Image Processing and Categorization

Xinyi Li, Yian Mo, Zhihe Shen

The purpose of our group project is to design and implement basic-level image processing and categorization. Our design consists of three general steps - image preprocessing and standardizing, categorization, and comparison, in order to return the closest k images. Each step contains multiple substeps conducted in sequence. To fine-tune the efficiency of run-time in our program, we focus on the preprocessing and categorization in order to reduce the number of comparisons needed within each category. Further thoughts on potential improvement of program performance will be mentioned at the end of the report.

Dataset

We find out our test data by googling "black and white shapes". The data contain different shapes' images with various qualities to satisfy our testing purpose for each substep. We import these images into Matlab and use built-in functions to convert them into binary matrices as the input for our python program later. We also change all 0's to 1's, 1's to 0's in the original matrices in correspondence to our definition of 0 as white and 1 as black.

Preprocessing and Standardizing

This step consists of six substeps: convert background, de-noise, rotate, border, scale and center. For each input matrix, we conduct these six functions in sequence to eventually get a new 50*50 matrix with a large shape in the middle of the frame, which enables pixel by pixel comparison between two matrices in the categorization stage.

- **Convert Background**

This function is to ensure each matrix we process later is black(1) in shapes and white(0) in the background. In this way we are able to detect the edges of the shapes by looping through the pixels to find the 1's on the outside, then rotate, scale or center it as needed. Our approach to detect whether a matrix has a black background is to scan through all the boundary pixels of the matrix and put them into a list. Then we check if >80% of pixels in the list are 1's. If it is, we determine that the matrix has black background. Then we loop through the matrix to change all 0's to 1's, 1's to 0's.

- **De-noise**

De-noise function cancels out the bad influence that random black pixels will have on our categorization and comparison results. For each black pixel in the matrix, we check the values of its up, down, left, right four neighbors. We define that if all four neighbors equal to 0, the current black pixel is a "noise" and we change it to white.

Our initial plan is to check diagonal neighbors as well to increase the accuracy of this function. However, if we use a 3*3 square to loop through the entire matrix, we may encounter indivisible row lengths and column lengths, which will cause extra trouble with rounding or cutting the extra edges. Another concern is run time complexity of the looping approach. With the above two consideration, we decide to only check four adjacent neighbors instead.

- **Rotate**

Rotate function is used to find the line best fit the image and then rotate the image if necessary to make the fit line in the vertical direction in the end. This is one of the standards that we want every image meets after preprocessing in order to help our later comparison. To find the line best fit the image, we first tried linear regression to find A and B in the equation Y= Ax + B. However, the result turned out to be incorrect and useless because it was not linear in that there are several y values according to single x values. Then we realize that the line connecting the top leftmost pixel and bottom rightmost pixel may be helpful though it is not the line perfectly dividing the image in half. Therefore, we decide to use that line as our fit line and calculate the degree of rotation using the arctan of its slope. For rotation, we use a function in scipiy library (scipy.ndimage.interpolation.rotate), which takes a matrix and a degree as arguments and returns the new matrix.

- **Border**

  The shapes in different test data can vary in sizes and locations in the frame. We write border function to cut out the shapes from original matrices so that we can enlarge or shrink it to fit them perfectly in the 50*50 frame. This way we can successfully categorize images which belong to the same type (e.g. triangles with different sizes and locations) into similar categories regardless of their actual shapes and locations in the original images.

  To detect edges of the shape, we scan each row and each column of matrix, respectively, document the x, y indexes of the first and last black pixels, calculate the length of longer edge out of two, then return a new square matrix containing the shape with the longer edge as the length of sides of the square matrix.

- **Scale**

After preprocessing the images, the dimension may change especially after the rotation process. Therefore, the Scale function is written to make every image have an identical dimension(50*50) in the end to facilitate the comparison. It includes two subfunctions: shrink and enlarge depending on the original image's size.

