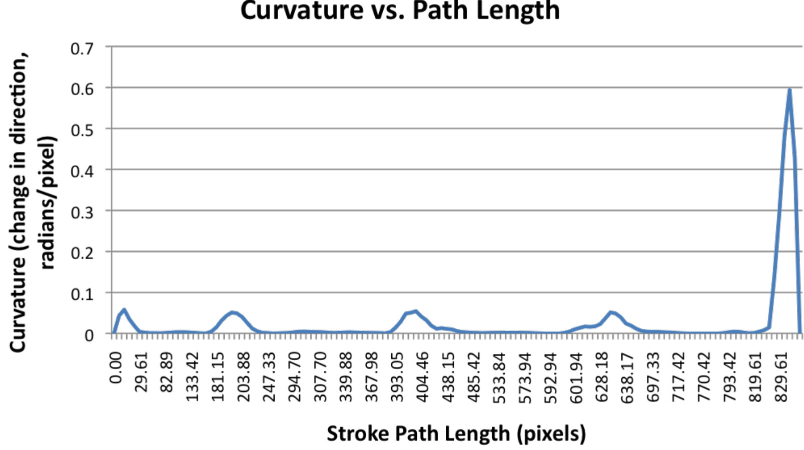


Fig. 17. A drawn square, with its 4 corners marked with red points. The direction and curvature graphs for this stroke are shown in Fig. 18(a) and 18(b), respectively.



(a) Direction graph. The direction of the stroke is relatively constant 4 times, which is when the 4 lines of the square are being drawn. The direction changes rapidly at the corners. The x-axis is the path length of the stroke, in pixels.



(b) Curvature graph. The curvature peaks five times during the stroke's drawing, once at each endpoint of the stroke, and once for each internal corner. The peaks at the endpoints are due to noise from when the user starts and stops drawing; they are referred to as "hooks" and will be discussed later.

Fig. 18. Direction and curvature and graphs for the stroke in Fig. 17.

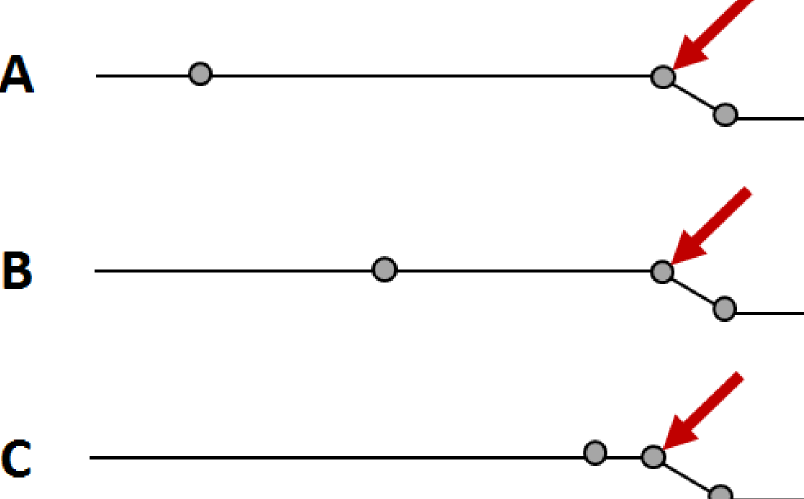


Fig. 19. These stroke examples, A, B, and C, show how path length can influence curvature values. Each of these examples are part of some stroke, where the 3 points marked are sequential points used to calculate the curvature of the indicated middle point. In these cases, the window of points we use to calculate curvature is $W = 1$, for simplicity. Without path length, all of the curvature values for the middle point would be equal. But, if we examine these strokes, we see that the spacing of the points matters for curvature. Suppose we sample points at a constant rate. In A, the spacing of the first and second points is so far apart that the third point could be considered an artifact due to the user's hand motions. In B, the first and second point spacing has shrunk, giving more weight to the curvature. Finally, in C, the spacing of the points is relatively even, which indicates that the user's stroke was more steady at this section. In essence, we use the path length to try and capture the user's *intention*.

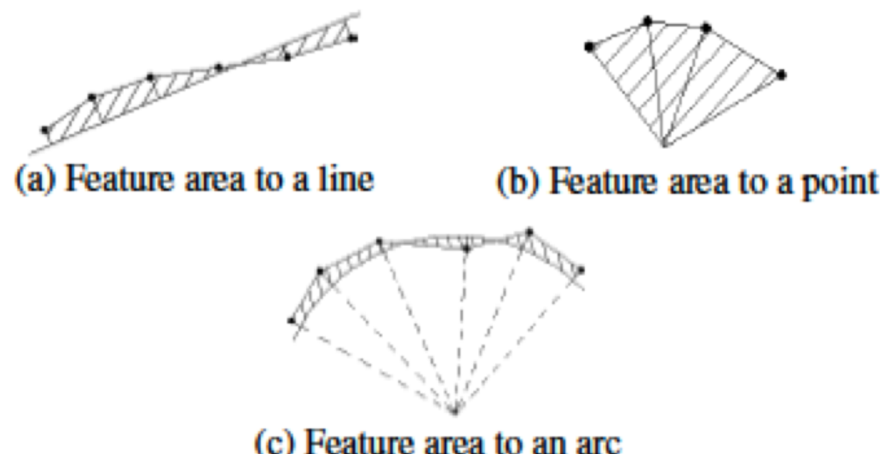


Fig. 20. Yu and Cai's error metric: feature area [40]. The feature areas is the area of the drawn stroke to some optimal primitive. The feature areas of lines, points, and arcs are shown here as dashed sections.

Yu and Cai created a corner finder that uses direction and curvature information to find the corners of a stroke [40]. The authors first try to fit a single primitive (line, arc, or circle) to the entire stroke. An arc is defined as a portion of a circle. If the primitive fit has too large an error, then the stroke is split at the point of highest curvature. The two resulting segments are then fit to primitives again, and the process is repeated until the entire error of the system is below a predefined threshold. Their system also introduces the idea of feature area, or the area of a drawn stroke segment in relation to a beautified version of the same segment (Fig. 20).

Sezgin *et al.* use the notion of pen speed to help determine stroke corners [41]. In their system, points of high curvature and low pen speed are considered corner candidates. The idea of using a user's drawing speed for segmentation has been around since the 1970s [44], where corners are found when the user slows down the pen. After Sezgin *et al.* obtain an initial collection of curvature and speed corners,

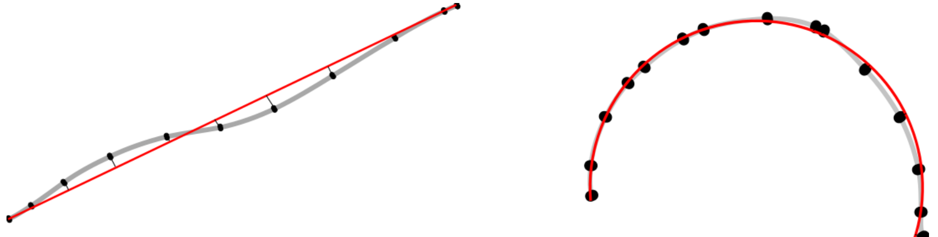


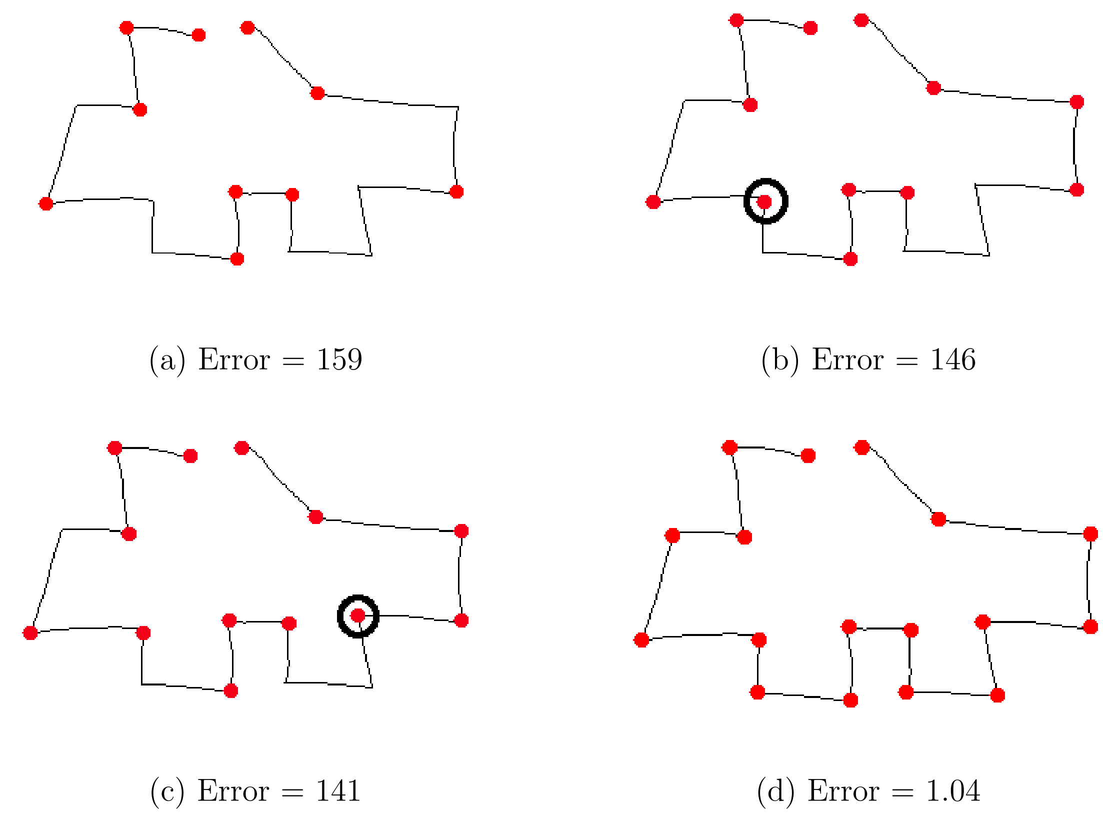
Fig. 21. The orthogonal distance squared error (ODSQ) is the summed, squared error between every point on a given stroke segment with the optimal representation, such as a perfect line or arc, fit to that segment. The ODSQ is often normalized by either the stroke segment length (Eqn. 2.6), or by the number of points in the stroke segment (Mean-squared Error).

their system greedily picks either the best curvature or speed corner, one at a time, and creates a new corner fit for the stroke using the picked corner and the previous corner fit. The best corner is determined by whichever corner lowers the error of the system the the most, where error is determined as the orthogonal distance squared, normalized by the stroke segment length (Fig. 21, Eqn. 2.6):

$$Error = \frac{1}{|S|}\sum_{i=0}^{N}(p_i - opt_i)^2 \tag{2.6}$$

This process of adding the best curvature or speed corner candidate to create a new fit is continued, and then a final polyline corner fit is chosen as the fit with the least amount of corners and an error below some developer-defined threshold (Fig. 22). After the polyline fit is generated, Sezgin *et al.* try to fit Bezier curves to segments that are not recognized as being lines (Fig. 23).

Kim and Kim propose new curvature metrics in their corner finding system [45]. These metrics, local convexity and local monotonicity, measure the curvature in the same direction at a point. Convexity is computed by summing all of the curvatures



(a) Error = 159

(b) Error = 146

(c) Error = 141

(d) Error = 1.04

Fig. 22. A walkthrough of Sezgin's segmentation algorithm. The walkthrough is for a polyline example so that the error metrics are easily seen for each segmentation. The error values are calculated using the normalized orthogonal distance squared error (Eqn. 2.6).

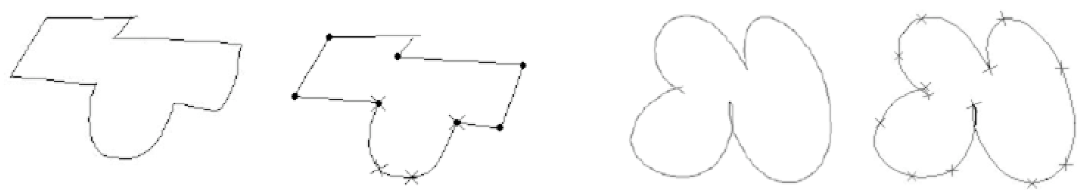


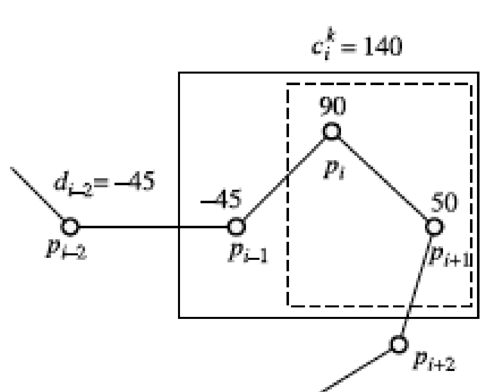
Fig. 23. Sezgin *et al.*'s algorithm first finds a polyline fit for a stroke, and then it fits Bezier curves to the remaining segments. The images here are taken from the original paper [41].

of the same sign within a window (Fig. 24(a)) , whereas local monotonicity looks at decreasing curvatures of the same sign around a point (Fig. 24(b)). Kim and Kim also have a different measure for the curvature at a point. Their system first resamples the points of a stroke to be equidistant from one another. Since the distance between consecutive points is now constant, a point's curvature value does not have to take into account path length changes, so the curvature at each point is equal to the direction change at that point.

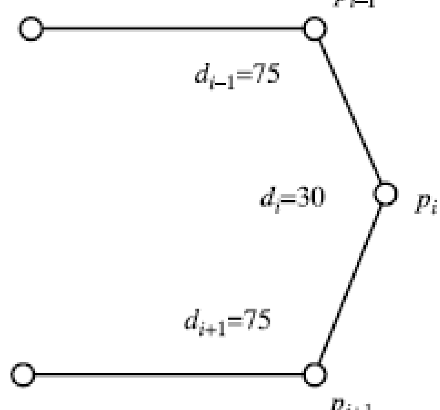
The corner finders previously mentioned all require developer set thresholds for different properties, such as curvature and speed thresholds or the interspacing distances for resampled points. Other corner finders avoid relying on hard-coded variables. Bandera *et al.* use a multi-pass algorithm to detect the curvature, or contour, scale for strokes of various sizes [46]. Other segmenters find the optimal noise-filtering scale to segment a stroke [47, 48]. This technique increasingly applies Gaussian filters to curvature data, and, as the filters smooth the data, the number of detected corners drops. The optimal scale is determined to be where the number of corners reduced by increasing the smoothing factor tapers off.

3. Direct Corner Finding Applications

Some applications can benefit directly from corner finding without relying on sketch recognition. Keyboard input on small-scale, mobile devices is a significant issue, and the ATOMIK keyboard and SHARK software seeks to add another input option for people using pen-based devices [49, 50]. A virtual keyboard is displayed to the user, and the user "gestures" over the keyboard to type, hitting every letter they want with their stroke. The location and movement of the gesture are used to determine the intended word (Fig. 25). Although the SHARK system does not currently use corner finding as explained, the addition of finding the key points and changes in a stroke's



(a) Local convexity. The local convexity sums the direction values of the same size around a given point, with the region of support bounded by some constant k. Here, with $k = 1$, the local convexity at point p_i is equal to the sum of the direction values at $p_{i-1}, p_i, and p_{i+1}$. Since p_{i-1} has the opposite sign of p_i, the convexity at point p_i is $c_i = 90 + 50 = 140$. This convexity region of support is denoted by the dashed boundary, and the full region of support is the solid rectangle.



(b) Local monotonicity. The direction values for the points are $d_{i-1} = 75$, $d_i = 30$, and $d_{i+1} = 75$. The local monotonicity sums the direction for decreasing curvatures of the same sign, so the local monotonic curvatures are $c_{i-1} = 105$, $c_i = 30$, and $c_{i+1} = 75$. The local convexities at these points would be $c_{i-1} = 105$, $c_i = 180$, and $c_{i+1} = 75$, for a supporting window of $k = 1$.

Fig. 24. Examples of local convexity and local monotonicity, as presented by Kim and Kim [45].

direction could provide a benefit to the system.

Corners can also be used as features during stroke classification. Gestures corresponding to musical notes can encode information in sharp direction changes (i.e., corners) of a stroke [51]. These direction changes were used to indicate note duration (Fig. 26). Corners are also used as a feature in MARQS when searching for previous sketches [52] and by Patel *et al.* to help identify sketched strokes belonging to shape

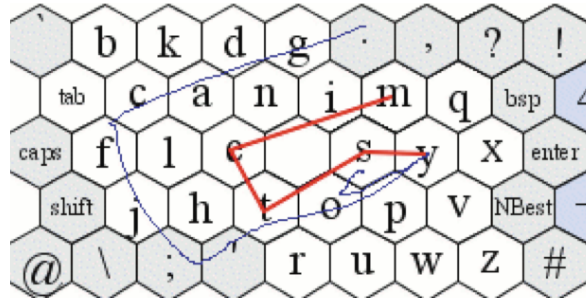


Fig. 25. A typing gesture in Shark² [49]. In this example, the user draws the word "system".

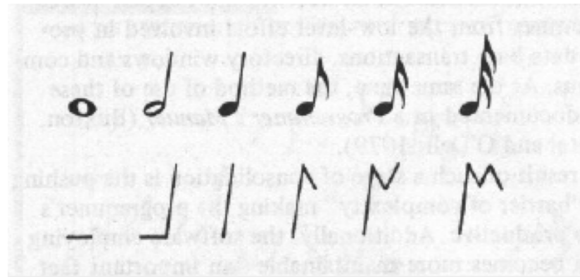


Fig. 26. Pen gestures can be mapped to musical notes based on the number of direction changes [51].

versus strokes composing text [53].

Other applications for corner finding are discussed in Appendix C.

4. Unsolved Problems

All segmenters can be improved in their segmentation accuracy. Since segmentation is often the first step toward recognizing a sketch, any errors in segmentation can percolate through a system and cause larger symbol and sketch errors.

No current corner finder has also found a good solution to finding the correct

corners and filtering incorrect corners due to noise. Sezgin *et al.* [41] and Stahovich [42] augment curvature-based approaches by using pen-speed, and, although this process does help pinpoint relevant corners, it can also be an additional source of noise. As Stahovich noted in his paper, false corners can be introduced if a user has a more constant drawing style than average, such as a trained calligrapher. In these cases, the user's more constant pen-speed causes even small speed fluctuations to introduce more false positive corners.

Most segmenters also use empirically found thresholds [11, 36, 40, 41, 42, 45]. Although these thresholds work well on each author's tested data, they can be susceptible to hardware changes (such as an increased sampling rate in digitizing pads), different user styles, or different domains. Instead, any segmentation thresholds should be trainable so that developers can easily find the correct threshold for a system.

Multiple primitive segmentation is also an unsolved problem. Some corner finders try to handle segmenting a stroke into both lines and arcs [40, 41, 42, 45], but multiple-primitive corner finder accuracies tend to be much worse than polyline corner finders. It is also debatable as to whether the construction of primitives should be left to low-level, primitive recognition, one step above segmentation.

We will address each of these problems during the motivation sections of our own segmenters (Chapters III-V), describing where the current segmenters are lacking and how our research has improved the field.

C. Primitive Recognizers

Primitive shape recognition and segmentation are often synergistic. Yu and Cai use the pen input's direction graph to segment a stroke into primitive lines, but they also utilize the direction graph to determine other basic primitives, such as circles

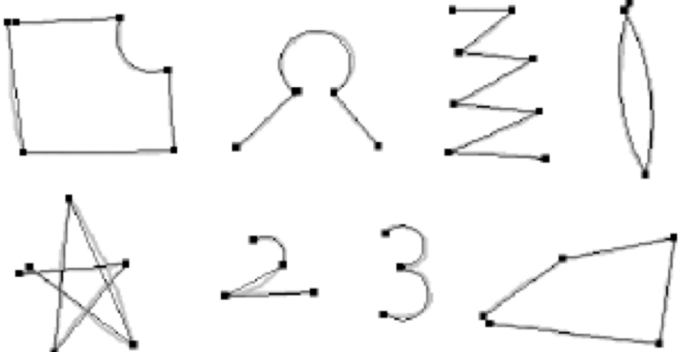


Fig. 27. Drawn strokes are beautified using Yu and Cai's segmenter and primitive recognizer [40].

and rectangles [40]. These primitives can then be *beautified* by replacing the original stroke segments with the optimal representation (Fig. 27). Sezgin *et al.* have a similar idea; they mention how recognition can be accomplished using simple, hand-crafted templates for primitives [41].

Hershberger and Snoeyink [37] and Hse *et al.* [54] also fit primitives to stroke segments in order to find the corners of a stroke for beautification purposes. Hershberger's algorithm is an extension of the Douglas-Peucker algorithm for line simplification [36]. Hse *et al.* fit both line segments and elliptical arcs to symbols using dynamic programming techniques [54].

Paulson's recent work took segmentation and primitive recognition one step further. The primitive recognizer, PaleoSketch, uses a trivial polyline segmentation along with finely-tuned heuristics in order to segment complex shapes [11]. PaleoSketch first breaks a given stroke into a series of line segments. Then, PaleoSketch analyzes the resulting segmentation using heuristics such as the number of line segments, the direction changes in the segments, and how the segments fit to an optimal shape (such

as fitting the polyline to an ellipse or helix).

CHAPTER III

MERGECF

One of our first attempts to improve stroke segmentation involved strokes composed of both lines and arcs [55]. We present a multi-pass corner finding algorithm called MergeCF that is based on continually merging smaller stroke segments with similar, larger stroke segments in order to eliminate false positive corners. MergeCF provides a substantial improvement over three benchmark corner finders.

A. Motivation

MergeCF work was inspired by the work of Sezgin *et al.* [41] and Stahovich [42]. These segmenters use curvature and speed values to segment a stroke. Sezgin *et al.*'s algorithm finds an initial set of corners that oversegments a stroke, and then they rank each corner based on some curvature metrics and greedily add the best corners, one at a time, to a final set of corners. Once the final set of corners has a fit error less than some developer-defined threshold, the algorithm stops and the final set of corners is returned.

Sezgin *et al.*'s greedy algorithm relies on the assumption that the correct corners to add to the system will always be ranked highest according to their metrics. Since the algorithm cannot backtrack and remove an added corner from the final set, the ranking metric must be perfect and rank every correct corner above every incorrect corner. We found that this ranking metric was susceptible to any noise or jitters in the user-drawn stroke, and MergeCF was created to try and account for this noise.

Stahovich also used pen speed and curvature to detect corners [42]. Unlike Sezgin *et al.*'s approach, Stahovich relies more heavily on chosen thresholds for average speed and curvature. Stahovich's algorithm does not build a fit greedily but seeks to remove

false positives by merging segments together. Their algorithm has many constraints as to when segments can be merged: strokes must be less than 20% of the length of their neighbors in order to be merged, merged pairs of strokes must be of the same primitive type, and the merged fit error must be no more than 10% than the fit errors of the two individual segments. When analyzing strokes, we found that these constraints are too rigid; we designed MergeCF to handle more general data.

B. Implementation

MergeCF utilizes curvature and speed differences within a stroke to obtain an initial corner segmentation for the stroke. We then repeatedly merge smaller stroke segments with longer segments, and, if the fit for the merged segment is less than 50% more than the sum of the individual segment errors, we eliminate the corner between the two segments.

1. Curvature and Speed Values

Our curvature and speed values are based on the equations given by Stahovich [42] and Yu and Cai [40]. These equations were discussed in Background chapter of this thesis, Chapter II, but are reproduced here.

