A review of text detection using stroke width constancy

David Ding

***Abstract*—In this paper, we explore using stroke-width information for text detection. We follow the general approach of Epshtein et al. but describe the process in more detail and explore different heuristics for refining our algorithm. We also describe methods of automatically detecting text for pictures with diverse image attributes.**

Text detection in natural images is an interesting computer vision problem with many important applications. For instance, to obtain street address information, Google takes street view images and tries to identify the addresses in the image. Text detection is also useful in building an autonomous robot that can navigate the world, enabling the robot to read signs, which convey a lot of information. Finally, text detection could be another way for users to input information on a smartphone (i.e. credit card numbers).

This problem is complicated by the fact that text could exist in many different forms in the world. There could be large text or small text, appearing in all sorts of different colors and illuminated in different ways. The text could also be on many different backgrounds, including non-flat ones. Text for different languages all look different, and indeed even within a language, text could appear in different font faces. Moreover, many non-text elements could easily be confused as text. For instance, a straight line could be interpreted as an "I" or "l", a circle as an "o", and a cross as a "t". Finally, for many practical applications, text detection must be efficient. For instance, a smartphone application allowing users to input credit card information using the camera must provide real-time feedback on whether the captured image contains a valid text region. If the text detection algorithm is too slow, the feedback is delayed, resulting in lag and poor user experience.

Epshtein et al. introduced an efficient method of finding text in images by considering stroke width[2]. In their paper, they introduced the stroke-width transform (SWT), which computes for every pixel the width of the stroke that the pixel belongs to (if any). They make the observation that text has relatively constant stroke width, whereas natural elements such as foliage has much more variability. After calculating the stroke width, they find regions with relatively constant stroke width, and then in a filtering stage, eliminate false positives with a variety of criterion. Finally, individual letters are chained together to find lines of text. Using this method, they got a precision of 0.73 and recall of 0.6.

This paper attempts to replicate the results of Epshtein et al., while coming up with new heuristics. The organization of the paper is as follows. Section I discusses previously published methods of text detection. Section II discusses our approach (modeled closely after that of Epshtein et al.) to finding text. This section is subdivided into Section II-A, which reviews the method of computing stroke width, Section II-B, which discusses heuristics that were used to eliminate false text elements, and Section II-C, which discusses our approach to chaining the letters together. Finally, Section III discusses our results.

We were not able to achieve the performance that Epshtein et al. achieved, mostly likely because our chaining algorithm is not as robust. However, we did find new heuristics that significantly improved upon the set of heuristics that were discussed in Epshtein et al. Moreover, we found that choosing parameters and getting the algorithmic details correct is difficult and can have significant impact on performance; we hope that our more detailed discussion would aid future implementers of text-detection algorithms.

I. Related Works

The standard dataset for the text localization probelm is the IDCAR 2003 dataset, a dataset with 258 training images (used to tune parameters in algorithms) and 251 test images [4]. These images come with human annotated bounding boxes, generally for each word but some times for the entire text region. These bounding boxes are fairly comprehensive: the recorded bounding boxes contain not only foreground main text elements, but also text in the background and small/almost illegible text. Performance on the test set is measured by precision, which measures how precisely the bounding boxes returned by the algorithm frames the text, and recall, which measures how much text area is included in the bounding boxes returned by the algorithm. To combine these to measures, Lucas et al. recommended

using f, the harmonic mean of the precision and recall ($\frac{2}{\text{precision}^{-1}+\text{recall}^{-1}}$). The rigorous definitions of precision and recall are given in Section III.

As mentioned in the introduction, Epshtein et al. introduced the idea of using stroke-width as an initial indicator for where text is. On the standard IDCAR 2003 dataset, they were able to achieve precision of 0.73 and recall of 0.6, with an overall f measure of 0.66. Most importantly, their algorithm is computationally efficient; compared to the second best method ($f = 0.62$), the algorithm ran more than 14 times faster.

Many other papers attempted to refine the result of Epshtein et al. For example, Yao et al. took the basic algorithm and added a few steps in the filtering and joining phases [6]. In the filtering phase, they used the basic heuristics introduced by Epshtein et al. and added a trained classifier (random forest) on a feature vector derived from the iamge that is scale invariant and rotation invariant. In the chaining phase, after the chains are formed, they use another random forest to accept or reject the chains. Their paper achieved similar results as Epshtein et al. on the IDCAR data set, but performed significantly better on other datasets containing images with text in arbitrary orientations.

