BogIr: Boggle Board Frame and Letter Detection within MATLAB

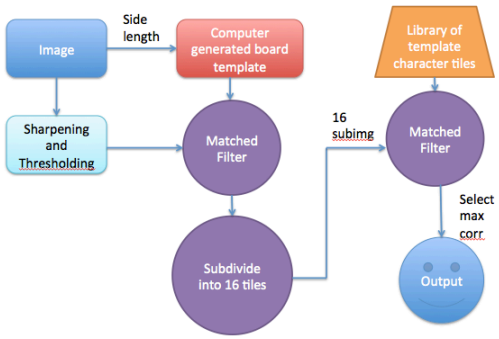
Richard Liu and Ja'Lon Sisson
ECE 488 – Digital Image Processing
Code: www.github.com/richrliu/BogIr

Introduction

The purpose of this project was to utilize the knowledge we have acquired over the duration of this semester and apply it towards the creation of a program that, when supplied with an image of a Boggle board, can detect the overall board, discriminate letters, and output the results. As a brief introduction to our project, Boggle is a game that involves a 4x4 matrix of letters, using which the players are supposed to generate a list of words in 3 minutes, and the player who has the better list in terms of quantity and complexity wins. An algorithm to detect the letters on a Boggle board using computer vision is important in quickly determining a lexicon for any given board, as players often argue about whether a certain word is valid or not. We capture an image of a randomized 4x4 Boggle board. Upon importation to MATLAB, we then attempt to detect the board itself using a matched filter, separate it into its gridded components, and identify the letters contained within. This task are accomplished by way of image pyramiding, matched filtering, normalization of images, and other methods that will be further explained in later sections.

This is an application of the field of optical character recognition (OCR), but since our application is very specific, we can simplify many aspects to develop a novel algorithm. While there exist previous non-academic attempts to accomplish this same goal, none of the methods we have discovered are quite as robust at detecting the Boggle board itself. It is of importance to note that one such attempt influenced our decision to have a library of lettered tiles to aid in our identification of them. As for the applications of our project, we plan on releasing this project to the open source community so that it will be possible to produce a lexicon from an image of a Boggle board as well.

Methods



Experimental Setup

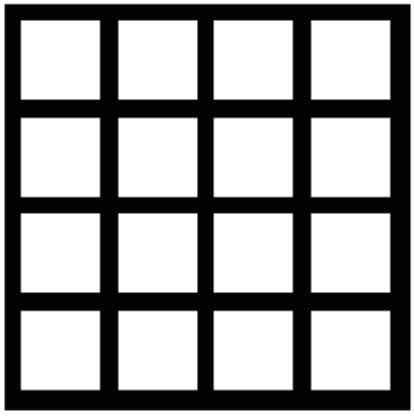
Before running the program, we needed a library of letter tiles with five instances of every letter, each rotated four times. This makes 20 images for each letter and 520 images in total. To generate these, we used a script (makeLetters.m) that processed, detected and subdivided an image of a board exactly the way the program would. The script then saved all sixteen detected letter tiles to images. After obtaining many of these images for each letter, we went through and selected five images for each letter manually, trying to capture as much translational variance as possible. We then used another script (saveLetters.m) to save a .mat file with a cell array containing the matrix representing each image of the aforementioned library, rotated four times each.

Preprocessing

In preprocessing the image we first sharpened the image using a MATLAB command (un-sharp masking). Following this preprocessing step, we then thresholded the image into high (255) and low (0). Pixels that were greater than 170 intensity we classified as high, and anything under that value was deemed as low. Ideally we would have liked to implement hysteresis, but unfortunately time constraints prevented this endeavor.

Board Detection

To detect the board itself, we employed the principles of a matched filter. For this step, the template was a computer-generated black and white image of a black square containing sixteen smaller tiles in a 4x4 array. A sample image is shown below.



In our case, the summation of the dot multiplication of two matrices, the image and square detector, was used to find the correlation. The region with the highest correlation to the square detector, which should be the ‘square’ Boggle board, will be the region that contains the highest overall correlation.

Correlation

$$E[X^i, Y^k] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^i y^k f_{X,Y}(x,y) dx dy$$

Covariance

$$cov[XY] = E[(X - E[X])(Y - E[Y])] = E[XY] - \mu_X \mu_Y$$

This is possible because the dot product (inner product) is equal to the covariance. With the covariance itself, with a mean of zero from normalization, being equal to the correlation.

To figure out where the Boggle board was in the image we used the image pyramid method. At this point we made several assumptions about the actual image.

