EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling (2011)
T. Hammond and A. Nealen (Editors)

Combining Corners from Multiple Segmenters

Aaron Wolin, Martin Field, and Tracy Hammond

Department of Computer Science and Engineering
Texas A&M University
{awolin,mfield,hammond}@cse.tamu.edu

Abstract

Pen-based interfaces utilize sketch recognition in order to allow users to sketch complex systems with intuitive input. In order to allow users to freely draw their ideas without constraints, the low-level techniques involved with sketch recognition must be perfected because poor low-level accuracy can impair a user's interaction experience. Stroke segmentation algorithms often employ single, specific techniques in their attempts to splice strokes into primitives used for visual shape representations. These algorithms each have their strengths and weaknesses, and different segmenters find and miss different corners.

We introduce a technique to combine polyline corner results from different segmenters by using a variation of feature subset selection. Our feature subset selection algorithm uses a sequential floating backward selection with a mean-squared error objective function in order to find the best subset of corners. By utilizing our combination method, we were able to achieve all-or-nothing accuracies of 0.926 on polyline stroke data.

1. Introduction

Many domains use drawings to convey meanings, such as circuit diagrams [AD04, KS04b], family trees [AD04], and UML diagrams [HD02]. Sketch recognition allows users to communicate these diagrams graphically to a computer while enabling a computer to understand a user's drawing intention.

In sketch-based interfaces, users interact with the computer through pen-based devices like Wacom pads and Tablet PCs. Users draw with their input styli, and the data is digitally recorded by a computer. Through sketch recognition, a computer can then understand these drawings, which allows users to intuitively draw complex graphical systems quickly and efficiently. This intelligent recognition of sketches can greatly simplify symbol and diagram-based domain interfaces so that users can interact with a system with minimal training.

A recent evaluation of a pen-based mathematics application showed that a low perception of recognition accuracy is one of the major obstacles to user adoption of complex sketch recognition systems [OLM*09]. This work shows that the users in their evaluation had low expectations for the computer's recognition accuracy, and, as such, all errors were attributed to the system, rather than the human. The low error tolerance by the human also manifested itself in another way: the users would not accept a learning curve and desired the system to work perfectly upon first use. Thus, in order to create effective sketch recognition user interfaces, the interfaces must truly be intelligent, or, more specifically, have high recognition accuracy. This paper focuses on elevating low-level recognition techniques to help improve overall recognition accuracy, making a more intelligent user interface.

In pen-based applications, interfaces are built upon strokes, where a *stroke* is defined as the path of a pen from pen-down on the screen to pen-up. Users draw these strokes to form symbols, which are then recognized in their respective domains. There are many ways users can draw even a simple symbol, and, as system developers, it is not practical to invent a separate technique to recognize every different drawing style, nor do we want to create a rigid system that requires a user to learn and conform to an unnatural drawing style. In order to prevent a steep learning curve and create an effective intelligent user interface, it is vital that a system allow for drawing flexibility.

Corner finding (also known as *segmentation*, *fragmenta-*

SBIM 2011, Vancouver, British Columbia, Canada, August 5 – 7, 2011.
© 2011 ACM 978-1-4503-0906-6/11/0008 $10.00

tion, or *cusp detection*) allows this drawing flexibility. Corner finding locates cusps or corners on strokes, enabling a system to break strokes into perceptually significant primitives, such as lines, arcs, curves, ellipses, spirals, helices, etc. [PH08]. These primitives then form the basis for a visual language used to build more complex shapes [HD05].

Corner finding therefore allows users to draw shapes with few constraints, and corner finding can then break multiple users' strokes into the same components. It does not matter how a user draws a square when corner finding always splits a user's strokes into the necessary 4 lines.

Individual corner finding algorithms are often specialized in one particular technique, such as using a specific curvature metric to detect corners [KK06, SSD01, WEH08]. All segmentation algorithms have one or more of the following drawbacks: (1) over-segmentation, or finding too many corners, (2) under-segmentation, or finding too few corners, or (3) incorrect segmentation, or finding the incorrect corners.

Another issue with past corner finder research is the usage of empirically determined thresholds. Empirically found thresholds are common in segmentation algorithms [DP73, KK06, PH08, SSD01, KS04a, WEH08, WPH09, YC03], but, we wanted to move toward completely trainable segmentation algorithms.

