Content-Based Image Retrieval (CBIR) Using Barcodes

Due on April 9th, 2021.

Prepared By:
Ridwan Hossain (100747897)
Ahmet Karapinar (100750048)
Michael Metry (100747141)
Joshua White (100747854)

Prepared For:
Shahryar Rahnamayan, PhD, P.Eng, SMIEEE, Professor
Ontario Tech University

Table of Contents

GITHUB Link: https://github.com/ahmetkca/hand-written-digit-barcode-generator

(contains a well formatted README file)

Problem Statement

The goal of this project is to design, implement, and perform comparative analysis by using the MNIST database to compare and retrieve similar images with the implementation of barcodes. The database consists of ten sets of ten handwritten digits ranging from 0 to 9 with each group of categorized digits being stored in their own folder. A barcode consisting of bars and spaces will store numerical values for each unique image. This application is widely used in many industries such as medical image sorting, mobile app games, storefront organization demonstrating its vast range of implementation .

The MNIST database contains 100 images of size 28 x 28 pixels consisting of ten images per digit. The task is to create a unique barcode for each image so that the program can perform the searching and comparison analysis.

Lastly, the program compares the generated barcodes to determine which image most closely matches the one which was searched. This is done by using hamming distance, a method of comparing barcodes at specific positions and increasing the hamming distance the more the two barcodes differ. Once every barcode has been compared to that of the searched number the algorithm will find the most similar image, the one with the lowest hamming distance. Furthermore, the analysis of the designed algorithms will use the Big-Oh notation to calculate the program’s efficiency of making comparisons, generating the barcodes, and performing the required calculations.

Explanation of Algorithms

Introduction

Our team had decided that our algorithm would refuse the use of metadata, such as labels, and strictly base the procedures for barcode generation and image searching on explicit visual image data. Given this requirement set by our team, our initial goal was an accuracy of 50% correct matches in the search algorithm, which appeared reasonable given the algorithms would work strictly with image data. The algorithms required to generate a barcode corresponding to a distinct MNIST image went through numerous phases of development, each requiring various changes to the algorithm to improve its accuracy. This section of the report will examine each phase of the development preceding the final algorithm and the changes that have been made.

Phase 1 - Initial Image Processing

Within the preliminary stages of development, our team decided to take each MNIST image and apply a method that would threshold each pixel within the image to either a fully white pixel (greyscale value of 255) or a fully black pixel (greyscale value of 0). This was done using the OpenCV library in python achieving sharper characteristics of each image while also mitigating unnecessary visual noise.

Phase 2 - Simplifying the Image into Divisions

Our team then made the decision to divide each MNIST image into four horizontal divisions, and as each image is 28 by 28 pixels, this meant that each horizontal division would be divided into 7 rows of 28 pixels. The devised plan was to take the sum of the pixel values (in this case, the sum of pixels, as they were thresholded to fully white or black) in each horizontal division, and for every image for a single projection, the average number of pixels in each division was taken , and that would be the threshold value for that specific division for that distinct projection angle. As a result, each projection angle would have four different thresholds for each horizontal division. A rough visual representation can be found in figure A.1 below. **(Note: the visual representation is not to scale, in our algorithm, each of the 4 divisions consisted of 7 rows of 28 pixels, whereas this figure depicts 3 rows of 12 pixels)**

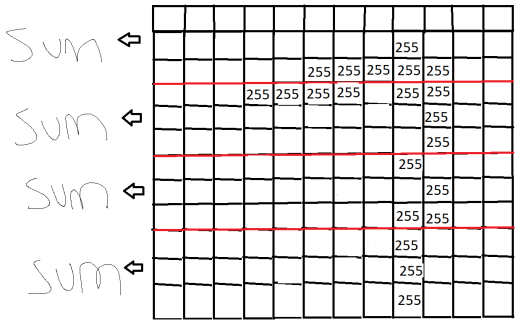


Figure A.1

Phase 3 - Stringing the Binary Bits Together

Our team had decided on having eight projection angles going from 0 degrees to 180 degrees of rotation. This meant that each projection would rotate the image in increments of 22.5 degrees, such that the projections would display the image at 0, 22.5, 45, 67.5, 90, 112.5, 135, and 157.5 degrees of rotation. In essence, the image itself would be rotated and sums of pixels would be obtained from horizontal divisions applied to that image once rotated. As the algorithm would be using eight projections at this stage of development, there would be eight sets of four divisions, thus, there would be four unique threshold values for each distinct projection angle. The inclination being that if the sum of pixels in a distinct horizontal division is above the threshold, it would become a binary digit 1 (represented as black on the visual barcode) and 0 otherwise. This meant that each projection of an image would produce four binary digits, and once every bit is strung together from all eight projections, the algorithm would generate a 32 bit barcode corresponding to that distinct MNIST image.

Phase 4 - Creating the Search Algorithm

To compare and search for the closest (but not matching) barcode, our team applied the concept of hamming distances to obtain the closest barcode to that of the query image, while omitting the matching barcode. To do this, the bit at each index of a barcode was compared to the bit at the same index of another barcode. Each time two bits at the same index are different, 1 is added to the hamming distance. As a result, the barcode that has the lowest hamming distance and is not equal to 0, will be the resultant image in the search.

Phase 5 - Testing of the Initial Draft of the Barcode Generator

After the completion of the first draft of the algorithm, initial testing provided relatively promising results, but still fell short of what our team had originally aspired. The barcodes that were being generated reflected useful characteristics of an image, such as thickness, angle of slant, curvature, and so on. However, the number of correct matches was still low, so our team decided to increase the number of projection angles we would take per image.

Phase 6 - Increasing the Number of Projections

At this stage of development, our team increased the number of projection angles from 8 to 16. As a result, the number of bits in a barcode had increased from 32 to 64. Since there were now more projections, it was necessary to recalculate the averages of pixel sums in each division for new thresholds in each projection. To do so, the same procedure as in phase 2 was implemented to calculate the new averages. This had marginally improved the accuracy of the results, as more correct matches were now being found when applying the search algorithm. However, results were still below our desired accuracy, leading to the decision to try and increase the number of projections even further to 32. After this point, it has seemed the possibility for the accuracy of our algorithm to improve by increasing the number of projection angles had plateaued, meaning a new method for increasing match percentage must be devised.