1. The Boggle board will always be roughly perpendicular (not diagonal) in the image.
2. The Boggle board has to cover approximately 1/16th of the image size. We stop pyramiding after a lower bound side length that roughly corresponds to this size.
3. The Boggle board image was taken against a darker background.
4. We used a stride of the window size divided by 21. This is because, according to our measurements, the black area between the white cubes was approximately 1/21 of the side length.

We started out using a square detection window of side length equal to the height and width of the image. For the image pyramid, we decreased the side length of the detection window four times successively, each time reducing the size by 25%. This produced a total of 5 window sizes, each with a new window side-length (nws) calculated from the formula below.

$$nws = original\ window * (0.75)^n$$

<u>Letter Detection</u>

Our letter detection algorithm was very similar to that of the board detection (matched filter), but with some extra processing steps. After we have detected our board, one of the first things we had to do was crop the image of the board so that the outer tiles bordered the edge of the window. Basically, we threw out all rows near the edges that had values of only 0. This would allow for much more accurate gridding of the detected board.

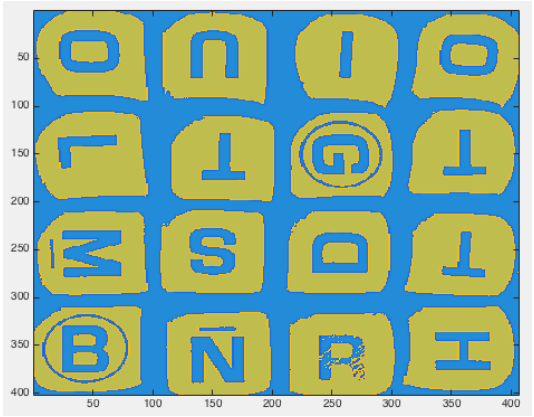
We then gridded the board into sixteen identically sized, non-overlapping sub-windows, with each one containing a letter tile. For each sub-image, we resized the window again in the method described above. To detect the character on the letter, we compared each sub-image with the aforementioned library of letter tile images. The key is that the library of letter tile images was also cropped such that the edges of the tiles border the edges of the image. As such, we tried to preserve as much translational invariance as possible. We then resized the library image to the same size as the detected sub-image, and then we calculated the correlation and picked the letter that had the

highest correlation. Both images were normalized to a total intensity of zero by subtracting the mean pixel intensity from every pixel in these respective images.

Results

Our board detector generally performs very well and most of the imperfections stem from the letter detectors. As an example, here are the results for an arbitrary image.

Detected board:



Detected Letters:

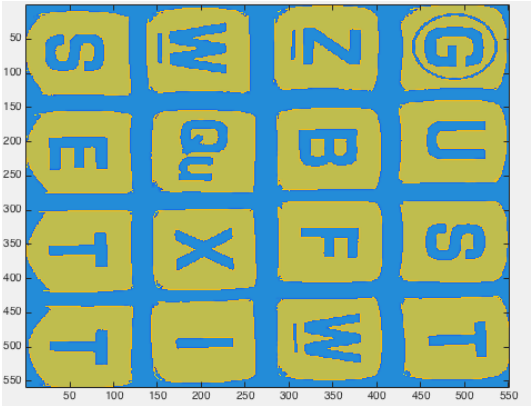
```
theLetters =

    'O'    'U'    'I'    'D'
    'L'    'T'    'G'    'T'
    'M'    'S'    'D'    'T'
    'B'    'N'    'B'    'H'
```

As we see, there are two mistakes, mistaking ‘O’ for ‘D’ in the top right, and ‘R’ for ‘B’ in the last row, third column.

This example just about sums up the performance of our detector: works great for board detection, gets most (14/16 or 15/16) letters correct every time, but often mistakes similar characters (O/D, R/B, S/C, U/C, F/E, etc).

Consider another example below:

Detected board:



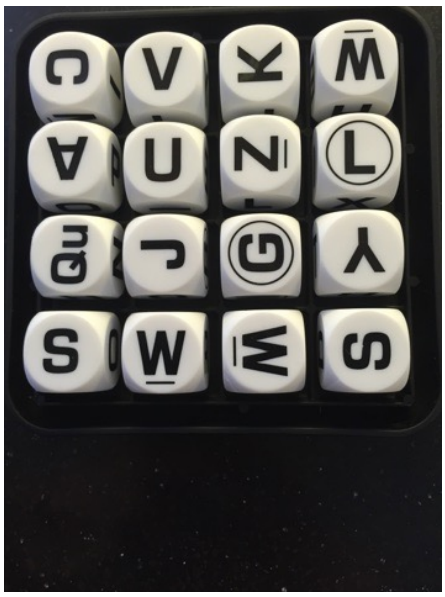
Detected Letters:

```
theLetters =

    'S'    'W'    'Z'    'G'
    'E'    'Q'    'B'    'U'
    'T'    'X'    'E'    'S'
    'T'    'I'    'W'    'T'
```

