Cairo University

Computer Vision (SBE3230)

Task 4

Name	Sec	BN
Dina Hussam Assem	1	28
Omar Ahmed Anwar	2	2
Omar Saad Mohamed	2	3
Mohamed Ahmed Ismail	2	16
Neveen Mohamed Ayman	2	49

Supervised By

Dr. Ahmed Badwy

TA: eng. Peter Emad & eng. Lila Abbas

Table of Contents

Introduction:

It’s an image processing implementation functions project implemented in C++ with a desktop application (Qt).

Algorithms Implemented And How They Work:

1) Optimal Thresholding:

1. Convert the input image to grayscale.
2. Calculate the initial threshold value by taking the average of the four corner pixels in the grayscale image.
3. Loop until the difference between the previous and current threshold values is less than 0.02.
4. Iterate through all the pixels in the image and separate them into foreground and background based on their intensity values relative to the threshold value.
5. Calculate the mean intensity values for the foreground and background pixels.
6. Calculate the new threshold value as the average of the mean foreground and mean background intensities.
7. Repeat steps 4-6 until the difference between the previous and current threshold values is less than 0.02.
8. Apply the final threshold value to the grayscale image to create a binary thresholded image.
9. Return the binary thresholded image.

2) Otsu Thresholding:

1. The function **otsu_thresholding** takes an input image of type cv::Mat and returns a thresholded image of the same type.
2. The input image is converted to grayscale.
3. A histogram of pixel intensities is computed using the **histo function**, which is described below.

    - **Histo function implementation:**

        1. The function **histo** takes an input image of type cv::Mat and returns a std::map<int, int> containing the histogram values.
        2. A map<int, int> named histogram is initialized to store the histogram values.

    3. A nested loop is used to iterate through every pixel in the image.
    4. That function is used to access the pixel value at coordinates (i, j) of the image.
    5. The pixel value is used as a key to increment the corresponding value in the histogram map.
    6. After all pixels have been processed, the histogram map is returned.

4. The variable size is assigned the total number of pixels in the grayscale image.
5. A loop is used to iterate through the histogram and compute the optimal threshold value using the Otsu algorithm.
6. Within the loop, variables w_b, mu_b, w_f, mu_f, and sigma_b are initialized and computed to represent **the within-class variance** and **between-class variance**.
7. The threshold value is updated if the computed between-class variance sigma_b is larger than the previous maximum.
8. The final threshold value is used to threshold the grayscale image using the **thresholding function**, which is described below.

- **Threshold function implementation:**

    1. The function **thresholding** takes two inputs: a grayscale gray_image of type cv::Mat and an integer threshold representing the threshold value.
    2. A new cv::Mat named threshold_image is created using the clone() function to create a copy of the input gray_image.
    3. A nested loop is used to iterate through every pixel in the image.
    4. That function is used to access the pixel value at coordinates (row, col) of the gray_image.
    5. If the pixel value is greater than the specified threshold, the corresponding pixel value in threshold_image is set to 255 (white), indicating the pixel belongs to the foreground.
    6. If the pixel value is less than or equal to the specified threshold, the corresponding pixel value in threshold_image is set to 0 (black), indicating the pixel belongs to the background.
    7. After all pixels have been processed, the threshold_image is returned.

9. The thresholded image is returned.

3) Global Spectral Thresholding:

1. Convert the input image to grayscale.
2. Calculate the histogram of the grayscale image using a helper function called histo.
3. Iterate over all possible pairs of threshold values by using pointers to the histogram map.
4. For each pair of threshold values, calculate the mean and probability of each class of pixels and the between-class variance.
5. If the between-class variance is greater than the current maximum variance, update the maximum variance and the threshold values.
6. Return the binary thresholded image using the selected threshold values by calling another helper function called double_thresholding.

4) Local Spectral Thresholding:

1. The function takes an input image and calculates the number of rows and columns in the image.
2. The function defines four regions of interest (ROIs) for each of the four parts of the image using the Rect class. Divide the image into 4 quarters.
3. The **spectral_thresholding** function is applied to each part separately.
    - **Spectral_thresholding discussed in point 3.**
4. Copy the thresholded parts back into the thresholded_image using the ROIs.
5. The function returns the final thresholded image.

5) K-means Segmentation:

K-means segmentation is a popular unsupervised clustering algorithm used for image segmentation. The goal of the algorithm is to partition an image into K segments based on the similarity of their pixel values.