Phase 7 - Scrapping the Use of Divisions

Our team had decided to no longer use divisions when generating the barcodes for the images because we felt that thresholding 7 rows of 28 pixels may have led to the loss of a substantial amount of data from the image. Instead, the algorithm was updated to use every single row of pixels so we would be able to extract more data from the image relative to before. We had revised the algorithm to use 8 projection angles once again, but now utilizing every row of pixels when generating the barcode.To obtain new thresholds, the algorithm was designed to take the sum of the pixel values across each row for each image for a distinct projection angle. The average number of pixels in each row was taken and that would be the threshold value for all rows for that distinct projection angle. Those thresholds would be applied to all rows of every one of the 8 projections, such that by the end, there would be 8 thresholds being utilized by the algorithm. When running the algorithm, there would be a bit corresponding to each row of each projection angle of an image. This would result in the creation of a 224-bit barcode for a distinct MNIST image.

Phase 8 - Testing the Revised Algorithm

Upon testing the revised version of the algorithm, results were much closer to the target. Whilst the accuracy was not the most precise, they were much closer to the general targets that had been initially set when beginning this project (see the **Required Measurements and Analysis** section of this report for further details).

Phase 9 - Final Changes

Our team had theorized that our initial process of thresholding the pixel values to either a full white or full black had resulted in the loss of data opposed to emphasizing the shape and characteristics of the images. To slightly improve the algorithm further, it was updated to no longer threshold pixels to a full white or black and instead retain any variations of grey pixels. This would allow the algorithm to obtain more unique data pertaining to each image which would allow images to be better separated from others. This had improved the accuracy by approximately 5% and had allowed our team to achieve our general target of roughly 50% that was set during the preliminary stages of development (see the **Required Measurements and Analysis** section of this report for further details). As such, due to time constraints, our team had considered this to be the final revision of our algorithm and development of it would be complete.

Required Measurements and Analysis

Retrieval Accuracy and Analysis of Results

The program exhibited a medium level of accuracy, correctly selecting the corresponding number for 58% of queried images. This means that the program can be considered a success as it exceeded the target value of 50% correct matches.

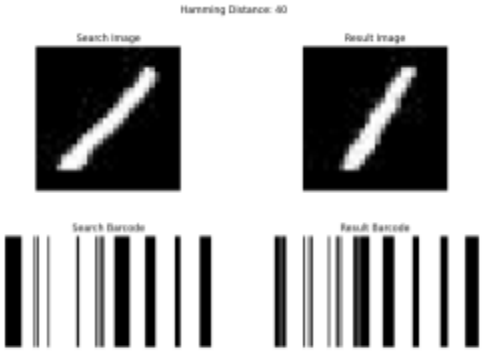


Figure B.1

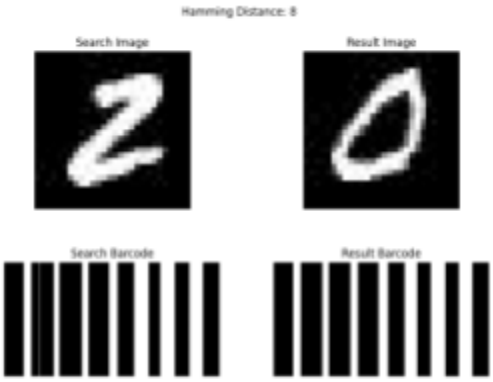


Figure B.2

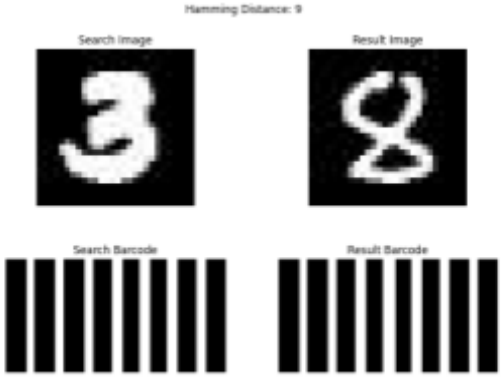


Figure B.3

Although the program was considered a success, the errors on the matching process can be observed and analyzed to explain why the program may miss-match certain characters. As seen in figure B.1, both “ones” exhibit a smiler slant, thickness and start/end points. Because there are multiple “ones” in the data set and these two were determined to be best matched, it can be stated that these factors will affect the hamming distance. This can explain the miss-match seen in figure B.2, as the left side and bottom of the zero match very closely to the middle and bottom of the two. Because the total sum of the pixels in a division is taken, the true thickness of the zero is more like that of the 2 then it would appear for the first projection. Both numbers having similar true thicknesses because of similar results for the first and final projection (unaltered projection and upside-down projection). Next, the projections of the side view are based primarily on the height of the image and the true thickness of the rotated image. Because the images of both 2 and 0 have similar heights and evenly distributed thickness, the projection of their side view also results in a similar portion in the center of their barcodes. For the angular projections of each image, it tends to fold in on itself resulting in a straight angled block of active pixels. This causes the second and third quartile

of projections to only accurately display the apparent thickness of the images. This can be seen by comparing the second and third quartile of the three comparisons (B.1, B.2, B.3), as they increase so does the thickness of the images.

From this information we can accurately assess what an image's barcode may look like or vice versa. An image with thick second and third quartiles will have a higher true thickness, while an image that has minor breaks all throughout its bars and droughts between them will be thinner. The angle of an image can be determined by the thickness of its first and last quartiles as the more an angle the image lays on the thinner its first and last projections will be.

Big-O Complexity Analysis

Search Algorithm:

```
101   for i, barcode in enumerate(barcodes):
102       print(imageLocations[i])
103       currentHMD = hammingDistance( barcode,selectedImageBarcode)
104       print(currentHMD)
105       if currentHMD == 0:
106           continue
107       elif currentHMD < minHMD:
108           minHMD = currentHMD
109           minBarcode = barcode
110           imageLoc = imageLocations[i]
111
```

In this snippet of the code, the search algorithm is being conducted to find the closest image. The beginning of the for-loop goes through an iteration of barcodes (variable n) for comparison between barcodes. For each barcode, it would go through an iteration of barcode length (variable m) and compare each bit at a time to see if there are any similarities between barcodes. This approach is referred to as the hamming distance algorithm and how similar two barcodes are dictated by how small the hamming distance is. Whenever two bits in the same index are different, 1 is added to the total distance, otherwise they are 0 when they are the same. Number of ones are then counted to get the hamming distance value. When it encounters the if and else statements, it will compare with the hamming distances of the current barcode we want to compare with the minimum hamming distance found so far. If the two have identical barcodes (if currentHMD == 0), it will skip this iteration (barcode) and continue with the next barcode in the barcode database and compare until the two barcodes have similar but not exact barcodes which is the minimum possible hamming distance that is not equal to 0. Hence, this runs in a big-Oh complexity of O(n*m) because we are comparing certain indices in a nested for-loop.

