FACULTY OF ENGINEERING AND APPLIED SCIENCE

SOFE 2715U: Data Structures

Course Project

Course Instructor: Dr. Shahryar Rahnamayan

1	Huzaifa Zia	---------

Table of Contents

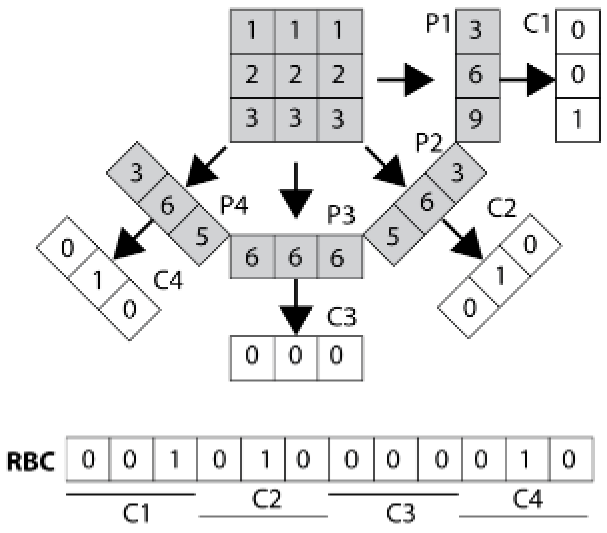
<div align="center">Appendix I</div>

To view the source code of this project visit the following link:
https://github.com/huzaifazia17/ContentBasedImageRetrieval

To view the barcodes generated in this project visit the following link:
https://github.com/huzaifazia17/ContentBasedImageRetrieval/blob/main/Barcodes.txt

To view the accuracy results of this project visit the following link:
https://github.com/huzaifazia17/ContentBasedImageRetrieval/blob/main/accuracy.txt

In this project, projections are referenced multiple times. The projections defined and used in this project are as follows:



Radon Barcode (RBC) – Projections (P1,P2,P3,P4) are binarized (thresholded) to generate code fragments C1,C2,C3,C4. Putting all code fragments together delivers the barcode **RBC**.

Introduction

This program is a content-based image retrieval system designed to compare barcodes of images and find other images with similar barcodes. It begins by taking in a list of 100 images, each 28 by 28 pixels. These images are contained in 10 folders with 10 images for each number from 0 to 9. The program will then take each image and convert it into a pixelated array using a python library named Pillow. Once these arrays have been created, five different projections are taken at varying angles for each generated array. The first of these projections will be the horizontal sum of each row. The second and fourth projections will be the sum of each of the positive and negative offset diagonals respectively. The third projection is the vertical sum of each column. Finally, the fifth projection is the reverse horizontal sum. These projections are then turned into barcodes by calculating the threshold value for each projection and are then stored in a dictionary.

The dictionary can then be used to search for the most similar image given a query image. At this point in the program, the user is asked to input an Image Class and Image Number in the format Image Class: Image Number. Finally, our search algorithm is used (discussed in more detail later).

Throughout this report, we discuss our barcode generating algorithm and search algorithm, as well as each algorithm's complexity, ways to improve accuracy and then give a conclusion on our results.

Algorithm Explanation

Barcode Generator:

Low Level Explanation of Barcode Generating Algorithm:
Algorithm BarcodeGenerator(Image Dictionary)
Input: Image Dictionary
Output: Dictionary of Barcodes