The K-means algorithm works by first randomly initializing K cluster centers, where each center represents a segment in the image. Then, it assigns each pixel in the image to the nearest cluster center based on its color value. After all pixels have been assigned, the algorithm computes the mean color value of each cluster, and updates the cluster centers accordingly. This process is repeated iteratively until convergence.

- **Gray images:-**

  Algorithm:

  1. Convert the input image to a 2D array of floats and normalize the values to be between 0 and 1.
  2. Initialize the cluster centers randomly using OpenCV's RNG class.
  3. Assign each pixel to the nearest cluster center using Euclidean distance.
  4. Update the cluster centers as the mean of the assigned pixels.
  5. Create the segmented image by mapping each cluster label to a grayscale value between 0 and 255.

- **RGB images:-**

  Algorithm:

  1. Convert the input color image to a 2D array of floats with 3 channels (for the red, green, and blue values).
  2. Initialize the cluster centers randomly using OpenCV's RNG class.
  3. Assign each pixel to the nearest cluster center using Euclidean distance as the distance metric.
  4. Update the cluster centers as the mean of the assigned pixels.
  5. Create the segmented image by mapping each cluster label to a color.

6) Region Growing Segmentation:

Region growing is a popular method used for image segmentation, which involves dividing an image into regions or segments based on similarities in pixel values or other image properties. The region growing approach starts by selecting an initial seed point or set of seed points and then iteratively adding neighboring pixels to the region if they meet certain similarity criteria.

The advantage of region growing is that it can produce accurate and detailed segmentations that can be tailored to specific applications. It is also a relatively simple and computationally efficient method that can be applied to a wide range of images. Additionally, region growing can handle noise and small variations in intensity, making it robust to some degree of image degradation and noise.

However, region growing also has some disadvantages. One of the main challenges is selecting an appropriate seed point or set of seed points, which can be difficult and time-consuming. Additionally, the performance of the method can be sensitive to the similarity criteria used, and it may be challenging to choose appropriate criteria for different types of images. Finally, the method can be sensitive to the initial conditions and may require fine-tuning to produce optimal results.

1. This is a function called **regionGrowingHelper** that performs region growing segmentation on an input image using a seed point and a threshold value. The function assigns a label to each pixel in the segmented regions and returns a colorized image where each region is shown in a different color
2. The function takes an input image, a seed point, a threshold value, a label number, and a colorflag as inputs.
3. The function creates an empty label matrix with the same size as the input image.
4. The function creates a queue for storing the neighboring points to be checked and initializes it with the seed point and its label number in the label matrix.
5. Depending on the colorflag, the function calls either **region_gray_img_det** or **region_color_img_det** to perform region growing segmentation on the input image.
    - **region_gray_img_det Implementation:**
        1. The function takes a queue of seed points, the number of rows and columns in the image, the input image, the label matrix, and a threshold value as inputs.
        2. The function initializes a variable newLabel to 0, which will be used to assign labels to the segmented regions.
        3. The function loops through the seed points in the queue and processes each pixel in the image by checking its 8-connected neighbors. For each neighbor of the current pixel that is within the threshold and has not been labeled yet, the function assigns the current pixel's label to the neighbor and adds it to the queue to be processed later.
        4. The function returns the label matrix.
    - **region_color_img_det Implementation:**
        - The function implemented at the same method of **region_gray_img_det** but on three channel image.
6. The function calls either **colorize_Region_Gray_Scale_Img** or **colorize_Region_Color_Scale_Img** to create a colorized output image.
    - **colorize_Region_Color_Scale_Img Implementation:**
        1. The function takes the number of rows and columns in the image, the input image, the label matrix, and an output image as inputs.
        2. The function loops through each pixel in the label matrix and assigns a color to the corresponding pixel in the output image based on its label.
        3. For each pixel in the label matrix that belongs to a segmented region (i.e., has a non-zero label), the function assigns the corresponding pixel in the output image the same color as the corresponding pixel in the input image.
        4. The function returns the output image.
    - **colorize_Region_Gray_Scale_Img Implementation:**
        1. The function takes the number of rows and columns in the image, the label matrix, and an output image as inputs.

    2. The function loops through each pixel in the label matrix and assigns a unique gray-scale color to the corresponding pixel in the output image based on its label.
    3. For each pixel in the label matrix that belongs to a segmented region (i.e., has a non-zero label), the function assigns a unique gray-scale color to the corresponding pixel in the output image based on its label. The color is generated by taking the modulus of the label value multiplied by some constant values with 255.
    4. The function returns the output image.
7. The function returns the colorized output image.