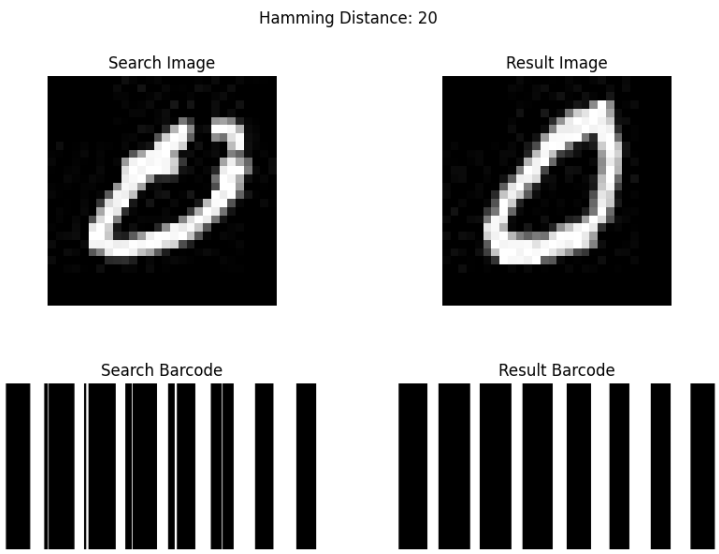
Barcode Generator:

```
def create_barcode(imagePath):
    barcode = []

    opcv = cv2.imread(imagePath, 0) # read image file as cv2 image

    img = Image.fromarray(opcv) # create image from thresholded 2d image array


    barcode = []
    degree = constants.MIN_DEGREE
    while degree < constants.MAX_DEGREE: # loop through MIN_DEGREE to MAX_DEGREE by STEP_DEGREE
        currentProjectionThreshold = int(degree / constants.STEP_DEGREE) # find the appropriate
threshold index
        rotated_image = img.rotate(degree) # rotate the image
        image2d = np.array(rotated_image) # get 2d representation of the rotated image


        for row in image2d:  # loop through each row in rotated image
            row_sum = 0  # initialize row pixel counter
            for pixel in row: # loop through each pixel in the row
                pixel = pixel / 255  # since we have either 0 or 255 as a pixel value divide this
number by 255 to get 0 or 1 which is there is pixel or there is not
                row_sum+=pixel # sum of pixels across a single row

            # thresholds the sum of the row to 1 or 0 based on calculated threshold
            if row_sum >= thresholds[currentProjectionThreshold]:
                barcode.append(1)
            else:
                barcode.append(0)

        degree += constants.STEP_DEGREE

    return barcode
```

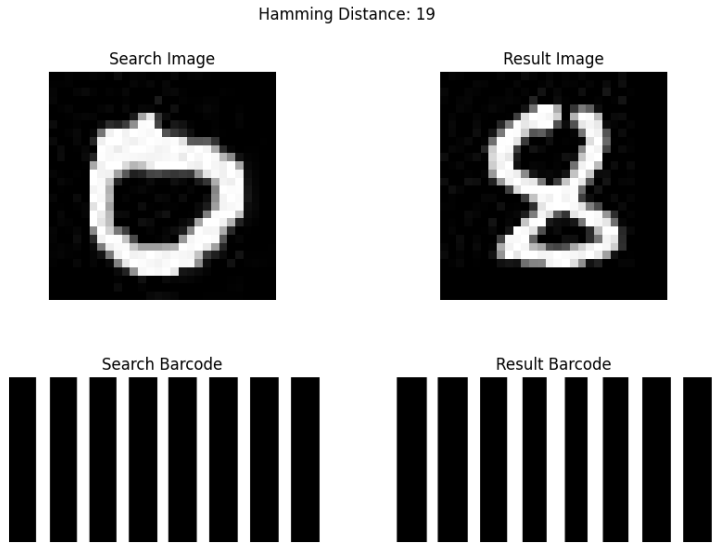
In this snippet of code, the barcode generator creates the barcodes for the 100 images in the MNIST database. Firstly, we would have to calculate threshold values for the barcode to be generated and used for comparison. Next, the while-loop (variable n) deals with rotating the images and it finds the most appropriate threshold index. For each image that is being rotated, it will perform the following nested for-loops, which will loop through each row in the rotated image (variable m). Meanwhile, it will loop through each pixel in the row and for each pixel it will divide the pixel value by 255 to get a pixel ratio which will be between 0 and 1 inclusive (variable a). Lastly, it will take the appropriate thresholds for the projection and the calculated row sum and if the calculated row sum is bigger than the projection threshold then adds 1 to the barcode, add 0 otherwise. Therefore, the complexity of this portion would be a O(n*m*a) since the while-loop is nested within nested for-loops which creates the barcode for each unique image, and it is displayed for each two images.

