Multi-Class Thresholding of Document Images

Algorithm Engineering 2022 Project Paper

Tristan Conrad
Friedrich Schiller University Jena
Germany
tristan.conrad@uni-jena.de

ABSTRACT

Low-quality scans or images of text documents can often be hard to read for humans, and hard to parse for OCR programs. To remedy this, the image is often cleaned up with a program before further use. However, although most text documents are purely in black and white, there are also some which utilize various gray values to display graphics for visual aid. Simply sorting into text and non-text via a threshold, even a dynamic one, will lose this information. This paper discusses an alternative approach to finding and cleaning up such groups of gray.

KEYWORDS

document image, thresholding method

1 INTRODUCTION

1.1 Background

In low quality scans of images, the main problem for most algorithms seeking to clean up the image presents itself in finding an appropriate threshold. Due to the lighting changing across the image, what may be clearly a white background in one part of the image may slowly shift to an extremely dark gray, almost as dark as the text, in another part.
The algorithm presented attempts not to find a threshold value, but to exploit the gradual shift in color to join together groups of pixels that form a gradient across the image, unifying their values and achieving a split between text and background in that fashion. By allowing multiple possible colors as the output, graphics using gray colors can also be cleaned up and outputted.

1.2 Related Work

The work of Bataineh et al. on adaptive thresholding allows for the binarization of images with an extremely robust thresholding formula that is calculated based on a window of surrounding pixels, achieving great accuracy even with extremely low contrast images. [1]
Bradley and Roth also devised a similar algorithm with the explicit goal of dealing with changes in lighting that form a gradient across the image. [2]
While both of these thresholding methods are powerful tools for the binarization of documents, my goal is to increase the number of possible output values in such a process. This paper borrows the main idea in the above works of viewing the pixels locally and grouping them based on local thresholds.

1.3 Outline

Section 2 will describe the algorithm in detail, providing small examples of the methods used. Section 3 briefly analyzes the performance of the sample implementation. Section 4 will evaluate both the algorithm and the implementation and propose some possible improvements on the algorithm presented. Section 5 closes with some final thoughts on the algorithm and the implementation.

2 THE ALGORITHM

2.1 Internal Representation of the Image

The program first reads an input image file and stores it as an array of RGB values for each pixel. The sample implementation reads PPM files for simplicity.
After reading each pixel, it also calculates a grayscale value, upon which further calculations will be performed.
A minimum and maximum grayscale value across the entire image can simultaneously be calculated and stored.

2.2 Splitting the Pixels into Blocks

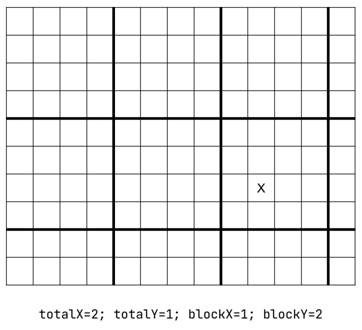

Figure 1: Splitting a 13×10 image into blocks with `blockSize` 4. The pixel marked with an ’X’ is assigned in the block’s pixel array with `block.pixels[2*4+1] = pixels[1*13*4+2*4+2*13+1]`, or in other words: the 9th index of the local block pixel array is assigned the pixel at the 87th index in the global pixel array.

To create an easily threadable program, the array of pixels is split into an array of blocks of pixels instead. The size of the $n \times n$ blocks is customizable as a parameter, and with a given width and height of an image file, will result in a block array

$$(\lfloor\frac{\text{width}}{\text{n}}\rfloor + \lceil\frac{\text{width mod n}}{\text{n}}\rceil) \times (\lfloor\frac{\text{height}}{\text{n}}\rfloor + \lceil\frac{\text{height mod n}}{\text{n}}\rceil)$$

in size.
To convert between the global index in the pixel array and the local index in the block array at x index `totalX` and y index `totalY` with a block width and height of `blockWidth` and `blockHeight`, we can use the formula

```
   block.pixels[blockY * blockWidth + blockX]
 = pixels[totalY * width * blockSize
    + totalX * blockSize
    + blockY * width + blockX]  (Figure 1)
```