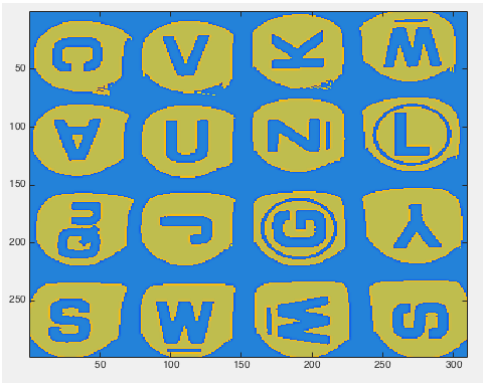
This example got 15/16 correct, mistaking the ‘F’ in the 3rd row, 3rd column for an ‘E’.

We did not develop a method of iteratively stress testing the detector, but that could be done with a library that maps images to the correct letters and repeatedly running the detector on every image and scoring the results against the ground truth. We will discuss the importance of the stress testing method in the Discussions section.

One of the major weaknesses of our algorithm was that it was easily affected by lighting and tilt/rotation. To see the effect of uneven lighting, consider the following image:



Though it may not be clear, this image was taking under much heavier lighting, and thus more of a shadow was cast on the area in between each Boggle tile. As we expect, the thresholded detected board detects the tiles as more circular in the 2D image.

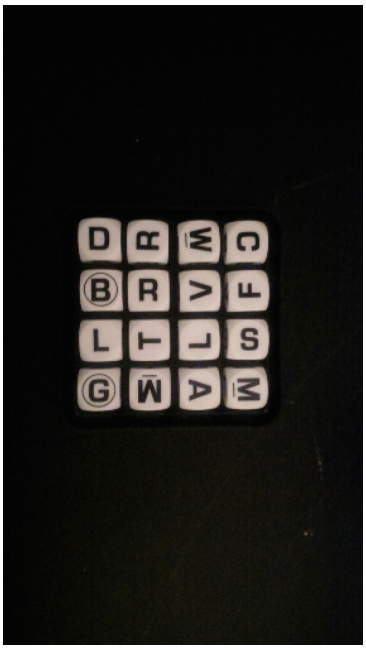


These tiles are indeed much more circular (or amorphous) than the mostly square tiles that we saw in the two previous examples.

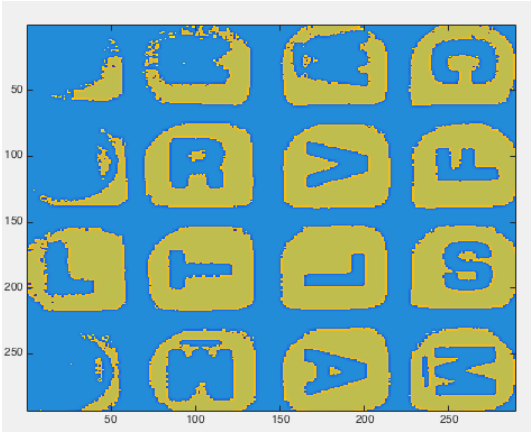
The detected letters got 12/16 correct, and are as follows:

```
letters =

    'O'    'V'    'K'    'W'
    'V'    'U'    'Z'    'L'
    'Q'    'J'    'G'    'E'
    'C'    'W'    'W'    'S'
```

Let us look at an example with poor lighting.



We expect this case to suffer a similar problem as the previous case – too much darkness will distort the shapes of the detected tiles. In fact, the actual experiment shows that this situation is a much more extreme case of the previous example.




```
theLetters =

    'R'    'W'    'N'    'O'
    'T'    'H'    'A'    'J'
    'E'    'J'    'E'    'S'
    'K'    'W'    'V'    'W'
```

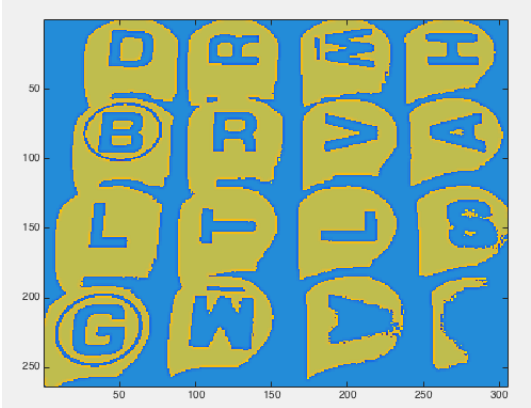
Due to the terrible lighting, thresholding ruined the detected board. The algorithm only scored 1/16 on this case, although it was close for 5 cases (mixed up W/M, O/C and A/V).