Search Results

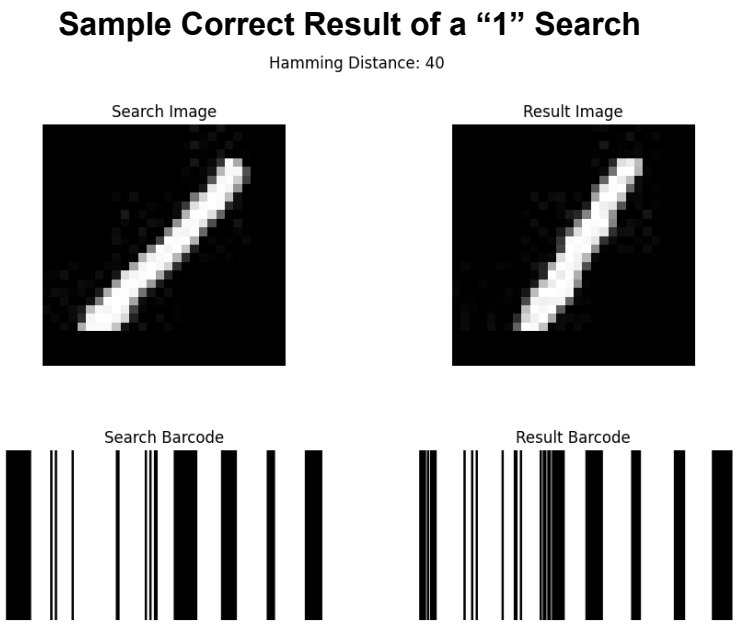
Sample Correct Result of a “0” Search

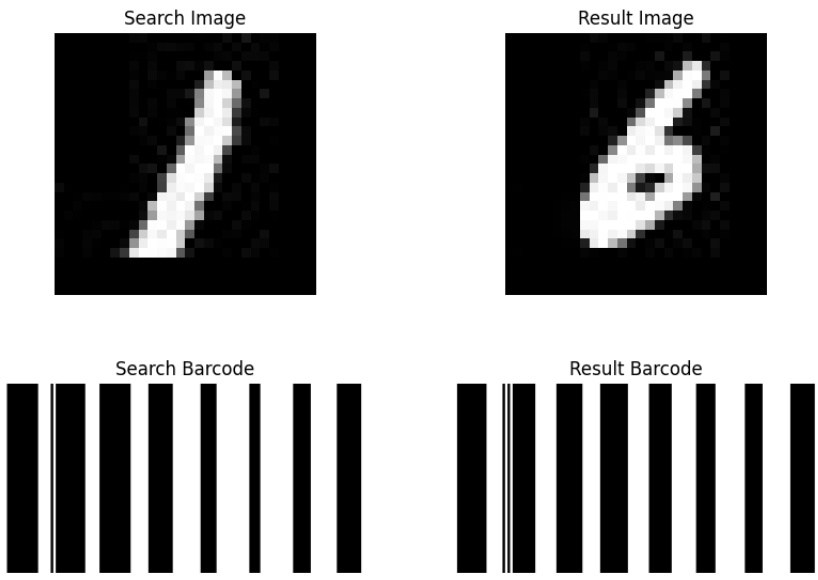


Sample Incorrect Result of a “0” Search



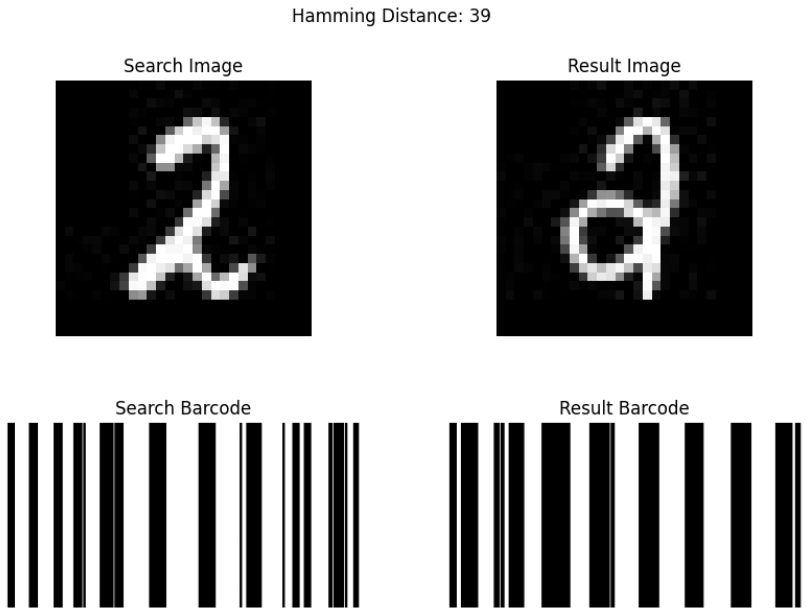
The algorithm might mistake this 0 for an 8 because it does not recognize the horizontal location of the pixels, so long as certain rows have a certain number of pixels. As you can see in the images, they have almost the same thickness. The threshold is simply looking for enough pixels such that if there are any more pixels past the threshold value, no further valuable information is supplemented to the barcode.



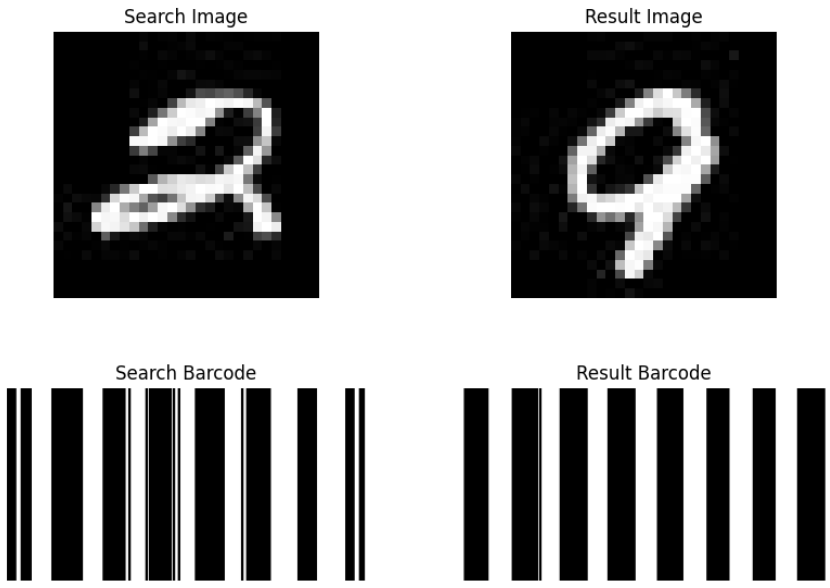


It is evident how similar the images are. One of the distinct properties of these images is that they are both slanted in the same direction and they have almost the same thickness which is enough for the threshold to determine that they must nearly be the same image.