The distance between two points is the Euclidean distance between the points, and the path length across a series of points $p_a, p_{a+1}, \ldots, p_b$ is taken to be the sum of the Euclidean distances between each pair of points (Eqn. 3.1).

$$pathLength(a, b) = \sum_{i=a}^{b-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{3.1}$$

where x_i and y_i are the respective x and y values of the point, p_i.

Curvature values for a point at stroke index i are taken to be the change in

direction (Eqn. 3.2) across a window of points, divided by the path length across the window (Eqn. 3.3).

$$direction_i = \arctan\left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) \tag{3.2}$$

$$curvature_i = \frac{\sum_{k=i-W}^{k+W-1} |direction_{k+1} - direction_k|}{pathLength(i - W, i + W)} \tag{3.3}$$

Speed at a point, p_i, is calculated as the path length change across the point, divided by the time (t) difference (Eqn. 3.4).

$$speed_i = \frac{pathLength(i - 1, i + 1)}{t_{i+1} - t_{i-1}} \tag{3.4}$$

2. Initial Fit

After we compute the curvature and speed values for each point, we find an initial set of corners by taking points that are local maxima (for curvature) and local minima (for speed), with respect to set curvature and speed thresholds. Our curvature local maxima are the greatest curvature values at a peak in the curvature graph; each maxima is bounded by the points at which the peak positively crosses the threshold until it decreases below the threshold. The local minima are found in a similar fashion. Curvature and speed corners are found separately and then combined into one set of corners.

In this implementation, our thresholds were set to find points that are local maxima above the average curvature and local minima below 90% of the average speed. An example of these thresholds in action can be seen in Fig. 28 and 29. These thresholds were found empirically using a set of 157 line and arc shapes. Empirically derived thresholds are common in segmentation algorithms [11, 36, 40, 41, 42, 45].

Our usage is justified given the existence of a separate dataset that we found our thresholds on before we begin testing.

In order to reduce the effects of noise, we iterate through the initial set of corners and check for points that are close together in proximity. If two corners are less than 15 pixels apart, then we remove the corner with the smallest curvature from the initial fit. We also remove corners closer than 10 pixels to the endpoints since these are most likely due to noise from stroke hooks from unwanted pen movements during pen-down and pen-up events.

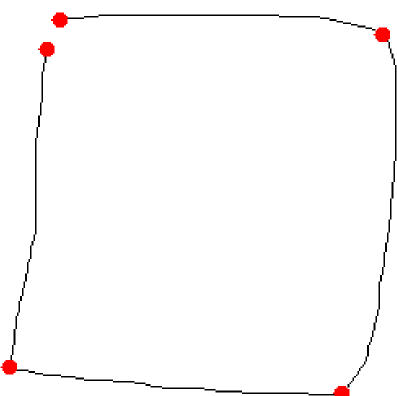
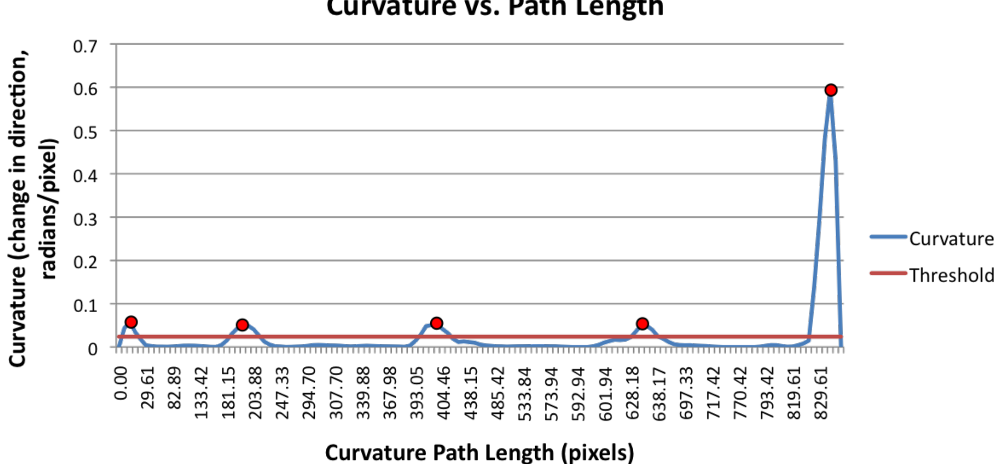
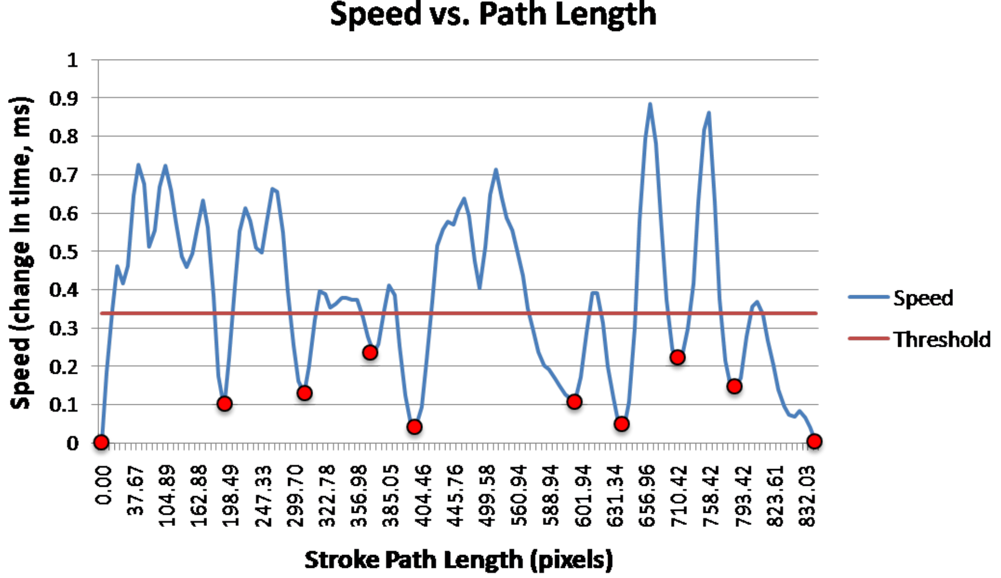


Fig. 28. The same sketched square from Fig. 17, reprinted here. The curvature and speed graphs for this stroke are shown in Fig. 29(a) and 29(b).



(a) Curvature graph. The average threshold is shown as a horizontal, red line. 5 local maxima (marked with red points) are above this threshold and are possible corners.



(b) Speed graph. The speed threshold of $0.90 \times avgSpeed$ is shown as a horizontal, red line. 10 local minima (marked with red points) are below this threshold and are possible corners.

Fig. 29. Curvature and speed graphs for the stroke in Fig. 28.

3. Merging Segments

Our initial corner fit tends to contain a few extraneous points that overfit the stroke. The main algorithm involved with our corner finding system is designed to eliminate these false positives, and MergeCF works on the assumption that corners surrounding the smallest segments are those more likely to be false positives overfitting the data.

The algorithm first finds the smallest stroke segment, checks if the segment can be merged with any of its neighbors, and then merges the segment with the best neighboring segment. The best segment is determined to be the segment that has the least primitive fit error (either line or arc) when combining the two segments.

The fit error calculations we use come from PaleoSketch [11]. The two primitives our system handles are lines and arcs, and the PaleoSketch primitive recognizer calculates line and arc fit errors by finding the mean squared error and the feature area error [40] for each segment.

We use a line test to discern which primitive fit to use for each segment. Lines must to pass a ratio test, where the test takes the Euclidean distance length between the two points and divides the value by the path length between the points [41, 42]. If this ratio is greater than a set threshold, then the segment is a line. If the ratio is less than the threshold, then the segment is considered an arc. The appropriate primitive fit errors are then computed. In our algorithm, the line ratio test threshold is equal to 0.95.

As an example, Fig. 30 shows a symbol with an initial corner fit containing three false positives (the circled points) and numbered stroke segments. Merging segment 5 with segment 4 would still result in an arc fit error that is not too much higher than the either segment 4 or 5's original error. Yet, merging segment 5 with segment 6 would produce a very high primitive error for either lines or arcs. Therefore, the best

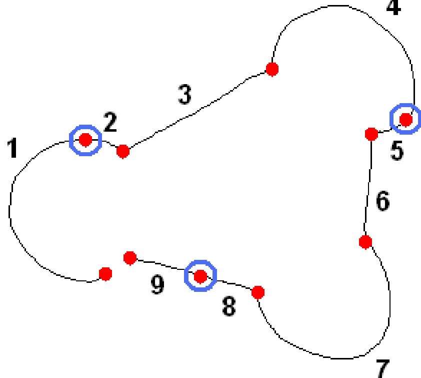


Fig. 30. Initial set of corners found for a stroke, which would split the stroke into 9 primitive lines and arcs. False positives are circled.

segments to merge are 4 and 5, and the circled point in between the two segments is removed from the corner set.

4. Algorithm

A more formal definition of our algorithm is as follows:

1. Calculate the path length of each segment between corners.
2. Calculate the average path length of the segments.
3. Sort the segments from step 1 in ascending order, based on the path lengths.
4. For each segment s shorter than the average segment:
   (a) Determine whether the segment is a line or an arc, using the line ratio test. Calculate the primitive fit error of the segment, $FitError_s$.

(b) Perform line ratio tests and calculate the appropriate fit errors of the segments to the left and to the right of segment s, if there are any. These are $FitError_{s-1}$ and $FitError_{s+1}$, respectively.

(c) Calculate the primitive fit error of the joined segments $s-1$ and s, which we will call $FitError_{left}$. Also, calculate $FitError_{right}$ from s and $s+1$.

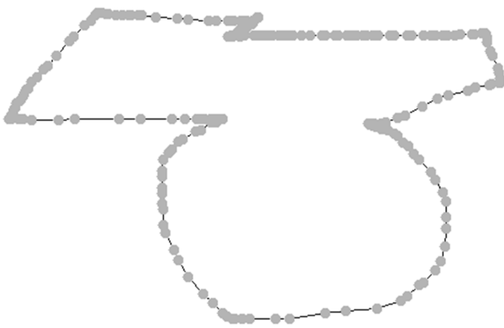
(d) If $FitError_{left} < FitError_{right}$ and $FitError_{left} < 1.5 * FitError_{s-1} + FitError_{s}$, then remove the corner between $s-1$ and s. Otherwise, preform similar checks for the right side of the segment.

5. Repeat steps 1-4, but, after each run, multiply the average segment by the number of runs. This steadily increases our "segment shorter than" threshold. Stop repeating once every segment is shorter than this threshold.

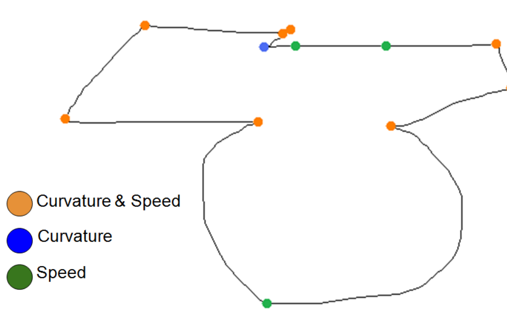
The threshold of 1.5 in step 4(d) was found by examining a set of training data. We analyzed our data and discovered that having the merged segment error be no more than 50% greater than the sum of the individual segment errors gave produced good segmentations.

MergeCF's algorithm underperforms when trying to merge two line segments. The primitive fit error for two individual line segments tends to be much lower than the fit error for the joined segments. After running the above algorithm, we iterate through our remaining segments and check specifically for two consecutive lines. If both lines have similar slopes, we merge the two segments together by eliminating the corner between them.

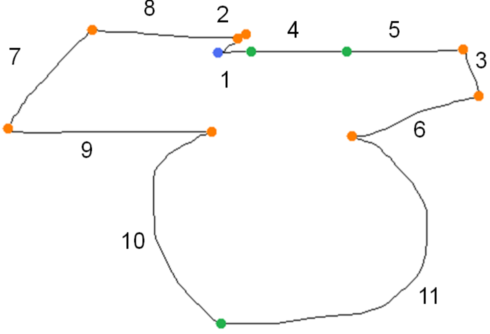
To provide further clarity for how our algorithm works, a visual walkthrough is presented in Fig. 31.



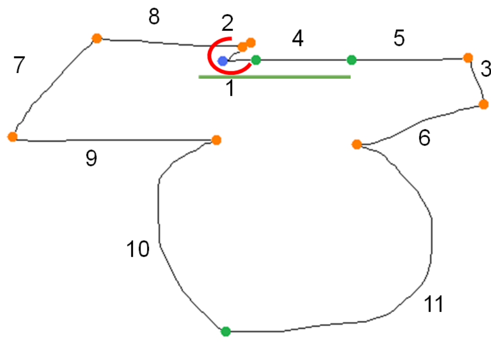
(a) Original points of the stroke.



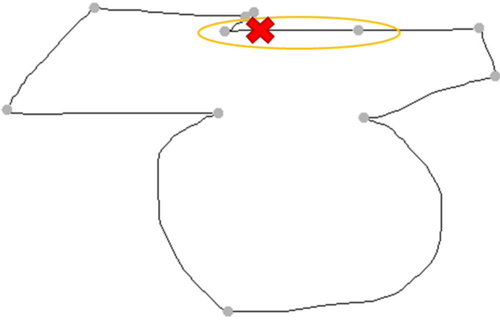
(b) Corners found using curvature and speed thresholds. Some points pass both curvature and speed tests.



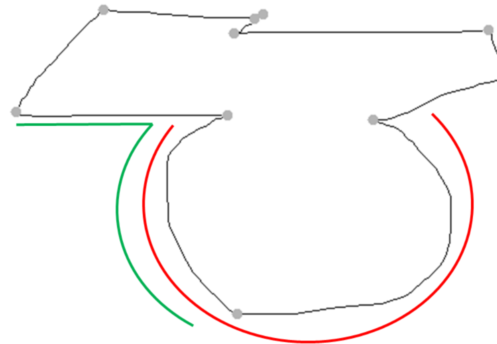
(c) Sort the segments by length.



(d) Check whether segment 1 can be merged with segments 2 or 4.



(e) Merge segments 1 and 4, eliminating the corner between them.



(f) Repeatedly check and merge segments until no more can be merged. This is the last merge before the algorithm terminates.

Fig. 31. A visual walkthrough of the MergeCF segmentation algorithm.

5. Intuition

Our decision to merge the smallest segments first works because of the inherent way that symbols are drawn. Complex (multiple-primitive) and polyline stroke symbols tend not to have segments that have extreme variance in length. Therefore, very small stroke segments attached to much longer segments are typically noisy data that should be removed. This is especially true of the small stroke hook segments near the endpoints.

If an initial corner fit contains few false negatives (i.e., missing corners), then the any error in the segmentation can be assumed to be from false positives. Now, if we assume that all of the stroke segments are drawn at the same scale (i.e., relatively same length), then any false positives would split a stroke segment of average length into smaller pieces. Therefore, the merging algorithm should start by examining the smallest stroke segments since they are most likely to contain false positives as end points. Continually increasing the threshold that determines which segments are small ensures that all stroke segments will eventually be evaluated.

Another reason why we want to merge smaller segments first is due to the way fit errors are calculated. Suppose a stroke consisting of an arc and a line has the initial fit shown in Fig. 32. If the algorithm started by merging the largest segments

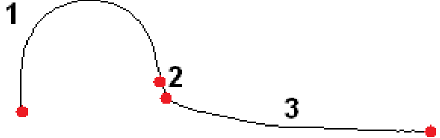


Fig. 32. Initial set of corners found for a stroke consisting of an arc and a line. Segment 2 is the smallest, unneeded segment and should be merged with Segment 1.

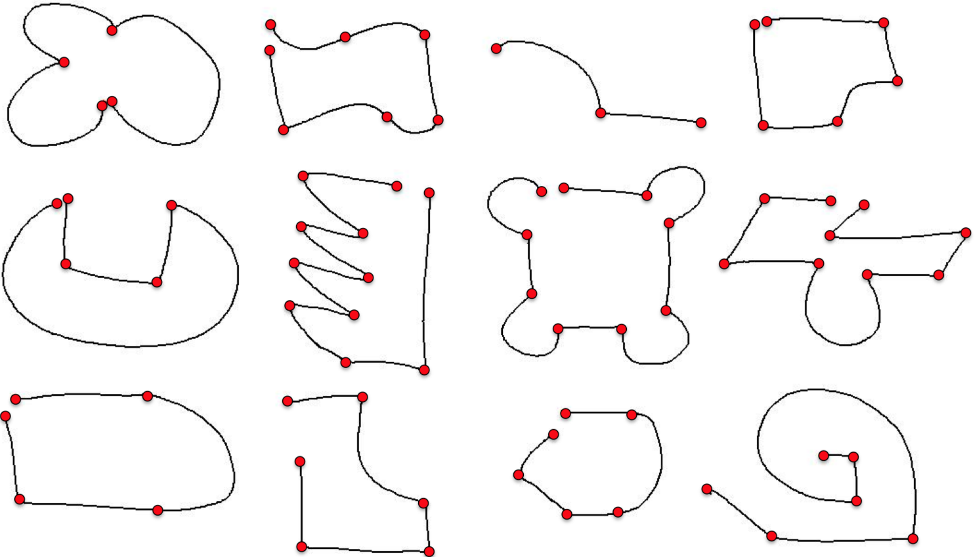
first, then segment 2 would be merged with Segment 3 since the error calculation for the line consisting of 2 and 3 is not substantially different than the line error for Segment 3 alone. In fact, Segment 2 can be considered a hook of Segment 3 since it is substantially smaller. A much better option would be to merge Segment 1 and 2 together to form a slightly larger arc. To avoid the problem of merging Segment 2 and 3 we let the smallest segments decide their best merging options.

C. Results

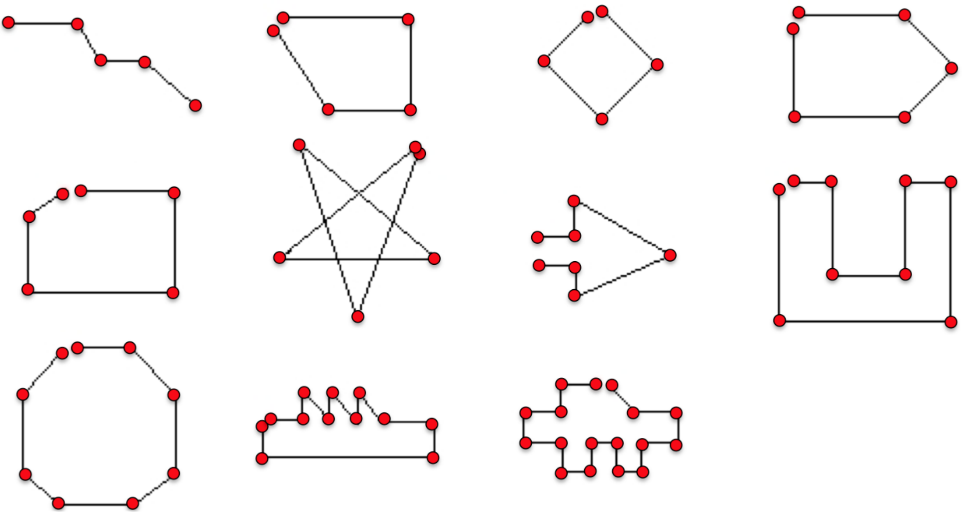
MergeCF was developed around a training data set consisting of 157 unistroke shapes drawn by five users taking a sketch recognition course at Texas A&M. The algorithm was tested on a different set of data based on the symbols presented by Kim and Kim [45]. This test set consisted of 501 complex shapes and polylines, each drawn with a single stroke (Fig. 33). During the collection of both data sets, users were asked to sketch a given shape with easily defined corners.

Results were gathered on three other corner finders as well, Sezgin *et al.*'s algorithm [41], Stahovich's algorithm [42], and Kim and Kim's algorithm [45]. We implemented each of these algorithms as presented in their respective papers, and we tested all of the corner finders on the same data sets.

The results in Table I show how our algorithm outperforms the baseline corner finding algorithms from Sezgin *et al.*, Stahovich, and Kim and Kim. Correctly classified examples of our test shapes can be seen in Fig. 34.



(a) Multiple-primitive, complex symbols with lines and arcs.



(b) Polyline symbols.

Fig. 33. Set of 23 symbols we used for testing. 6 users drew each of these symbols up to 4 times each. 12 of the symbols contained both lines and arcs (a), and 11 of the symbols contained only lines (b). Due to some users quitting the study early and other data collection issues, the total number of symbols collected was 501. Red dots indicate the corners.

Table I. Results for MergeCF and three baseline corner finders. The results are for a set of 501 shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs. The metrics are discussed in Section C.1.

	MergeCF	Sezgin	Stahovich	Kim
False Positives	233	300	2081	233
False Negatives	113	384	161	479
True Positives (Correct Corners)	3299	3028	3251	2933
True Negatives	124,951	124,884	123,105	124,951
Total Correct Corners	3412	3412	3412	3412
Accuracy	0.997	0.995	0.983	0.994
All-or-Nothing Accuracy	0.667	0.415	0.0818	0.327
F-measure	0.950	0.898	0.744	0.892
Sensitivity (a.k.a. Recall)	0.967	0.887	0.953	0.860
Precision	0.934	0.910	0.610	0.926
FDR	0.066	0.090	0.390	0.074
Avg. time for all 501 strokes (ms)	30,200	162	430	336
Avg. time per stroke (ms)	60.3	0.323	0.858	0.670
All-or-nothing / Avg. time per stroke	0.0111	1.28	0.0953	0.488

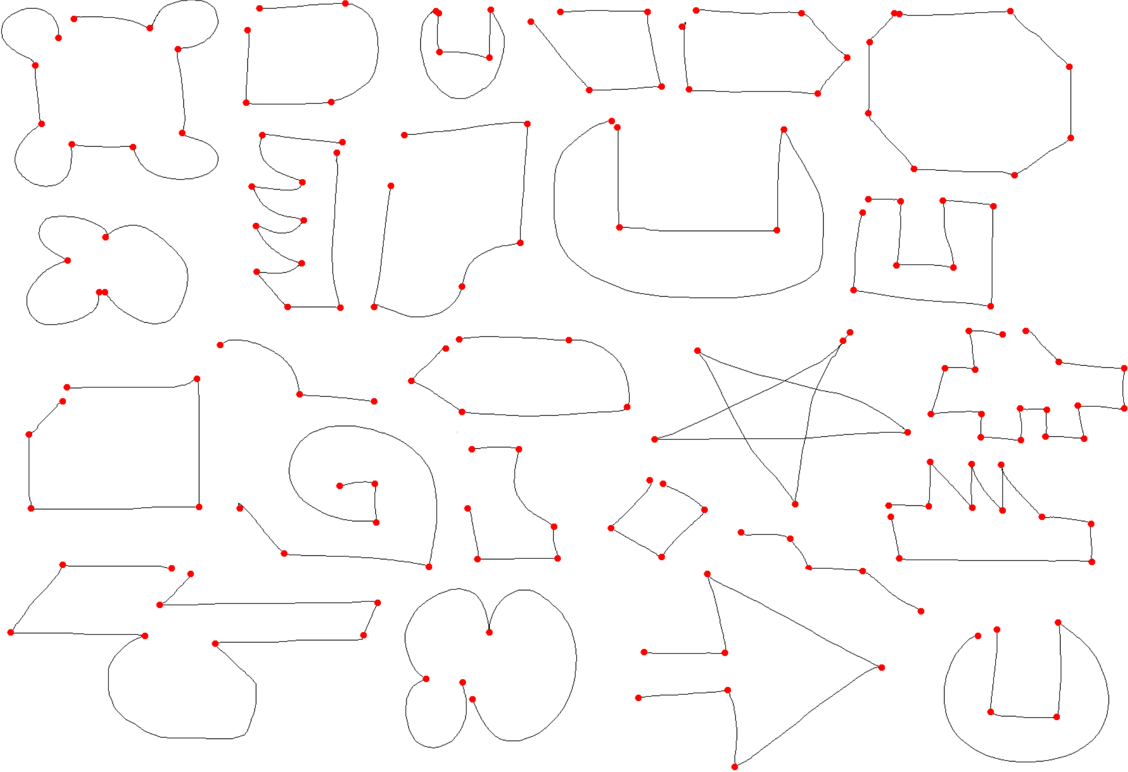


Fig. 34. Examples of correctly classified symbols by MergeCF. These symbols come from the set of 501 complex and polyline shapes drawn by six users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper.