2.3 Assigning Pixels to Groups

Pixels that have a similar grayscale value locally should be part of the same group in the end, and so we define a number of groups for each block, the maximum number of groups is also a possible parameter of the program. The groups of each block partition the contained pixels, such that every pixel belongs to exactly one group (Figure 2).
The calculation for whether a pixel is "locally close enough" in value to a group depends on another input parameter, a threshold value that will add a pixel to a group if it is within a certain range of the average of all pixel grayscale values added to the group thus far. If all the groups are already initialized and it wasn't locally close enough to any of them, it will instead be added to the group whose average it was closest to.

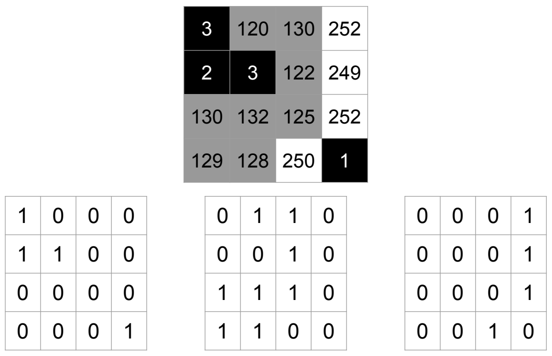

Figure 2: Assigning pixels in a block into distinct groups. The pixels in each group are added to a group based on their grayscale proximity with other pixels in the group, and each group stores this information a simple boolean array.

2.4 Merging the Groups

This follows a similar procedure to the assigning of pixels to groups, but will now instead link adjacent groups together if their averages of all grayscale values within them are close enough to each other. When groups are linked like this, the average of all these groups (weighted by the amount of pixels in each group) is also calculated, resulting in an average grayscale value for a large, linked grouped of pixels spanning across the entire image. (Figure 3)
The sample implementation iterates through all the groups, setting a flag as they are added to a linked list containing all the linked groups across a certain gradient. Other implementations are certainly possible, and recursively building up larger linked groups is likely to be far more parallelizable, though also complex to implement.

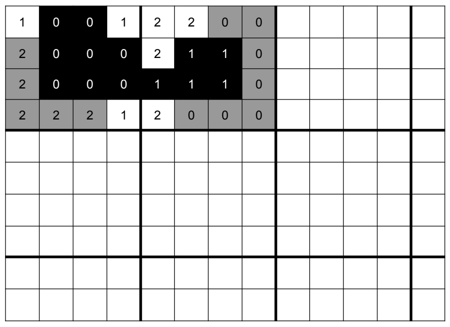

Figure 3: Merging the groups in the top-left corner will require linking the block groups `{0, 1}`, `{1, 2}`, and `{2, 0}`. The sample implementation appends every new merged group to a linked list.

2.5 Recoloring the Groups

Utilizing the minimum and maximum grayscale values calculated at the start, we can now rescale the grayscale values of the linked groups (which is likely to be very limited in low quality images) to the full spectrum including pure white and pure black. (Figure 2)

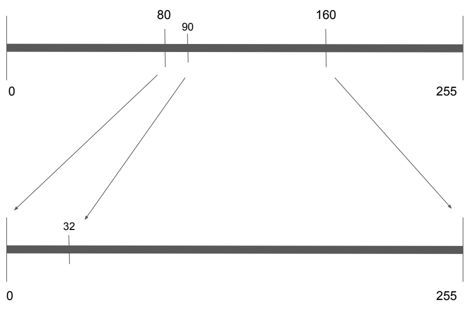

Figure 4: Rescaling the color values across all groups. In this example, the image only contained grayscale values between 80 and 160, with the current group having a grayscale value of 90. After rescaling the entire image, it now utilizes the entire full grayscale value spectrum, rescaling the value 90 to a dark value (as it was dark relative to the rest of the image).