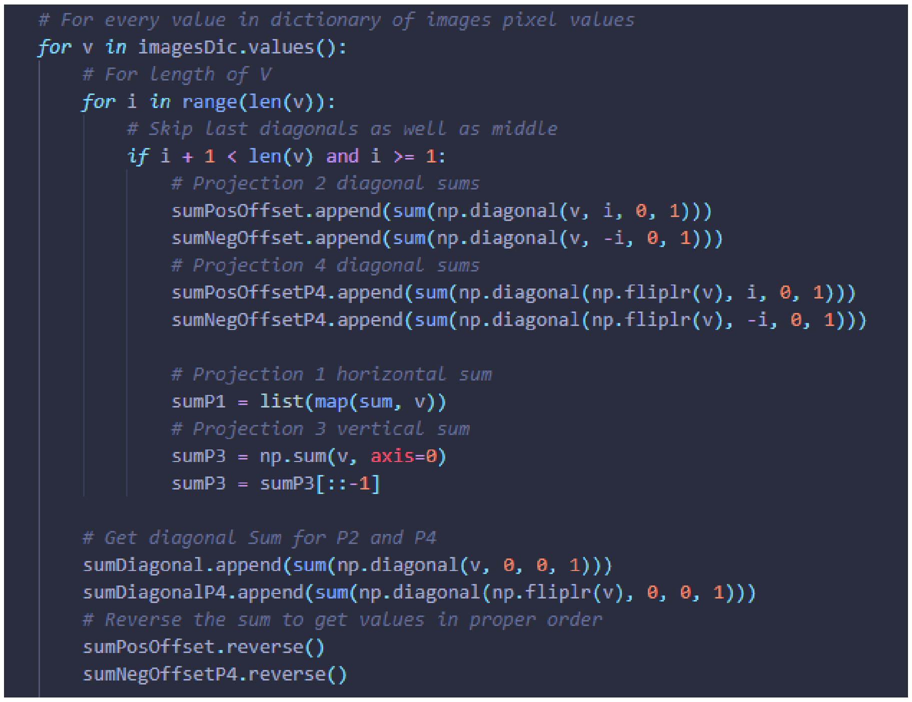
```
For each value in dictionary of images
    For  length of dictionary of images
        If last and middle diagonals are skipped
            Compute projection 2 diagonal sums
            Compute projection 4 diagonal sums
            Compute projection 1 horizontal sum
            Compute projection 3 vertical sum
        Get diagonal sum for projection 2 and 4
        Set projections
        p1= sum of horizontal projections
        p2= sum of projection 2 diagonals
        p3= sum of vertical projections
        p4= sum of projection 4 diagonals
        p5=sum of horizontal projections reversed
        Set Barcodes
        p1C= Call generate_c and return barcode for p1
        p2C= Call generate_c and return barcode for p2
        p3C= Call generate_c and return barcode for p3
        p4C= Call generate_c and return barcode for p4
        p5C= Call generate_c and return barcode for p5
        If counter has reached 10 images
            Increment image counter to next folder
            Set counter to zero
        Initialize barcodes into a dictionary with image key
        Increment counter by 1
        Set All lists to zero
    Return Barcode
```

Explanation:

The algorithm begins by creating variables to store the folder of the image, a counter for each individual image as well as a main counter for each folder. After declaring these variables, a for loop is run that goes through each value in the dictionary of pixelated arrays. This dictionary was created earlier by converting each 28x28 image into a pixelated array. The first function this for loop completes is to ensure the middle and last diagonals of our search are skipped. We will add the middle separately and we do not include the edge diagonals in our projection. Once this is accomplished we begin by finding the diagonal sums for projections 2 and 4. This is done by dynamically finding the sum of each diagonal offset by using numpy's built in diagonal method which takes in the following parameters: list, offset, axis1, axis2, integer value. After this, we find the horizontal and vertical sum for the first and third projections respectively by using python's built in sum method for each value of the dictionary.Then, we find the fifth projection by reversing projection 3. Now we can get the diagonal sum for projections 2 and 4 using the same method to get all the diagonals and begin the process of setting the projections. Refer to the Image below for the source code for more clarification.

The projections are set by concatenating the previously calculated projections. Once the projections have been set to the calculated sums from earlier we can begin generating barcodes by finding the threshold of each of the projections which is done by taking in the projections, finding the largest value, iterating through each separate value and subbing in one if the value is greater than the threshold, else subbing a zero. Once our barcodes have been generated, we store them in a dictionary. We use counters to label each barcode with a key in the form, Image Class: Image Number. At the end of iteration all the lists are cleared. For more clarification on how the barcodes are stored see the image below:

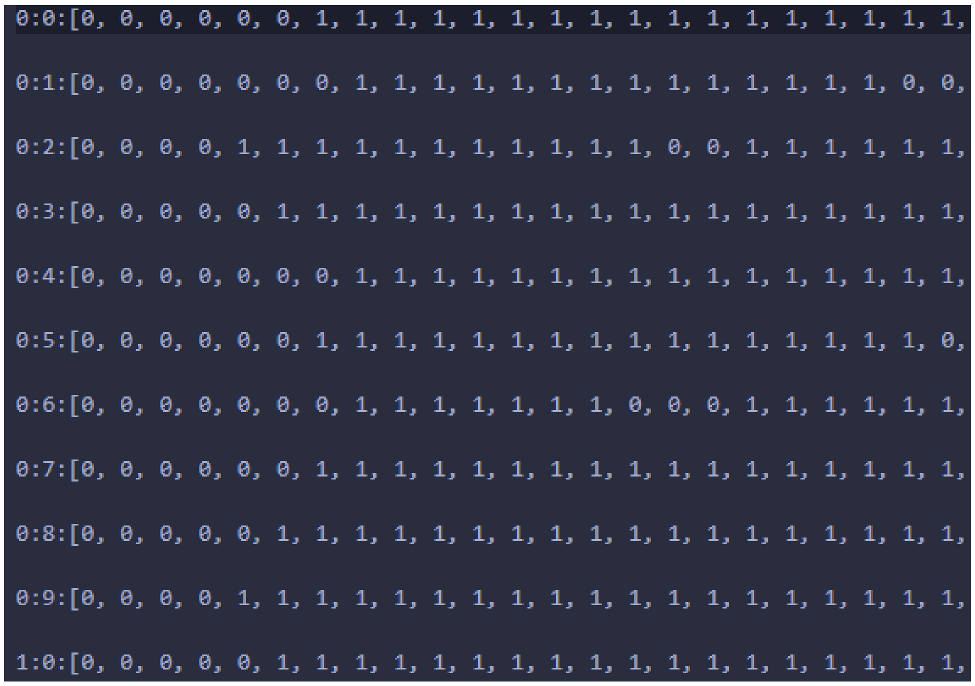
Note: The image does not show the full output



```
# For every value in dictionary of images pixel values
for v in imagesDic.values():
    # For length of V
    for i in range(len(v)):
        # Skip last diagonals as well as middle
        if i + 1 < len(v) and i >= 1:
            # Projection 2 diagonal sums
            sumPosOffset.append(sum(np.diagonal(v, i, 0, 1)))
            sumNegOffset.append(sum(np.diagonal(v, -i, 0, 1)))
            # Projection 4 diagonal sums
            sumPosOffsetP4.append(sum(np.diagonal(np.fliplr(v), i, 0, 1)))
            sumNegOffsetP4.append(sum(np.diagonal(np.fliplr(v), -i, 0, 1)))

            # Projection 1 horizontal sum
            sumP1 = list(map(sum, v))
            # Projection 3 vertical sum
            sumP3 = np.sum(v, axis=0)
            sumP3 = sumP3[::-1]

    # Get diagonal Sum for P2 and P4
    sumDiagonal.append(sum(np.diagonal(v, 0, 0, 1)))
    sumDiagonalP4.append(sum(np.diagonal(np.fliplr(v), 0, 0, 1)))
    # Reverse the sum to get values in proper order
    sumPosOffset.reverse()
    sumNegOffsetP4.reverse()
```



```
0:0:[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0:1:[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,

0:2:[0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,

0:3:[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0:4:[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0:5:[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,

0:6:[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1,

0:7:[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0:8:[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0:9:[0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1:0:[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Search Algorithm:

Low Level Explanation of Barcode Generating Algorithm:
Algorithm: Image Search (Barcode Dictionary, Key)
Input: Barcode Dictionary and Key
Output: Retrieval Accuracy

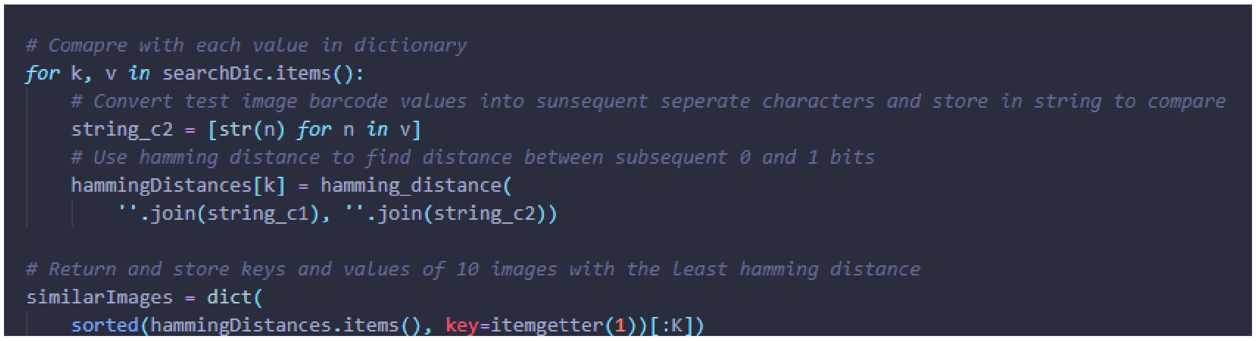
Initialize Dictionary to store all hamming distances
Initialize K as 10
Set search dictionary equal to barcode dictionary
Get key and image class to be searched
Set Image class to Key
Initialize hit and accuracy to zero
Remove query image from dictionary
Set StringC1 to converted barcode of query image