Neumann and Matas consider text detection and recognition in a unified algorithm [5]. To identify text regions, they find a set of Maximally Stable Extremal Regions, under the assumption that regions containing text are extremal in some scalar projection of pixel values. Then, they consider sequences of such regions and calculate the probability that the sequence contains text (this probability is calculated by a classifier trained on Windows font data), returning the sequences that achieve local maxima in this probability. Their algorithm performs less well than Epshtein et al. on the problem of text localization ($f = 0.57$), but they were able to achieve robust reading on the images also.

II. Method

A. Stroke-Width Transform

Stroke widths are calculated in the manner described in Epshtein et al. [2]. The algorithm takes an input image and returns an out image of the same size as the input image, where the value of each pixel is the width of the stroke that the pixel belongs to (or some max value if the pixel doesn't belong to any strokes). The basic idea of the algorithm is to find parallel edges and to assume that such edges appear as the edges of a stroke. To find edges, we use the Canny edge detector. Then, at each edge pixel, we compute the gradient, which points from the lower intensity side of the edge to the higher intensity side and is perpendicular to the edge. If the stroke is brighter than the background, we follow the gradient until we hit another edge pixel, and if the stroke is darker than the background, we follow the opposite of the gradient. After we hit another edge pixel, we compute the gradient there, and if the two gradients are roughly opposite of each other, that means that the two edges are roughly parallel, and we can set the stroke width for every pixel along the path to be the minimum of the current value for the stroke width at that point and the distance between the two parallel edges. We repeat this for every edge pixel until all stroke width are computed.

The above steps will compute a stroke width for every pixel, but at corners, stroke widths could be misbehaved, since the two edges of the strokes change direction at different paces. To smooth out the anomalous effect of corners, we find the strokes again by looking at edge pixels and following the gradient. Now, we compute the median of the stroke width of pixels along the path, and replace the stroke width of each pixel with this median if the current value is larger than the median.

There are numerous details with this basic algorithm to tune. First, the search direction is flipped depending on whether we are looking for dark text on light background or light text on dark background. To detect text in both instances, we must try both possibilities. Note that we cannot have a single SWT hold the stroke width of both light strokes and dark strokes, since in images, light and dark strokes typically alternate (caused by the spaces between the letters). Hence, if we set the stroke width at each pixel to be the minimum of the light and dark stroke, text would appear as an indistinct blob as both the letters and the gaps of the letters get classified as a stroke. This decreases the overal performance of the text detection algorithm. Moreover, it does not suffice to choose one SWT per image, since the same image could contain both light and dark stroke.

Second, the dependence of the SWT on intensity values necessitates a careful choice of which intensity values to feed into the algorithm. Depending on lighting conditions and the color of the text, strokes could be more visible in one channel versus another. In order to extract the best quality strokes, we attempt to compute the SWT on the red, green, blue, and grayscale channels. By considering all the channels, we improved the f-measure from 0.44 to 0.49. The disadvantages of this approach is extra computation time and the required extra step of removing duplicate bounding boxes.

Third, the results of the SWT depends greatly on the threshold values chosen for the Canny edge detector. If the edge detector is too sensitive, lots of extraneous

(a) (b)

Fig. 1: The stroke-width transform of a sample image. Figure 1a corresponds to searching along the gradient, and Figure 1b corresponds to searching opposite the gradient.

edges will be returned, which can interfere with the process of finding a parallel edge. Moreover, the algorithm will take more time to run, since SWT runs in time linear in the number of edge pixels. On the other hand, if the edge detector is not sensitive enough, smaller text elements will not be found, and performance on blurry images will be poor. In our experiments, we found that an optimal result is achieved if we first smooth the image with a Gaussian filter of standard deviation 3 pixels, and then apply the Canny detector with lower threshold approximately 0.06 and upper threshold approximately 0.2. As seen in Figure 4 though, this can cause blurry and small text to be missed.

Fourth, since edges of a stroke are rarely perfectly parallel, we must define an acceptable error threshold. Epshtein et al. recommended that the gradients to the two edges should form an 180 degree angle with an error of up to 30 degrees. In our experiments, however, we found that an error threshold of 30 degrees caused us to miss a significant number of strokes in text. Using 60 degrees for our threshold, we got $f = 0.49$, as opposed to the 0.42 we got from 30 degrees.