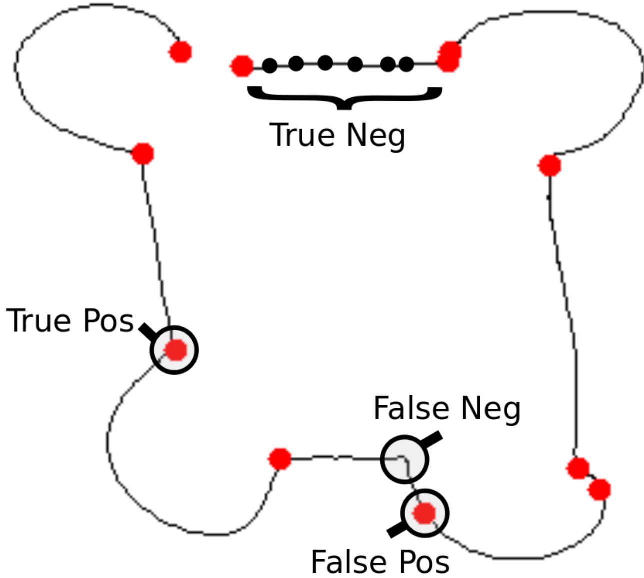


Fig. 35. A stroke with examples of true positives, true negatives, false positives, and false negatives highlighted.

1. Accuracy Metrics

We use different metrics to determine the performance of each corner finder. In the table, false positives are extraneous, unnecessary corners in a segmentation, whereas false negatives are missed corners (Fig. 35). The following equations are described using true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).

The first metric is the corner finder's sensitivity or recall, also known as "correct corners found" accuracy [41]. This accuracy is calculated by dividing the number of correct corners found by the total number of correct corners that a human would perceive (Eqn. 3.5). This metric is also known as recall in some literature.

$$TP/(TP + FN) \tag{3.5}$$

This sensitivity metric does not discount false positives and only penalizes for false negatives. Therefore, a system that returns every point possible as a corner would achieve a perfect 1.00 sensitivity since all of the correctly perceived corners would be found. Another problem with using the metric is the ability to count end points as corners. Technically, the end points of a stroke could be considered segmentation points since they are used for generating the primitives, and in some cases the end point of a stroke could be shifted if there are large, noisy hooks that do not accurately represent the primitives. These end points are typically given, so counting them as corners can artificially inflate the segmenter's sensitivity.

Traditional accuracy is the second metric we use to compare corner finders (Eqn. 3.6).

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.6}$$

The main issue with this traditional accuracy equation is the use of true negatives, which are incorrect corners that were not found by the segmenter. Strokes can have hundreds or thousands of points, but only a few of these points are corners. Therefore, the number of true negatives is quite large in comparison to the other variables, causing the Eqn. 3.6 accuracy to be close to 1.00 as the number of true negatives grows. Even worse, as the number of points in the test set increases, the overall accuracy for every segmenter approaches 1.00 and makes the calculation irrelevant.

We use a different accuracy measure to counteract the issues in sensitivity and accuracy: all-or-nothing accuracy. All-or-nothing implies that only the minimum number of corners to segment a figure are found in order for a stroke to be considered

correctly segmented. In other words, for a stroke to be counted a correct stroke it must have no false positives or negatives. This accuracy is calculated by taking the number of correctly segmented strokes divided by the total number of strokes. For corner finding, all-or-nothing accuracy is a more important accuracy measurement since any recognition errors can frustrate users, and we do not want users to become agitated if their strokes they do not segment correctly. From a user's point of view, the computer is either correct or it is wrong, and we wanted to model this behavior in our results.

We we also measure the number of true positives expected in each segmentation through precision:

$$TP/(FP+TP) \tag{3.7}$$

and we measure the false discovery rate (FDR) for each segmenter, which determines what percentage of false positives expected in a segmentation. It is the opposite of precision:

$$FP/(FP+TP) \tag{3.8}$$

An F-measure can be calculated using the precision and recall scores for each segmenter (Eqn. 3.9). The F-measure is often used in information retrieval systems, such as Internet searching, where the number of irrelevant documents for a query is much higher than the number of relevant documents. Segmenters behave in a similar manner; there are significantly more points in a stroke that are not corners than points that are corners. The F-measure then calculates performance based on a weighted average of the precision and recall values for a segmentation.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3.9}$$

Finally, we have a metric that divides the all-or-nothing accuracy by the average time taken per stroke. This metric provides us with a measure of how well a segmenter performs while penalizing longer run-times.

D. Discussion

Our corner finder significantly improves corner detection over the three benchmark systems in our all-or-nothing accuracy measure. MergeCF finds fewer false positives and negatives than our Sezgin *et al.*, Stahovich, and Kim and Kim implementations, and the all-or-nothing accuracy is over twice that of the previous best corner finders'.

MergeCF performs better than the other corner finders for a few reasons. Sezgin *et al.*'s algorithm assumes that the best corners, or the correct ones, will always be ranked higher than any false positives. This assumption is often invalid on complex shapes where minor speed differences and line noise can greatly affect the the author's corner metrics. Noisy arcs are the main culprit in this issue and produce many false positives along subtle bumps or peaks in the arc. Also, since Sezgin *et al.*'s algorithm chooses the fit with the least number of corners below a certain threshold, it is often the case where correct corners are missing from the final segmentation if the threshold is too high for a shape. If the majority of the corner fits are below the threshold, then the corner fit with the least number of corners can be a poor choice.

Stahovich's algorithm produces many false positives. Although Stahovich's empirical thresholds for merging and splitting worked well for their testing hardware and dataset, we found the thresholds to be much too strict. The thresholds rarely allowed the merging segments that needed to be combined, and the algorithm often

split segments that should have remained whole. MergeCF's thresholds for merging are much more lenient, but it is also our technique to sort strokes by length before we begin merging that helps out algorithm avoid many of these false positives.

Kim and Kim's corner finding algorithm produces many false positives and false negatives, mainly due to sensitive thresholds present in their system. Their algorithm oversmooths the data by using resampled points as well as smoothing curvature metrics, and when the data is too smooth, points with a high curvature have only slightly higher curvature values than points with average curvature, causing false positives. Similarly, oversmoothed data causes corners at obtuse angles between primitives to be missed.

To summarize, MergeCF avoids the issues of these baseline corner finders by:

- Having an initial fit with few false negatives
- Evaluating individual corners and segments at a local level
- Using inherent properties of false positives to examine short segments first
- Performing multiple passes through the segments to ensure that each segment is eventually evaluated and merged if necessary.

1. Algorithm Speed

We ran real-time tests on each of the segmenters to compare their relative speeds. The time values in Table I are in milliseconds and were averaged over 20 full runs of the test data. We performed 20 runs so that any influence of background computational processes would be lessened. Each time value counts only the segmentation time of the stroke; the times to load the stroke and switch to the next file are not included in the calculation. The time calculations were performed using a Mac Pro with a pair

of quad-core, 2.8GHz Intel Xeon processors with 10.0 GB of RAM, running on OS X 10.6.2.

MergeCF ran at approximately 60 ms per stroke, which is slower than the other segmentation algorithms by two orders of magnitude, but it still provides real-time segmentation for each stroke. 60 ms per stroke is much faster than a human's reaction time, so the stroke can be segmented before any perceivable lag.

The all-or-nothing accuracy over average time per stroke metric also shows that MergeCF does not provide the best performance with respect to time. MergeCF's metric value is 0.0111, an order of magnitude lower than the best, Sezgin *et al.*'s value of 1.28. Yet, MergeCF does have a 50% increase in all-or-nothing accuracy over Sezgin *et al.*'s algorithm on the test set.

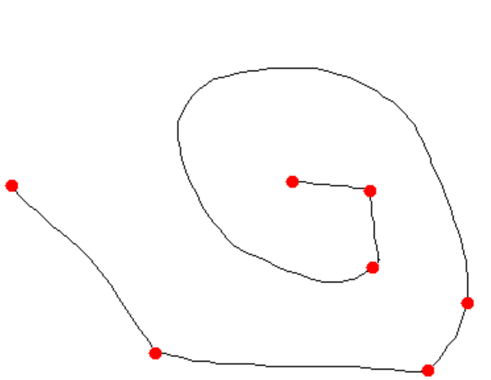
E. Limitations and Future Work

Although MergeCF improves segmentations considerably, there are still a few key issues with the algorithm.

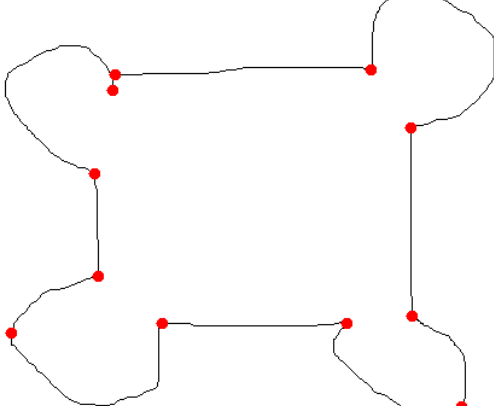
1. Arc Issues

First, merging two smaller arcs together can be difficult since arcs are classified as sections of circles, and the error associated with arcs tends to be high. Appending two slightly offset arcs often produces a shape that has a considerably higher error than either of the individual arcs (Fig. 36(a)).

Another issue with trying to segment primitives into arcs is that arcs are often noisy. Users often draw arcs with sharper points of curvature than intended (Fig. 36(b)). These sharp points are difficult to remove from the final set of corners due to their high curvature, and the arc merging issues discussed above prevent the points



(a) Merging two arcs together can often be difficult, since appending two arcs can often fit an elliptical or spiral shape better than a circle.



(b) Arcs often contain sharper points of curvature than the user intended, such as the two arcs in the bottom portion of this symbol.

Fig. 36. Issues with segmenting arcs.

from being removed.

We tried to prevent this jump in error by introducing an additional primitive: curves. A Bezier curve could be created to approximate two adjacent arc segments, and if the curve fits the segments well, then the corner between the segments would be eliminated. The primitive recognizer we use from [11] already has a definition for curves, so we modified our main merging algorithm to handle the error associated with these primitives. Unfortunately, segmenting lines, arcs, and curves produced slightly poorer segmentations (Table II). The all-or-nothing accuracy decreases slightly when adding curves, but the number of negatives greatly increases. These results do not bode well for segmenting many primitives at once.

Table II. Results comparing MergeCF with lines and arcs and MergeCF with lines, arcs, and curves.

	MergeCF	MergeCF with Curves
False Positives	233	188
False Negatives	113	225
True Positives (Correct Corners)	3299	3187
True Negatives	124,946	124,996
Total Correct Corners	3412	3412
Accuracy	0.998	0.997
All-or-Nothing Accuracy	0.667	0.647
Sensitivity (a.k.a. Recall)	0.971	0.934
Precision	0.954	0.944
FDR	0.046	0.056

Table III. Results for MergeCF and our baseline algorithms on polyline-only data. There are 244 polyline-only strokes in our test set of 501 strokes.

	MergeCF	Sezgin	Stahovich	Kim
False Positives	24	29	799	22
False Negatives	3	162	12	242
True Positives (Correct Corners)	1838	1679	1830	1599
True Negatives	56,620	56,615	55,844	56,622
Total Correct Corners	1841	1841	1841	1841
Accuracy	1.00	0.997	0.986	0.995
All-or-Nothing Accuracy	0.914	0.594	0.135	0.443
F-measure	0.992	0.946	0.818	0.923
Sensitivity (a.k.a. Recall)	0.998	0.912	0.993	0.868
Precision	0.987	0.983	0.696	0.986
FDR	0.013	0.017	0.304	0.014

2. Polyline Data

On the opposite end of the spectrum, we tested MergeCF on the polyline-only data in our test set (Fig. 33(b)). Out of the 501 strokes in our test set, 244 of them are polyline only. MergeCF performed admirably on this data, having an all-or-nothing accuracy of 0.914 (Table III). The second best accuracy, from Sezgin *et al.*'s algorithm, was less than half that of MergeCF's.

The results from Table III indicate that MergeCF performs much better on polyline data than on line and arc data. In fact, the all-or-nothing accuracy on the 257 line-arc symbols in our test set is 0.431, which is less impressive than the all-or-notthing accuracy of 0.667 on the entire 501 symbols.

Furthermore, PaleoSketch has shown that accurate primitive recognition can occur when using only a polyline segmentation [11]. In PaleoSketch, each available primitive (line, arc, curve, ellipse, etc.) tries to fit itself to a given stroke and returns an error based on its fit. A decision tree then returns the best primitive mapping to a stroke. One aspect of the decision tree involves segmenting each stroke using a polyline segmenter and using the number of segments returned to help classify the stroke. For instance, a stroke must have more than 3 polyline segments in order to be considered as a possible arc.

Therefore, PaleoSketch can also handle polylines and complex symbols by using only a simple polyline segmenter. As long as the polyline segmenter can find the correct points at which two primitives meet, PaleoSketch can then recursively try different combinations of primitive fits on the resulting groups of segments until a decent fit is found, similar to how Yu and Cai perform segmentation [40].

3. Implementation

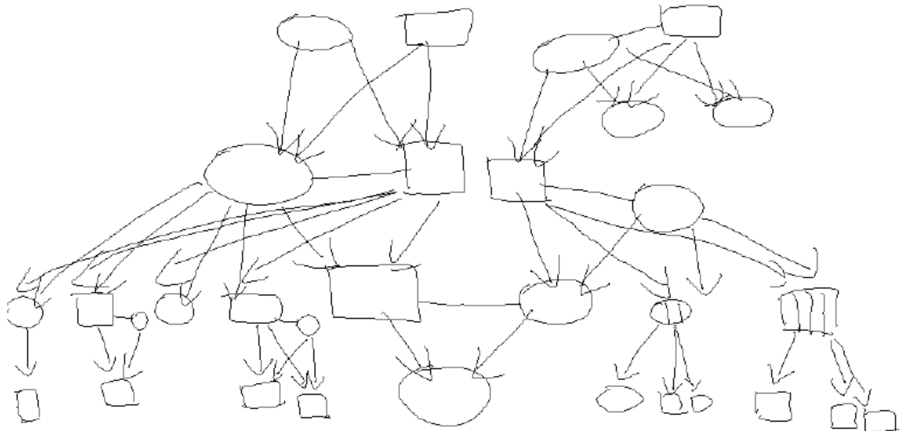
MergeCF is relatively difficult to implement. PaleoSketch uses multiple line and arc tests in order to calculate the primitive fit errors. Since MergeCF has very high accuracy on polyline data compared to complex, line-arc data, we might be performing too much work to accurately segment strokes into polylines.

4. Test Set

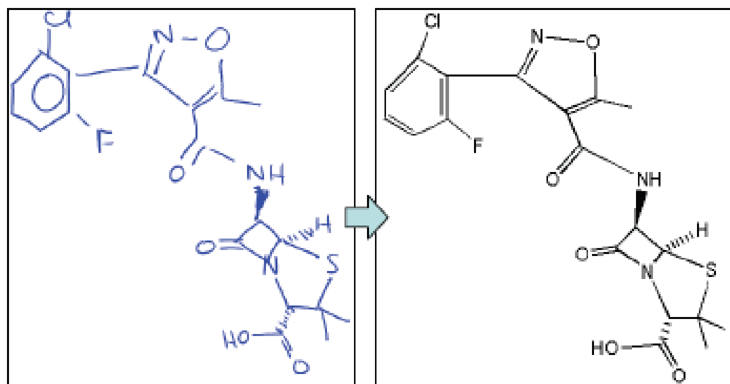
Finally, the symbols we use to test MergeCF are the same symbols used by Sezgin *et al.* [41] and Kim and Kim [45]. We used these symbols to have better comparability in our testing against other algorithms, but the symbols themselves are made to be more difficult than the typical symbols sketch recognition domains require.

Graphical, diagramatic domains are often composed of symbols formed with simple polygons (triangles, rectangles, starts), ellipses, and connectors (lines, arrows). For instance, family trees contain rectangles, ellipses, lines, and arrows [3, 56] (Fig. 37(a)). Chemistry symbols contain capital letters, polylines, and lines [4] (Fig. 37(b)). Even a more difficult domain, such as course of action diagrams, are composed of rectangles, diamonds, polylines, lines, ellipses, waves, and arrows (Fig. 38). In each of these domains, there are no symbols that reach the complexity of the line-arc symbols we test on.

We are not saying that testing on a difficult test set like Fig. 33 is a problem. In fact, we *want* to show that our corner finding algorithms perform well in extreme situations. But, if real-world diagram data is sketched using polylines and ellipses, then we should focus on those aspects of segmentation first. We should, therefore, work to improve polyline accuracy to be almost perfect in these extreme cases; PaleoSketch can then handle any ellipses.



(a) Family tree data, presented by Cates and Davis [56].



(b) Chemistry symbols, presented by Ouyang and Davis [4].

Fig. 37. Sketched symbols from different, real-world domains. The domains consist mainly consist of shapes formed from lines, polylines, and ellipses.

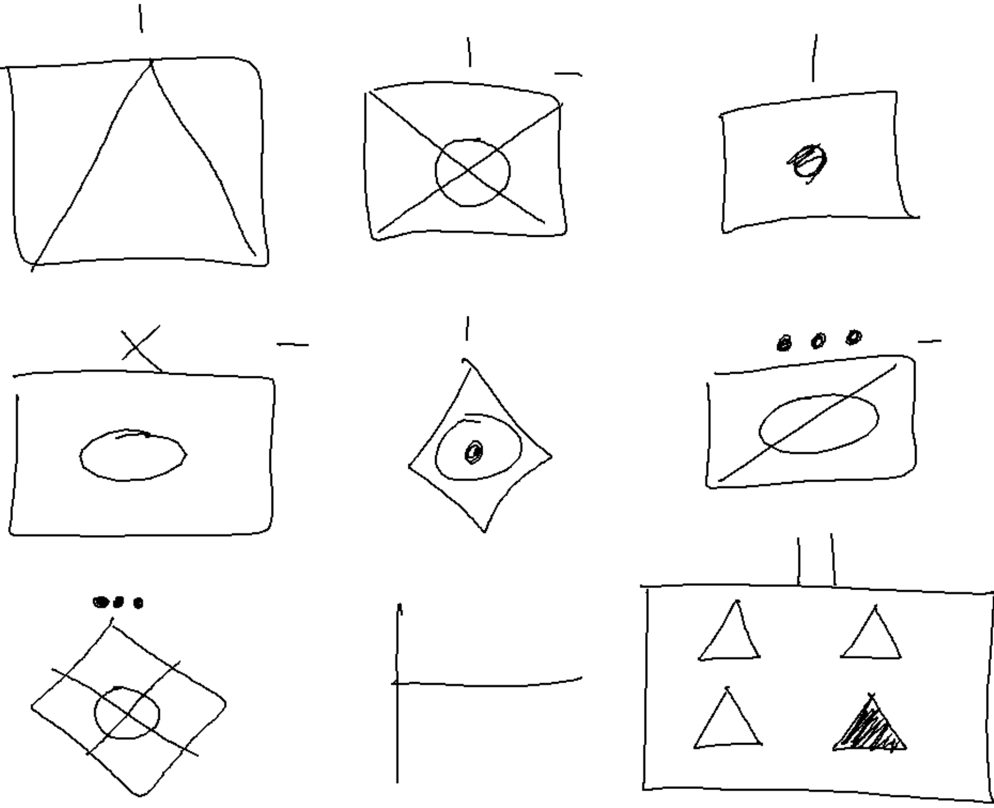


Fig. 38. Examples of military course of action (COA) symbols. The symbols can be described using primitives and simple shapes such as lines, ellipses, triangles, rectangles, diamonds, and dots.

5. Contributions

In summary, MergeCF contributes to the sketch recognition community by:

- Creating a multiple primitive recognizer that outperforms the current baselines.
- Efficiently eliminating false positive corners by merging the smallest stroke segments first.
- Demonstrating how primitive recognizers can be used to improve stroke segmentations.

6. Directions for Future Work

After we analyzed MergeCF, we came to a few conclusions about corner finding:

1. It is hard to produce acceptable corner finding accuracies when working with multiple primitives.
2. The more primitives being segmented, the more difficult segmentation becomes.
3. Symbols in real-world data often focus on polylines and other simple primitives. More extreme cases like those found in our test set are rarer, but necessary if we want to push the field past basic diagram recognition.
4. Segmenting a stroke into polylines and then using a primitive recognizer, such as PaleoSketch, might be a better solution than segmenting strokes directly into multiple primitives.

It is for these reasons that our later corner finders focused solely on polyline segmentation. The next algorithm we will present, entitled ShortStraw, moves to both simplify and improve upon previous polyline-only techniques while retaining the high polyline accuracy levels we can achieve with MergeCF.

CHAPTER IV

SHORTSTRAW

ShortStraw is designed to be simple to understand and easy to implement. As such, the entire algorithm can be discussed in detail in the paper, and pseudocode for the algorithm is also presented in Appendix A.

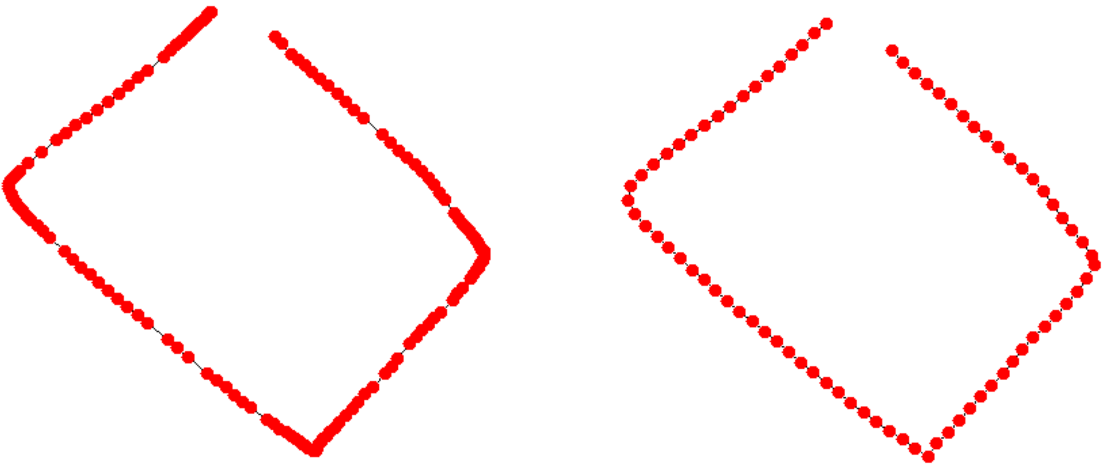
A. Motivation

We transitioned to working with polyline segmentation algorithms after we discovered the limitations of multi-primitive segmenters to be quite expansive (See Chapter III, Section E).

We built MergeCF while taking the Fall 2007 Sketch Recognition course at Texas A&M. During this time, many of the graduate students struggled with the implementation of algorithms like Sezgin *et al.* [41] and Kim and Kim [45]. The curvature and primitive fit calculations were not always easy to understand or implement.