After creating and implementing multiple segmentation algorithms, we realized that no one segmentation 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 [Wol96, WM97].

We propose a new technique to overcome these deficiencies by combining the corners found from many individual segmentation algorithms using feature subset selection methods. In this algorithm, the features are the corners themselves.

2. Previous Work

2.1. Relevant Corner Finders

Much work has been done in corner finders that employ a single, specific technique. The two main types of corner finders are polyline corner finders and multi-primitive corner finders.

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 current stroke direction [PH08, SG80, WD84]. 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 [WEH08], 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 [DP73,HS92]. These techniques work well for non-self-intersecting strokes, but can produce many false positives. Although it is sometimes important to represent primitives other than lines, polyline segmentation is still an important and widely applicable technique. For example, PaleoSketch uses polyline segmentation as a preprocessing step toward recognizing multiple primitives.

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 [KK06, SSD01, KS04a, WPH09, YC03] and finding points of low pen speed [SSD01, KS04a, WPH09]. Noise is the main issue of these corner finders; local or global thresholds for curvature and speed corner choosing are highly susceptible to outliers.

There is currently 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 [SSD01]. 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 feature subset 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 [MG63]. 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 [Cha72], beam searches, or branch-and-bound algorithms [SS93]. We use a sequential floating backward selection (SFBS) algorithm to utilize both forward and backward searching, and since we should not have hundreds or thousands of corners per stroke we do not need to use more bounded approximation algorithms.

2.2. Relation to Primitive Finding

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

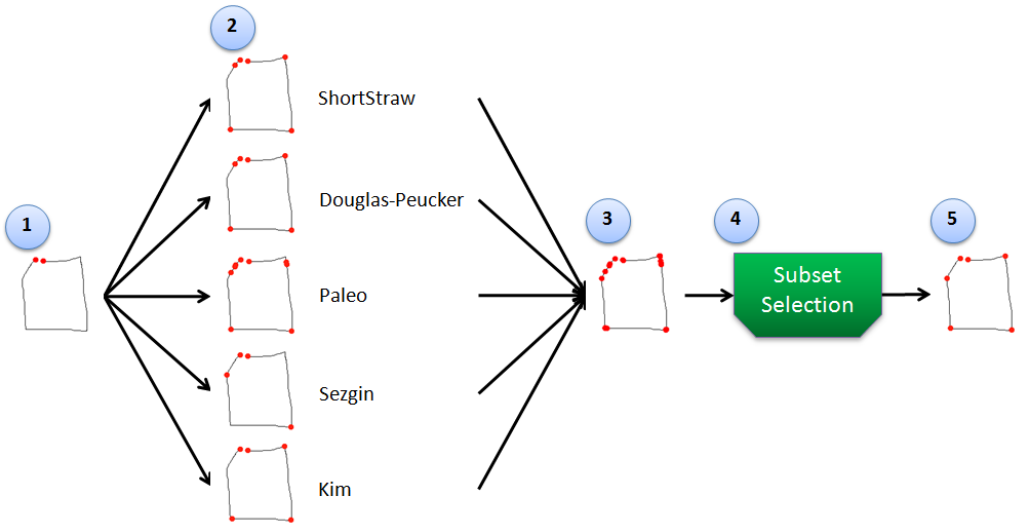

Figure 1: *Corner Subset Selection Process: (1) Take an input stroke, (2) segment the stroke using five different techniques, (3) combine the corners from all the techniques into one set, (4) pass the combined corner pool to our subset selection algorithm, and (5) output the best subset found.*

the direction graph to determine other basic primitives, such as circles and rectangles [YC03]. Sezgin et al. have a similar idea; they mention how recognition can be accomplished using simple, hand-crafted templates for primitives [SSD01].

Paulson's recent paper in IUI took segmentation and primitive recognition one step further. The primitive recognizer, called PaleoSketch, uses a trivial polyline segmentation along with finely-tuned heuristics in order to segment complex shapes [PH08]. 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). Improving polyline segmentation can therefore benefit complex segmentation.

3. Corner Subset Selection

We created our corner combination algorithm to segment polyline strokes. These polyline segmentation techniques can then be used to build multiple-primitive recognizers, such as PaleoSketch [PH08].

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" (CSS). A mean-squared error objective function is used in the algorithm. The overall process is outlined in Figure 1, and the following sections describe the steps in detail.