7) Agglomerative Segmentation:

Agglomerative clustering is a popular method used for image segmentation, which involves dividing an image into clusters or segments based on similarities in pixel values or other image properties. The agglomerative clustering approach starts by considering each pixel as a separate cluster and then iteratively merging the two closest clusters until a stopping criterion is met. This method is widely used in computer vision and image processing applications.

Agglomerative clustering is a versatile method that can produce accurate and detailed segmentations that can be tailored to specific applications. It is also a relatively simple and computationally efficient method that can be applied to a wide range of images. Additionally, agglomerative clustering can handle noise and small variations in intensity, making it robust to some degree of image degradation and noise.

One of the main disadvantages of agglomerative clustering is that it can be computationally expensive, particularly for large datasets or images. Additionally, the performance of the method can be sensitive to the similarity criteria used, and it may be challenging to choose appropriate criteria for different types of images. Finally, the method can be sensitive to the initial conditions and may require fine-tuning to produce optimal results.

Algorithm Description:

The provided code implements the agglomerative clustering method for image segmentation. The code includes the following functions:

1. **euclidean_distance**: This function calculates the Euclidean distance between two points.
2. **clusters_distance**: This function calculates the maximum distance between any two points in two groups of points (or clusters).
3. **initial_clusters**: This function initializes a set of clusters from a given matrix of pixels.

4. **fit**: This function performs the agglomerative clustering to segment the image into k clusters.
5. **predict_cluster**: This function predicts the cluster number of a given point.
6. **predict_center**: This function predicts the center of the cluster that a given point belongs to.
7. **image_preparation**: This function prepares the input image for segmentation.
8. **image_color_segmentation**: This function performs agglomerative clustering to segment the image into k clusters and creates a new image with each pixel assigned to its corresponding cluster center.

- The **fit** function takes in four parameters: an integer k, a matrix pixels containing pixel values, a vector labels to hold the cluster labels for each pixel, a map centers to hold the centers of each cluster, and a map cluster_map to hold the cluster number for each pixel. The function initializes a set of clusters using the initial_clusters function and then repeatedly merges the two closest clusters until there are only k clusters remaining. The function assigns a cluster number to each pixel and computes the center of each cluster.

- The **predict_cluster** function takes in two parameters: a map cluster_map containing the cluster number for each pixel and a vector point containing pixel values. The function returns the cluster number that the given pixel belongs to.

- The **predict_center** function takes in three parameters: a map centers containing the centers of each cluster, a map cluster_map containing the cluster number for each pixel, and a vector point containing pixel values. The function returns the center of the cluster that the given pixel belongs to.

- The **image_preparation** function takes in an input image and returns a one-dimensional array of pixels.

- The **image_color_segmentation** function performs agglomerative clustering to segment the image into k clusters and creates a new image with each pixel assigned to its corresponding cluster center. The function takes in an integer k, a matrix pixels containing pixel values, and a matrix resized_image containing the resized input image.

Overall, the provided code implements the agglomerative clustering method for image segmentation and can be used as a starting point for developing more complex image processing applications.

8) Mean Shift Segmentation:

1. The input color image is first cloned and converted from the BGR color space to the Lab color space.

2. The function then iterates over each pixel in the image using two nested for-loops. For each pixel, it defines a **hypersphere centered** at the pixel with a radius determined by the distance bandwidth parameter.

3. The function then initializes a mean shift point PtCur to represent the center of the hypersphere. The mean shift **point is a struct** that holds the (x, y) coordinates of a pixel in the image and its Lab color values.

    - **Point Structure explanation**: The Pixel struct is a data structure used to hold a pixel's (x, y) position in an image and its color values in the Lab color space, which are represented by the l, a, and b fields.

4. The function then performs mean shift clustering on the hypersphere around PtCur using the **meanShift_setPoint**, **meanShift_PointLab**, **meanShift_ColorDistance**, **meanShift_PointAccum**, **meanShift_PointScale**, and meanShift_copyPoint_F2S functions. These functions are part of the implementation of the mean shift algorithm.

    - Helper functions implementation:

        1. **meanShift_setPoint:** This function takes a pointer to a Pixel struct and sets its x, y, l, a, and b fields to the values passed as arguments: px, py, pl, pa, and pb respectively.

        2. **meanShift_PointLab:** This function takes a pointer to a Pixel struct that represents a pixel's color values in the Lab color space and modifies its l, a, and b fields to be in a normalized range.

