Text Detection on Nokia N900 Using Stroke Width Transform

Saurav Kumar, Andrew Perrault
{sk2357, arp86}@cornell.edu
CS4670 – Computer Vision
December 14, 2010

Introduction

We implemented the stroke width transform (SWT) text detection algorithm from [1] on the Nokia N900. The purpose of the algorithm is to segment out likely regions of text from an image, in order to clean the input of an optical character recognition algorithm. We followed the general framework of [1], but deviated from it in several places. Overall, our implementation is not as successful as [1], as we encountered several problems that are not mentioned in that paper.

Figure 1: stages of the SWT algorithm. a) original image b) Canny edge detection c) stroke width detection d) connected components e) chaining f) final detected text

Approach

In this section, we briefly outline the algorithm in [1] and our alterations to it. The first stage of processing aims to extract the stroke width at every pixel. To do this, the key insight is that letters have roughly parallel sides. To find the edges of the letters we use cvCanny. Then, we calculate the gradient at every edge pixel. Depending on whether the text is light-on-dark or dark-on-light, the gradient will either point into the letters or out of them. Our algorithm depends on the gradient pointing into the letters, which is why it is not possible to detect light-on-dark and dark-on-light text simultaneously, but it can be run twice on

the same image to achieve that effect. The original mentions some capability for doing this, but it is difficult to tell how it is implemented.

We follow the gradient of each edge pixel until we hit another edge pixel, where we check if the gradient is pointing in roughly the opposite direction. Here is the first point where we differ from [1]. We allow the gradient directions to be up to 90° apart, instead of 30°. Figure 2 is a SWT image with the tolerance of 30°. There are several points where the letters have sides that are not near parallel: the joins of the "A", "3", "x" and "2."

Figure 2 (top): low tolerance stroke labeling fails at joins of letters. Figure 3: results of the completed stroke labeling stage

If the ray satisfies these conditions, each pixel along it is marked with the minimum of its previous stroke width and the length of the ray. After all rays are marked, we go through each ray and assign to each pixel whose stroke width is higher than the ray's median the median value. This is done to reduce the effect of perpendicular joins, which generally will have very high stroke widths. Figure 3 shows the results of our stroke width labeling process, where a darker color corresponds to a shorter stroke with.

The next stage of the algorithm is forming connected components. We do this by considering two pixels as neighboring if the ratio of their stroke widths is between 1/3 and 3. This is different from the original algorithm, which appears to do some kind of tracing of the rays formed in the first stage. We used the boost graph libraries for this.

We use some of the heuristic rules from [1] to filter the connected components. The most important rule is that we filter components based on the ratio of their SW variance to their SW mean. Letters tend to have lower SW variance. Another important rule is checking if a component's bounding box contains more than two other components' centers, which eliminates signs and frames that otherwise have very uniform stroke width. We find a minimal-size bounding box by rotating the component and use this to filter out long, thin components, such as lines separating sections of text.

There are two rules from the paper we did not use. The first is checking whether the height of a component is greater than 300 pixels, which was not working well on images taken with the phone. The second is checking the ratio of the diameter of a component to its SW mean. This rule may be useful, but we weren't having many problems with false positives so we decided to focus on other parts of the process instead.

The final stage of the algorithm is chaining, which has two parts. The first is to use heuristics to find eligible letter pairings. We check every pair of letters and if it meets certain conditions, we label it as an eligible pair. The conditions are similar median stroke width, similar font height, similar color and center-center distance less than three times the width of the wider letter. After these pairings have been collected, they are formed into chains in the following way: the pairs are sorted by distance and then merged if they share an end (i.e. a component) and have a similar direction. Chaining ends when no more merges are possible.

In the chaining stage, specifically in merging, we get vastly different results from the paper. In a single word in a series of lines of text, letters will often be paired with six other components, two forward, two back and one above and below. This results in many boxes drawn around each word; as even when the correct box has been formed, there are pairings left over. Also, chains will form vertically. However, we were able to use our word markings to convert text to a binary image that could easily be read by OCR.

Results

Figure 4 shows the results of our algorithm on high-resolution images taken with camera. Figure 5 shows the results on lower resolution camera images. It is possible to take higher resolution pictures on the camera, but the processing time for the stroke width stage becomes quite high, in the minutes, which is not acceptable for a mobile app. The stroke width labeling is by far the most computationally intensive step of

the algorithm, and it also requires the most resolution. To turn this into an effective mobile app, we would propose the following pipeline: extract the Canny edges on the phone, blur and downsample the image significantly, and then send the two pieces to the server for further processing. The edge detection is what requires the most image resolution, but it is also quite a fast step. Since we will blur before calculating the gradients, it makes sense to blur before sending the images.