3.1. Step 1: Segmenters Used

Our algorithm starts by taking all of the corners found from five segmentation algorithms: Douglas-Peucker [DP73], ShortStraw [WEH08], PaleoSketch [PH08], Sezgin *et al.*'s [SSD01], and Kim and Kim's [KK06]. 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 succeptible to missing corners at obtuse angles and finding extraneous corners at segments that have noisy "bumps" (See Figure 1 segmentations).

The other two segmenters, Sezgin *et al.*'s and Kim and Kim's, are multiple-primitive segmenters that try to split strokes into lines, arcs, and curves. 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 five segmentation algorithms are combined together, and duplicate corners are removed.

3.2. 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 algorithms attempt to select the dimensions with the most information and discard those that contribute the most noise to the system.

In our case, the features of a stroke are the set of points we

use to describe that stroke. Our goal is to describe a stroke using the smallest number of points without losing any important information about the stroke. For a stroke composed of straight lines (since we are investigating polyline segmentation), the two endpoints and the internal corners are enough to completely describe the stroke, since we know there is a straight line connecting adjacent points. By running the segmenters described in the previous step, we generate a candidate set of points, and wish to discard any points that contribute noise (i.e., they are not actual corners).

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. If three features, when individually removed from a system, would reduce the system's performance by 20%, 10%, or 1%, we would remove the feature that only reduced performance by 1%. 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. Although SFBS is a greedy algorithm, which may be less accurate than a dynamic programming approach, the ability to "float" and reintroduce removed features helps alleviate some issues caused by greedy selection, and the algorithm is fast enough to run in real-time.

To determine the performance of the system, and hence choose 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. In the MSE Eqn. 1, 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.[†]

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

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, since they must always be kept.

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 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. 2). 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

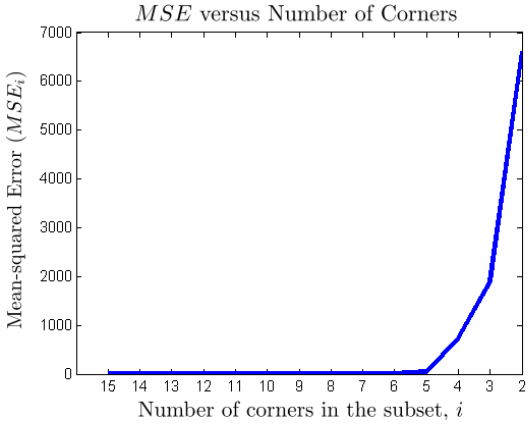

Figure 2: *Mean-squared error (MSE) of the stroke in Fig. 1. As corners are removed, the MSE has little change until critical corners are removed. In this example, the correct number of corners is 6, so critical corners are removed starting at $n_{corners} = 5$.*

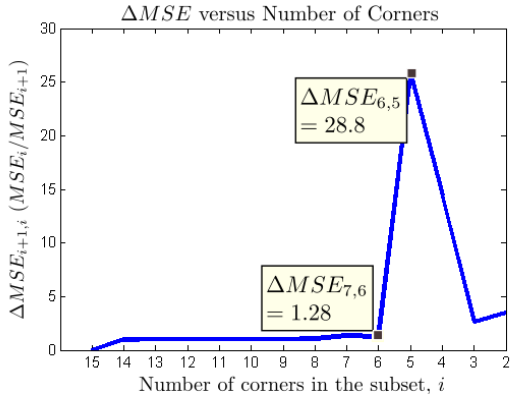

Figure 3: *Mean-squared error ratio described in Eqn. 2. The ratio plot is for the stroke in Fig. 1 whose MSE plot is shown in Fig. 2. The ratio is essentially a derivative of the MSE, which deviates only slightly until a critical corner is removed at $n_{corners} = 5$.*

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. 2). 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

[†] We use the term "performance" when discussing objective functions in feature subset selection algorithms. Error is the other side of the same metric – we want to maximize performance and minimize error.