Sample Correct Result of a “2” Search

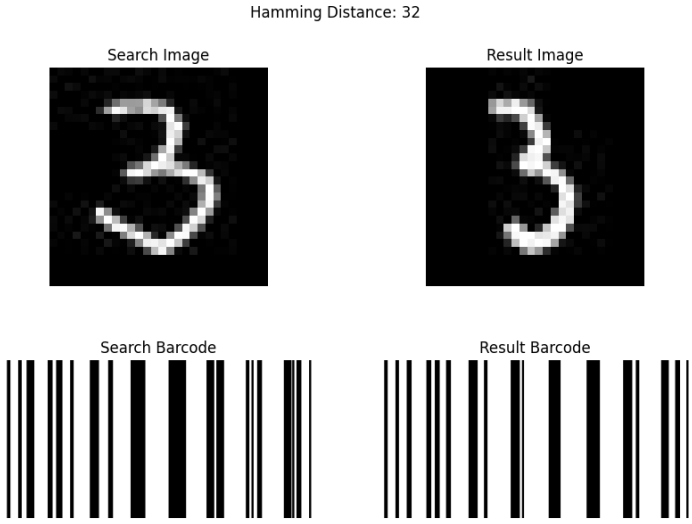


Sample Incorrect Result of a “2” Search

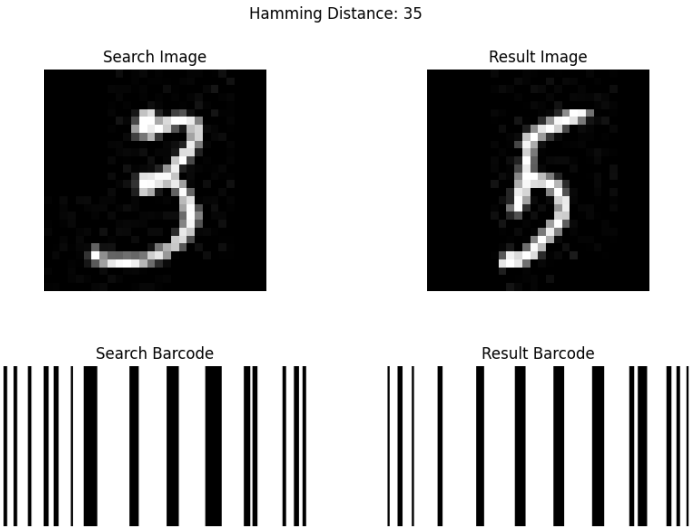


One of the differences between these images is that the left image does not have a distinct tail like the left image has, however, the distinct shape that the head of a nine has is present in both images.

Sample Correct Result of a “3” Search

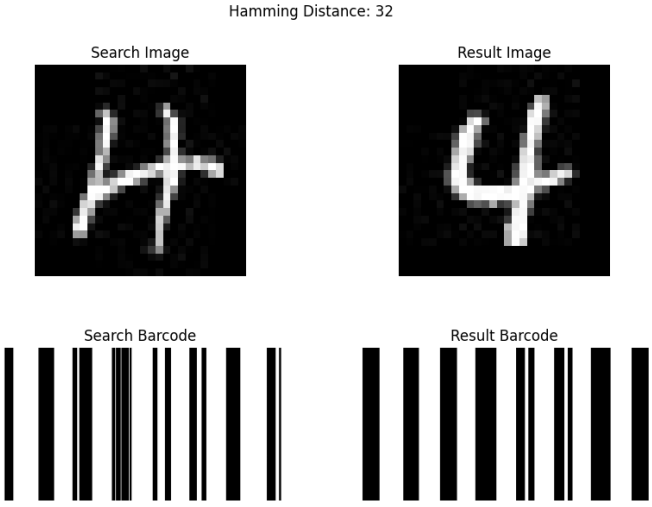


Sample Incorrect Result of a “3” Search

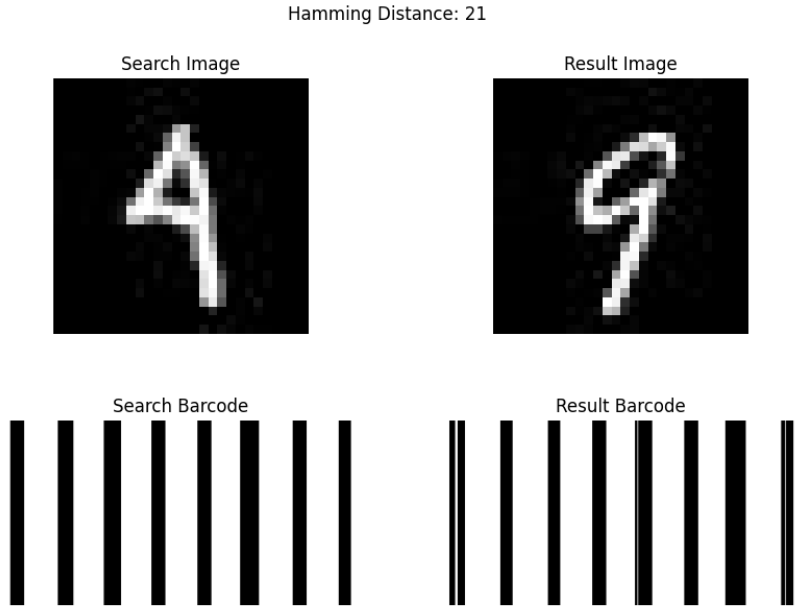


There does not seem to be much similarity at first glance but if we consider how the algorithm deciphers these images, it is possible to interpret some clear connections between them. Both images have a similar bottom curve, and their top portion is also similar. The only difference is the location of the middle component of the top curve, where the algorithm does not interpret horizontal pixel locations, thus, the location of that specific portion does not make a difference through the eyes of the algorithm.