We will discuss potential improvements to solve the lighting problem in the Discussion section.

Consider the following example of a slightly tilted image.

We immediately notice two things about this image. One is that some tiles, due to perspective, appear larger than others and this will definitely throw off the gridding of the detected board. Two, tilt naturally

lends itself to lighting gradients, which will affect the performance as described in the previous two examples.

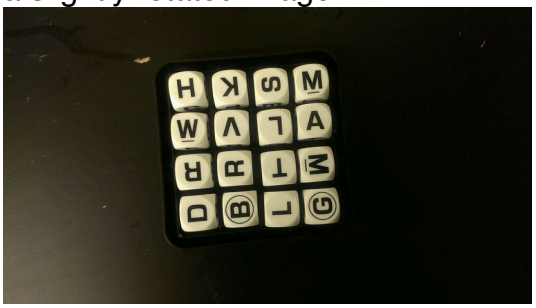


We see that due to lighting differences, the two lower right tiles were heavily mutilated.

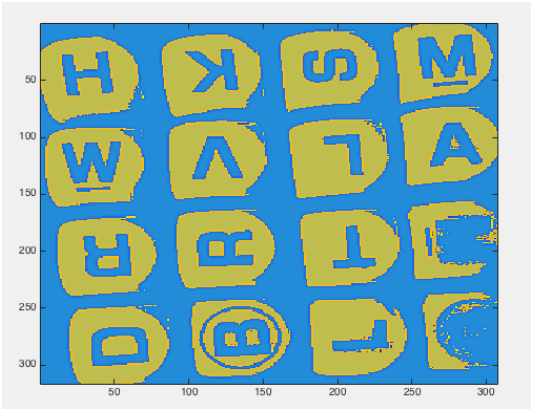
```
theLetters =

    'R'    'K'    'P'    'T'
    'Q'    'Z'    'Z'    'J'
    'X'    'K'    'H'    'Z'
    'D'    'E'    'X'    'Y'
```

The algorithm detected 0/16 of the letters correctly for a tilted example. We conclude that a perspective shift is heavily detrimental to the performance of the detector.

Now consider the following image of a slightly rotated image.



As in the previous example, we expect the detected board to be improperly gridded which will be heavily detrimental to the matched filter recognition of each individual letter.



```
theLetters =

    'H'    'K'    'S'    'P'
    'R'    'A'    'O'    'X'
    'P'    'U'    'T'    'Q'
    'R'    'P'    'H'    'H'
```

In this example, we get 4/16 correct, with 3 “understandable” mistakes (V/A, R/P, B/P, all similar looking letters).

In all the examples above, we see one of the strengths of our algorithm – board detection always seems to work, with the matched filter being an effective way to detect the Boggle board.