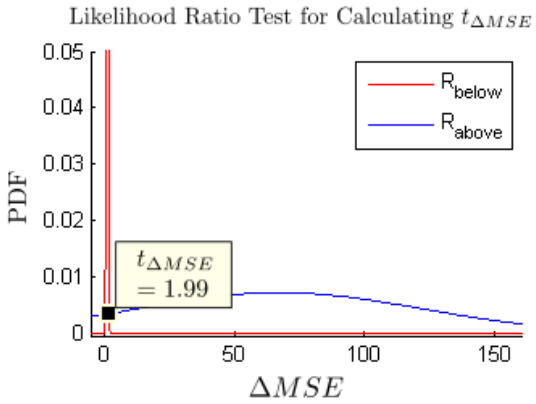

Figure 4: *Two Gaussian distributions are created from the R_{below} and R_{above} ratios for each set of training data. The optimal threshold is then found to be at the distributions' intersection; in this case, the threshold $t_{\Delta MSE} = 1.99$. Note that R_{below} is a much narrower distribution 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.*

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. 2). This change 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.

$$\Delta MSE_{i+1,i} = \frac{MSE_i}{MSE_{i+1}} \tag{2}$$

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. 3). Therefore, we found a threshold t_{elbow} where the first instance of $\Delta MSE > t_{elbow}$ would indicate that we are severely affecting the mean-squared error of the system and have already found the best subset.

3.3. 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 a possible threshold value or above a possible threshold value.

The median of each collection is found, and then the median absolute deviation (MAD) is computed (Eqn. 3). These median values are then substituted for the mean and standard deviation, respectively, when computing a Gaussian distribution for R_{below} and R_{above}.[‡]

$$MAD = median(|R_i - median(R)|) \tag{3}$$

In Eqn. 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 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. 4). 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 of r below the threshold as oversegmented, and all subsets after ΔMSE 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. 5.) After training, we found $t_{\Delta MSE} = 1.99$.

4. Results

Our algorithm was tested on a set of 244 polyline strokes drawn by 6 users. These strokes are a set of the 11 symbols (Figure 6), where each user drew the symbol up to four times. The data was gathered by the authors of the algorithm ShortStraw [WEH08], which is one of the algorithms we compare to. There are only 244 symbols instead

[‡] 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.

	CSS	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
Recall	1.00	0.985	0.989	0.903	0.912	0.868
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
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

Table 1: *Results for our corner subset selection algorithm (CSS) and the five 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.*

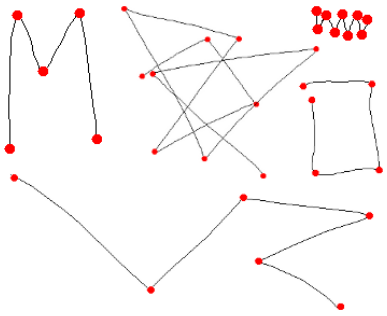

Figure 5: *A subset of the 216 random, single-stroke polyline shapes used for training. The polylines ranged from 2-line to 10-line shapes.*

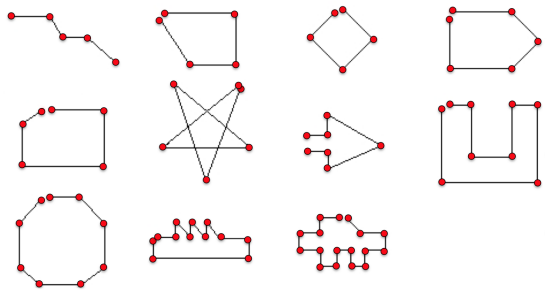

Figure 6: *Symbols used for testing. These 11 symbols were drawn up to 4 times each by 6 different users.*

of 264 ($4 \times 11 \times 6$) due to a bug in the data collection program, where the authors of ShortStraw noted that 20 of the drawn symbols experienced data corruption and could not be parsed.

These 11 symbols were also evaluated by Sezgin *et al.* [SSD01] and Kim and Kim [KK06]. This testing set is different than our training set.

The results for our CSS algorithm compared to the five individual algorithms are organized in Table 1. 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 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.

In the table, false positives are extraneous, unnecessary corners in a segmentation, whereas false negatives are missed corners.

Recall, also known as correct corners accuracy, is calculated by dividing the correct corners found by the total number of corners possible:

$$TP/(TP+FN) \tag{4}$$

Traditional accuracy is calculated as:

$$(TP+TN)/(FP+FN+TP+TN) \tag{5}$$

The standard view of accuracy suffers from the fact that there are many true negatives in segmentation, which are points in the stroke that are not corners. Therefore, accuracy is close to 100% for all segmenters.