Sample Correct Result of a “4” Search

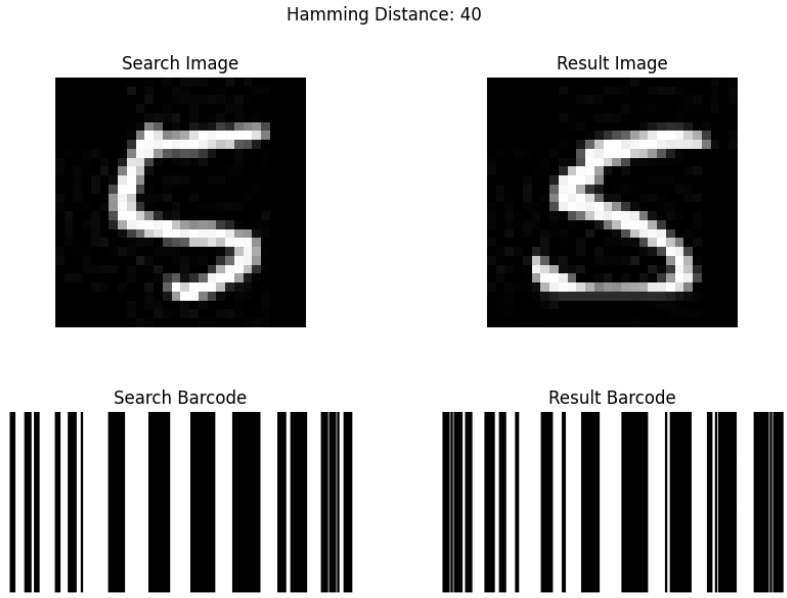


Sample Incorrect Result of a “4” Search

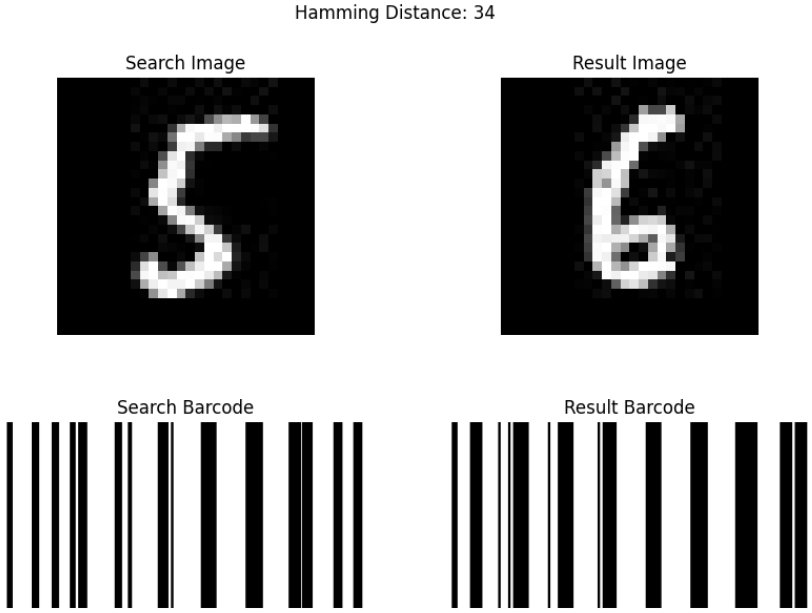


This failed result was the most similar one among other failed results. You can see they both exhibit a shape like that of a nine, with the only difference being that the left image contains sharper edges whereas the right image is curvier. This makes perfect sense because our algorithm can not distinguish between sharper edges and curvy shapes, simply the pixel counts.

Sample Correct Result of a “5” Search

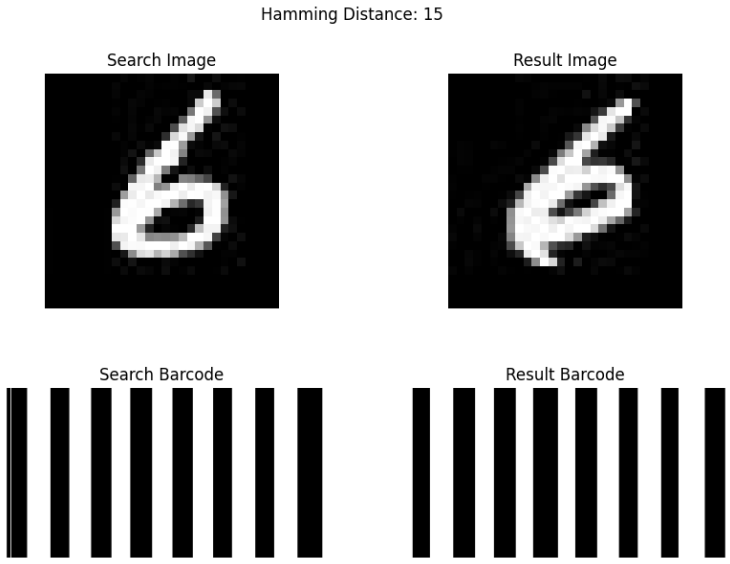


Sample Incorrect Result of a “5” Search

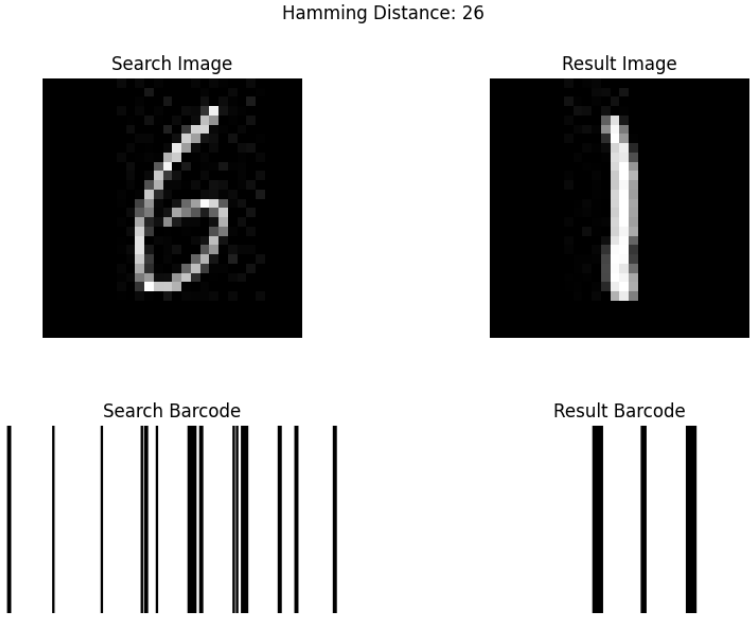


This was another one of the similar failed results. You can see the left image, which is a 5, is simply missing a portion of the bottom loop that would transform it into a 6. Other than this difference, both images are of similar thickness, shape, and are both slanted in the same direction as well.