We wanted to (1) make a segmentation algorithm that was easy to implement and (2) make the algorithm still perform better than previous work. Similar to how Wobbrock *et al.* created an easy-to-use template matching algorithm with $1 [23], we wanted to provide sketch recognition students, researchers, and developers with simple but powerful techniques to segment strokes.

The algorithm we created, entitled ShortStraw, is founded on the concept that curvature in polylines can be approximated with little computation. Like Wobbrock *et al.* and Kim and Kim, we discovered that resampling points in a stroke allowed us to reduce segmentation complexity. The next few sections will explain ShortStraw in detail.



(a) Original points of the stroke (b) Resampled points of the stroke

Fig. 39. The original points in 39(a) are varied in distance away from each other, whereas the resampled points in 39(b) are interspaced evently.

B. Resampling

The first step to ShortStraw involves resampling the points of a stroke to be evenly spaced apart (Fig. 39). Resampling points is necessary in ShortStraw, for reasons that will be discussed in Section IV.C.1.

The algorithm for resampling points is based on the algorithm presented in $1 [23]. Although the resampling algorithm remains the same, we determine the interspacing distance of the resampled points differently.

In ShortStraw, points are resampled based on the diagonal length of the stroke's bounding box. The interspacing distance is equal to the diagonal divided by a constant factor (Fig. 40). In our implementation, this constant was empirically determined to be 80. We found that increasing the value caused too much noise, whereas decreasing the constant created oversmoothed strokes.

In essence, this interspacing distance is an indication of the scale of a stroke. Human perception of what constitutes a significant change in a symbol varies with scale [57, 48], and we wanted to support drawing at different scales.

We impose a lower-bound on our resample spacing of 0.5 pixels. In some cases, if a user drew a dot or accidentally tapped the screen, the resample size could be close to 0.0 and caused infinite loops.

The original points of the stroke can be resampled once we have calculated the interspacing distance, S. First, an empty set of points, called *resampled*, is created to store any new resampled points. The first point in the original point set, $points_0$, is then appended to *resampled*. A distance holder D is initialized to 0.

The main algorithm is as follows:

1. The Euclidean distance d between two consecutive points, $points_{i-1}$ and $points_i$, is added to D.

2. If D is less than the interspacing distance S, then we increment i by 1 and repeat from step (1).

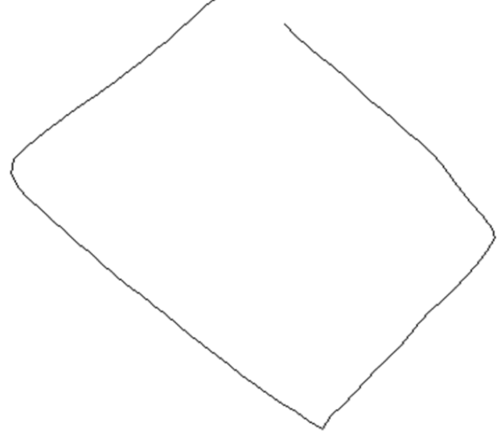
3. Otherwise:

   (a) Create a new point q that is located approximately S Euclidean distance away from the last resampled point. q_x and q_y are calculated to be $(S - D)/d$ distance between $points_{i-1}$ and $points_i$.

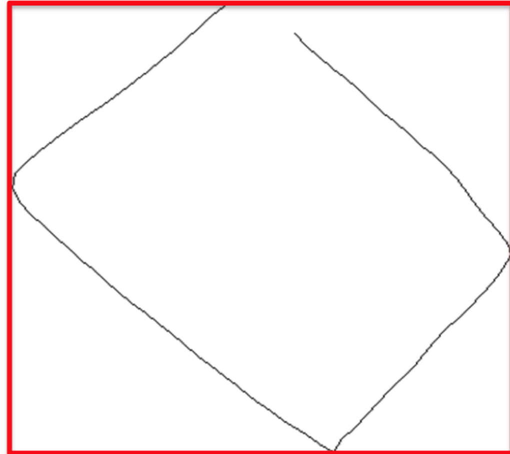
   (b) Append q to *resampled*, and insert q before $points_i$.

   (c) Repeat from step (1) without incrementing i.

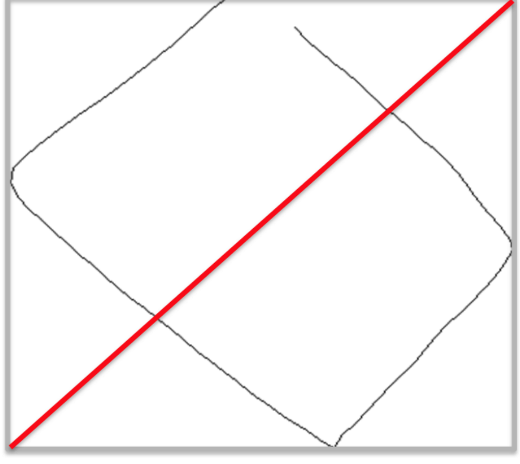
The main algorithm loop terminates when $i > |points|$. The algorithms for both the interspaced distance calculation and the point resampling can be found in Appendix A.



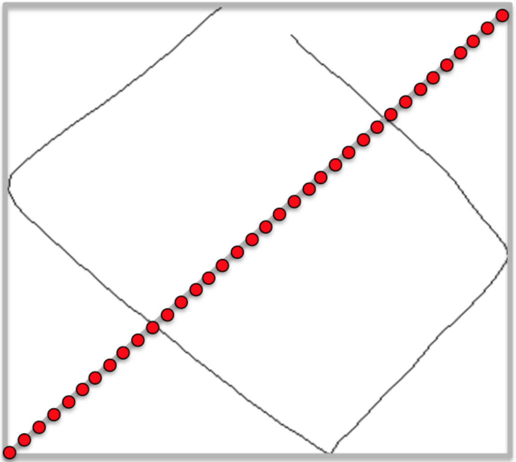
(a) A stroke is drawn by a user.



(b) The bounding box for the stroke is computed.



(c) The bounding box diagonal is computed.



(d) A constant number of points is fitted to the bounding box diagonal. The spacing between these points is then taken as the resample spacing for the entire stroke.

Fig. 40. An example demonstrating how the interspacing distance for the resampled points is calculated. Note that we fit 80 points to the diagonal in our implementation, but, for image clarity, we only fit 36 points to the diagonal in 40(d).

C. Corner Finding

ShortStraw finds corners using both a bottom-up and top-down approach. The bottom-up approach attempts to build corners from primitive information, whereas the top-down approach looks at higher-level patterns to determine possible insertion or deletion of corners.

1. Bottom-Up

ShortStraw finds corners in a stroke based on the length of the "straws". A straw for a point at p_i is computed as:

$$straw_i = |p_{i-W}, p_{i+W}| \tag{4.1}$$

where W is the constant size of the window and $|p_{i-W}, p_{i+W}|$ is the Euclidean distance between the points p_{i-W} and p_{i+W}. As a stroke starts to bend at a corner, the straws between points will begin to shorten, and the local minimum straw at point index k is a likely corner.

To find the initial corner set, all the straws are first computed for points p_W to $p_{|points|-W}$. The median straw size is then found and a threshold t is set to be equal to the $0.95 \times median$. For each $straw_k \in straws$, if $straw_k$ is a local minimum below the threshold t, then k is a corner. We set the window size $W = 3$. These numerical values were empirically determined to be the most effective at helping locate correct corners. An example of finding corners from straws is seen in Fig. 41.

From these equations, it follows that the straw length must remain relatively constant throughout the stroke in order for the correct corners to be found. Resampling the points of a stroke assures that our algorithm will have a static straw length for the majority of the stroke, whereas the straws of non-resampled points (such as

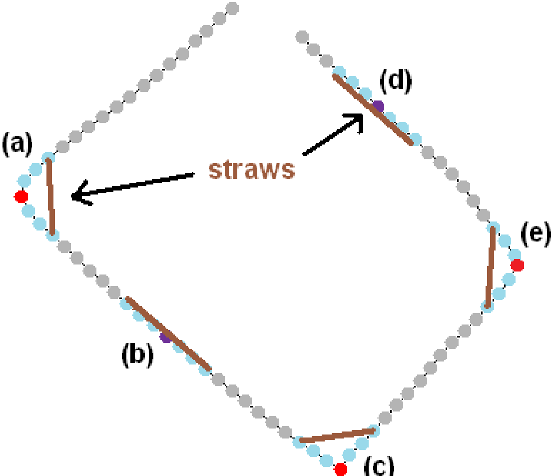


Fig. 41. An example of "straws" in a stroke. The points (a-e) all have a window of ±3 points. the distance at endpoints at these windows forms a straw, with the shortest straws being at points (a), (c), and (e). These points are considered corners. Points (b) and (d) have straws that are close to the median straw length, so these points are not initial corner candidates.

in Fig. 39(a)) would be highly variant.

2. Top-Down

After the initial set of corners is found by taking the shortest straws, some higher-level processing is run on the stroke to find missed corners and remove false positives.

ShortStraw first checks to see if each consecutive pair of corners passes a line test. Two points at indices a and b pass a line test if the chord distance and the path distance between the two points are relatively equal. We represent this equality through the ratio:

$$r = \frac{\textsc{Distance}(points, a, b)}{\textsc{Path-Distance}(points, a, b)} \tag{4.2}$$

where $0.0 \leq r \leq 1.0$, since the squared distance between the two points will never be greater than the squared path distance. If the ratio in Eqn. 4.2 is above a developer-set threshold, then the segment between the points at a and b is considered to be a line. This line test is the same one that we used in MergeCF. In our system, this threshold is set to 0.95 (See Appendix A for the functions to compute DISTANCE, PATH-DISTANCE, and the IS-LINE test).

If the stroke segment between any two consecutive corners c_m and c_n does not form a line, then there must be additional corners in-between c_m and c_n. Missing corners are assumed to be approximately halfway between the c_m and c_n. Since these potential corners are below the original threshold t, the threshold is relaxed and the new corner to add is taken to be the point with the minimum straw that is in the middle half of the stroke segment. This process of adding corners is repeated until all of the stroke segments between pairs of consecutive corners are lines.

A collinear check is then run on subsets of triplet, consecutive corners. If the three corners are collinear, then the middle corner is removed from the corner set. This process checks and removes false positives. Three consecutive corners c_l, c_m, and c_n are deemed collinear if the stroke segment between c_l and c_n passes an IS-LINE test.

Finally, we check for hooks near the endpoints of the stroke. If we find corners close to the endpoints, then we assume that they were the result of hook noise and remove them. We determine the distance at which we stop checking for hooks based on the equation:

$$hookThreshold = \min(\text{GET-DIAGONAL}(points) \times 0.10, 15) \tag{4.3}$$

In Eqn. 4.3, we use the bounding box diagonal of the stroke to determine the

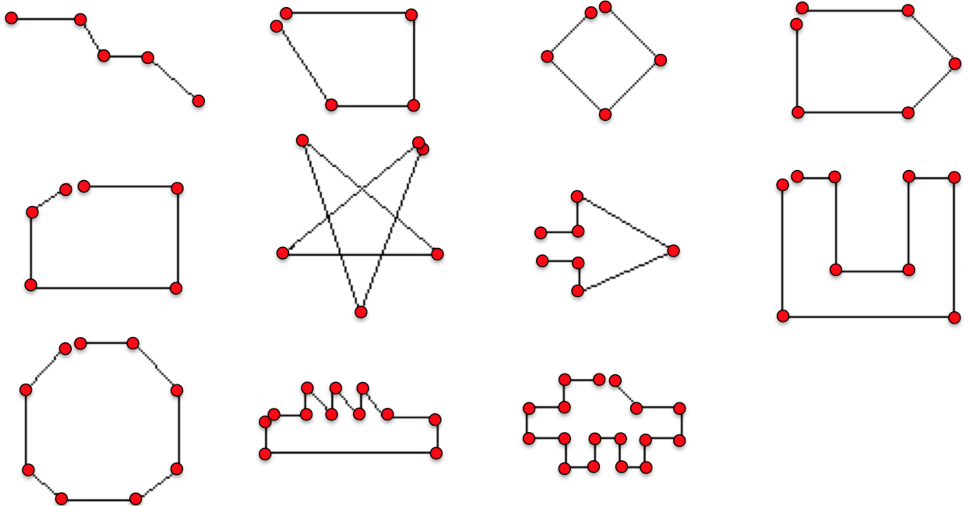


Fig. 42. The 11 polyline symbols used during corner finder testing. These symbols were drawn up to 4 times each by 6 different users, resulting in 244 polyline strokes.

relative scale and size of the stroke in question. If a stroke is very large, then it is possible that the user had a relatively large hook near the endpoints. We bound the hook threshold to be between $0 \leq hookThreshold \leq 15$ pixels so that we do not remove correct corners as the stroke's diagonal distance grows.

It is important to note that the final corners returned are from the resampled points. If a domain requires the original points of a stroke to be used, a developer implementing ShortStraw can map resampled corners to original points simply by taking each corner found and searching for the closest original point to that corner.

D. Results

To test ShortStraw, we used the polyline data from the data set collected for MergeCF (Fig. 33). These polyline symbols are reproduced in Fig. 42 and are based on the

symbols tested by Kim and Kim [45]. A single set of these 11 symbols contains 37 right, 16 obtuse, and 12 acute angles. This test set consists of 244 polyline strokes.

For direct polyline corner finder comparisons, we implemented Douglas-Peucker's algorithm using their "Method 2" algorithm [36], and we tested PaleoSketch's polyline segmenter [11].

We also tested an implementation of Sezgin *et al.*'s corner finder since it is a baseline for many sketch recognition algorithms [41], Kim and Kim's algorithm since our dataset was based off the images in their paper [45], and MergeCF for a comparison to our previous work.

Each of these baseline algorithms was implemented to provide the best accuracy possible. This required us to implement some functionality not mentioned in the original papers. All of the algorithms have filters to remove close or overlapping corners.

We used two the same two accuracy metrics, correct corners accuracy and all-or-nothing accuracy, described in Chapter III, Section 1. The results from our tests can be found in Table IV, and examples can be seen in Fig. 43[1].

E. Discussion

ShortStraw has a substantial improvement over our four baseline corner finders: Douglas and Peucker's, PaleoSketch's, Sezgin *et al.*'s, and Kim and Kim's. The all-or-

[1]Our results for ShortStraw here are different than the results originally presented in our SBIM 2008 paper [58]. Since the writing of the original ShortStraw paper, we have tweaked the ShortStraw threshold values, specifically the increase of the number of points on the resampled diagonal from 40 to 80. We also added the ability to check for hooks, or extraneous corners located at the end of a stroke. These tweaks improved segmentation accuracy on our newer, real-world datasets and translated well to our older SBIM test set. These changes were implemented locally before an improvement to ShortStraw, entitled iStraw [59] was either known or published (See Section F, Appendix B). The ShortStraw algorithm reflects these changes in Appendix A.

Table IV. Results for ShortStraw and our comparison corner finders. The results are for a set of 244 polyline shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs.

	ShortStraw	Douglas-Peucker	Paleo	Sezgin	Kim	MergeCF
False Positives	6	86	26	29	22	24
False Negatives	28	20	178	162	242	3
True Positives (Correct Corners)	1813	1821	1663	1679	1599	1838
True Negatives	56,638	56,558	56,618	56,615	56,622	56,620
Total Correct Corners	1841	1841	1841	1841	1841	1841
Accuracy	0.999	0.998	0.997	0.997	0.995	1.00
All-or-Nothing Accuracy	0.881	0.816	0.705	0.594	0.443	0.914
F-measure	0.991	0.972	0.942	0.946	0.923	0.992
Sensitivity (a.k.a. Recall)	0.985	0.989	0.903	0.912	0.868	0.998
Precision	0.997	0.955	0.985	0.983	0.986	0.987
FDR	0.003	0.045	0.015	0.017	0.014	0.013
Avg. time for all 244 strokes (in ms)	228	34	872	64	156	11,800
Avg. time per stroke (in ms)	0.934	0.139	3.57	0.262	0.639	48.4
All-or-nothing / Avg. time per stroke	0.943	5.07	0.197	2.27	0.693	0.0189

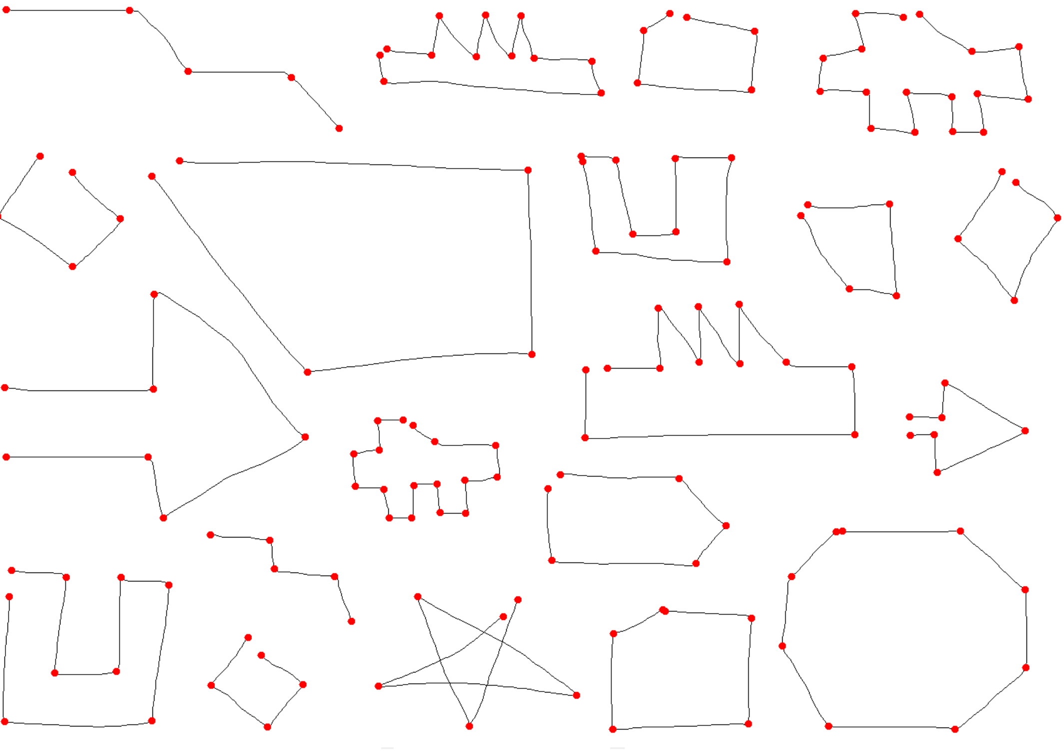


Fig. 43. Examples of correctly classified symbols by ShortStraw. These symbols come from the set of 244 polyline shapes drawn by six test users. The size ratio between the symbols has not been altered, although each symbol is similarly scaled so that the entire image will fit in the paper.

nothing accuracy for ShortStraw is over 15% better than that of the second-best baselines, our Douglas-Peucker implementation and PaleoSketch segmenter, over 25% better than Sezgin *et al.*'s algorithm, and almost twice that of our Kim and Kim's implementation. Furthermore, ShortStraw greatly improves upon the correct corners accuracy metric; our algorithm has less false positives and negatives than the other segmenters.

The only exception to ShortStraw's improvement is with MergeCF. Our MergeCF algorithm outperforms ShortStraw, but this performance comes at a cost.

1. Simplicity

MergeCF is a much more complicated algorithm than ShortStraw, combining curvature techniques from Sezgin *et al.* [41] and primitive recognition from PaleoSketch [11]. The PaleoSketch primitive recognizer uses multiple error fit techniques, such as least-squared regression and feature area [40]. Although some of these techniques could be culled from MergeCF if we tweak the algorithm to only work with polylines, the core algorithm is still more complex than Sezgin *et al.*'s algorithm, which many graduate students in the Fall 2007 Sketch Recognition class at Texas A&M had difficulty implementing.

The implementation of ShortStraw, on the other hand, is very simple, and we provide the entire algorithm in Appendix A. We had a sophomore undergraduate student unfamiliar with sketch recognition read our paper and code our algorithm. After completion, the student mentioned that the algorithm was "fairly easy to implement", and the entire time to read the paper, understand the algorithm, finish the implementation, and debug and test the code took the undergraduate took only 5-6 hours.

PaleoSketch's segmenter and the Douglas-Peucker algorithm are also relatively

simple to implement, but ShortStraw improves upon their accuracy.

2. Complexity and Time

ShortStraw has some other benefits that have not been previously mentioned. ShortStraw is not computationally intensive, so it can be easily used on mobile devices such as PDAs or touch-screen cell phones. A quick analysis of ShortStraw shows that resampling the points takes only $O(n)$ time and $O(n)$ memory. Calculating the straws for each point also runs in $O(n)$ time, as well as finding the initial corner fit. The only two sections of the algorithm that do not run in linear time include calculating the median straw length (which can run as quickly as $O(n \log n)$ with an efficient sorting algorithm), and the Post-Process-Corners function, which runs in time $O(cn)$ where c is the number of corners found in the stroke. In the very unlikely case that every stroke point is a corner ($c = n$) AND all of the corners were missed during initial processing (requiring each stroke point to be added as a corner via the Halfway-Corner function that searches for a corner under relaxed constraints), this function, and, thus, the entire algorithm, has a worst case scenario of $O(n^2)$ running time.

We again evaluated the approximate runtime of our various segmenters. Douglas-Peucker is the fastest segmenter, running at approximately 0.139 millisecond per stroke. ShortStraw runs at around two-thirds of a millisecond per stroke. The MergeCF and PaleoSketch segmentation algorithms are the the slowest, but they still runs in real-time.

With ShortStraw, we also have a decent All-or-nothing Accuracy / Avg. Time per Stroke metric at 1.42. This indicates that the segmentation performance we achieve with ShortStraw is not counterbalanced by a long run-time.

3. Potential Optimizations

To further reduce ShortStraw's computation time, the Euclidean distance measurement for calculating the straw length can be replaced with a squared distance measurement. This eliminates the need to perform over n square root calculations since the actual length of the straw is not important, only the straw's relation to the median straw length. We refrain from performing that step in the description of the algorithm to make the explanation easier to conceptualize for quick understanding and implementation. All additional distance calculations after the straws are computed, such as the path distance calculations in the Is-Line function, must then use the squared distance measurement as well to remain in the same scale as the straws.

The Path-Distance equation can also be optimized. Since the points are resampled, the path distance to any one point is equal to the resampled spacing multiplied by the point index. We discovered that with very long strokes this calculation can produce some unwanted noise, due to double precision and rounding issues.

These optimizations are not entirely necessary since the algorithm runs in real-time for our data set, but some sketch recognition domains and real-world applications might require very large stroke segmentation or large batch segmentation, and we want to show that ShortStraw is a robust baseline segmenter that can be fine-tuned for different scenarios.

4. Offline Possibilities

Another important aspect of ShortStraw is that the corner finding algorithm does not use any temporal information. Our corner finder could therefore be used in conjunction with systems that reconstruct strokes from static, offline images [60, 61], whereas the algorithm in [41] relies on speed information to locate corners.

In these offline cases, the ordering of the points in the stroke must be preserved in order for ShortStraw to function correctly. The work by Rajan, a previous member of the Texas A&M Sketch Recognition Lab, maintains stroke point ordering [61].

5. Relation to Curvature

Both Sezgin and Kim's corner finders are designed to work with complex fits as well as polylines, whereas ShortStraw is designed only for polylines. Our algorithm is not designed to work well with arc and curvy segments since the median straw length of strokes with high curvature vary widely.