Finally, we calculate all-or-nothing accuracy, which implies that the corner finding algorithm must segment a given stroke with no false positives or negatives in order for the stroke to be counted correct. All-or-nothing accuracy is calculated by summing every stroke that has no false positives and negatives, and then dividing that by the total number of strokes.

5. 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 any of the five combined segmenters.

All segmentation techniques perform well in recall and accuracy. With recall, algorithms that return every corner as correct will have a recall of 1.00, so this metric is not indicative of true performance, but it does indicate that the segmentation algorithm can find the necessary corners. Accuracy is a standard metric, but fails to highlight the performance differences between the algorithms. In segmentation, accuracy has a limit of 1.00 as the number of points in a stroke increases, due to this increasing the number of true negatives (Eqn. 5). It is for these reasons that we favor comparing segmentation algorithms using all-or-nothing accuracy.

All the correct corners must be found by the original techniques; our presented solution does not add any additional corners. Because CSS does not find any false negatives, this indicates that the five original algorithms do find the correct corners when pooling their segmentation results.

The errors in our CSS algorithm are due to thresholding errors, where the algorithm finds false positives because the ΔMSE has surpassed the $t_{\Delta MSE}$ threshold before the correct number of corners has been reached. This issue with thresholding errors is an important limitation of our combination algorithms. The No Free Lunch theorems we mentioned during our introduction state that if an algorithm performs better on one set of test cases, it will perform worse on others [Wol96, WM97]. Although we used this as a motivation for using multiple segmentation 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 we have demonstrated that the approach can be beneficial to stroke segmentation.

Overall, a lower $t_{\Delta MSE}$ will produce more false positives, while a higher $t_{\Delta MSE}$ will produce more false negatives.

CSS does take longer to run for each stroke, but 15.2 ms per stroke is still real-time for individual or small batch stroke segmentation. Each stroke could be segmented before a user could perceive a visual lag. This increase in time is due to running the original five algorithms, as well as the time required to perform feature-subset selection.

In the Introduction, we claim that there is not one silver bullet for corner finding. This claim also holds true for the CSS method. However, since CSS uses labeled training data to calculate the optimal $t_{\Delta MSE}$ for one particular dataset, it is easy to create a plethora of 'bullets' to use in different domains. The automated training makes CSS easier to customize than algorithms with hard-coded parameters.

The CSS algorithm could also be used as a tool for segmentation algorithm designers to identify areas for improvement in their algorithms. The algorithm in question will produce a set of candidate corners and CSS will select a subset of those corners. If the pruned corners follow some trends, the algorithm could be adjusted to eliminate those false positives. Additionally, CSS may select some corners from other algorithms that were not included in the original set. These extra corners could also be analyzed to determine certain types of corners that the algorithm is weak at detecting.

5.1. Gaining Intuition: Why Do We 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 run CSS with every point as a corner, thus eliminating the need to run multiple corner finders?

The authors tried this very technique, and the results from this implementation are not positive. After training, we found our $t_{\Delta MSE} = 2.718$. 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 drastically 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 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. Using existing algorithms to filter the initial set of points considered proves to be an effective method of finding an initial estimate for the corner set. Since the runtime of the SFBS is dependent on the total number of candidate corners, it is important to prune this set as much as possible. As shown in Table 1, applying other segmentation algorithms never filtered out actual corners during our tests, but only provides corners accepted by at least one individual algorithm.

6. Future Work

The combination techniques that we use to improve segmentation accuracy can also be extended to other recognition techniques. For instance, if different recognizers can output similar error values for recognizing low-level primitives such as arcs, ellipses, or squares, then our optimal chooser algorithm could select the best recognition interpretation based on a set of given criteria.

Similarly, 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.

7. Conclusion

Combining corner finder results through the individual corners or whole interpretations improves the accuracy of polyline segmentation. The all-or-nothing accuracy for our corner subset selection algorithm is much greater than any of the individual segmenters. This improvement to low-level accuracy benefits systems that rely on geometric languages to describe shapes.

8. Acknowledgements

The authors would like to thank Dr. Gutierrez-Osuna and the Sketch Recognition Lab members at Texas A&M University for their help with reviewing our paper and gathering data. This work is supported in part by DARPA CSSG and NSF 0757557.