```
For each value in barcode of dictionaries
    Convert the barcode into separated characters
    Set StringC2 to the converted barcode generated
    Initialize and set array to store hamming distances between converted barcodes
Set values of 10 images with lowest hamming distance to an array
For every image class
    For every image
        Set image class to key
        If query image class=image class
            Increment hit by 1
Set retrieval accuracy
accuracy = (number of times image class and query image class match/number of classes) * 100
Display accuracy
```

Explanation:

The algorithm is for the ten key return method and begins by making a dictionary to store the hamming distances between the query image barcode and the rest of the barcodes in the dictionary. We also initialize the variable k to 10 to represent the number of images per folder. The next step is to copy the barcode dictionary onto a separate search dictionary and set the query image equal to whatever value the user inputted. We then declare and initialize a variable for the number of hits we receive as well as a variable for the retrieval accuracy of the query image. To ensure we do not accidentally include the image the user themselves asked to be searched, we remove it from the copied barcode dictionary. The first step in the search process is to convert the query image into separate characters and store it into a string. This way we can calculate hamming distance. We now create a for loop to go through the search dictionary and convert every barcode in our copied dictionary from a barcode into separate characters and store it into a string. For each conversion, we also measure the hamming distance between the converted barcoded and the converted query image from before by calling the hamming distance function which essentially increments the further the distances from subsequent number one’s in each barcode. Then we return and store the ten converted images with the smallest hamming distance into a dictionary.For more clarification please refer to the image below:

To determine the number of hits found, we must compare the keys of the query image and the images we've determined to have the least hamming distance. For every match between keys, we increment our hit variable by one as this means both the query and the similar image are located in the same number folder. Once each of the 10 images has been checked for a hit we divide the hit variable by 10 and multiply by 100 to determine the retrieval accuracy. The algorithm for the one key return method is very similar, but instead of saving the ten lowest hamming distances in a dictionary and then checking to see how many are in the same class as the query image, this method, returns True if the only the image with the smallest hamming distance key is the same as the query image. The results are then output to the user.



