Data Structures

Course Project: Content-Based Image Retrieval (CBIR) Using Barcodes

Group Members

Mohsin Rehman -100788028

Due Date: Thursday, April 14th, 2022

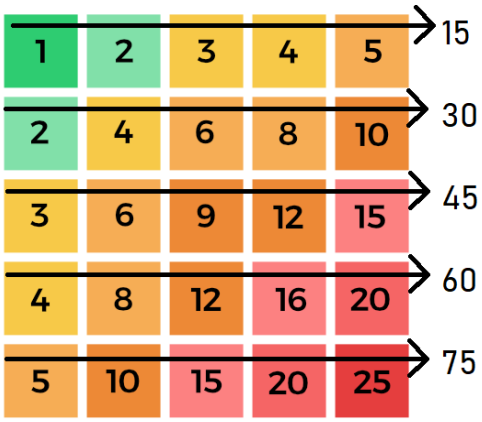
Introduction

Within the duration of this project, we were given a dataset of 100 28x28 pixel images that represented digits ranging from 0 to 9 (10 for each digit). We were then given the task of writing write an algorithm that searches for the most similar image in the dataset and returns the number corresponding to it. Projections across the image array at a variety of angles will be used to create barcodes and search the query for the nearest image.

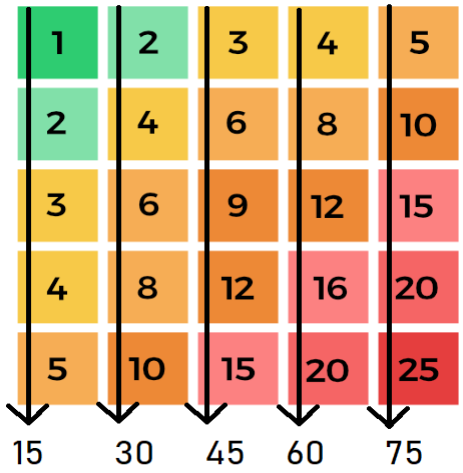
Algorithms Explanation in detail

The first Algorithm to be explained is the Barcode_Generator Algorithm. The images given are 28x28 (n x n) so we will convert them into arrays where the indexes will be based on their color values ranging from 0-255. Once we have obtained the NxN(28 x 28) array made from the image we will then compute the projections. For the 90 degrees projection, we will take the sums of the values going across the rows of the array (shown in figure 1-1) and store these values in a new array. To build upon, the 180-degree projections will take the sum going down the column of the array(as shown in figures 1-2) after we will store the sums in an array. In the case of the 45-degree projection, we get the sum of the indexes going for the diagonals going from the top right to the bottom left. We get the sum of math.floor(n/2) diagonals from the main diagonal to get (n-1) values (shown in figures 1-3) and store these values in a new array. For the 135 degrees projection, it is identical to the 45-degree projection, however, we get the sums of the diagonals from the top right to the bottom left(shown in figures 1-4). Moreover, we added 2 more projections than initially intended and these are the 22.5 and 112.5-degree projections. These 2 projections work identically to the 45 and 135-degree projections however they move across an additional row value to decrease the slope (shown in figure 1-5). After obtaining all the sum arrays we then convert them into projections. To do this we take the average of each sum array and set the values under the average to be zero and the rest to be 1. After the projections are obtained we then store them into a text file that will act as a query.

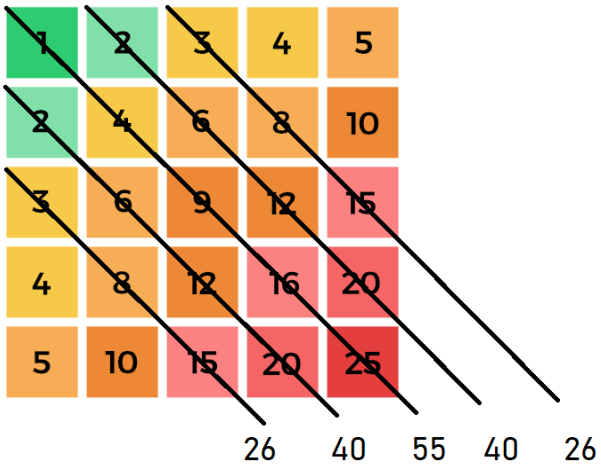
The second algorithm that will be explained is the Search Algorithm. After all the projections for the images in the MSNIT dataset are stored in the query, we will convert this text file into a string. We will then convert this string into an array where each line is considered an index. The projection in the Search Algorithm will be compared to all lines in the text query, however, it will not be compared to the lines corresponding to the same image. During the comparisons, the hamming distance for each image will be calculated (so there will be 99 hamming distances in total) and they will be stored in an array. We will then calculate the average hamming distances corresponding to each number (i.e all 10 hamming distances for the ten (0) images) and store them in a new array. After the averages have been stored, we will return the index containing the lowest number and that will be the generated output.