B. Finding letters

After computing the stroke-width, we seek to identify letters as regions of constant stroke width. We define a graph G where the vertices are the pixels in the image. An edge connects two pixels if they are adjacent and the ratio of their stroke width is between $\frac{1}{3}$ and 3. We then use depth-first search to find connected components in this graph. The connected components returned by this algorithm will have similar stroke width locally. However, since two adjacent vertices could differ by up to a factor of 3 in their stroke width, as we walk from one end of the component to the other, we could encounter significant variance in stroke width. We thus also enforce a global constraint on stroke width deviation: each component is valid only if the standard deviation of the stroke width across the component is at most half the mean stroke width.

Fig. 2: Bounding boxes for individual components.

Constancy of stroke-width eliminates most of the connected components, leaving behind a small set of remaining bad components that must be filtered out with other heuristics. The simplest but most effective heuristic is the size of the component. Because of noise and spurious edges, there will be lots of components with very few pixels. These components are unlikely to be text, so we eliminate them. Another heuristic mentioned in the paper is the aspect ratio; we eliminate all regions whose aspect ratio is greater than 10. This eliminates lines, which occur frequently in images but otherwise

could easily be confused as text. One heuristic mentioned by the paper that we did not use was that the median stroke width is at most 10% of the diamater of the connect component. We found this measure rejected many valid text boxes, for example text with bold font.

The above heuristics were all first discussed by Epshtein et al. [2]. We also added a new heuristic, distinguishability from the background, that we found significantly improved detection accuracy. Typically, in images, the pixels for the letter differ significantly in intensity from the background, whereas for strokes that were calculated from spurious edges, the distinction is less obvious. Hence, we require the median intensity of pixels in the component to be at least 0.13 higher than the median intensity of the other pixels in the bounding box (in a scale where the maximum intensity value is 1). This heuristic alone raised the f measure from 0.36 to 0.49 by increasing precision from 0.32 to 0.52. The end result of the filtering is shown in Figure 2.

C. Chaining

After identifying components that likely correspond to letters, we chain together these components to form lines of text. As noted by Epshtein et al., since letters rarely appear isolated, line formation provides yet another value clue on whether a component corresponds to text or not [2].

To chain together the bounding boxes, we consider all pairs of bounding boxes to find candidates to join together. We join bounding boxes only if

- The median stroke width of the two components differ by less than 50%.
- The median intensities of the two components differ by less than 0.11.
- The ratio of the heights of the two boxes is between 0.5 and 2.
- The distance between the bounding boxes is at most half the height of the two boxes.
- The slope of the line connecting the centroids of these boxes is less than 0.25.

These criterion are usually satisfied by usual text. Consecutive characters tend to have similar stroke width, intensities, and size. They tend to be close together, and for English and most Western languages, to lie in horizontal lines more frequently than vertical. We greedily chain together the bounding boxes until we could find no more boxes to chain. This algorithm is semantically identical to the algorithm provided by Epshtein et al. [2].

This entire pipeline produces a series of bounding boxes representing words. Recall, however, that when we were computing stroke-width, we forced a choice of dark text on light background or vice versa, and we also chose a specific color channel to work with. This choice was not covered in detail by Epshtein et al., so here we provide our own algorithm. To eliminate this arbitrary choice and to improve our overall accuracy, we repeat the entire pipeline for the gray, red, blue, and green channels for both dark on light and light on dark, a total of eight iterations. This will in general give many duplicate bounding boxes, as well as significantly different bounding boxes that nevertheless try to capture the same text. For instance, we noted that the gaps between letters could be interpreted as strokes, and in many cases, these "strokes" can be classified as letters. However, because these strokes satisfy far fewer properties than the strokes corresponding to the actual text, and because the size of these strokes differ in general from the size of the text (they could extend to some other boundary that frames the text for example), the quality of these bounding boxes is often significantly poorer. We thus need an algorithm that can robustly detect duplicates and choose the best bounding boxes.

Our algorithm judges the quality of bounding boxes by keeping track of a vote while merging the bounding boxes for each component. The vote of a final output bounding box is the number of components that were merged to form this box. We then find bounding boxes with significant overlap (40% of the smaller box), and selects the box with the higher vote.