```python
# Comapre with each value in dictionary
for k, v in searchDic.items():
    # Convert test image barcode values into sunsequent seperate characters and store in string to compare
    string_c2 = [str(n) for n in v]
    # Use hamming distance to find distance between subsequent 0 and 1 bits
    hammingDistances[k] = hamming_distance(
        ''.join(string_c1), ''.join(string_c2))

# Return and store keys and values of 10 images with the least hamming distance
similarImages = dict(
    sorted(hammingDistances.items(), key=itemgetter(1))[:K])
```

Required Measurements & Analysis

Barcode Generator:

Pseudocode:
Algorithm BarcodeGenerator(Image Dictionary)
Input: Image Dictionary
Output: Dictionary of Barcodes

```
imageCounter ← 0                                        1
Counter ← 0                                             1
mainCounter ← 0                                         1
P1 ← []                                                 1
P2 ← []                                                 1
```

```
P3 ← []                                                              1
P4 ← []                                                              1
P5 ← []                                                              1
P1C ← []                                                             1
P2C ← []                                                             1
P3C ← []                                                             1
P4C ← []                                                             1
P5C ← []                                                             1
Barcode ← []                                                         1
Barcodes Dictionary ← {}                                             1

For every value in ImagesDictionary() do:                            n +1

    For i in range of length(value) do:                              n(n-1)

        If  i + 1 < length(value) and i >= 1 then:                   n*n

(n*(n-1))/2     Projection 2 Diagonal Positive Offset ←+ Sum(np.diagonal(value, i, 0,
1))
(n*(n-1))/2     Projection 2 Diagonal Negative Offset ←+ Sum(np.diagonal(value,-
i,0,1))
(n*(n-1))/2     Projection 4 Inverse Diagonal Positive Offset
                        ←+ Sum(np.diagonal(Flip(value), i, 0, 1))
(n*(n-1))/2     Projection 4 Inverse Diagonal Negative Offset
                        ←+ Sum(np.diagonal(Flip(value), -i, 0, 1))
(n*(n-1))/2     Projection 1 Horizontal Sum ←+ List(map(sum, value))
(n*(n-1))/2     Projection 3 Vertical Sum ←+ Reverse(np.Sum(value, axis=0))

        Projection 2 diagonal←+ sum(np.diagonal(v,0,0,1))
    n*n
        Projection 4 diagonal ←+ sum(np.diagonal(Flip(v),0,0,1))     n*n
        Reverse(Projection 2 Diagonal Positive Offset)               n*n
        Reverse(Projection 4 Inverse Diagonal Negative Offset
                        ←+ Sum(np.diagonal(Flip(value), -i, 0, 1)))  n*n

        P1 ← Projection 1 Horizontal Sum                             n*n
        P5 ← Projection 5 Reverse Horizontal Sum
n*n
        P2 ← Projection 2 Diagonal Positive Offset + Projection 2 diagonal +
                Projection 2 Diagonal Negative Offset                      n*n
        P3 ← Projection 3 Vertical Sum                                     n*n
        P4 ← Projection 4 Diagonal Positive Offset + Projection 4 diagonal +
                Projection 4 Diagonal Negative Offset                      n*n

        P1C ← Generate_C(P1)                                               n*n
        P2C ← Generate_C(P2)                                               n*n
        P3C ← Generate_C(P3)                                               n*n
        P4C ← Generate_C(P4)                                               n*n
        P5C ← Generate_C(P5)                                               n*n

        Barcodes ← P1C + P2C + P3C +P4C + P5C
n*n
        If Counter % 10 == 0 then:                                         n*n
                imageCounter ←+ 1                                            n
                Counter ← 0                                                  n
        mainCounter ← (imageCounter-1, Counter)
n*n
        Barcodes Dictionary [mainCounter] ← Barcodes                       n*n

        Clear All Lists                                                      1

Return Barcodes Dictionary                                                   1
```

Big-O Complexity: Therefore the Big-Oh Complexity is O(n^2)

Search Algorithm for Ten Key Return Method:
Big-O Complexity:

Pseudocode:
Algorithm: Image Search (Barcode Dictionary, Key)
Input: Barcode Dictionary and Key
Output: Retrieval Accuracy

```
hammingDistances ← {}
1
K ← 10
1

Search Dictionary ← Barcodes Dictionary.Copy()
1
queryImage ← Barcodes Dictionary[key}
1
queryImageClass ← key
1
queryImageClass ← queryImageClass [0]
1
Hit ← 0
1
retrievalAccuracy ← 0
1

Delete search dictionary[key]                                                1
String queryImage_c1 ← [Str(n) for n in queryImage]
1