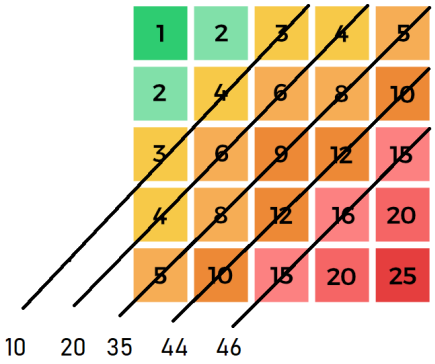
(Figure 1-1: 90-degree projection sum)



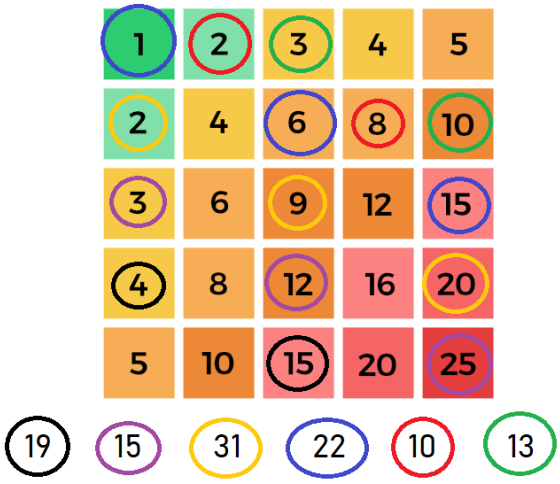
(Figure 1-2: 180-degree projection sum)



(Figure 1-3: 45-degree projection sum)



(Figure 1-4: 135-degree projection sum)



(Figure 1-5: 22.5-degree projection sum)

Psuedocode For Algorithms

Pseudocode For Barcode Generator

Barcode Generator(image array, image name, folder number){

22.5degreeproj ← empty array
45degreeproj ← empty array
90degreeproj ← empty array
112.5degreeproj ← empty array

```
    135degreeproj ← empty array
    180degreeproj ← empty array

    projectionArray ←empty array

    projectionArray [0] ← get22.5degreeproj(22.5degreeproj)
    projectionArray [1] ←get45degreeproj(45degreeproj)
    projectionArray [2] ←get90degreeproj(90degreeproj)
    projectionArray [3] ←get112.5degreeproj(112.5degreeproj)
    projectionArray [4] ←get135degreeproj(135degreeproj)
    projectionArray [5] ←get180degreeproj(180degreeproj)
  String s ← image name
  s += new line
    For x in range 6


          s += arrayToString(projection[x])
        s += new line
    s +="image corresponds to" + image number ← this is to help get the accurate hamming distance values in search algorithm
    Store s in a text file.
```

(figure 2-1: pseudocode for barcode generator)

```
Search Algorithm (projection, image name):

    Convert query text file to string
    Convert String into an array corresponding to lines
    hammingArray ← empty array
    hamming_distance ← 0
    projectionsArray ← empty array
    avgArray ← empty array
    numOf0 ← 0
    numOf1 ← 0
    numOf2 ← 0
    numOf3 ← 0
    numOf4 ← 0
    numOf5 ← 0
```

```
numOf6 ← 0
numOf7 ← 0
numOf8 ← 0
numOf9 ← 0
For x in range length of TextFileArray:

        If TextFileArray[x] contains image name:
                For j in range 6 do
                        TextFileArray[x+6] ← " "        ←-- removes image from list

        If TextFileArray[x] contains "["  #opening for list

                projectionsArray.add(convertLineIntoIntArray(TextFileArray[x]))
        If TextFileArray[x] contains "image correspond to x" then
                numOfX++ #put x here instead of number to avoid 10 if statements

For x in range projectionsArray
        K == x%6  #keeps looping the first projection values for comparison
        For j in range length of projectionsArray[x]

                If projections[k][j] !== projectionsArray[x][j]
                                hamming_distance += 1


                if(k==0 and j==0 and x!==)

                        hamming_distance ← 0

hammingArray.add(hamming_distance)

LengthArray= [numOf0, numOf1, numOf2,.....numOf9]
Avg ← empty array
 Length ← 0
For x in range 10
        Sum ←0
        For j in range LengthArray[x]

                sum+= hammingArray[x]
        Leng +=LengthArray[x]
        Avg.add(sum/LengthArray[x])

Output the number corresponds to min index of Avg array
```