There were multiple parameters for our entire algorithm, such as the cutoff angle for SWT, the thresholds for the Canny edge detector, and the threshold for distinguishability from the background. To choose the optimal such parameters, we wrote a Python program that takes in Matlab files, chooses different values for the parameters, and runs the modified Matlab files on the training set, recording the values for each parameter that maximizes the f measure. Since we had so many parameters, we did not do a full grid search over all combinations of parameters but optimized each parameter one by one.

III. Results

Figure 3 shows the result of our text detection algorithm on multiple sample images from the IDCAR 2003 data set [4]. As we see, the algorithm successfully handles variations such as text size and color. We also see that the bounding boxes that the algorithm draws are fairly tight, and there are few false positives. However, the algorithm has trouble detecting some smaller text fragments that appear in the images. For example, in

(a) (b) (c)

(d) (e) (f)

Fig. 3: Text detection in sample images from the IDCAR 2003 data set.

Figure 3b, while the large characters "THE RAB BUTLER BUILDING" are successfully detected, the sign in the background is not. In Figure 3d, the url on the pen is not detected, and in Figure 3e, many smaller text elements were lost. The primary reason for this is the blurring step, which adversely affected edge detection for small text and hence resulted in inaccurate stroke width determination. This can be seen in Figure 4, where the edge detector failed to find edges for several letters that appeared in the image. However, without blurring, many noise elements become classified as text. In the future, it might make sense to run this algorithm on various successively blurred images (i.e. for several images in the Gaussian pyramid) to detect text at all scales.

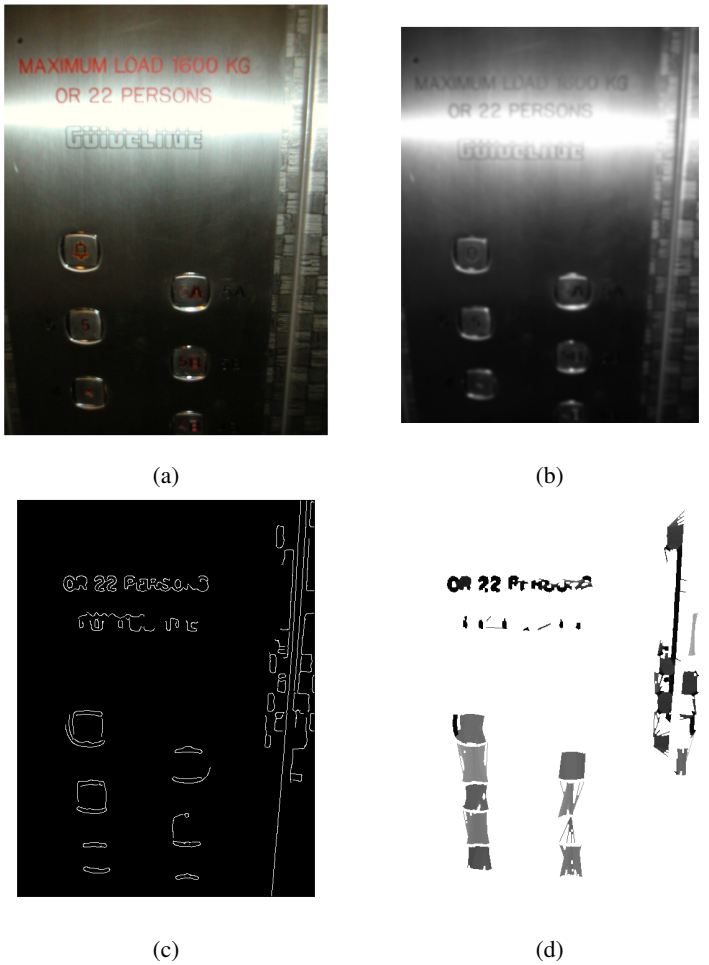
Figure 3e shows another deficiency with our text detection algorithm: it was unable to group the "R" with "CD". The reason for this is that the bounding box for the preceding hyphen had too small of a height and hence was not chained with the "CD". Moreover, the distance between the "R" and "CD" was too great for the chaining algorithm to consider them. Because "R" was an isolated component, the algorithm eliminated it. In general, our algorithm will fail to detect one character words. Epshtein et al. solved this problem by allowing for components farther apart to be chained together, so they drew bounding boxes around lines rather than words [2]. After finding the bounding boxes for the lines, they then separate the line into words by examining the spacing between words to determine the intra-word letter distance versus the inter-word letter distance; this step is not necessary in general, but was necessary in order to compare with the IDCAR 2003 data set. For simplicity, we did not implement this step, and since labled bounding boxes in the IDCAR 2003 dataset are bounding boxes for words, we had to tune our algorithm to draw bounding boxes around words to allow for accuracy comparisons.