Although ShortStraw does not explicitly use the word "curvature", each straw or chord length is in essence a simplified form of curvature. Instead of calculating curvature as the change in tangent across a series of points, a straw is a more naive representation for how bent a series of points are. If we were to redescribe our algorithm in terms of curvature, on a global scale we resample using a large number of points, and then we progressively "compute curvature" over an expanse of 7 points (our straws). The intuition behind the improvement gained from this algorithm compared to other algorithms is that we are able to effectively smooth the stroke to remove noise without the common problem of removing corner precision:

- **Smooths out noise:** Both resampling and computing straw lengths across 7 points cause the algorithm to be less susceptible to the pixelized noise commonly prevalent in stroke points.

- **Keeps corner precision:** Because the resampled stroke still contains a large number of points, and, because the system progressively computes the straw lengths by moving only one resampled stroke point at a time, the algorithm is able to keep the corner precision which is usually lost during stroke smoothing.

F. Extensions

ShortStraw has already had an impact in the sketch recognition community. At the University of Central Florida, Dr. LaViola had students code ShortStraw during a homework assignment in his Fall 2008 class on pen-based user interfaces [62]. During this class, one of Dr. LaViola's students, Yiyan Xiong, used ShortStraw as a base for a new algorithm, entitled iStraw. iStraw tweaked ShortStraw's thresholds and introduced the ability to handle curved segments in the segmenter [59]. With Xiong's additions, the accuracy for polyline segmentation climbed to almost perfect accuracy.

The results from iStraw are important because they (1) show that the results from the ShortStraw algorithm are reproducible, (2) show that ShortStraw is extensible, and (3) show that the original curvature metrics in ShortStraw work well across many different users, data sets, and hardware.

Although iStraw performs better than ShortStraw, the algorithm itself is much more complicated than ShortStraw's. This sacrifices implementation speed and simplicity for higher segmentation accuracy; depending on the situation, either option can be preferred. A larger discussion of iStraw can be found in Appendix B.

G. Contributions

ShortStraw provides numerous benefits and improvements over other polyline corner finders.

- ShortStraw is more accurate than most current segmenters when analyzing polyline strokes.

- ShortStraw runs in real-time, and the segmenter is faster than MergeCF, which is the only corner finder we compared to that has a better all-or-nothing accu-

racy than ShortStraw on polyline data.

- ShortStraw is easy to code, and the entire algorithm is provided in Appendix A.

H. Limitations and Future Work

Finally, we wanted to highlight the various limitations of the ShortStraw algorithm. The primary hindrance to ShortStraw's accuracy is the reliance on global thresholds. The main thresholds in our system are the use of 80 points for calculating the resampled bounding box diagonal, the IS-LINE test threshold of 0.95, and the straw size threshold of $0.95 \times median$.

In our published paper on ShortStraw [58], the resampled bounding box threshold was set to 40 points. Since the writing of that paper, our lab has worked on a real-world application of sketch recognition with course of action diagrams. We tweaked our resampling threshold to be at 80 points. This new threshold better accounts for the course of action data (Fig. 38), and, in combination with our hook detection addition, it resulted in a better all-or-nothing accuracy metric on our official testing set. These results demonstrated that the best thresholds for ShortStraw are dependent on the training set and domain we use. Researchers implementing ShortStraw should therefore tweak this threshold to work with the dataset they wish to segment.

Another issue with the way we choose our resampling threshold is that we are assuming that drawn strokes do not vary heavily in scale. ShortStraw's resampling does not work well when we have strokes where the length of the stroke segments fluctuates wildly, such as the stroke in Fig. 44. In this case, the stroke's larger scale causes the resampled points to be spread out, which produces issues when ShortStraw

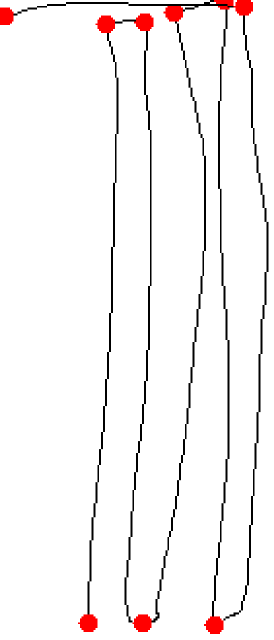


Fig. 44. The resampled points in this stroke are too far apart to accurately find the correct corners in the small horizontal segments at the bottom of this stroke.

tries to find the corners for the small horizontal segments. We could always set the resample spacing of a stroke to be a small constant, such as forcing points to be 1 pixel apart, but then we find many additional corners due to noise.

To counteract global thresholds, we tried to account for local changes and outliers by using a fraction of the median straw length. Although the median value is locally determined, the value 0.95 is still constant across all data and domains. Again, we do not claim that 0.95 is the best general threshold across all domains; we simply found that 0.95 was a sufficient threshold for ours.

ShortStraw's window of ± 3 points is one more global threshold that could possibly be eradicated. In Teh and Chin's corner finder [63], they vary the window for

each point examined during corner finding calculations. Although having a scaling window can increase the accuracy for finding points that are corners, ShortStraw was designed to be as simple as possible while still providing high polyline accuracy.

Thresholding issues are prevalent across all corner finders. No one segmenter has golden thresholds that work for every dataset, but ShortStraw's thresholds are easy for developers to fine-tune and tweak. Nonetheless, after creating both MergeCF and ShortStraw, we realized that no single solution will accurately segment every stroke. This conclusion led us to our next work: combining results from multiple segmenters.

CHAPTER V

COMBINING CORNER FINDERS

A. Motivation

Both of our previous algorithms, MergeCF and ShortStraw, relied on empirically found thresholds that worked well for our real-world data, such as course of action diagrams, and our test set. Empirically found thresholds are common in segmentation algorithms [11, 36, 40, 41, 42, 45], but, after building two corner finders that required some manual tweaking of threshold values, we wanted to move toward completely trainable segmentation algorithms.

During our research of previous corner finding algorithms, we noticed that most segmenters employ a single, specific technique. These algorithms work well for most cases, but each has a notable weakness in detecting certain corners. For instance, some polyline corner finders employ a linear search along a stroke to find points that deviate heavily from the direction of the current stroke direction [11, 31, 32]. These types of corner finders work well for strokes that contain sharp, acute angle changes, but more obtuse direction changes are harder to detect. Polyline corner finders that use local curvature values, such as [58], also suffer from this obtuse angle issue. Other polyline corner finders use simple trigonometry techniques to recursively detect points that deviate the most from the current polyline representation [36, 37]. These techniques work well for non-intersecting strokes, but intersecting strokes can cause some false positives to be found.

More complex corner finders try to distinguish between multiple primitives such as lines, arcs, and curves. The main techniques for detecting the corners of multiple-primitive strokes are to use curvature values at points [40, 41, 42, 45, 55] and finding

points of low pen speed [41, 42, 55]. Noise is the main issue of these corner finders; local or global thresholds for curvature and speed corner choosing are highly susceptible to outliers.

After analyzing MergeCF and ShortStraw, we realized that no one method will be a silver bullet that would preform best in all cases. In fact, Wolpert's work in No Free Lunch theorems state that if an optimization algorithm performs better than average in a certain class of problems, then it will perform worse in another class of problems [64, 65].

We have seen no method to combine multiple corner finder techniques. The closest algorithm is Sezgin *et al.*'s algorithm that picks the "best" corners found from the speed and curvature of the stroke [41]. Points of slow speed are considered to be corners since users slow down when changing direction; likewise, points of high curvature are considered corners. The algorithm ranks each speed and curvature corner by a metric and then greedily picks the next best corner. This is in essence a sequential forward search algorithm for feature subset selection where the corners are features. This technique often introduces errors into the final segmentation due to the choice of objective function (ranking speed and curvature points individually) and the inability to backtrack. Our approach extends using subset selection techniques in segmentation by both improving the objective function using a global mean-squared error criteria and allowing for both forward and backward searching.

There are many feature subset selection techniques, the most basic of which are forward and backward searches [66]. These searches greedily add or remove the best or worst features, respectively. Better results can be obtained by allowing both forward and backward searching, such as by using dynamic programming techniques [67], beam searches, or branch-and-bound algorithms [68]. We use a sequential floating backward selection (SFBS) algorithm to utilize both forward and backward searching,

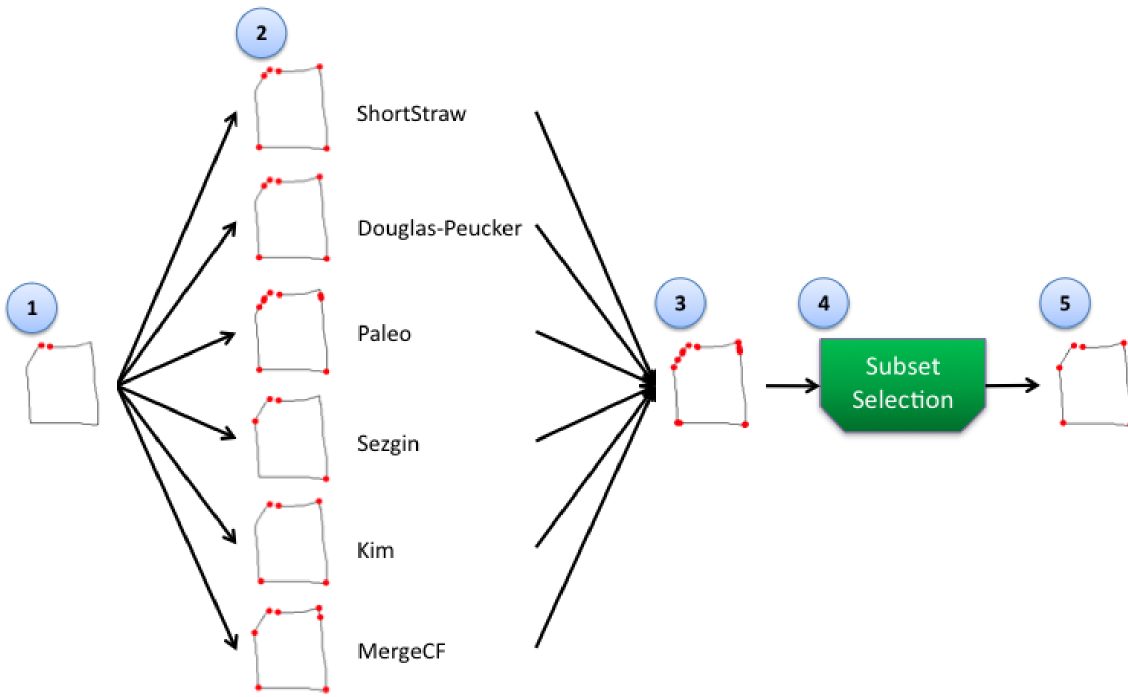


Fig. 45. Corner Subset Selection Process: (1) Take an input stroke, (2) segment the stroke using six different techniques, (3) combine the corners from all the techniques into one set, (4) pass the combined corner pool to our subset selection algorithm, and (5) output the best subset found. None of the original segmentations are correct, but the final subset has the correct 6 corners.

and, since we should not have hundreds or thousands of corners per stroke, we do not need to use more bounded approximation algorithms.

B. Implementation

We created our corner combination algorithm to segment polyline stroke data. These polyline segmentation techniques can then be used to build multiple-primitive recognizers, such as in PaleoSketch [11].

To combine corners from multiple segmenters we use a feature subset selection

algorithm where the features are the corners; we coined this technique "corner subset selection". A mean-squared error objective function is used in the algorithm. The overall process is outlined in Fig. 45, and the following sections describe the steps in detail.

1. Step 1: Segmenters Used

Our algorithm starts by taking all of the corners found from five segmentation algorithms: ShortStraw [58], Douglas-Peucker [36], PaleoSketch's segmentation algorithm [11], Sezgin *et al.*'s [41], Kim and Kim's [45], and MergeCF. The first three segmenters (Douglas-Peucker, ShortStraw, and PaleoSketch's) are polyline corner finders that rely on simplified line tests to determine if a segment between two corners is a line. These finders are often susceptible to missing corners at obtuse angles and finding extraneous corners at segments that have noisy "bumps" (See Fig. 45 segmentations).

The other three segmenters are multiple-primitive segmenters that try to split strokes into lines and arcs. Sezgin *et al.*'s use of speed helps find subtle corners where the user slowed down their drawing. The local curvature values in Kim and Kim's algorithm can often find corners that the other, global-threshold algorithms have missed.

The results from all six segmentation algorithms are combined together, and duplicate corners are removed.

2. Step 2: Subset Selection

Feature subset selection is a technique used for dimensionality reduction in pattern classification problems. Pattern classification often uses data that was gathered in high-dimension feature-spaces, where each feature contributes one dimension to the space. Transforming these spaces into lower dimensions is a key component of pattern

classification research, since using fewer dimensions can allow classification algorithms to train and run in less time.

Feature subset selection techniques find the most significant dimensions of a feature-space, allowing researchers to use fewer dimensions while producing comparable classification accuracies. In a sequential forward selection (SFS) algorithm, the subset of features, F_S, we will select starts empty. Features are greedily added from the entire set of features, F, to this subset, one at a time, based on an objective function that measures the performance of the system on a set of training data. The objective function calculates if adding a feature $f_i \in F$ to F_S will improve the system's performance. The feature that improves the system's performance the most is added to our subset, and the process continues until no features remain. During this procedure, the algorithm stores a copy of every different subset we create and the performance measure of that subset. The final subset of features in is then determined based on the feature subset that maximizes the system's performance and minimizes the number of features.

The key component to feature subset selection is that it is a greedy algorithm. This makes the algorithm less accurate than a dynamic programming approach, but it also allows dimensionality reduction to occur in real-time.

There are other approaches to subset selection. In our implementation, we use a sequential floating backwards selection (SFBS) technique that starts with the entire set of features ($F_S = F$) and greedily removes the feature in F_S that contributes the least to the system's performance. At each step, we can also add a previously removed feature back into F_S if the performance of the system will increase; bookkeeping techniques prevent the constant removal and addition of the same feature. The ability to "float" and reintroduce removed features helps alleviate some issues caused by greedy selection. Again, every subset and its corresponding performance is recorded,

and the subset that maximizes the performance while minimizing the number of features is chosen.

In our subset selection step, corners themselves are the features to select a subset from. In essence, the corners of a polyline stroke are a feature-space that describe the polyline stroke, and we want to reduce the number of corners so that we have maximize the polyline description while minimizing the error of the polyline fit.

To determine which corner to remove, the corner subset selection algorithm uses an objective function that looks at the mean-squared error (MSE) between the actual stroke segments and the optimal polyline created through linking consecutive corners.

The mean-squared error of a segment is computed as the average difference between every closest vertical pair of points in the original stroke and optimal polyline, squared (Eqn. 5.1, Fig. 46, 47, and 48). In the MSE equation, p_i represents a point in the original stroke at index i, opt_i is the closest vertical point on the optimal polyline, and N is the number of original points.[1]

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(p_i - opt_i)^2 \tag{5.1}$$

The corner that affects the mean-squared error the least is then removed from the current subset. A copy of the subset is stored for future reference, and the process continues on the remaining corners. The endpoints of the stroke are omitted from consideration.

At each step the corner subset selection algorithm also determines if adding a previously removed corner back into the system will be better than removing another corner. If the mean-squared error for the system is reduced when adding a corner

[1]We used the term "performance" when discussing objective functions in the Feature Subset Selection Overview. Error is the other side of the same metric. Overall, we want to maximize performance and minimize error.

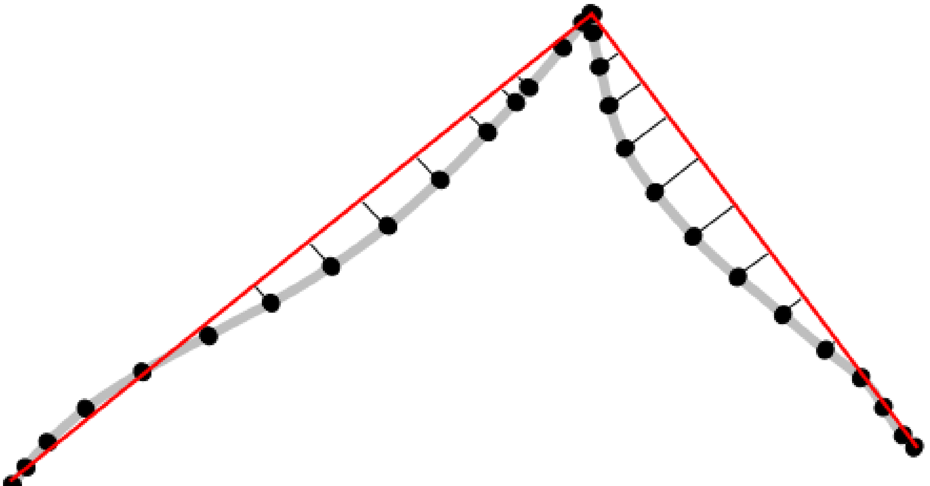


Fig. 46. This figure shows an example of the error between the original stroke (gray stroke, black points), and a representation based on a stroke's corners (red lines). To calculate the mean-squared error, the distances (black lines) between the original points and the optimal polyline are squared, summed, and then averaged.

back to the system, then the corner is replaced. It's important to note that this step occurs often due to the nature that oversegmented strokes tend to have a lower mean-squared error than strokes with fewer segments.

The algorithm terminates once the only two corners remaining are the endpoints of the stroke. The best subset occurs at the "elbow" of the mean-squared errors, where the mean-squared error for removing a point suddenly jumps (Fig. 47 and 48). Because we want our algorithm to handle strokes at different scales, and because strokes with at larger scales typically have higher mean-squared errors than strokes at smaller scales, we normalize the subset data by looking at the change in mean-squared error, ΔMSE, instead of the error itself. We first find the ΔMSE between the subset with $i+1$ corners and the subset with i corners (Eqn. 5.2). This change

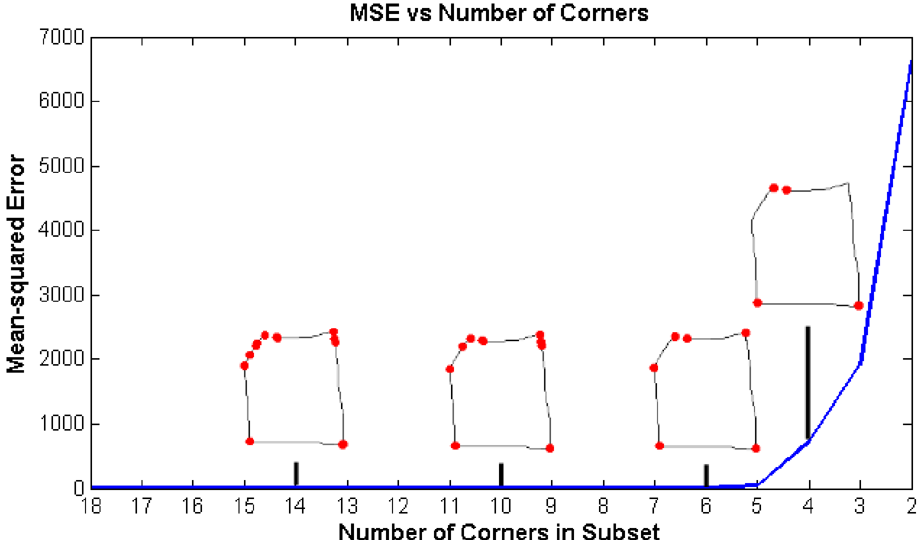


Fig. 47. Mean-squared error (MSE) of the stroke in Fig. 45. As corners are removed, the MSE has little change until critical corners are removed. In this example, the correct number of corners is 6, so critical corners are removed starting at $i = 5$. The segmentations at $i = 14, 10, 6$, and 4 are shown here to illustrate how the subsets change as the number of corners in a subset decreases.

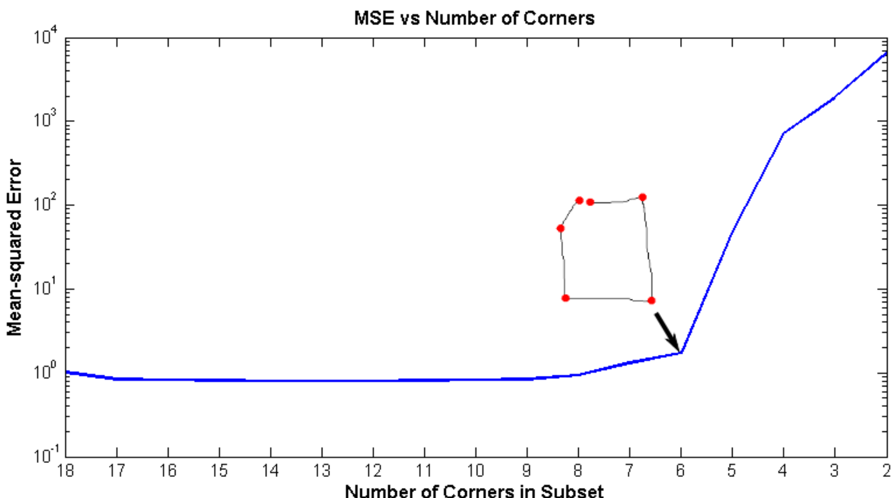


Fig. 48. This is the same data from Fig. 47, but with a log scale for the MSE.

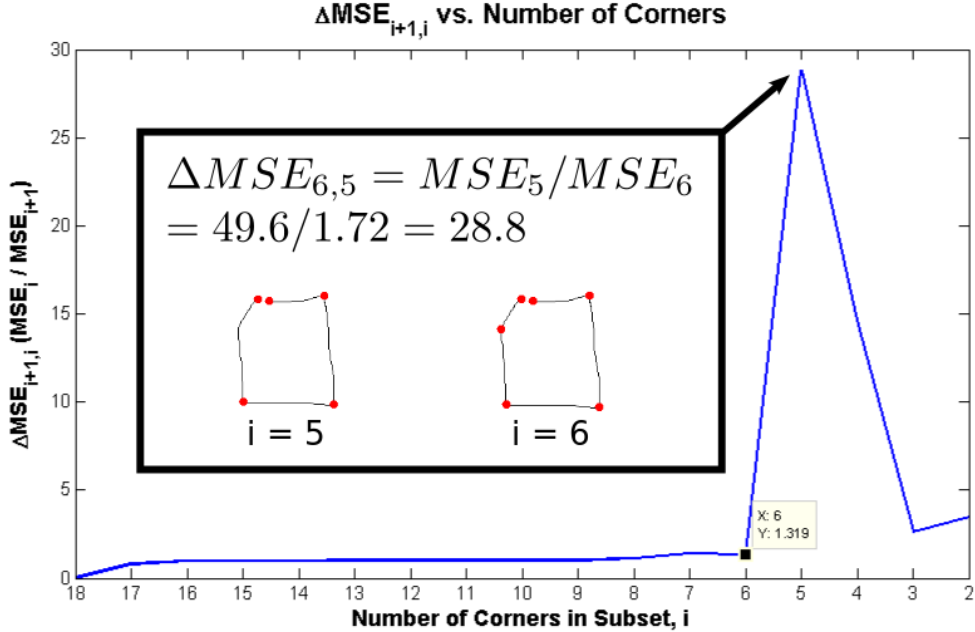


Fig. 49. ΔMSE described in Eqn. 5.2. This chart is for the stroke in Fig. 45, whose MSE plot is shown in Fig. 47. ΔMSE is essentially a derivative of the mean-squared error, which deviates only slightly until a critical corner is removed at $i = 5$. The ΔMSE from $i = 6$ to $i = 5$ is calculated to be 28.8.

in error is calculated for all $i = C, C-1, C-2, ..., 3$, where C is the total number of combined corners that we started with. We stop at $i = 2$ since the final two corners are endpoints and will never be removed.

$$\Delta MSE_{i+1,i} = \frac{MSE_i}{MSE_{i+1}} \tag{5.2}$$

Initially, the mean-squared error remains almost constant as erroneous corners are removed, so $\Delta MSE_{i+1,i}$ is close to 1.0. When a crucial corner is removed from the subset, ΔMSE should jump significantly (Fig. 49). Therefore, we found a threshold $t_{\Delta MSE}$ where the first instance of $\Delta MSE > t_{\Delta MSE}$ would indicate that we are severely

affecting the mean-squared error of the system and have already found the best subset.