An informal survey of our results on the computer (in this section as well as the appendix) shows precision of 97% and recall of 85%, calculated per character. The phone images are a much more limited set because of resolution, but among those we collected, there was only one false positive (around 99% precision) and the recall was 100%.

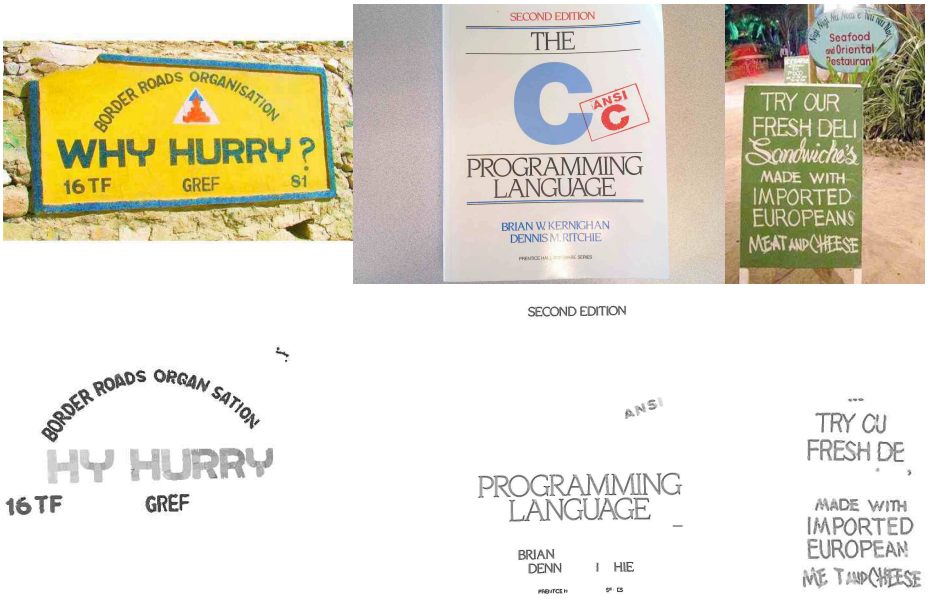


Figure 4: Results of SWT on high-resolution images.

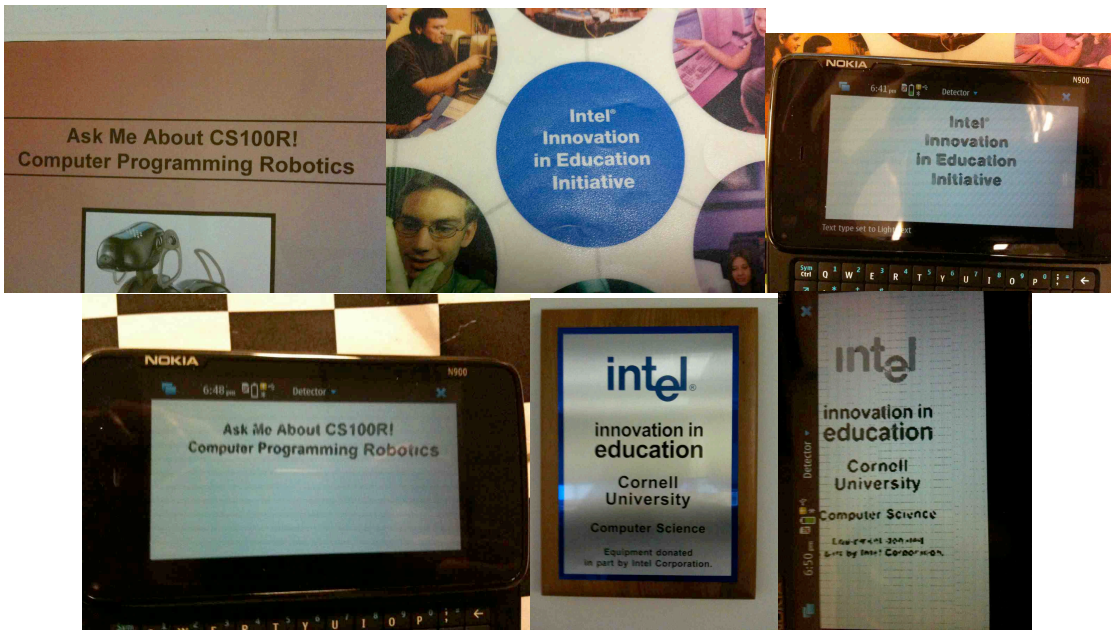
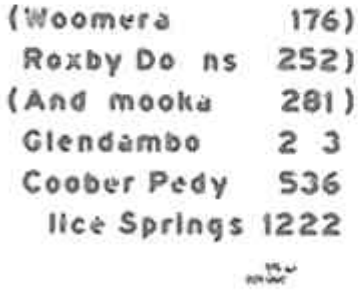


Figure 5: Results of SWT on low-resolution images.