Another input parameter, the amount of colors we want to map the input image to, will define how these groups will be colored in the end. We begin by dividing the final grayscale spectrum into a number of sectors based on the `maxColors` parameter. Each linked group will then map all the contained pixels to the grayscale value which is the closest endpoint in the current sector.

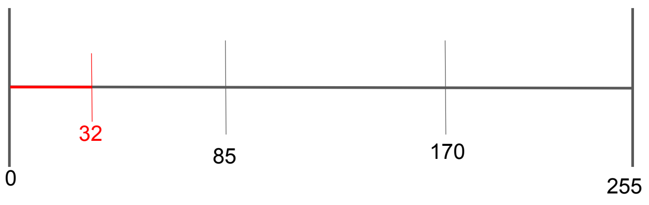

Figure 5: Recoloring a group based on its newly rescaled grayscale value. In this example there are `maxColors`= 4 total colors that the output image will map pixels to. The rescaled value of 32 is closest to the possible output value of 0, and will thus map to a purely black pixel.

3 PERFORMANCE

The runtime of the program is heavily dependant on the filesize, the block size, and the number of groups. However, threading and parallalization have lead to significant speed gains in the methods for splitting the image into blocks and assigning pixels to groups within each block. Due to implementing the linked groups as a linked list, parallelization for merging and recoloring the groups became hard to implement, and an implementation with another data structure will likely see even more significant performance boosts.

4 EVALUATION

While the algorithm succeeds in splitting the input image into various color groups, it doesn't necessarily always do so in a way that could be deemed a success. The input paramaters often need adjusting, making this a finicky implementation not quite ready for real-world use.
There are multiple possible ways to improve the algorithm and implementation that are worth considering.

4.1 Possible Improvements

- Instead of using a constant threshold for both adding a pixel to a group and for linking groups together, a variation of one of the thresholding methods mentioned in Section 1.2 could lead to more consistent behaviour.
- When recoloring the pixels, sorting the groups by grayscale value and making sure that the extreme output values of 0 and 255 are set by the appropriate linked groups could prevent some cases where the background is mapped to a gray as a result of the average simply being too dark.
- Implementing the merged groups as boolean pixel maps across the entire image would lead to a much simpler parallelization and significant speed gains.

4.2 Identifying Objects in an Image

Somewhat surprisingly, the algorithm incidentally performs quite well in helping identify broad objects in more generic images (e.g. of landscapes). Figure 6 shows an example of a fantasy landscape image mapped to just 4 colors. (Image Source: [3])

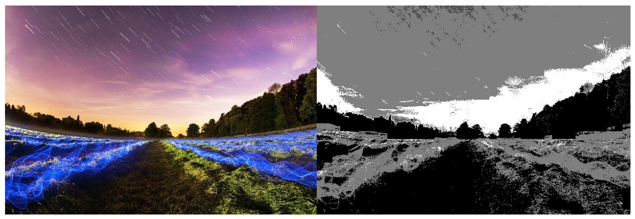

Figure 6: Left is the input image, right is the output image with `maxColors` = 4. The overall structure of the image is clearly discernible.

5 CONCLUSIONS

This paper presented an approach to cleaning up low-quality images while allowing multiple output values. The algorithm attempts to be easily parallelizable and scalable, while being simple enough to easily implement and extend. There are four different input parameters (size of blocks, number of groups, threshold for joining, number of colors) that the user may define, but given some more thought, there may be a way to calculate an optimal value for the first three parameters within the program.

REFERENCES

[1] Bilal Bataineh, Siti N. H. S. Abdullah, K. Omar, and M. Faidzul. 2011. Adaptive Thresholding Methods for Documents Image Binarization. In *Pattern Recognition*, José Francisco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, Cherif Ben-Youssef Brants, and Edwin Robert Hancock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–239.

[2] Derek Bradley and Gerhard Roth. 2007. Adaptive Thresholding using the Integral Image. *J. Graphics Tools* 12 (01 2007), 13–21. https://doi.org/10.1080/2151237X.2007.10129236

[3] FileSamples. 2022. Sample PPM Files. https://filesamples.com/formats/ppm [Online; accessed July 11, 2022].