References

[AD04] ALVARADO C., DAVIS R.: Sketchread: A multi-domain sketch recognition engine. In *Proceedings of UIST '04* (2004), pp. 23–32. 1

[Cha72] CHANG C. Y.: Dynamic programming as applied to feature subset selection in a pattern recognition system. *ACM Annual Conference 1* (1972), 94–103. 2

[DP73] DOUGLAS D. H., PEUCKER T. K.: 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 10*, 2 (1973), 112–122. 2, 3

[HD02] HAMMOND T., DAVIS R.: Tahuti: A geometrical sketch recognition system for uml class diagrams. In *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding* (Palo Alto, California, United States of America, 2002), Association for the Advancement of Artificial Intelligence (AAAI) Press, pp. 59–66. 1

[HD05] HAMMOND T., DAVIS R.: Ladder, a sketching language for user interface developers. *Elsevier, Computers and Graphics 28* (2005), 518–532. 2

[HS92] HERSHBERGER J., SNOEYINK J.: Speeding up the douglas-peucker line-simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling* (1992), pp. 134–143. 2

[KK06] KIM D., KIM M.-J.: A curvature estimation for pen input segmentation in sketch-based modeling. *Computer-Aided Design 38(3)* (2006), 238–248. 2, 3, 6

[KS04a] KARA L. B., STAHOVICH T. F.: An image-based trainable symbol recognizer for sketch-based interfaces. In *Making Pen-Based Interaction Intelligent and Natural* (Menlo Park, California, October 21-24 2004), AAAI Fall Symposium, pp. 99–105. 2

[KS04b] KARA L. B., STAHOVICH T. F.: Sim-u-sketch: A sketch-based interface for simulink. In *AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces* (2004), pp. 354–357. 1

[MG63] MARILL T., GREEN D.: On the effectiveness of receptors in recognition systems. *IEEE Transactions on Information Theory* (1963), 11–17. 2

[OLM*09] OCONNELL T., LI C., MILLER T., ZELEZNIK R., J.J. LAVIOLA J.: A usability evaluation of algosketch: a pen-based application for mathematics. In *sbim* (2009), pp. 149–157. 1

[PH08] PAULSON B., HAMMOND T.: Paleosketch: Accurate primitive sketch recognition and beautification. In *Proceedings of the International Conference on Intelligent User Interfaces* (Canary Islands, Spain, 2008), Association for Computing Machinery (ACM) Press, pp. 1–10. 2, 3

[SG80] SKLANSKY J., GONZALEZ V.: Fast polygonal approximation of digitized curves. *Pattern Recognition 12*, 5 (1980), 327–331. 2

[SS93] SIEDLECKI W., SKLANSKY J.: *On automatic feature selection: In Handbook of Pattern Recognition and Computer Vision*. World Scientific Publishing Co., 1993. 2

[SSD01] SEZGIN T. M., STAHOVICH T., DAVIS R.: Sketch based interfaces: Early processing for sketch understanding. In *Proceedings of 2001 Perceptive User Interfaces Workshop (PUI'01)* (Orlando, FL, November 2001). 2, 3, 6

[WD84] WALL K., DANIELSSON P.: A fast sequential method for polygonal approximation of digitized curves. *CVGIP: Graphical Models and Image Processing 28*, 2 (November 1984), 220–227. 2

[WEH08] WOLIN A., EOFF B., HAMMOND T.: Shortstraw: A simple and effective corner finder for polylines. In *SBIM '08: Proceedings of the 5th Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2008), pp. 33–40. 2, 3, 5

[WM97] WOLPERT D., MACREADY W.: No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* (1997), 67–82. 2, 7

[Wol96] WOLPERT D.: The lack of a priori distinctions between learning algorithms. In *Neural Computation* (1996), pp. 1341–1390. 2, 7

[WPH09] WOLIN A., PAULSON B., HAMMOND T.: Sort, merge, repeat: An algorithm for effectively finding corners in hand-sketched strokes. In *Proceedings of the Eurographics Symposium on Sketch-based Interfaces and Modeling (SBIM)* (2009), pp. 93–99. 2

[YC03] YU B., CAI S.: A domain-independent system for sketch recognition. In *GRAPHITE '03: Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* (Melbourne, Australia, 2003), Association for Computing Machinery (ACM) Press, pp. 141–146. 2, 3