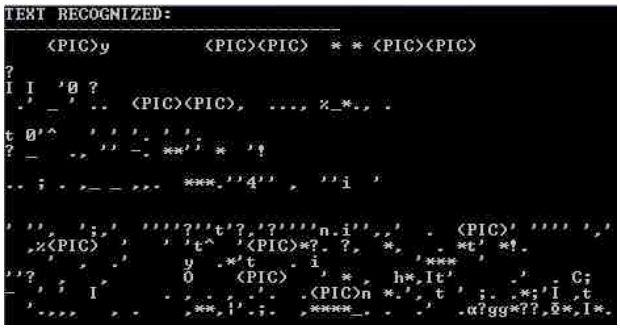


Figure 6a: character recognition from original image and from SWT-detected text

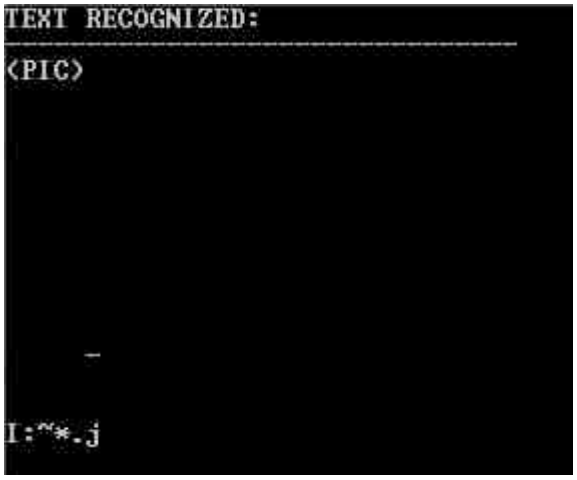


Figure 6b: character recognition from original image and from SWT-detected text

We ran character recognition with the OCR System developed at ASPIRE [2]. We used the Aspire trial version dynamic link library. It can be seen from the results that the OCR system doesn't perform well on the natural images due to the fact that majority of OCR engines are designed for scanned text and so depend on segmentation which correctly separates text from background pixels. This process is simple for scanned text documents, but much harder in natural images due to a wide range of imaging conditions, such as color noise, blur, occlusions, etc. thus drastically affecting the performance of OCR algorithms.

Conclusions/Future Work

The most important improvement to the algorithm would be robustness to stray strokes. In the current implementation, the extra lines connecting to the "T" and "k" of "thank" in Figure 7 would cause those letters to become unreadable. Likewise, the stray line above "CHRISTIAN" causes a great deal of trouble in filtering and chaining the components. These lines need to be removed without interfering with the good components. The sign image has this problem to an even greater extent.

Figure 7**:** Stray strokes in stroke width labels

The lettering becomes attached to outline of the sign, causing large amounts of good text to be thrown out. Because the strokes are not separate components, we would have to do this detection in the stroke width labeling stage, before the components themselves are filtered. The sign image also shows a second problem,

inability to detect light and dark text in the same image. If the algorithm were truly robust in each case, the results of could be combined.

Figure 8: Variable letter thickness failure cases

Figure 8 shows our problems with detecting stylized text. Besides the problem of "loose" strokes, the text has variable thickness, and thus a high variance. One way of solving this would be to allow stroke width to vary slowly across the image, not using just one score, like variance. In our current model, text within one standard deviation of the mean stroke width must be within ±25% of the mean value. This is clearly not the case for cursive writing, which probably doesn't fit to a normal well on a letter-by-letter basis.
The letters also tend to overlap, which causes them to be thrown out.

There are many cool applications for OCR in natural scenes. One would be creating a tag cloud of Flickr. Another would be an augmented reality application that tags store signs with links to websites or something similar.

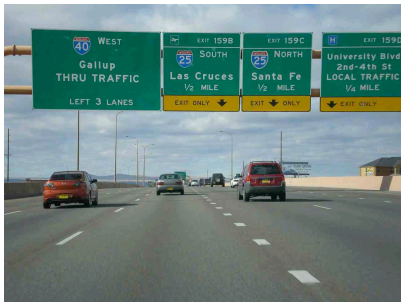
Division of Work

Saurav implemented the stroke width labeling stage, prepared the presentation, set up the OCR Software environment for comparison and wrote about the OCR software part of report. Andrew did components, chaining, phone implementation and wrote the other sections of the paper.

References

[1] Epshtein, B., Ofek, E., and Wexler, Y. Detecting text in natural scenes with stroke width transform. *CVPR '10* (oral), 2010.
[2] Aspire OCR: http://asprise.com/home/

Appendix



Additional text detection on the computer (above) and phone.