Sample Correct Result of a “6” Search

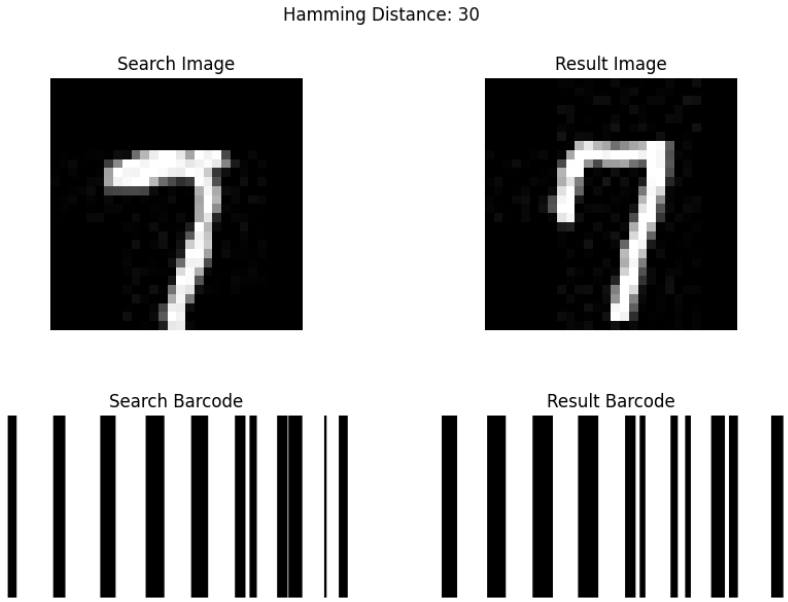


Sample Incorrect Result of a “6” Search

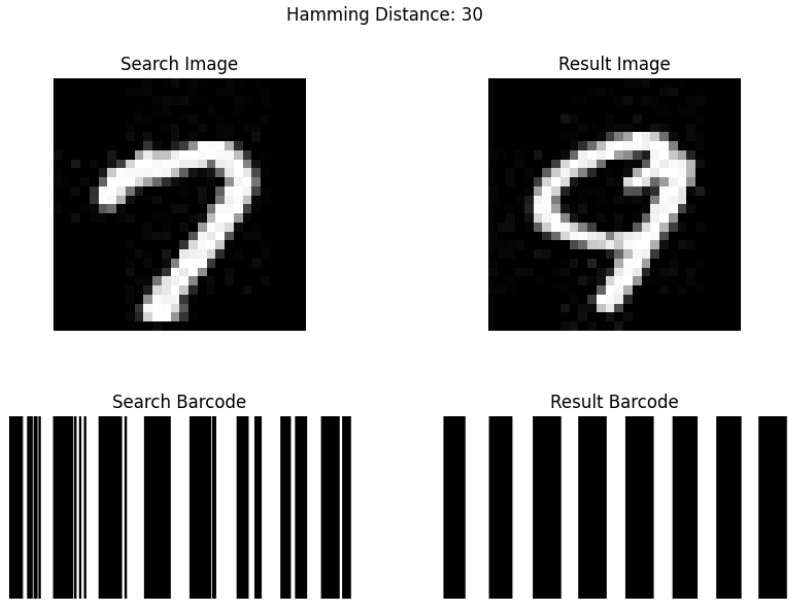


The images do not seem to be similar at first, but through the eyes of the algorithm, it is simply looking directly from left to right across each row. When viewing the images from that perspective, one can see it is just a vertical line even though the left image is wider than the right. It is because the algorithm is simply looking at how many pixels there are in each row and they seem to have roughly the same number of pixels in each row.

Sample Correct Result of a “7” Search

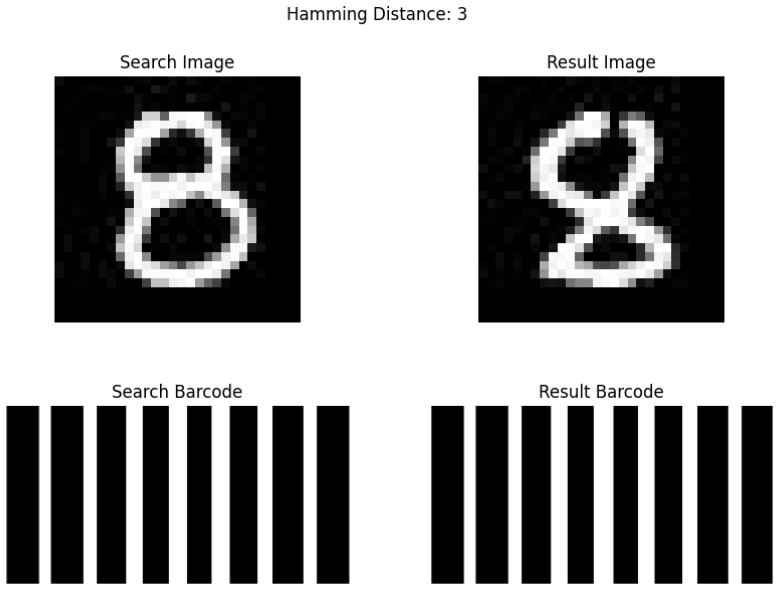


Sample Incorrect Result of a “7” Search

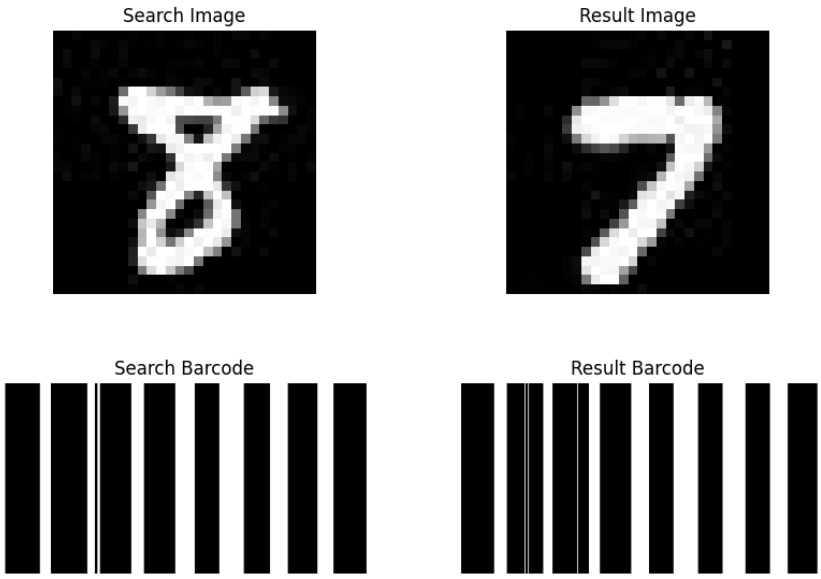


As seen above, the top loop of the right image is complete whereas the top loop of the left image is partially completed. It is also visible on the barcode that the missing part has its fingerprint, as there are thin white lines in place of the black filled lines on the right image’s barcode. Those thin white bars represent that missing portion, which seems to be the only visible difference between them.