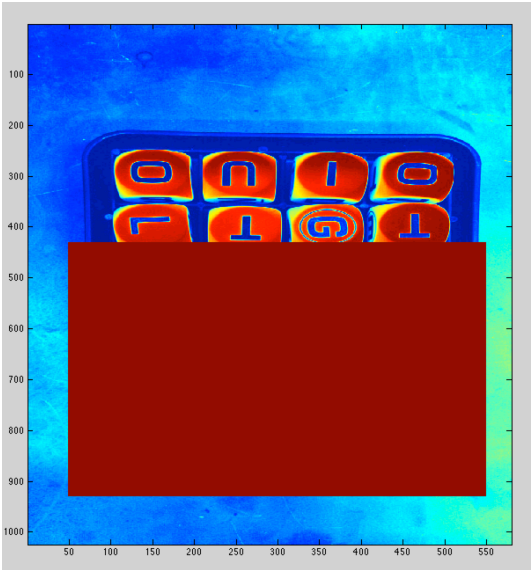
Discussion

Design Decisions

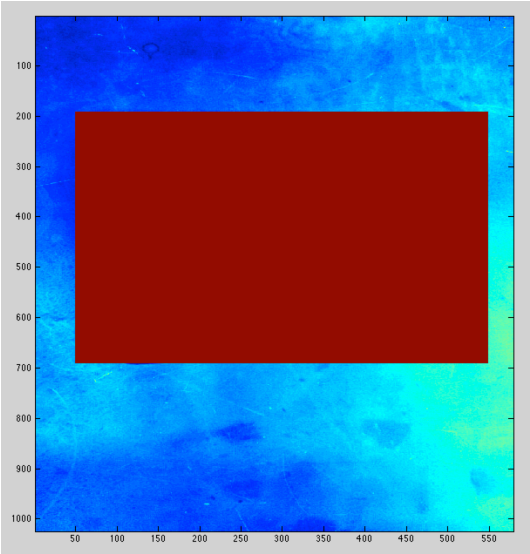
The first design decision we had to make was on how to preprocess and clean our data. It quickly became clear through various tests that thresholding with parameters high for intensities above 170 pixels and low below that value made the board clearer, and easier to detect. Another enhancement we made before applying the thresholding method

was the sharpening of the image. This operation made the edges of each letter tile easier to detect. While we did attempt to use histogram equalization, we noticed it was detrimental to performance; with our assumption being that it did not respond well to lighting gradients. That is to say, areas that were lighter, comparatively, were much more likely to be thresholded into the low values incorrectly.

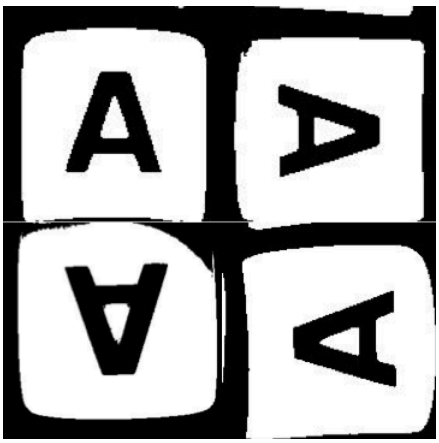
To detect the Boggle board frame itself we decided that either an approach using minimum RMSE with a matched filter to detect the frame, or the maximum correlation with a matched filter to detect the Boggle frame would be effective. We first attempted with minimum RMSE, but one issue we encountered was with larger detection windows, which would produce shifted detection relative to the actual placement of the Boggle board. This was evident in window sizes above 490x490 pixels, for one specific image as shown below.



The correlation on the other hand was robust enough to handle these increased window sizes.



Another design decision we chose was in the detection of the actual letters. After creating a grid of the board, we could detect the letter by comparing it to an existing library. For this stage also, we could either try to use a matched filter with minimum RMSE or a matched filter with maximum correlation. We first tested RMSE, but for inexplicable reasons, there was a particularly high frequency of the letter ‘L’, as well as the confusion of the program of the letters ‘T’, with ‘I’ and ‘Y’. We found that performance improved drastically after switching to using maximum correlation with normalized intensities.



Improvements

This algorithm, although generally good, is not reliable and there are several improvements that could be made to deal with some noticed imperfections.

Lighting invariance

We saw that our algorithm was generally unable to deal with less than ideal lighting schemes effectively. When there was a lot of light, there were too many shadows formed in between the tiles that affected the shape of each letter tile after thresholding. Thus, using a matched filter to compare a circle tile against a square template is guaranteed to yield unreliable results. One improvement to deal with this situation is a small morphological operation where, after the letter sub-image is cropped down to the approximate size of the tile, pad all the edges with high (255) values. In this manner, regardless of what "shape" the detected tile is, we can force it to become a square.

This solves the less extreme cases of lighting. But is there any way to improve all lighting conditions, even the extreme ones? We believe there may be a way in preprocessing the image. We determined empirically that histogram equalization generally decreased performance. However, for images with extremely low variance, histogram equalization may actually help. We also believe there is a method of *dynamic thresholding* instead of setting a fixed threshold at 170 pixel intensity. This preprocessing algorithm would ideally scan the image to determine the range of pixel intensity values in the detection windowed sub-image, and set a threshold based on that range.

We also believe using hysteresis thresholding instead of a single threshold could improve performance.

Rotation Invariance

This is a decently covered topic in OCR, but for our application we believe a simple change should lead to a lot of improvement. Instead of rotating each image in the "library" 4 times at 90 degree intervals, we could rotate each image at smaller intervals to capture more rotation.

Future Applications

We believe that our work is a foundational algorithm for future applications that will allow people to find solutions in word games. Although we applied computer vision to Boggle specifically, an extension into other word games like Scrabble or Upwords is not outside the realm of possibility.