We measured the quality of our text detection algorithm using the standard precision, recall, and f-measures for the IDCAR data set. These values are defined as follows. If R_1 and R_2 are two rectangles, define $m_p(R_1, R_2)$ to be the area of the intersection divided by the area of the minimum bounding box containing box rectangles. Now, if S is a set of rectangles and R is a rectangle, define

$$m(R, S) = \max\{m_p(R, R')|R' \in S\},$$

the largest match of R with a rectangle in S. Then, if E denotes the estimated bounding boxes for words, and T denotes the true bounding boxes that were labeled in the data set, we define

$$\text{precision} = \frac{\sum_{R \in E} m(R, T)}{|E|}$$
$$\text{recall} = \frac{\sum_{R \in T} m(R, E)}{|T|},$$



(a) (b)

(c) (d)

Fig. 4: A sample image for which text detection failed to identify any bounding boxes. Figure 4a shows the original image, and Figure 4b shows the blurred grayscale version. Figure 4c is the result of the Canny edge detector using the thresholds as described in Section II-A, and Figure 4d is the result of the SWT. Note that the first line of text is totally missing in the output of the edge detector.

and f to be the harmonic mean,

$$f = \frac{2}{\text{precision}^{-1} + \text{recall}^{-1}}.$$

The results for our implementation is listed in Table I, along with results from several other text-detection algorithms. Our algorithm did not perform as well as state of the art systems, but it did perform at a reasonable ballpark of the latest results.

Algorithm	Precision	Recall	f
Our system	0.53	0.46	0.49
Yao et al. [6]	0.69	0.66	0.67
Epshtein et al. [2]	0.73	0.6	0.66
Yi et al. [7]	0.71	0.62	0.62
Chen et al. [1]	0.6	0.6	0.58
Zhu et al. [3]	0.33	0.4	0.33
Kim et al. [3]	0.22	0.28	0.22

TABLE I: Results on the IDCAR 2003 dataset.

Our algorithm was implemented in MATLAB for ease of implementation and to leverage the debugging tools available. Unfortunately, in the computation of the stroke-width transform, we were unable to take advantage of the vectorized operations that Matlab offers, since the algorithm is fundamentally iterative on each edge pixel. Moroever, we had to implement our own connected components algorithm because our definition of connectedness is fuzzy and because we needed a list of components and the vertices in each component. Because we were unable to use the optimized vectorized functions

in MATLAB, our code ran slowly, making parameter optimization a hassle. We applied optimizations, such as removing `sub2ind` and `ind2sub` calls (which took up almost half the time), and we also scaled the image down so that the the maximum dimension is at most 1080 pixels, which significantly helped the performance of the algorithm without significantly degrading the quality of the output. In the future, we would implement the algorithm in C++ using the opencv library, since the bottleneck in development was testing the effect of various changes on accuracy.

IV. Conclusion

We implemented the algorithm described in Epshtein et al. and were able to get comparable but worse results. We described in detail choices that were made in the design of the algorithm and the impact they had in the final text detection accuracy. Finally, we described methods of improving accuracy by looking at different color channels, and proposed methods of finding text at different scales by using different levels of the Gaussian pyramid.

References

[1] X. Chen and A. Yuille. Detecting and reading text in natural scenes. In *Proc. CVPR*, 2004.

[2] B. Epshtein, E. Ofek, and Y. Wexler. Detecting text in natural scenes with stroke width transform. In *Proc. CVPR*, 2010.

[3] S. M. Lucas. Idcar 2005 text locating competition results. In *Proc. IDCAR*, 2005.

[4] S. M. Lucas et al. Idcar 2003 robust reading competitions. In *Proc. IDCAR*, 2003.

[5] L. Neumann and J. Matas. Text localization in real-world images using efficiently pruned exhaustive search. In *Proc. IDCAR*, 2011.

[6] C. Yao et al. Detecting texts of arbitrary orientations in natural images. In *Proc. CVPR*.

[7] C. Yi and Y. Tian. Text string detection from natural scenes by structure-based partition and grouping. *IEEE Trnas. IP*, 2011.