3. Step 3: Training and Testing

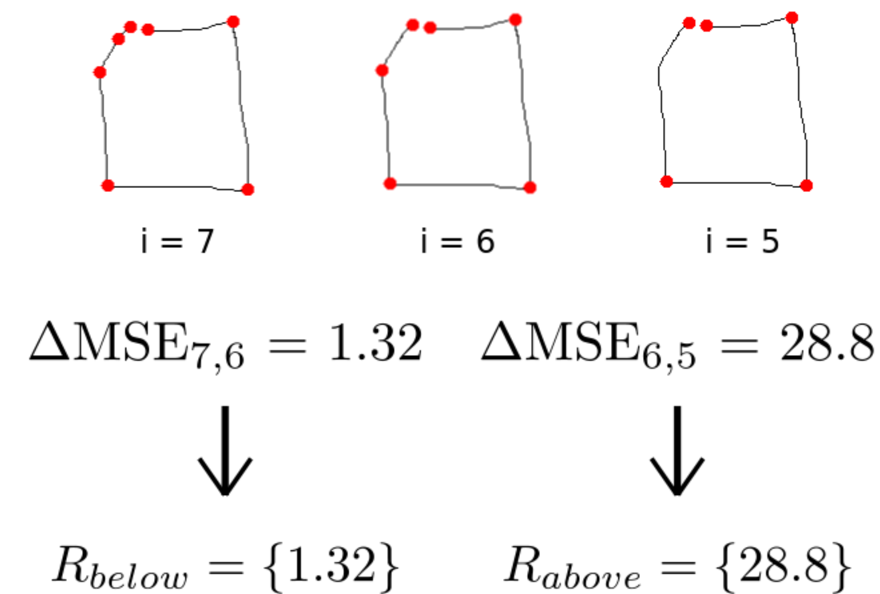
The correct number of line segments in each stroke is known during the training process. For each shape, after all the subsets are found during the SFBS process, the ΔMSE is calculated for the change in error from the first oversegmented subset to the correct subset, and from the correct subset to the first undersegmented subset. For example, if the correct number of corners to segment a shape into equals n, then $n+1$ is the first oversegmented subset, and $n-1$ is the first undersegmented subset. Each training shape's $\Delta MSE_{n+1,n}$ and $\Delta MSE_{n,n-1}$ values are stored during the training process in separate collections, R_{below} and R_{above}, respectively. These collections indicate that the ΔMSE value's are either below than a possible threshold value or above a possible threshold value (Fig. 50).

The median of each collection is found, and then the median absolute deviation (MAD) is computed (Eqn. 5.3). These median values are then substituted for the mean and standard deviation, respectively, when computing a regular Gaussian distribution for R_{below} and R_{above}.[2]

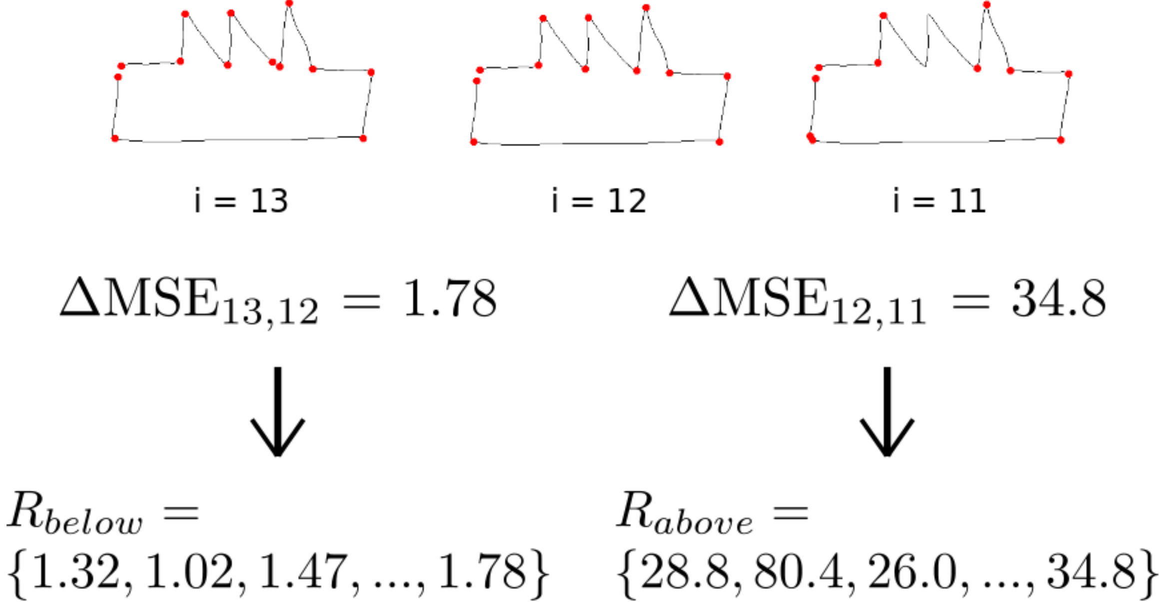
$$MAD = median(|R_i - median(R)|) \tag{5.3}$$

In Eqn. 5.3, R is the set of data (in our case, the ΔMSEs), and R_i is one value in R. The MAD is then the median of every $R_i \in R$ differenced with the median of

[2]We originally used the mean and standard deviation of R_{below} and R_{above} to compute Gaussian distributions, but we found large fluctuations in these values based on the data chosen for training and testing. If we trained using k-fold cross validation, this corresponded to large differences in thresholds between folds (such as some thresholds being orders of magnitude larger than others) and eventually led to inaccurate training. Using the median and median absolute deviation helped stabilize the trained thresholds and produced reliable results.



(a) The first training example's ΔMSE's below and above a correct segmentation are calculated and stored in their respective collections.



(b) Training continues, storing every ΔMSE around a correct number of corners into their respective collections. Note that $i = 11$ has removed two correct corners and added one incorrect corners, due to this segmentation having a lower MSE than if the algorithm only removed one correct corner. We allow this by using SFBS instead of a one-directional searching algorithm

Fig. 50. An example of how R_{below} and R_{above} are generated during training.

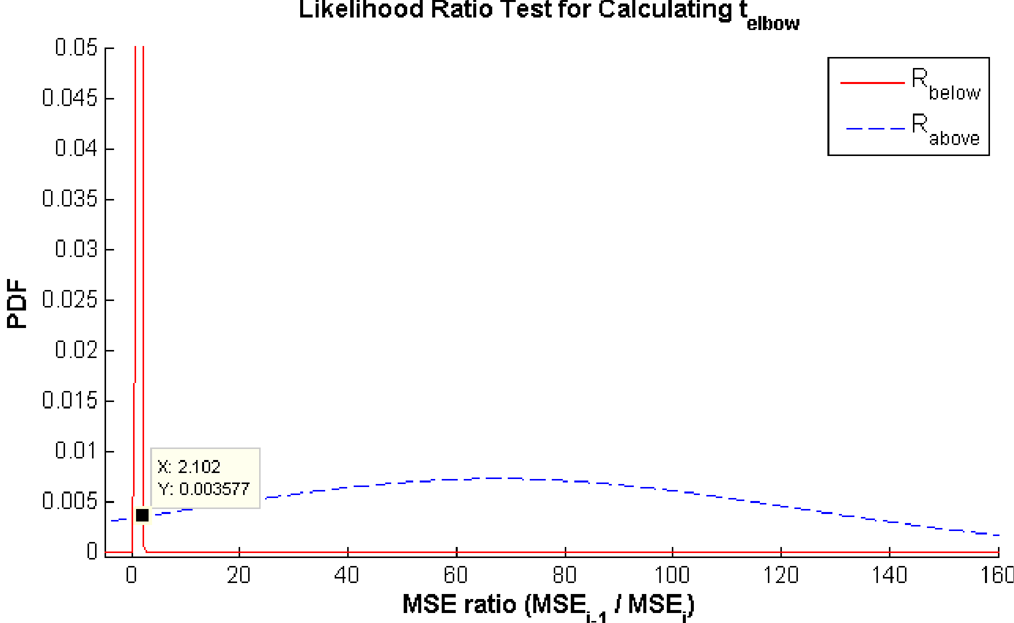


Fig. 51. Two Gaussian distributions are created from the R_{below} and R_{above} ratios for each set of training data. The optimal threshold is then found to be at the intersection of these two Gaussians; in this case, the threshold would be $t_{\Delta MSE} = 2.102$. Note that R_{below} is a much narrower Gaussian distribution than R_{above}'s, and the probability density for R_{below} goes to approximately 1.6. We chose a smaller y-axis in order to highlight the intersection of R_{below} and R_{above}.

R itself. This is similar to how standard deviations are calculated, but with medians instead of means.

The threshold, $t_{\Delta MSE}$ is determined to be at the point where the Gaussian probability densities for the two ΔMSE distributions are equal (Fig. 51). This process is equivalent to a likelihood ratio test that finds the best decision boundary minimizing the Bayes risk between two choices.

Intuitively, we can classify every subset with a ΔMSE below the threshold as

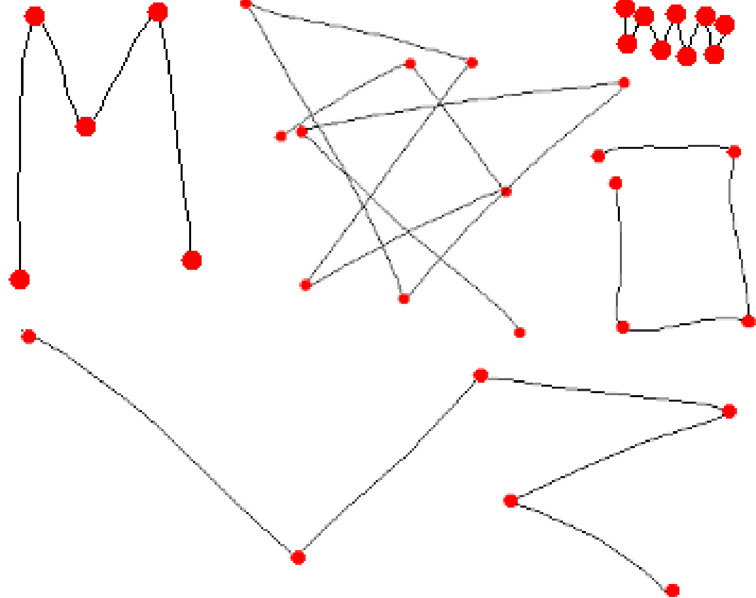


Fig. 52. A subset of the 216 random polyline shapes used for training. The polylines ranged from 2-line to 10-line shapes. The only drawing constraints were the number of line segments in each polyline and that the shape must be drawn with one stroke. Some users drew common symbols ('M' and square), others drew common patterns (zigzag), and a few drew random patterns of lines.

oversegmented, and all subsets after r has jumped above $t_{\Delta MSE}$ to be undersegmented. The subset before $\Delta MSE > t_{\Delta MSE}$ is the best subset of corners that perceptually segments the stroke into polylines.

To calculate the threshold, $t_{\Delta MSE}$, we train our corner subset selection algorithm on a set of 216 polyline strokes. The strokes were randomly drawn by 6 different users and range in difficulty from having only 2-line polylines to having 10-line polylines (Fig. 52). After training, we found the ΔMSE threshold, $t_{\Delta MSE} = 1.99$.

4. Algorithm Summary

Below are algorithm summaries for segmenting a single stroke and for the training of the $t_{\Delta MSE}$ threshold.

a. Single-stroke Segmentation

1. Calculate corners from multiple segmentation algorithms.
2. Merge all corners from the segmenters into one set.
3. Pass the full set of corners into the SFBS algorithm.
   (a) Calculate the MSE for the current set of corners. Store this value in an array, MSE. Store the subset in a list, *subset*.
   (b) Remove the corner that affects the MSE the least.
   (c) Check if adding a previously removed corner will reduce the MSE.
      i. If so, add the previous corner back into the set of corners.
      ii. If not, continue removing corners.
4. Calculate the $\Delta MSE_{i+1,i}$ for each MSE_i/MSE_{i+1}.
5. Find the first ΔMSE that is above $t_{\Delta MSE}$. Return the corresponding $s \in subset$ that corresponds to this ΔMSE. This is the set of corners used to segment the stroke. If no ΔMSE is above $t_{\Delta MSE}$, return the first subset we found, $subset_0$.

b. Training Algorithm

1. For a set of known training data, run the CSS process (above algorithm) on each stroke.

(a) Since we know the number of segments in each piece of training data, we know the MSE at which the stroke should be segmented correctly. For instance, suppose the correct number of corners in a segmentation is n. Then the ΔMSE at which a large increase is seen should be at MSE_{n-1}/MSE_n.

(b) Store MSE_n/MSE_{n+1} in R_{below}

(c) Store MSE_{n-1}/MSE_n in R_{above}

2. Calculate the median absolute deviation (MAD) of R_{below} and R_{above}. Use the median and distributions as Gaussians.

3. Find the ΔMSE value at which the two distributions for R_{below} and R_{above} intersect (i.e., a Bayes likelihood test). This value is the threshold, $t_{\Delta MSE}$.

Note that the training algorithm can be run using any pattern recognition training techniques, such as training on a separate set of data, using k-fold cross validation, leave-one out, or bootstrapping.

C. Results

Our algorithm was tested on a set of the same set of 244 polyline strokes that we tested MergeCF and ShortStraw on (Fig. 53). This testing set is different than our training set.

The results for our corner subset selection algorithm compared to the five individual algorithms are organized in Table V. Each of the five baseline algorithms were implemented by ourselves, so the resulting accuracies may not match those of the original papers. The ground-truth segmentation was determined by human recognizers, where a correct segmentation is determined to be perceptually correct. Because

Table V. Results for our corner subset selection algorithm (CSS) and the six original finders we used. The results are for a set of 244 polyline shapes drawn by six different users. The average times, in milliseconds, were found by averaging over 20 runs.

	CSS	ShortStraw	Douglas-Peucker	Paleo	Sezgin	Kim	MergeCF
False Positives	22	6	86	26	29	22	24
False Negatives	0	28	20	178	162	242	3
True Positives (Correct Corners)	1841	1813	1821	1663	1679	1599	1838
True Negatives	56,625	56,638	56,558	56,618	56,615	56,622	56,620
Total Correct Corners	1841	1841	1841	1841	1841	1841	1841
Accuracy	1.00	0.999	0.998	0.997	0.997	0.995	1.00
All-or-Nothing Accuracy	0.922	0.881	0.816	0.705	0.594	0.443	0.914
F-measure	0.994	0.991	0.972	0.942	0.946	0.923	0.992
Sensitivity (a.k.a. Recall)	1.00	0.985	0.989	0.903	0.912	0.868	0.998
Precision	0.988	0.997	0.955	0.985	0.983	0.986	0.987
FDR	0.012	0.003	0.045	0.015	0.017	0.014	0.013
Avg. time for all 244 strokes (in ms)	15,500	228	34	872	64	156	11,800
Avg. time per stroke (in ms)	63.5	0.934	0.139	3.57	0.262	0.639	48.4
All-or-nothing / Avg. time per stroke	0.0146	0.943	5.07	0.197	2.27	0.693	0.0189

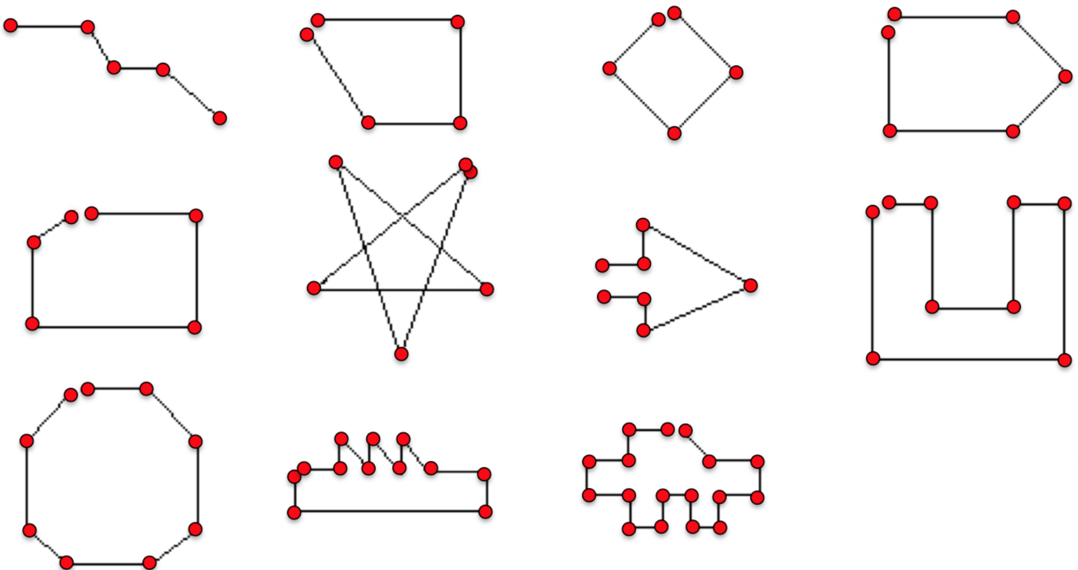


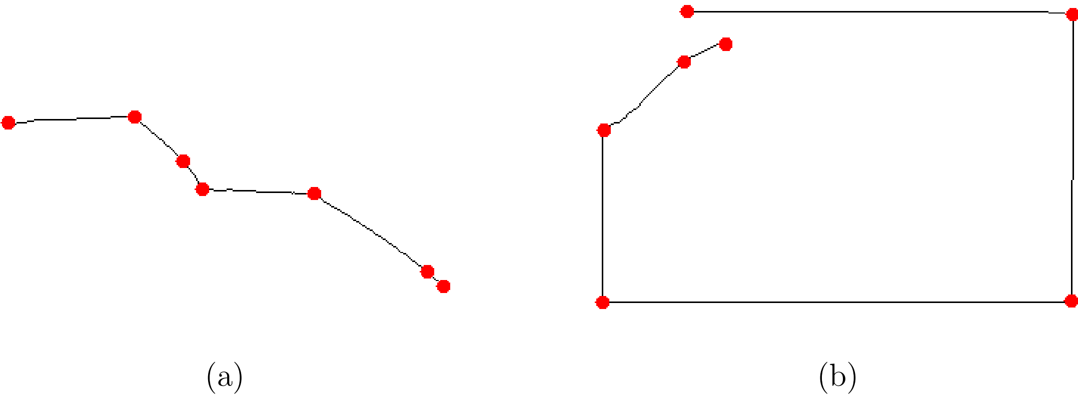
Fig. 53. The 11 polyline symbols used during corner finder testing. These symbols were drawn up to 4 times each by 6 different users, resulting in 244 polyline strokes.

the data we were working with consisted of polylines, the segmentations were fairly obvious. For any segmentations that were not obvious, we had more than one person outside of our authors examine the data and provide their input as to whether the segmentation was correct.

We use the same accuracy metrics for analyzing our combination algorithm: correct corner accuracy (with and without endpoints) and all-or-nothing accuracy.

D. Discussion

Our corner subset selection algorithm performs better than any of the individual algorithms in most accounts. The combination algorithm finds less false negatives, more correct corners and, most importantly, has a higher all-or-nothing accuracy than



(a) (b)

Fig. 54. Issues with thresholding in our corner subset selection algorithm. In both of these cases, the mean-squared error of the system would rise considerably (i.e., above our found $t_{\Delta MSE}$) if any corner was removed.

any of the six combined segmenters.

Overall, the corner subset algorithm succeeds at combining the hard-to-find corners of the other segmenters, picking those that only contribute the most to the global mean-squared error reduction of the optimal fit. One example is seen in Fig. 45, where no one technique finds the correct all-or-nothing fit, with some fits finding more corners than necessary and others finding too few.

1. Thresholding

The largest errors seen with our corner subset selection algorithm are false positives in the final segmentation. These are due to mean-squared ratio threshold errors (Fig. 54).

This issue with thresholding errors is an important limitation of our combination algorithm. The No Free Lunch theorems we mentioned during our motivation for creating CSS state that if an algorithm performs better on one set of test cases, it

will perform worse on others [64, 65]. Although we used this as a motivation for combining the corners from different algorithms, the fact is that any combination or ensemble algorithm also falls under this theorem's grasp. Merging the results from different algorithms will not automatically eliminate thresholding issues, and it can introduce new ones. But, approaching segmentation from the subset selection angle will be beneficial to segmentation performance.

Another issue with the corner subset selection algorithm is that all of the correct corners must be found by the original techniques. Our segmenter does not find any additional corners, so any false negatives that are present in all six original segments will automatically be false negatives in our corner subset selection algorithm.

2. Complexity and Time

Our CSS algorithm is by far the slowest of the segmentation algorithms we run, and it takes an average of 63.5 ms to segment a stroke. The large runtime is due to the algorithm computing segmentations from all 6 of our comparison segmenters before the subset selection process can even begin.

The CSS algorithm does run in real-time for each stroke, since we can still segment a stroke before a human could perceive a visual lag.

3. Significance

The all-or-nothing accuracy results between CSS and MergeCF are not statistically significant ($\chi^2 = 0.208, p > 0.5$). But, when we eliminate MergeCF from the CSS algorithm, we can retrain the system and do find significant results (Table VI). The $t_{\Delta MSE}$ for CSS without MergeCF did not change to three significant figures, so $t_{\Delta MSE} = 1.99$.

In this case, our all-or-nothing accuracy of 0.926 for CSS without MergeCF is

statistically significant from the next best, ShortStraw's 0.881 ($\chi^2 = 4.73, p < 0.05$). The other benefit from not using MergeCF is that the average time per stroke decreases from 63.5 ms to 15.2 ms. Thus, CSS without MergeCF has better performance than MergeCF while also taking less time to segment each stroke.

These results indicate that there might be an optimal set of segmenters to use with the CSS algorithm. The subset selection technique will only work well when there are no false negatives in the initial, pooled set of corners from other segmenters. If we can have a set of segmenters that complement each other well and do not have many false negatives, then we can reduce the time it takes to segment a stroke using CSS while retaining all of the algorithm's accuracy benefits.

E. Gaining Intuition: Why Do We Need to Run Existing Segmentation Algorithms?

Our CSS algorithm starts with a selection of corners gathered by existing corner finders. One question that has been asked is: Why not just start with all points and perform CSS on that, thus eliminating the need to run on multiple corner finders?

We tried this very technique, and the results from this implementation are shown in Table VII under the "CSS with All Points" column. After training, we found our $t_{\Delta MSE}$ for this system to be 2.718. Our final all-or-nothing accuracy was 0.0, and the number of false positives we found was enormous and on the average of 107 false positives per stroke. Also, rather than causing a speedup (by preventing the need to call multiple segmentation algorithms), using every point actually dramatically slows down segmentation from real-time to approximately 4 seconds a stroke. This shows that our method of calling multiple recognizers is both more accurate and faster than applying it to all points.

When we analyzed the data to determine why this happened, we realized that

Table VI. Results for our corner subset selection algorithm (CSS) without using MergeCF as an original corner finder. We find these results to have comparable performance to CSS with MergeCF, with the added benefit of less segmentation time per stroke.