Sample Correct Result of an “8” Search

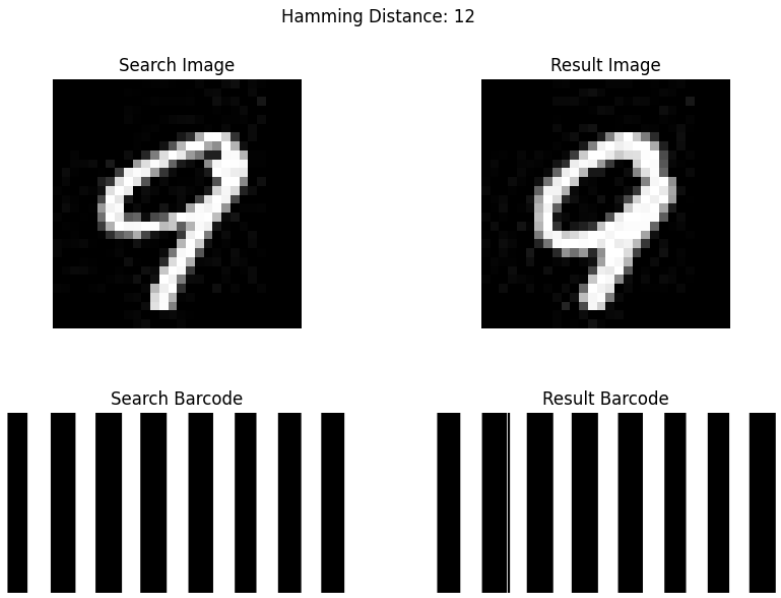


Sample Incorrect Result of an “8” Search

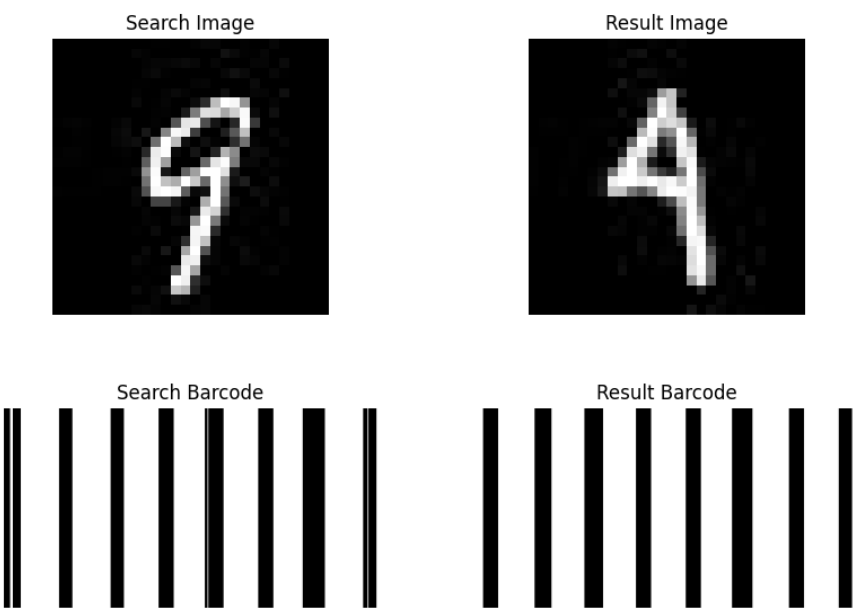


As seen above, these images share a key component between them. Both images contain a horizontal line at the top, with a middle portion spanning the top right corner to the bottom left. The image on the right is missing some features which the image on the left contains, however, since the algorithm focuses on the number of pixels opposed to the location of them, the extra thickness of the right image can make the algorithm think there are features which may be like that of the left image.

Sample Correct Result of a “9” Search



Sample Incorrect Result of a “9” Search



Refer to **Sample Incorrect Result of a “4” Search** for details.

Concluding Remarks

After completing the project, our team has developed a successful algorithm that generates barcodes to describe an image, as well as searches and compares two images. However, despite the success, there were some struggles faced throughout the development process. Firstly, the most difficult task was downloading the sample MNIST dataset and having the editor process all the images. Also, there was some knowledge that our team had not been introduced to initially, and as such, thorough research was conducted. For instance, our team researched how the radon barcode functioned regarding calculations for the projections and calculating the pixel values after they have been thresholded. After generating barcodes for each image, the code was executed to verify if the barcodes themselves were realistic, but many errors had occurred throughout development. This section will further examine the contingencies that had occurred throughout the algorithm's evolution, such as some of the challenges faced by our team, examine possible areas of improvement, and provide concluding remarks on this project.

The biggest challenge faced in the development process was how to generate sufficient threshold values. As we discussed in the **Explanation of Algorithms** section, initially we were planning to have bigger divisions, for example, 7 rows of pixels per division, opposed to the currently used method of utilizing the sums of every individual row. Our group was planning to find a threshold value for each division in each projection. We created an algorithm to create an image representation of the barcode so that we would have the ability to manually check each individual barcode. From these manual examinations, we had received unexpected barcodes, for example, all black or all white barcodes. These meant that the threshold values we calculated were insufficient to extract useful data for some digits. As an example, for the digit 1, the barcodes were all white and for the digit 8, the barcodes were mostly all black. Our team decided to discard the bigger division and stick with the use of every row of pixels and have 1 threshold being applied across all rows for each projection. After thresholds were calculated, it was easy to apply them to each image and create the barcodes.

One of the largest downfalls that the algorithm exhibited was its inability to distinguish the horizontal location of pixels. This had made it so when the algorithm had summed the pixels in a row, it simply took the total count and thresholded it. Given more time, the algorithm could be altered to also consider the gaps in between white pixels. This would aid the software in piecing together the varying spaces in between filled portions of an image. This would help the algorithm to have some form of idea as to where the white pixels are located. In addition to this, another area of improvement could be to have the algorithm consider the transitioning of pixels as the image is rotated. By looking at the change in pixels as the image is rotated, this could allow the algorithm to have better awareness of the locations of the pixels. It would also be possible to implement a voting system to improve search results, but as stated, our team had wished to strictly use image data within the generation and searching of barcodes and not implement any use of metadata. Metadata was only used to perform accuracy tests within the algorithms.

In conclusion our project met all benchmarks that it had sought after, and it had produced formidable results. Our algorithm had successfully created a barcode for each image, ensured that the barcodes generated sufficiently reflected key characteristics of an image, correctly calculated the hamming distance between the queried image's barcode and all other barcodes from the data set, tried to output the next closest image to the query image, and finally, the program has correctly matched greater than 50% of the queried images to an image of the same digit. Through testing, it was proven that all one hundred images would produce a unique barcode, and through automated testing, it was shown that the program correctly calculated hamming distances and could therefore correctly compare the hamming distance of two barcodes. Finally, through a comparison of results it was proven that the program matched 58% of the queried images to the correct corresponding digit's image. As a result, the algorithm had met and exceeded the minimum requirements set by our team.