(figure 2-2: pseudocode for barcode generator)

Measurements and Analysis:

Accuracy/Hit Ratio:

The accuracy was obtained by running both algorithms for all images in the dataset(100 times) and checked if the image imputed was matched to its correct number. Every time the correct number was matched we incremented the hit ratio variable by 1. Our driver code test all 100 values in 1 run and outputted a hit ratio of 60/100 or 61%. The search results and values obtained for each image can be seen in figures 4-1 to 4-10.

Run time Complexity:

The Run time complexity for the barcode generator was calculated to be O(n^3) as one of the methods called in the algorithm runs for O(n^3) time, and that was the method with the largest run time complexity. The n-value in the run time corresponds to the length of the image array in the case of Barcode_Generator. However, in the case of Search_Algorithm the run time was calculated to be O(n^2) where n was the number of projections in the Query text file. However as we stated above the driver code test all 100 values in 1 run, and these algorithms are stored within nested for loops, so the run time for our driver code that runs both methods is O(n^5).

```
def Barcode_Generator(numzero, projection, name): #one of the main algorithms that generates the barcodes/projections and prints them out

    temp = [] #empty list temp that will temporarily store the barcodes generated for each angle to be appeneded into another list

    ttpf_projection(numzero, temp) #calls function to get the 22.5 projections   method runs in O(n^2)
    projection.append(temp) #append projection into the projection list

    temp = [] #set temp to be empty again so it can store the value of the next barcode

    ff_projection(numzero, temp) #calls function to get the 45 projections    method runs in O(n^3)
    projection.append(temp) #append projection into the projection list

    temp = [] #set temp to be empty again so it can store the value of the next barcode

    ninty_proj(temp, numzero) #calls function to get the 90 projections  method runs in O(n^2)
    projection.append(temp) #append projection into the projection list
    temp = [] #set temp to be empty again so it can store the value of the next barcode

    otpf_projection(numzero, temp)  #calls function to get the 112.5 projections    method runs in O(n^2)
    projection.append(temp) #append projection into the projection list
    temp = [] #set temp to be empty again so it can store the value of the next barcode

    otf_projection(numzero, temp) #calls function to get the 90 projections   method runs in O(n^2)
    projection.append(temp) #append projection into the projection list
    temp = [] #set temp to be empty again so it can store the value of the next barcode

    oneeighty_proj(temp, numzero) #calls function to get the 180 projections
```

(figure 3-1: Run time for methods in barcode generator)

```
for i in range(len(listOfProjections)):    for loop and nested for loop run in O(n^2) time

    for j in range(len(listOfProjections[i])):

        k = i % 6;

        if (projection[k][j] != listOfProjections[i][j]):
            sum += 1

        if ((k == 0) and (i != 0) and (j == 0)):
            distance.append(sum)
            sum = 0
```

(figure 3-2: Run time calculations for Search Algorithm)

Search Results

Image Name	Image Number	Outputted Number	Hit Ratio
10007	0	1	0
10010	0	0	1
10017	0	0	2
10032	0	1	2
10039	0	0	3
10123	0	0	4
10156	0	7	4
10207	0	0	5
10231	0	0	6
10242	0	0	7

(figure 4-1: Table For Search Results For Algorithm For Number 0)

Image Name	Image Number	Outputted Number	Hit Ratio
1000	1	1	8
10006	1	1	9

10009	1	1	10
10040	1	1	11
10042	1	1	12
10076	1	8	12
10175	1	1	13
10189	1	1	14
10190	1	1	15
10192	1	1	16

(figure 4-2: Table For Search Results For Algorithm For Number 1)

Image Name	Image Number	Outputted Number	Hit Ratio
1011	2	4	16
1020	2	7	16
10000	2	2	17
10027	2	2	18
10031	2	2	19
10062	2	2	20
10068	2	2	21
10111	2	1	21
10285	2	8	21
10297	2	2	22

(figure 4-3: Table For Search Results For Algorithm For Number 2)

Image Name	Image Number	Outputted Number	Hit Ratio
10016	3	3	23
10029	3	3	24

10045	3	3	25
10049	3	3	26
10117	3	8	26
10121	3	3	27
10140	3	7	27
10179	3	1	27
10193	3	3	28
10299	3	9	28