	CSS (w/o MergeCF)	ShortStraw	Douglas-Peucker	Paleo	Sezgin	Kim
False Positives	21	6	86	26	29	22
False Negatives	0	28	20	178	162	242
True Positives (Correct Corners)	1841	1813	1821	1663	1679	1599
True Negatives	56,625	56,638	56,558	56,618	56,615	56,622
Total Correct Corners	1841	1841	1841	1841	1841	1841
Accuracy	1.00	0.999	0.998	0.997	0.997	0.995
All-or-Nothing Accuracy	0.926	0.881	0.816	0.705	0.594	0.443
F-measure	0.994	0.991	0.972	0.942	0.946	0.923
Sensitivity (a.k.a. Recall)	1.00	0.985	0.989	0.903	0.912	0.868
Precision	0.989	0.997	0.955	0.985	0.983	0.986
FDR	0.011	0.003	0.045	0.015	0.017	0.014
Avg. time for all 244 strokes (in ms)	3710	228	34	872	64	156
Avg. time per stroke (in ms)	15.2	0.934	0.139	3.57	0.262	0.639
All-or-nothing / Avg. time per stroke	0.0609	0.943	5.07	0.197	2.27	0.693

Table VII. Results for modifications of the corner subset selection algorithm. Here we have our original version (CSS), our subset selection technique applied to all points, and our subset selection technique applied to an oversegmented set.

	CSS	CSS with All Points	CSS with Oversegmentation
False Positives	22	26040	43
False Negatives	0	0	1
True Positives (Correct Corners)	1841	1841	1840
True Negatives	56,625	30,604	56,601
Total Correct Corners	1841	1841	1841
Accuracy	1.00	0.555	0.999
All-or-Nothing Accuracy	0.922	0.00	0.857
Avg. time for all 244 strokes (in ms)	15,550	965,000	69,800
Avg. time per stroke (in ms)	63.5	3950	286

using all of the points initially causes the mean-squared error of the stroke to be 0.0, since each initial segment in the stroke will be composed of two consecutive points. Therefore, removing possible corners from the stroke segmentation causes large spikes in the mean-squared error ratio, and our training process cannot find a good, stable threshold.

A subtle modification to the "all points" approach would be to use one corner finding technique to heavily oversegment the stroke. This oversegmentation would have significantly fewer points to send to our subset selection algorithm, but it should still have a high probability of containing the correct corners. To do this, we found an initial set of corners using a modification of ShortStraw where we loosen the thresholds to produce more false positives. The results from this oversegmentation can be seen in Table VII under the "CSS with Oversegmentation" column.

The CSS with Oversegmentation algorithm performs better than CSS with All Points, but it performs worse than the original CSS algorithm, and, ironically, it even performs worse than ShortStraw alone. The oversegmentation approach suffers from the same issues as using every point: the starting segmentation can have a very low mean-squared error, which can cause some large jumps in the mean-squared ratio. Even with loosening the ShortStraw thresholds, we still will run into issues where ShortStraw is simply not good at finding certain corners. The algorithm also runs much slower than the original CSS algorithm, and encroaches on running longer than real-time for single-stroke processing, where real-time would indicate that the user perceives no lag or delay in recognition.

Thus, Table VII shows the benefits of our CSS algorithm and why first running existing segmenters plays such an important part in the success of our algorithm.

F. Future Work

The combination technique that we use to improve segmentation accuracy can also be extended to other recognition techniques. For instance, if a sketched diagram is drawn with many strokes and is composed of multiple symbols and connectors, running different recognition algorithms on the sketch could find different overall interpretations. Our subset selection algorithm could find the best overall sketch interpretation, given an objective function that models the likelihood of the components found by different algorithms.

We also envision enhancing the subset selection's mean-squared error approach by incorporating probabilities of corners. If many segmenters find the same (or similar) points as corners, then those corners should have a lower chance of being removed from the final segmentation. Corners that are only introduced by a single segmenter would have a greater chance of being false positives. Using this information could hopefully eliminate the few false positives we find in our final segmentations.

Even though we shifted our focus toward polyline segmentation, we briefly examined how our algorithms could find corners in strokes that contain both lines and arcs. The main issue with this extension is that the mean-squared error objective function does not work well when evaluating multiple-primitive segmentations; what a user perceptually sees as an arc might be better segmented into a series of polylines based on mean-squared error. Similarly, polylines that are more obtuse, such as in octagons, are often segmented into a seemingly random series or arcs. Further work is needed to discover what objective functions can be used for combining multiple-primitive segmentation results.

G. Contributions

Our combination approach to segmentation

- Trainable segmentation technique
- Improves upon individual segmentation techniques in all of our tested metrics, finding less false positives, false negatives, and has a greater all-or-nothing accuracy
- Has great extensibility potential through new objective functions or additional segmenters

CHAPTER VI

CONCLUSION

We have presented three new techniques for corner finding. Our three segmenters are uniquely different from each other and show how we have progressed the field of corner finding in multiple primitive segmentation, polyline segmentation, and combining algorithms.

In MergeCF, our multiple primitive segmentation algorithm helped reduce the amount of noise in a stroke by removing the smallest segments. We showed that the improvement over previous multi-primitive segmentation algorithms is substantial, but the issues we documented with multiple primitive segmentation were too substantial to ignore. When more types of primitives were added to the MergeCF segmentation algorithm, the number of false positives and negatives increased and the all-or-nothing accuracy decreased. After analyzing MergeCF and discussing alternatives, such as PaleoSketch's post-segmentation primitive recognizer, we came to the conclusion that polyline segmentation should be sufficient.

Our new polyline segmenter introduced was entitled ShortStraw. ShortStraw has the benefit of being both a powerful polyline corner finder and simple to code. The algorithm uses a polyline-specific form of curvature based on chord lengths. The all-or-nothing accuracy of ShortStraw was much higher than other segmenters, but we realized that the algorithm does have a few shortcomings, specifically in that the found thresholds might not be sufficient for every domain.

After creating two new segmentation techniques and implementing many corner finders from previous work in the field, we decided to approach the problem of segmentation from a new direction. Instead of trying to create a segmentation algorithm that produced very high accuracy in all cases, we wanted to utilize every segmentation

algorithm's corner finding capabilities. We created a combination algorithm based on a feature subset selection technique found in pattern recognition. Our Corner Subset Selection algorithm picks the best corners from each segmenter and outperforms every individual algorithm's accuracy, finding many fewer false negatives and having an all-or-nothing accuracy above 92%.

Our results show a steady progression in segmentation accuracy. The sketch recognition community will greatly benefit from our work, as already evidenced by the modifications of ShortStraw presented at SBIM 2009 [59].

REFERENCES

[1] T. Hammond and R. Davis, "Tahuti: A geometrical sketch recognition system for UML class diagrams," in *Papers from the 2002 AAAI Symposium on Sketch Understanding*, Stanford, California, March 2002, pp. 59–68.

[2] L. B. Kara and T. F. Stahovich, "Sim-U-Sketch: A sketch-based interface for SimuLink," in *Proceedings of the Working Conference on Advanced Visual Interfaces*, 2004, pp. 354–357.

[3] C. Alvarado and R. Davis, "SketchREAD: A multi-domain sketch recognition engine," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 2004, pp. 23–32.

[4] T. Y. Ouyang and R. Davis, "Recognition of hand drawn chemical diagrams," in *Proceedings of the 22nd National Conference on Artificial Intelligence*, 2007, pp. 846–852.

[5] J. LaViola, Jr. and R. Zeleznik, "MathPad2: A system for the creation and exploration of mathematical sketches," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 432–440, 2004.

[6] R. Zeleznik, T. Miller, C. Li, and J. Laviola, Jr., "MathPaper: Mathematical sketching with fluid support for interactive computation," in *Proceedings of the 9th International Symposium on Smart Graphics*, 2008, pp. 20–32.

[7] T. O'Connell, C. Li, T. S. Miller, R. C. Zeleznik, and J. LaViola, Jr., "A usability evaluation of AlgoSketch: A pen-based application for mathematics," in *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, 2009, pp. 149–157.

[8] Palm Inc., "Palm website," http://www.palm.com, September 2009.

[9] Wacom, "Wacom website," http://www.wacom.com, September 2009.

[10] Lenovo, "Lenovo website," http://www.lenovo.com, September 2009.

[11] B. Paulson and T. Hammond, "PaleoSketch: Accurate primitive sketch recognition and beautification," in *Proceedings of the 13th International Conference on Intelligent User Interfaces*, 2008, pp. 1–10.

[12] D. Rubine, "Specifying gestures by example," in *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, 1991, pp. 329–337.

[13] A. C. Long, Jr., J. A. Landay, L. A. Rowe, and J. Michiels, "Visual similarity of pen gestures," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2000, pp. 360–367.

[14] Palm Computing, "Suddenly Newton understands everything you write," *Pen Computing Magazine*, p. 9, January 1995.

[15] I. S. Mackenzie and S. X. Zhang, "The immediate usability of Graffiti," in *Proceedings of Graphics Interface*, 1997, pp. 129–137.

[16] J. A. Landay and B. A. Myers, "Interactive sketching for the early stages of user interface design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1995, pp. 43–50.

[17] M. W. Newman, J. Lin, J. I. Hong, and J. A. Landay, "DENIM: An informal web site design tool inspired by observations of practice," *Human-Computer Interaction*, vol. 18, no. 3, pp. 259–324, 2003.

[18] O. Bimber, L. M. Encarnaçao, and A. Stork, "A multi-layered architecture for sketch-based interaction within virtual environments," *Computers & Graphics*, vol. 24, pp. 851–867, 2000.

[19] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge, "Comparing images using the Hausdorff distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 9, pp. 850–863, 1993.

[20] M.-P. Dubuisson and A. K. Jain, "A modified Hausdorff distance for object matching," in *Proceedings of the 12th International Conference on Pattern Recognition*, 1994, pp. 566–568.

[21] L. B. Kara and T. F. Stahovich, "An image-based trainable symbol recognizer for sketch-based interfaces," *Computers & Graphics*, vol. 29, no. 4, pp. 501–517, 2005.

[22] A. Wolin, B. Eoff, and T. Hammond, "Search your mobile sketch: Improving the ratio of interaction to information on mobile devices," in *Papers from the 2009 Intelligent User Interfaces Workshop on Sketch Recognition*, 2009.

[23] J. O. Wobbrock, A. D. Wilson, and Y. Li, "Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes," in *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, 2007, pp. 159–168.

[24] M. Gross, "The electronic cocktail napkin: A computational environment for working with design diagrams," *Design Studies*, vol. 17, no. 1, pp. 53–69, January 1996.

[25] T. Hammond and R. Davis, "LADDER, a sketching language for user interface developers," *Computers and Graphics*, vol. 28, pp. 518–532, 2005.

[26] S. E. Palmer, *Object Perception: Structure and Process*, chapter Reference Frames in the Perception of Shape and Orientation, pp. 121–163, Erlbaum, 1989.

[27] S. E. Palmer, *Vision Science: Photons to Phenomenology*, MIT Press, 1999.

[28] D. Groome, *An Introduction to Cognitive Psychology: Processes and Disorders*, Psychology Press, 1999.

[29] E. J. Gibson, *Principles of Perceptual Learning and Development*, Prentice Hall, 1969.

[30] D. H. Hubel and T. N. Wiesel, "Brain mechanisms and vision," *Scientific American*, vol. 241, no. 3, pp. 150–162, 1979.

[31] J. Sklansky and V. Gonzalez, "Fast polygonal approximation of digitized curves," *Pattern Recognition*, vol. 2, no. 5, pp. 327–331, 1980.

[32] K. Wall and P.-E. Danielsson, "A fast sequential method for polygonal approximation of digitized curves," *Graphical Models and Image Processing*, vol. 28, no. 2, pp. 220–227, November 1984.

[33] B. K. Ray and K. S. Ray, "Determination of optimal polygon from digital curve using L1 norm," *Pattern Recognition*, vol. 26, no. 4, pp. 505–509, April 1993.

[34] Yoshisuke K. and W. A. Davis, "Polygonal approximation by the minimax method," *Computer Graphics and Image Processing*, vol. 19, no. 3, pp. 248–264, July 1982.

[35] P.-C. Chung, C.-T. Tsai, E.-L. Chen, and Y.-N. Sun, “Polygonal-approximation using a competitive hopfield neural-network,” *Pattern Recognition*, vol. 27, no. 11, pp. 1505–1512, November 1994.

[36] D. H. Douglas and T. K. Peucker, “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,” *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[37] J. Hershberger and J. Snoeyink, “Speeding up the Douglas-Peucker line-simplification algorithm,” in *Proceedings of the 5th International Symposium on Spatial Data Handling*, 1992, pp. 134–143.

[38] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink, “Approximating polygons and subdivisions with minimum link paths,” *International Journal of Computational Geometry and Applications*, vol. 3, pp. 383–415, 1993.

[39] S.-T. Wu, M. Rocío, and G. Márquez, “A non-self-intersection Douglas-Peucker algorithm,” in *Brazilian Symposium on Computer Graphics and Image Processing*, Oct. 2003, pp. 60–66.

[40] B. Yu and S. Cai, “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*, 2003, pp. 141–146.

[41] T. M. Sezgin, T. Stahovich, and R. Davis, “Sketch based interfaces: Early processing for sketch understanding,” in *Papers from the Workshop on Perceptive User Interfaces*, 2001.

[42] T. F. Stahovich, “Segmentation of pen strokes using pen speed,” in *Papers*

from the 2004 AAAI Symposium on Making Pen-Based Interaction Intelligent and Natural, 2004.

[43] P. Agar and K. Novins, "Polygon recognition in sketch-based interfaces with immediate and continuous feedback," in *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, 2003, pp. 147–150.

[44] C. F. Herot, "Graphical input through machine recognition of sketches," in *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques*, 1976, pp. 97–102.

[45] D. H. Kim and M.-J. Kim, "A curvature estimation for pen input segmentation in sketch-based modeling," *Computer-Aided Design*, vol. 38, no. 3, pp. 238–248, 2006.

[46] A. Bandera, C. Urdiales, F. Arrebola, and F. Sandoval, "Corner detection by means of an adaptively estimated curvature function," *Electronics Letters*, vol. 36, no. 2, pp. 124–126, 2000.

[47] A. Rattarangsi and R. T. Chin, "Scale-based detection of corners of planar curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 4, pp. 430–449, April 1992.

[48] T. M. Sezgin and R. Davis, "Scale-space based feature point detection for digital ink," in *Papers from the 2004 AAAI Symposium on Making Pen-Based Interaction Intelligent and Natural*, 2004, pp. 145–151.

[49] S. Zhai, P.-O. Kristensson, and B. A. Smith, "In search of effective text input interfaces for off the desktop computing," *Interacting with Computers*, vol. 17,

no. 3, pp. 229–250, 2005.

[50] P. O. Kristensson and S. Zhai, "Shark^2: A large vocabulary shorthand writing system for pen-based computers," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 2004, pp. 43–52.

[51] W. Buxton, R. Sniderman, W. Reeves, S. Patel, and R. Baecker, "The evolution of the SSSP score editing tools," *Computer Music Journal*, vol. 3, no. 4, pp. 14–25, 1979.

[52] B. Paulson and T. Hammond, "MARQS: Retrieving sketches learned from a single example using a dual-classifier," *Journal on Multimodal User Interfaces*, vol. 2, no. 1, pp. 3–11, July 2008.

[53] R. Patel, B. Plimmer, J. Grundy, and R. Ihaka, "Ink features for diagram recognition," in *Proceedings of the 4th Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2007, pp. 131–138.

[54] H. Heloise, M. Shilman, and A. R. Newton, "Robust sketched symbol fragmentation using templates," in *Proceedings of the 9th International Conference on Intelligent User Interfaces*, 2004, pp. 156–160.

[55] A. Wolin, B. Paulson, and T. Hammond, "Sort, merge, repeat: An algorithm for effectively finding corners in hand-sketched strokes," in *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, 2009, pp. 93–99.

[56] S. Cates and R. Davis, "A new approach to early sketch processing," in *Papers from the 2004 AAAI Symposium on Making Pen-Based Interaction Intelligent and Natural*, October 2004, pp. 29–34.

[57] O. Veselova and R. Davis, “Perceptually based learning of shape descriptions,” in *Proceedings of the 19th National Conference on Artificial Intelligence*, San Jose, California, 2004, pp. 482–487.

[58] A. Wolin, B. Eoff, and T. Hammond, “ShortStraw: A simple and effective corner finder for polylines,” in *Proceedings of the 5th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, June 2008, pp. 33–40.

[59] Y. Xiong and J. LaViola, Jr., “Revisiting ShortStraw: Improving corner finding in sketch-based interfaces,” in *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, 2009, pp. 101–108.

[60] Y. Qiao and M. Yasuhara, “Recovering dynamic information from static handwritten images,” in *Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition*, 2004, pp. 118–123.

[61] P. Rajan and T. Hammond, “From paper to machine: Extracting stokes from images for use in sketch recognition,” in *Proceedings of the 5th Eurographics Workshop on Sketch-Based Interfaces and Modeling*, June 2008, pp. 41–48.

[62] J. LaViola, Jr., “CAP 6938 topics in pen-based user interfaces: Assignment 2,” http://www.eecs.ucf.edu/courses/cap6938/fall2008/penui/handouts/asgn2.pdf, September 2008.

[63] C. H. Teh and R. T. Chin, “On the detection of dominant points on digital curves,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 8, pp. 859–872, 1989.

[64] D. H. Wolpert, “The lack of a priori distinctions between learning algorithms,” *Neural Computation*, vol. 8, no. 7, pp. 1341–1390, October 1996.

[65] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, April 1997.

[66] T. Marill and D. Green, “On the effectiveness of receptors in recognition systems,” *IEEE Transactions on Information Theory*, vol. 9, no. 1, pp. 11–17, 1963.

[67] C. Y. Chang, “Dynamic programming as applied to feature subset selection in a pattern recognition system,” in *Proceedings of the ACM Annual Conference*, 1972, pp. 94–103.

[68] W. Siedlecki and J. Sklansky, *Handbook of Pattern Recognition and Computer Vision*, chapter 1.3.1 On Automatic Feature Selection, pp. 63–87, World Scientific Publishing Co., 1993.

APPENDIX A

SHORTSTRAW PSEUDOCODE

This section contains the full algorithm for ShortStraw in object-oriented pseudocode. The variable *points* contains a sequential series of (x, y) points, whereas *corners* contains a set of indices that reference points. For example, $corner_i = j$ indicates that $point_j$ is the ith corner found. Arrays start at index 0.

Main body where the corner finding functions are called. Takes in a series of original, non-resampled *points* and returns the corners for the resampled points.

Input: A series of original, non-resampled points

Output: The corners for the resampled points

MAIN(*points*)

1: S ←DETERMINE-RESAMPLE-SPACING(*points*)

2: *resampled* ←RESAMPLE-POINTS(*points*, S)

3: *corners* ←GET-CORNERS(*resampled*)

4: **return** *corners*

Determines the bounding box diagonal of the points

Input: A series of points

Output: The bounding box diagonal size for the points

GET-DIAGONAL(*points*)

1: $topLeft.x \leftarrow \text{MIN}_x(points)$

2: $topLeft.y \leftarrow \text{MIN}_y(points)$

3: $bottomRight.x \leftarrow \text{MAX}_x(points)$

4: $bottomRight.y \leftarrow \text{MAX}_y(points)$

5: $diagonal \leftarrow \text{DISTANCE}(bottomRight, topLeft)$

6: **return** *diagonal*

Determines the interspacing pixel distance between resampled points

Input: A series of points

Output: The interspacing distance for the resampled points

DETERMINE-RESAMPLE-SPACING(*points*)

1: $diagonal \leftarrow \text{GET-DIAGONAL}(points)$

2: $S \leftarrow \max(diagonal/80.0, 0.5)$

3: **return** S

Resamples the *points* in a stroke to be interspaced S pixel distance away from each other

Input: A series of points and an interspacing distance

Output: The resampled points

RESAMPLE-POINTS($points, S$)

1: $D \leftarrow 0$
2: $resampled \leftarrow points_0$
3: **for** $i \leftarrow 1$ **to** $|points|$ **do**
4: $d \leftarrow$ DISTANCE($points_{i-1}, points_i$)
5: **if** $D + d \geq S$ **then**
6: $q.x \leftarrow points_{i-1}.x + ((S - D)/d) \times (points_i.x - points_{i-1}.x)$
7: $q.y \leftarrow points_{i-1}.y + ((S - D)/d) \times (points_i.y - points_{i-1}.y)$
8: APPEND($resampled, q$)
9: INSERT($points, i, q$)
10: $D \leftarrow 0$
11: **else**
12: $D = D + d$
13: **end if**
14: **end for**
15: **return** $resampled$

Finds the resampled points that correspond to corners within the stroke

Input: A series of resampled points

Output: The resampled points that correspond to corners

```
GET-CORNERS(points)
 1: corners ← ∅
 2: APPEND(corners, 0)
 3: W ← 3
 4: for i ← W to |points| − W do
 5:   straws_i ← DISTANCE(points_{i−W}, points_{i+W})
 6: end for
 7: t ← MEDIAN(straws) × 0.95
 8: for i ← W to |points| − W do
 9:   if straws_i < t then
10:     localMin ← +∞
11:     localMinIndex ← i
12:     while i < |straws| and straws_i < t do
13:       if straws_i < localMin then
14:         localMin ← straws_i
15:         localMinIndex ← i
16:       end if
17:       i ← i + 1
18:     end while
19:     APPEND(corners, localMinIndex)
20:   end if
21: end for
```

22: APPEND($corners, |points|$)

23: $corners \leftarrow$ POST-PROCESS-CORNERS($corners, straws$)

24: **return** $corners$

Checks the corner candidates to see if any corners can be removed or added based on higher-level polyline rules

Input: A series of resampled points, an initial set of corners, and the straw distances for each point

Output: A set of corners post-processed with higher-level polyline rules

---

POST-PROCESS-CORNERS(*points*, *corners*, *straws*)

1: $continue \leftarrow$ FALSE

2: **while** $\neg continue$ **do**

3: $continue \leftarrow$ TRUE

4: **for** $i \leftarrow 1$ **to** $|corners|$ **do**

5: $c_1 \leftarrow corners_{i-1}$

6: $c_2 \leftarrow corners_i$

7: **if** $\neg$IS-LINE($points, c_1, c_2$) **then**

8: $newCorner \leftarrow$ HALFWAY-CORNER($straws, c_1, c_2$)

9: INSERT($corners, i, newCorner$)

10: $continue \leftarrow$ FALSE

11: **end if**

12: **end for**

13: **end while**

14: **for** $i \leftarrow 1$ **to** $|corners| - 1$ **do**

15: $c_1 \leftarrow corners_{i-1}$

16: $c_2 \leftarrow corners_{i+1}$

17: **if** Is-Line($points, c_1, c_2$) **then**

18: Remove($corners, corners_i$)

19: $i \leftarrow i - 1$

20: **end if**

21: **end for**

22: $hookThreshold \leftarrow \min(\text{Get-Diagonal}(points) \times 0.10, 15)$

23: **while** $|corners| > 1$ **and** Distance($points_0, corners_1$) $< hookThreshold$ **do**

24: Remove($corners, corners_1$)

25: **end while**

26: **while** $|corners| > 2$ **and** Distance($points_{|points|-1}, corners_{|corners|-2}$) $< hookThreshold$ **do**

27: Remove($corners, corners_{|corners|-2}$)

28: **end while**

29: **return** $corners$

Finds a corner roughly halfway between point indices a and b

Input: The straw distances for each point, two point indices a and b

Output: A possible corner between the points at a and b

Halfway-Corner($straws, a, b$)

1: $quarter \leftarrow (b - a)/4$

2: $minValue \leftarrow +\infty$

3: **for** $i \leftarrow a + quarter$ **to** $b - quarter$ **do**

4: **if** $straws_i < minValue$ **then**

5: $minValue \leftarrow straws_i$

6: $minIndex \leftarrow i$

7: **end if**

8: **end for**

9: **return** $minIndex$

Computes the Euclidean chord distance between the points at indices a and b

Input: A series of points and two indices, a and b

Output: The Euclidean (chord) distance between the points at a and b

---

DISTANCE($points, a, b$)

1: $\Delta x \leftarrow points_b.x - points_a.x$

2: $\Delta y \leftarrow points_b.y - points_a.y$

3: **return** $\sqrt{\Delta x^2 + \Delta y^2}$

Computes the Euclidean path distance between the points at indices a and b

Input: A series of points and two indices, a and b

Output: The path (stroke segment) distance between the points at a and b

---

PATH-DISTANCE($points, a, b$)

1: $d \leftarrow 0$

2: **for** $i \leftarrow a$ **to** $b - 1$ **do**

3: $d \leftarrow d$ + DISTANCE($points_i, points_{i+1}$)

4: **end for**

5: **return** d

Determines if the stroke segment between the points at indices a and b form a line

Input: A series of points and two indices, a and b

Output: A boolean for whether or not the stroke segment between points at a and b is a line

---

IS-LINE($points, a, b$)

1: $threshold \leftarrow 0.95$

2: $distance \leftarrow$ DISTANCE($points_a, points_b$)

3: $pathDistance \leftarrow$ PATH-DISTANCE($points, a, b$)

4: **if** $distance/pathDistance > threshold$ **then**

5:     **return** TRUE

6: **else**

7:     **return** FALSE

8: **end if**

APPENDIX B

ISTRAW

Xiong and LaViola Jr. extended our ShortStraw algorithm [58] when creating their own segmenter, IStraw [59]. The IStraw authors analyzed ShortStraw and wanted to make two main modifications: (1) improve aspects of ShortStraw to account for some of the algorithm's limitations, and (2) add curvature segmentation to the algorithm.