For key, value in Search Dictionary.Items() do:                              n-1
n       String testImage_c2 ← [Str(n) for n in value]
n       hammingDistances[key] ←
hamming_distance(Join(queryImage_c1),Join(testImage_c2))

similarImages  ← dict(sorted(hammingDistances.items(), key←itemgetter(1))[:K])      1

For key, value in similarImages.items() do:                                  n+1
        imageClass ← key[0]
n
```

```
        If queryImageClass == ImageClass do:                                n
            Hit ← + 1                                                      n

retrievalAccuracy ← Hit/K *100                                             1

Return retrievalAccuracy                                                   1
```

Big-O Complexity: Therefore the Big-Oh Complexity is O(n)

Search Algorithm for One Key Return Method:
Big-O Complexity:

Pseudocode:
Algorithm: Image Search (Barcode Dictionary, Key)
Input: Barcode Dictionary and Key
Output: Hit or Miss

```
hammingDistances ← {}
1

Search Dictionary ← Barcodes Dictionary.Copy()
1
queryImage ← Barcodes Dictionary[key}
1
queryImageClass ← key
1
queryImageClass ← queryImageClass [0]
1
Hit ← False
1

Delete search dictionary[key]                                              1
String queryImage_c1 ← [Str(n) for n in queryImage]
1

For key, value in Search Dictionary.Items() do:                            n-1
```

```
n        String testImage_c2 ← [Str(n) for n in value]
n        hammingDistances[key] ←
hamming_distance(Join(queryImage_c1),Join(testImage_c2))

similarImageKey  ← min(hammingDistances, key=hammingDistances.get)        1
ImageClass ← similarImageKey[0]                                            1

If queryImageClass == ImageClass do:                                       n
         Hit ← True
n

Print(The Key of the most similar image is: similarImageKey)