➔ **Shrink**: For an n x n image shrinking to 50 x 50 image, we compute R, the ratio of transformation If n is divisible by 50, than ratio R = n/50. Else, the ratio R= n/50+1. Then for each 1 x 1 pixel in the new image it corresponds to R x R matrix in the original image and the value takes the majority of R x R matrix. For the case that n is not divisible by 50, when we loop till the border pixels, it may happen the out of bound error. In this situation, we only have to compute the majority of the matrix that is left.

➔ **Enlarge**: For an n x n image enlarging to 50 x 50 image, we compute R, the ratio of transformation. If 50 is divisible by n, than ratio R = 50/n. Else, the ratio R= 50/n+1. Then for each 1 x 1 pixel in the old image it corresponds to R x R matrix in the new image and the value in R x R matrix is as same as the 1x 1 matrix. For the case that 50 is not divisible by n, we will finally get a matrix that is larger than 50x 50. Then we simply cut a 50 x 50 matrix from it.

- **Center**

  After all the previous steps, it is possible that the shapes are not perfectly in the center of the matrix. Even slight dislocation may affect our pixel by pixel comparison results, so we want to ensure each shape can be in the center as much as possible by using center function.

  First, we calculate the x,y indexes of the center of the shape by dividing sum of row(col) indexes of all black pixels by the number of black pixels in total. We calculate the index difference between center of the shape and center of the matrix. Then we create a new 50*50 matrix with all 0's inside. For all pixels at index [m][n] in original matrix that are black, paint pixels at index [m+rowdiff][n+coldiff] in new matrix to black.

Categorization

After preprocessing the dataset, every image should have become standardized as we want and we would be able to have a good comparison based on that. But we can't compare them one by one, which will result in incredibly terrible runtime. Therefore, we use decision tree to further categorize the database according to symmetry, convexity and area in order to decrease comparisons. After the categorization, each time when we try to find the closest k images for an image, we only need to compare it with the images in the same category instead of all the images. This way largely reduces the runtime.

- **Symmetry**

  Symmetry function is simply used to determine whether an image is symmetric or not. It checks the symmetry in both top-bottom and left-right direction. It any of these two directions indicates symmetry, then the image is symmetric. The algorithm is that it loops through the row or column and find the corresponding row or column pixel to see whether the two is same and calculate a correct rate. If the correct rate is larger than 80%, we say it is symmetric.
- **Convexity**

  Our shapes can be convex or nonconvex. We check their convexity to further categorize the images. First, we add locations of all black pixels to a list. We randomly pick pairs of pixels. We find the connecting lines between those pairs by calculating the slope and y-intercept. Then we iterate through those lines from small x value to large x value and check if all pixels between those pairs are also black. If they are all black, the shape re likely to be convex. If any of the pixels re white, then the shape must be nonconvex.

  The number of pairs of pixels that we randomly pick can not be too small, because we may skip some white pixels that are on the connecting line between those pairs. The number also can not be too large, because it would largely increase the runtime of our program. So we decide to choose 600 pairs after testing on our datasets.
- **Area**

  Another standard we use to categorize those images is the percentage of black pixels. This is reasonable because after our preprocessing, images belonging to the same category should have similar percentage of black pixels. The way we did was simply iterating through the matrix to get the number of black pixels and then calculating its percentage to the total number of pixels. We use 50% as a separator to categorize shapes.

Comparison

We calculate the squared error pixel by pixel as the distance between two images. Originally, we used KD Tree to generate the output matrix (python sklearn library). However, it was difficult to convert index to file names. So we instead sort by distance to get the closest k images.

Summary and Future Improvement

Our program is designed for general image comparison and we are not aware of the database size or image size beforehand. Therefore, we only chose to have 8 categories for the decision tree. If facing a very large database, we can add more layers to have a larger decision tree and more categories. This will make each category have less images for comparison.There also are some other improvements we can do in the future, which are limited by the time for now. For example, there are some repetitive loops in each preprocess step. We may further reduce the runtime by combining some of those steps and decreases the loop times.We may also considering filling the shapes that are empty in the middle so that an empty circle can stay in the same category with the filled circle. We may also improve our comparison method using KD tree rather than sort to improve run time a little bit, which may be more helpful facing a much larger dataset.