A. Modifications

IStraw introduced modifications to ShortStraw that enhanced the accuracy of the corner finder. Many of the modifications were small, such as changing a threshold slightly. We will discuss only the main additions here.

1. Corners From Speed

IStraw adds points of low speed to the corner set, similar to Sezgin *et al.* [41] and Stahovich [42]. The speed information is computed on resampled points, which means that the speed of each resampled point must be extrapolated from the original point data.

2. Consecutive Collinear Tests

When ShortStraw runs collinear tests on point triplets, it is possible that the algorithm can remove a correct corner. After an initial segmentation, ShortStraw runs collinear tests to see if a series of three points forms a line; if so, then we remove

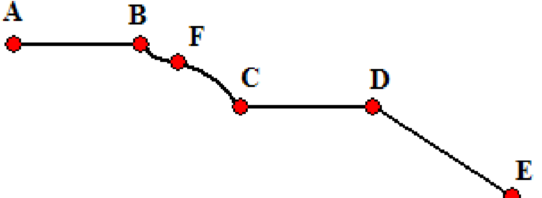


Fig. 55. An example of collinear line test issues in ShortStraw. In ShortStraw, the $A-B-F$ collinear test will eliminate a correct corner, B, before the $B-F-C$ tests remove the false positive, F. This figure was created by Xiong and LaViola Jr. [59].

the middle point because it is unneeded. ShortStraw can sometimes remove an unintended, correct corner due to the line thresholds being too lenient.

The example that Xiong and LaViola Jr. provided is shown in Fig. 55. The collinear tests will first start by examining $A - B - F$. $A - F$ forms a line segment under relaxed threshold choices, which would cause the correct corner, B, to be removed from the final segmentation. To compensate for this error, IStraw runs two sets of collinear tests. The first run has stricter IS-LINE thresholds so that the $A - B - F$ collinear test is less likely to remove B. The second test relaxes the line test thresholds to be equal to their original ShortStraw values.

3. Hook Removal

In some cases, ShortStraw might find corners close to the endpoints of the stroke. These corners are considered part of noisy hooks in a stroke, and IStraw removes them. Note that in the version of ShortStraw presented in this thesis, we also remove hooks.

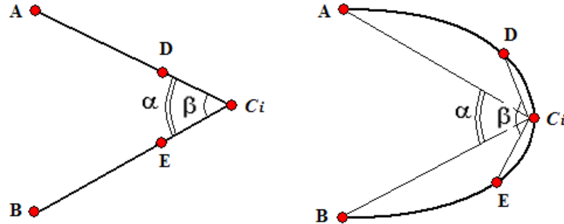


Fig. 56. At each corner, c_i, IStraw evaluates two angles, α and β, around a window of resampled points. If c_i is a correct corner, such as in the figure on the left, $\beta - \alpha$ is close to 0. If c_i is part of a curve, then $\beta - \alpha$ is greater than 0. This figure was created by Xiong and LaViola Jr. [59].

4. Addition of Curves

IStraw additionally checks for curve primitives. IStraw originally segments a stroke into polylines by using ShortStraw (with IStraw's additional modifications). The algorithm then checks whether each corner, $c_i \in corners$, is part of a curve by examining a window of points around the corner. Fig. 56 demonstrates this process. The two chords $A - c_i$ and $B - c_i$ form angle α. The chords $D - c_i$ and $E - c_i$ form angle β. If $\beta - \alpha$ is approximately equal to 0, then the points A, D, and c_i are collinear; similarly, B, E, and c_i are collinear. If $\beta - \alpha$ is greater than 0, then the points are not collinear and c_i is part of a curve. The threshold, t_a, for which $\beta - \alpha > t_a$ implies that c_i is a curve was empirically determined by Xiong and LaViola Jr. to be between 14 and 33 degrees. The threshold is dynamically chosen by the function $t_a = 10 + 800/(\alpha + 35)$.

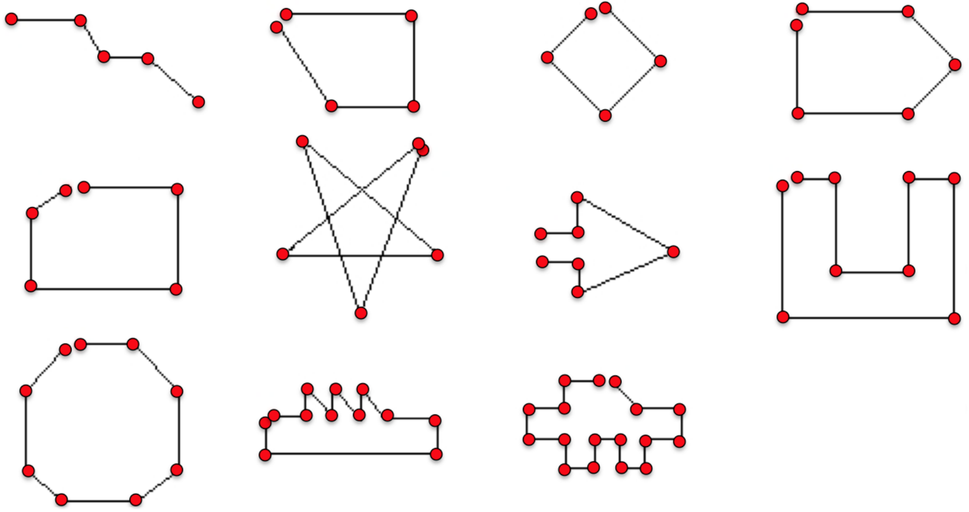


Fig. 57. The 11 polyline symbols used for testing in our ShortStraw evaluation.

B. Presented Results

Xiong and LaViola Jr. had multiple evaluations of their IStraw algorithm. Their first evaluation compared ShortStraw with IStraw using our test set of 244 polyline strokes, from the 11 symbols shown in Fig. 57. Xiong and LaViola Jr. also collected data from 15 additional users at the University of Central Florida. The data the users drew included the symbols in Fig. 57 and Fig. 58. This new dataset contained 656 polyline strokes and 590 strokes containing curvature, for a total of 1246 strokes.

Note that Xiong and LaViola Jr. compared their results to our SBIM 2008 paper on ShortStraw [58], not the ShortStraw algorithm with slight modifications presented in this thesis.

The results from various comparisons are arranged in Tables VIII, IX, and X.

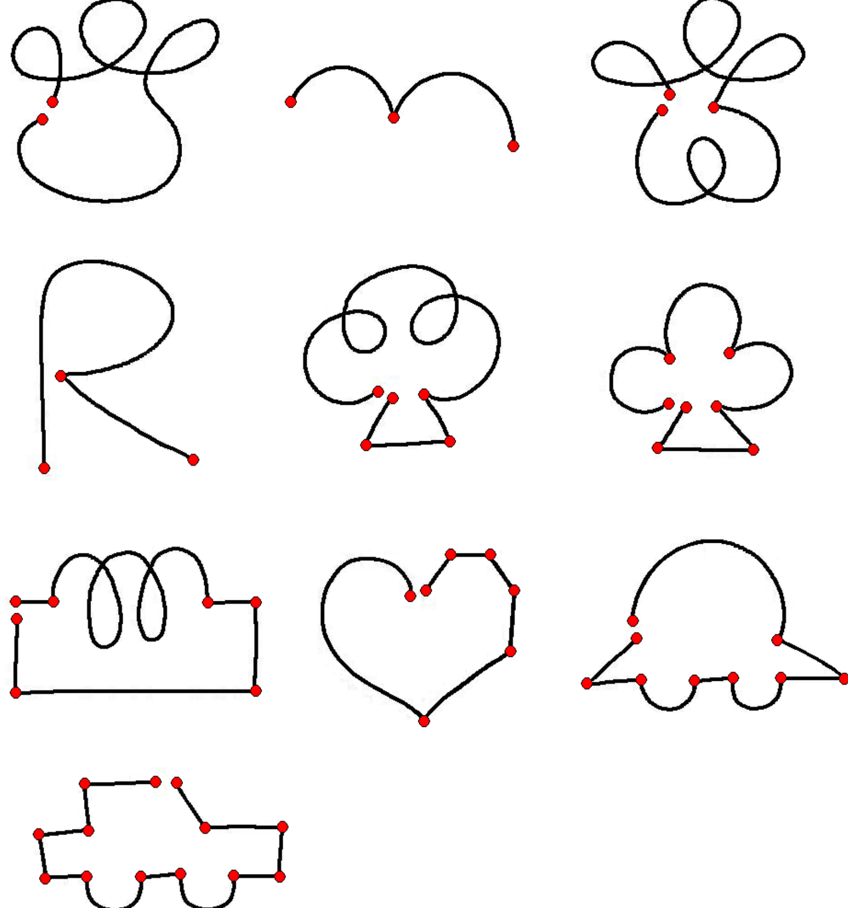


Fig. 58. The 10 line and curve symbols Xiong and LaViola Jr. collected. This figure was presented in their SBIM 2009 paper [59].

Table VIII. Results comparing ShortStraw to IStraw on the 244 original polyline test strokes, as presented by Xiong and LaViola Jr. [59]. We added a comparison to the ShortStraw algorithm we present in this thesis. IStraw-C is IStraw with curve detection deactivated. The 244 strokes were not tested with IStraw's curve detection turned on. The number of correct corners has been changed to 1841 from 1842 in the original paper; 1842 was a typo in ShortStraw [58].

	ShortStraw (SBIM)	ShortStraw (Thesis)	IStraw
False Positives	32	6	2
False Negatives	38	28	1
Correct Corners	1804	1815	1840
Total Correct Corners	1841	1841	1841
Correct Corners Accuracy	0.979	0.984	0.999
All-or-Nothing Accuracy	0.741	0.881	0.998

Table IX. Results comparing ShortStraw (from SBIM 2008 [58]) to IStraw with curve detection. These values are for the 656 polyline strokes in Xiong and LaViola Jr.'s dataset [59].

	ShortStraw (SBIM)	IStraw
False Positives	32	1
False Negatives	93	21
Correct Corners	5059	5131
Total Correct Corners	5152	5152
Correct Corners Accuracy	0.983	0.996
All-or-Nothing Accuracy	0.838	0.968

Table X. Results comparing ShortStraw (from SBIM 2008 [58]) to IStraw with curve detection. These values are for the 1246 strokes in Xiong and LaViola Jr.'s dataset [59]. The dataset contains both polyline-only data (Fig. 57) and the line and curve data (Fig. 58).

	ShortStraw (SBIM)	IStraw
False Positives	8326	29
False Negatives	127	58
Correct Corners	8497	8566
Total Correct Corners	8624	8624
Correct Corners Accuracy	0.985	0.993
All-or-Nothing Accuracy	0.441	0.940

C. Discussion

IStraw does better than ShortStraw in all of these cases, and it has very high (above 0.94) all-or-nothing accuracy for every dataset used. The original ShortStraw algorithm from SBIM 2008 performs well against Xiong and LaViola Jr.'s new dataset of polyline symbols. ShortStraw performs poorly on curvature symbols, since it is a polyline only algorithm.

The IStraw segmenter's all-or-nothing accuracy results are impressive for the data. Their accuracy is the highest we have seen reported, and Xiong and LaViola Jr. should be commended for this work. Yet, their results do not undermine ShortStraw's (or MergeCF's or Corner Subset Selection's) inherent value. Correct corners accuracy is equivalent to recall, and all of the algorithms perform similarly in this metric.

IStraw's main drawback compared to ShortStraw is in code complexity. One of ShortStraw's main benefits was that it could be coded very quickly by even novices to sketch recognition. IStraw has many additional components, some of which have undefined thresholds (i.e., speed thresholds for slow corners were never given), and

the algorithm has no accompanying pseudocode.

Many of the changes from ShortStraw to IStraw were minor, such as small threshold tweaks or using the *mean* straw length to calculate the straw threshold instead of the *median*. These changes can improve ShortStraw in certain situations, but any minor threshold changes will improve segmentation on some cases while hurting segmentation on others. This is another example application of the No Free Lunch theorems [64, 65]. The major adjustments in IStraw do demonstrate a remarkable improvement in the segmentation accuracy, and it would be beneficial in future papers from Xiong and LaViola Jr. to know which modification (corners from speed, consecutive collinear tests, or hook removal), provided the greatest impact to polyline segmentation.

The curvature data that IStraw handles is not necessarily curve primitives. In the sketch recognition literature, a curve is typically defined as a sequence of points that can be modeled with a Bezier curve of some order [41]. The data in Fig. 58 has some curvature sequences like helixes that can not be easily described mathematically. Instead, we can say that IStraw handles "curvy" data, which is different than curves. Curvy data can provide a perceptually correct segmentation for many symbols, such as the bottom left symbol in Fig. 58. But, it can also lead to some awkward segmentations. For instance, the 'R' symbol does not have a corner connecting the left vertical line to the 'R's arc (Fig. 59). Depending on the domain the segmenter is being used in, the distinction between "curves" and "curvy" might be insignificant or highly important. Application developers need to be aware of this difference, but, if the issue is inconsequential, then IStraw is a very alluring segmenter.

Lastly, we wanted to reiterate our Corner Subset Selection's benefits. Although we do not report as high all-or-nothing accuracies as IStraw, our subset selection technique is fully trainable and has great extensibility. IStraw (and other ShortStraw-

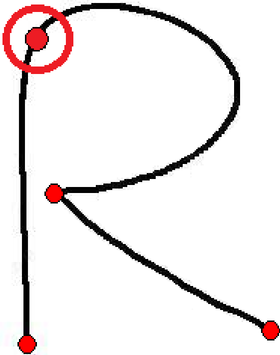


Fig. 59. The ‘R’ symbol from Fig. 58 should have another corner where the left vertical line and arc meet (circled here). This corner is missing from the IStraw symbols due to Xiong and LaViola Jr.’s recognition of “curvy” data, rather than curves.

based techniques) are sure to be weak segmenters of some domains and symbols. In these cases, other corner finders might fare better, and the subset selection technique can accommodate these issues by using many different segmentation algorithms. Our Corner Subset Selection segmenter can also be trained to be either user-specific or domain-specific; all the algorithm needs is training data for the user or domain in question.

APPENDIX C

APPLICATIONS

Corner finding can be used in a variety of applications, as mentioned during Chapter II. Here, we will mention how the Sketch Recognition Lab at Texas A&M Univesity uses corner finding.

D. Geometric-based Recognizers

The main focus of our work has been to use corner finding as a low-level step in large sketch recognition systems. The system that we use in the lab is entitled SLOTH, and it is an extension of the geometric-based system (See Chapter II.A.3) presented in LADDER [25]. SLOTH has four steps during sketch recognition:

1. Segment a stroke (or set of strokes) using polyline segmenters.
2. Send the segmentations into PaleoSketch for low-level primitive recognition.
3. Try to build shapes from the resulting set of primitives. For each shape description in a domain:
   (a) Check whether a shape descriptions has the required component primitives (the <`componenentList`> section in Fig. 60).
   (b) If so, then evaluate the primitives using the defined constraints (the <`constraintList`> section).
   (c) Calculate a confidence score for the shape based on how well the shape's primitives obey the constraints.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<shapeDefinition name="infantry" description="Infantry">

    <componentList>
        <!-- the frame [rectangle] -->
        <component name="rectangle" type="Rectangle" />

        <!-- the infantry -->
        <component name="posLine" type="Line" />
        <component name="negLine" type="Line" />
    </componentList>

    <constraintList>
        <!-- CONTAINS RELATIONSHIPS -->
        <constraint name="Contains">
            <param component="rectangle" />
            <param component="posLine" />
        </constraint>

        <constraint name="Contains">
            <param component="rectangle" />
            <param component="negLine" />
        </constraint>

        <!-- LINE ORIENTATIONS -->
        <constraint name="PositiveSlope">
            <param component="posLine" />
        </constraint>

        <constraint name="NegativeSlope">
            <param component="negLine" />
        </constraint>

        <!-- SIZE RELATIONSHIPS -->
        <constraint name="SameSize">
            <param component="posLine" />
            <param component="negLine" />
        </constraint>
        . . .
    </constraintList>
</shapeDefinition>
```

Fig. 60. Part of the shape description for the military course of action symbol, Infantry. The shape description specifies the primitives that the shape contains (1 rectangle, 2 lines), as well as the constraint interactions between the primitives.

4. Choose the shape that has the best confidence score.

In SLOTH, segmentation is a key component in performing recognition. Without segmentation, users would need to draw each primitive separately in order for the system to build shapes from primitives. With segmentation, users can draw multiple primitives in a single stroke, and corner finding with PaleoSketch allows us to separate and recognize the drawn primitives.

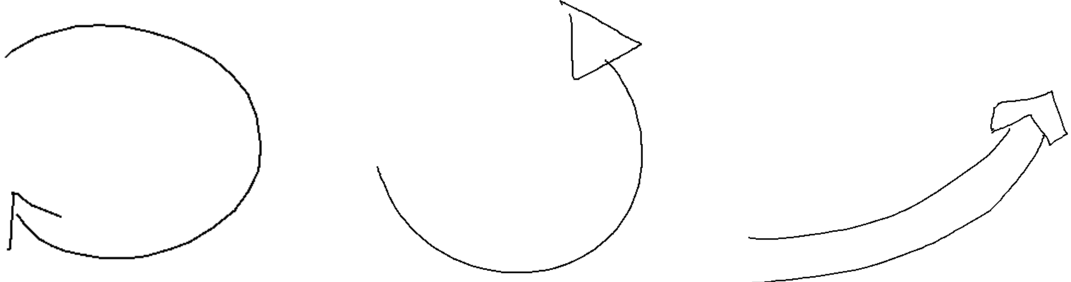
E. Corners as Features in Arrow Recognition

We also use segmentation as a feature when creating domain-specific algorithms. A recent project for DARPA had the Sketch Recognition Lab recognize military course of action symbols. Many of the symbols in the course of action data can be recognizable by a geometric-based system. That is, the symbols have a well-defined structure.

Some symbols in the course of action data do not have a rigid structure. For instance, arrows can have arbitrary paths which are difficult to describe (Fig. 61).

We created an arrow-specific recognizer that uses corners as a feature for recognition. We used segmentation to distinguish between arrow heads based on the number of segments each arrow head had. With only 2 segments (i.e., 3 corners), we discerned

Fig. 61. Two arrows can have different, arbitrary paths that indicate the attack direction of units in course of action diagrams.



(a) An arrow with a standard, 'V' arrow head.

(b) An arrow with a triangular arrow head.

(c) An arrow with an outlined arrow head.

Fig. 62. The three types of arrow heads we use segmentation to help recognize.

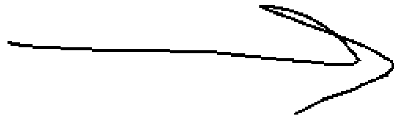
an arrow head to be a standard 'V' shape. With 3 segments that formed a closed polygon, we classified an arrow head as being triangular. Finally, there were outline arrow heads that were typically segmented into 6 primitive lines (Fig. 62).

We also use segmentation to distinguish between types of arrows. In course of action diagrams, a Task, Fix arrow and a Ground Supporting Attack arrow are similar except for the arrow's path (Fig. 63). We can use the number of polyline segments in an arrow's path in order to differentiate between the two arrows.

Similarly, in Task, Follow and Assume and Task, Follow and Support arrows, the tail end of the arrow differs by only one segment (Fig. 64). We utilize the number of segments (along with our arrow head prediction and other features), in order to confidently recognize these arrows.

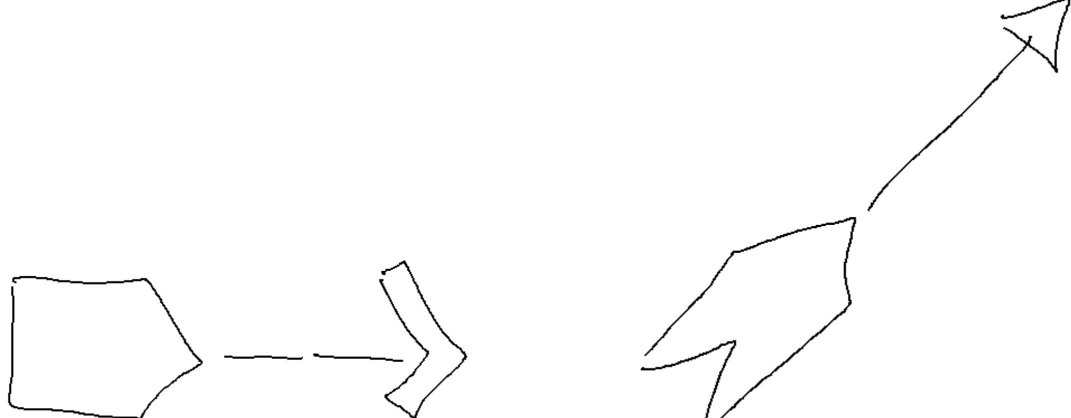
The simplicity of these approaches is that with only segmentation and some rudimentary constraints, we were able to create a domain-specific recognizer to use within a more complicated system. When low-level techniques like segmentation become more reliable and accurate, developers can use the number of corners in a

(a) A Task, Fix arrow.



(b) A Ground Supporting Attack arrow.

Fig. 63. These two arrows, Task, Fix (63(a)) and Ground Supporting Attack (63(b)), differ only in the number of segments in the arrow's path.



(a) A Task, Follow and Assume arrow.

(b) A Task, Follow and Support arrow.

Fig. 64. These two arrows, Task, Follow and Assume (64(a)) and Task, Follow and Support (64(b)), have a different number of segmentations in their tail. The number of segments, 5 and 6, respectively, is one feature that helps classify the arrows.

stroke as an additional feature that can improve recognition.

VITA

Aaron David Wolin received his B.S. in Computer Science from Harvey Mudd College in May 2007. He entered the graduate department of Texas A&M the same year, earning a Master's degree in Computer Science in May 2010. His interests include pen-based computing, pattern recognition, and HCI.

Aaron can be reached at the Department of Computer Science and Engineering, Texas A&M University, TAMU 3112, College Station, TX 77843-3112. His email address is awolin@gmail.com.