Return Hit                                                                 1
```

Big-O Complexity: Therefore the Big-Oh Complexity is O(n)

Search Results

Brief Introduction:

There are a few ways to test the accuracy of this program.The first method is the single key return method, which will return a hit if the most similar image to the query image is in the same class. The second method determines accuracy by seeing how many of the ten most similar images are in the same class as the query image. Both methods calculate accuracy using the hamming distance function to find barcodes with the least hamming distance which in turn means that they are the most similar barcodes and Images.

1.Single Key Return Accuracy Explanation and Results:

In the Single Key Return method the program first asks the user to input the Image Class and Number they wish to search. The program then takes the key of the query image and removes it from the dictionary of barcodes. Then the program iterates through all keys and values of the barcodes dictionary while converting the barcodes into a subsequent list of characters and comparing it to the query image using the hamming distance. At the same time the hamming distances are stored into a separate dictionary with the same keys. After the iteration is complete, the lowest hamming distance with its key is extracted from the dictionary and returned. If the key is the same as the key of the query image then it is a hit otherwise a miss. The program was then tested for results on all 100 Images. The results are as follows: Out of all 100 images searched, ***51%*** of them returned an image with the same class. These results are about average. Ways the accuracy can be improved are discussed later in this report.

2.Ten Key Return Accuracy Explanation and Results

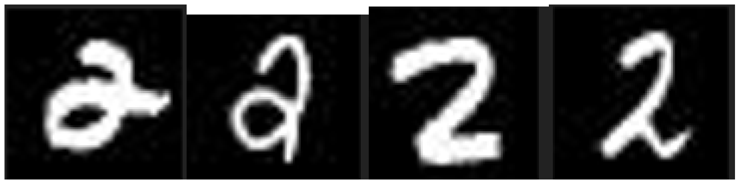
In the Ten Key Return method the program first asks the user to input the Image Class and Number they wish to search. The program then takes the key of the query image and removes it from the dictionary of barcodes. Then the program iterates through all keys and values of the barcodes dictionary while converting the barcodes into a subsequent list of characters and comparing it to the query image using the hamming distance. At the same time the hamming distances are stored into a separate dictionary with the same keys. After the iteration is complete, the barcodes with the ten lowest hamming distance with its keys are returned and stored into a list. The 10 keys are now compared with the key of the query image in a for loop. If the key is the same then it is a hit otherwise it is a miss. The retrieval accuracy is calculated by dividing the number of hits by ten and then returning the value. Out of all 100 Images searched each Images accuracy was noted and stored into a file. It can be viewed in the following link: (https://github.com/huzaifazia17/ContentBasedImageRetrieval/blob/main/accuracy.txt).

The overall Program accuracy was 33%.

This method of measuring the accuracy is more accurate in determining the program accuracy than the first method as it takes into account the ten images with the least hamming distance, the reason being there are ten images in each class. An overall accuracy of 33% is about average and ways the accuracy can be improved are discussed later in this report.

Ideas to Improve Accuracy

The accuracy of our search algorithm is dependent on the number of hits we count over the number of potential hits we have. This means that to improve accuracy we must either increase our number of hits or decrease our potential pool of matching candidates. As simply limiting the images we look at when determining the lowest hamming distances does nothing to improve our algorithm we will ignore that option. Therefore we must look at ways to ensure the images our program provides are all in the same number folder as the image the user inputs. First, let it be clear that the accuracy of this program was affected by the images of each number looking significantly different. For example, all two's were not drawn the same way, with some looking more like the number 8 or the letter a and z. Refer to the images below. All four images were in the set of Number two:



The first way this could be accomplished is by increasing the number of projections created for each image. This means that each image will have a longer and more unique barcode to compare hamming distances against. The further complication of the barcode ensures that the 10 images with the lowest hamming distance will be in the same folder.

Furthermore, another way to improve the accuracy of the algorithm is to add more images to search through. This means that we will have significantly more barcodes to compare against, thus increasing the probability that the ten images with the lowest hamming distance are in the query image folder.

A third way to improve the accuracy would be to have higher-quality images in the set. For example 1000 x 1000 pixel images rather than 28 x 28 pixels. This will give the algorithm a much larger area to work on which will increase the accuracy by giving more differentiation to the images as a result of the larger pixel density. In addition to this, a good portion of data is lost by the threshold process. The consequence of using this design method means that we only look at the highest values produced by our barcode and make everything else zero. Rather than this, if we instead converted the entire projection into another number base, we would be able to differentiate between many more values than just 1 and 0.

Additionally, the division of each image and the individual inspection of each resulting sub-image would allow us to greatly increase hit accuracy. Rather than setting our projections on the entire image, we can repeat the projection process multiple times in different areas of the photo. This results in more barcodes to compare between images, thus a better chance of finding an image with a low hamming distance. Furthermore, a quick sure-fire way to improve accuracy without too drastically changing the code is to refine the threshold process. Incorporating the division of the image with this idea allows us to make a larger amount of smaller projections rather than a larger amount of longer ones. Thanks to these shorter projections, we can use the threshold process with projections as small as two values, thus letting us keep the binary conversion without sacrificing accuracy.

Conclusion

The purpose of this project was to design, implement and test real-world searching algorithms. To accomplish this we used a set of 100 images to search for the most similar image in the data set, given that a user inputs a query image. The similarities between images were determined by first converting each of the images into a barcode through the use of projections and threshold values. Once accomplished each of these barcodes was separated into separate strings to accurately compare the hamming distance between the query barcode and the current barcode being checked. The images with the lowest hamming distance will be stored to then check for retrieval accuracy. This is checked by determining which of the 10 cases with the lowest distance originate from the same number folder as the query image. The larger this number is, the better the retrieval accuracy of our search algorithm.

Our barcode and search algorithm have a Big-Oh complexity of O(n^2) and O(n) respectively, which can be improved by using more complex data structures for the insertion and editing of the key and values such as trees and hash tables instead of dictionaries that we have implemented. Furthermore, our retrieval accuracy was 33% in the ten key return method, and 51% in the one key return method, which is about average considering the size of the data set, and the major difference between each image in the set as described in the Ideas to improve accuracy section of the report.

Thus, this project has given insight into how content-based image retrieval can be implemented on a small scale and an idea of how larger-scale programs as such work, like CAT scans and X-rays.