(figure 4-4: Table For Search Results For Algorithm For Number 3)

Image Name	Image Number	Outputted Number	Hit Ratio
1004	4	4	29
1009	4	4	30
10026	4	9	30
10070	4	4	31
10086	4	4	32
10097	4	4	33
10160	4	4	34
10205	4	6	34
10212	4	4	35
10289	4	9	35

(figure 4-5: Table For Search Results For Algorithm For Number 4)

Image Name	Image Number	Outputted Number	Hit Ratio
10001	5	9	35
10030	5	3	35

10037	5	8	35
10047	5	3	35
10051	5	5	36
10057	5	4	36
10069	5	4	36
10100	5	4	36
10103	5	4	36
10184	5	4	36

(figure 4-6: Table For Search Results For Algorithm For Number 5)

Image Name	Image Number	Outputted Number	Hit Ratio
10021	6	6	37
10066	6	6	38
10072	6	6	39
10087	6	6	40
10102	6	6	41
10110	6	6	42
10129	6	6	43
10134	6	6	44
10144	6	6	45
10151	6	6	46

(figure 4-7: Table For Search Results For Algorithm For Number 6)

Image Name	Image Number	Outputted Number	Hit Ratio
1022	7	7	47
10019	7	7	48

10038	7	7	49
10056	7	1	49
10073	7	7	50
10091	7	7	51
10157	7	7	52
10161	7	9	52
10165	7	4	52
10223	7	9	52

(figure 4-8: Table For Search Results For Algorithm For Number 7)

Image Name	Image Number	Outputted Number	Hit Ratio
10	8	9	52
1025	8	8	53
10024	8	3	53
10041	8	1	53
10101	8	1	53
10109	8	1	53
10112	8	8	54
10115	8	1	54
10139	8	0	54
10176	8	4	54

(figure 4-9: Table For Search Results For Algorithm For Number 8)

Image Name	Image Number	Outputted Number	Hit Ratio
100	9	9	55
10003	9	9	56
10083	9	7	56

10120	9	9	57
10126	9	9	58
10131	9	9	59
10135	9	4	59
10181	9	7	59
10210	9	9	60
10236	9	9	61

(figure 4-10: Table For Search Results For Algorithm For Number 9)

Ideas to improve the retrieval accuracy:

For our current algorithm we have made a variety of changes to improve the hit ratio. The first being we increased the projections by a number of 2 (22.5 and 112.5) and this increased our hit ratio from 54% to 55%. Additionally, we took the averages of the hamming distances rather than taking the lowest hamming distance value and this increased our hit ratio to 60%. However, we could implement new ideas to improve our hit ratio further such as using the sum arrays and calculating the euclidean distance. This could potentially improve our hit ratio because the loss of data when converting to a projection can account for a decreased hit ratio. To build upon this, we could split the images into 4 equal parts and calculate the barcode for each of them, and then compare to increase the accuracy of the code. The issue pertaining to this solution is that it will increase the run time complexity and lower the overall efficiency of the program. The last solution is to increase the size of the dataset with more images so that there is less variance with the projections. This solution will not increase the run time complexity by a lot as both methods would still run in O(n^3) and O(n^2) to their respective n values but these respective n values may increase.

Conclusion:

During the span of this project, we were given a dataset of 100 images that represented handwritten numbers, and had to create algorithms that would find the most similar number that corresponds to the image. The two algorithms that were developed were, Search_Algorithm and Barcode_Generator. The algorithm Barcode Generator would generate barcodes corresponding to the 22.5,45,90,112.5,135,180 degree projections and store them into a Query text file. The run-time complexity for the Barcode Generator algorithm was O(n^3) where n is the length of the array. To build upon, the Search Algorithm would compare an image's barcode to all 99 other images in the Query and obtain the hamming distances corresponding to each one. After obtaining the hamming distances, the Search Algorithm would then get the averages of the hamming distances corresponding to each number and return the lowest index in the average array. The run-time complexity for the Search Algorithm was O(n^2) where n is the number of projections in the Query. Moreover, after running both of these algorithms for all 100 images in the data set we got an accuracy of 60%. We intended to improve on this code in the future by using a variety of methods such as an increased dataset and using the euclidean

distance instead of hamming distance. In sum, we have created two algorithms that would take in an image corresponding to a handwritten number and output what that number is with a 60% accuracy rate.