        3. **meanShift_ColorDistance:** This function takes two Pixel structs that represent the color values of two pixels in the Lab color space and computes the Euclidean distance between them in the Lab color space.

        4. **meanShift_PointAccum:** This function takes two Pixel structs as input: mainPt and Pt. The function then updates the values of mainPt by adding the corresponding values from Pt to it.

        5. **meanShift_PointScale:** This function takes a pointer to a Pixel struct and a scale factor as input. The function then scales the values of the x, y, l, a, and b fields of the Pixel struct by the scale factor.

5. The mean shift algorithm iteratively shifts the center of the hypersphere towards the mode of the color values of the pixels within the hypersphere until convergence. The convergence criteria are determined by the MS_MEAN_SHIFT_TOL_COLOR and MS_MEAN_SHIFT_TOL_SPATIAL constants.

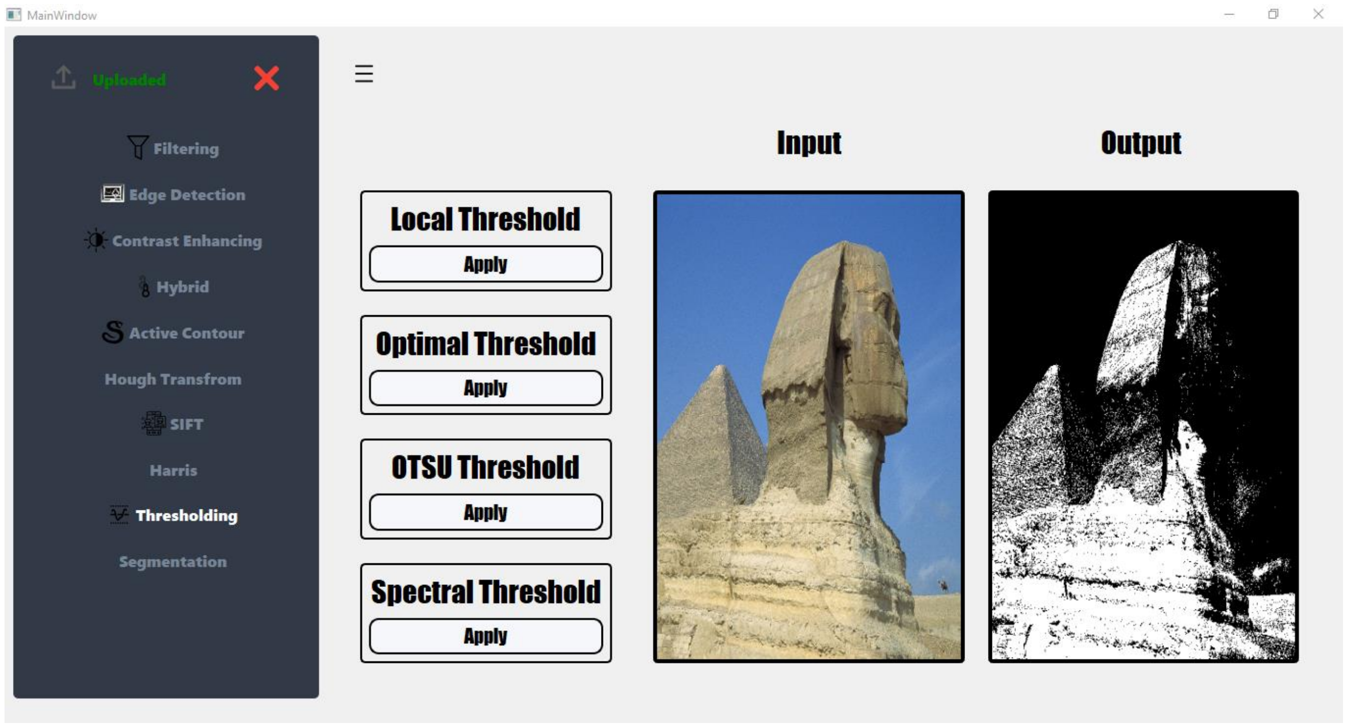
6. After the mean shift clustering is performed on all pixels, the function performs **region growing** to group pixels with similar colors into segments. It starts from an unlabeled pixel and uses a stack to grow the region by recursively adding neighboring pixels that have similar colors. The color similarity is determined by the color bandwidth parameter.

7. For each segment, the function computes the mode of the Lab values for all pixels in that segment and assigns the mode as the Lab value for all pixels in that segment.

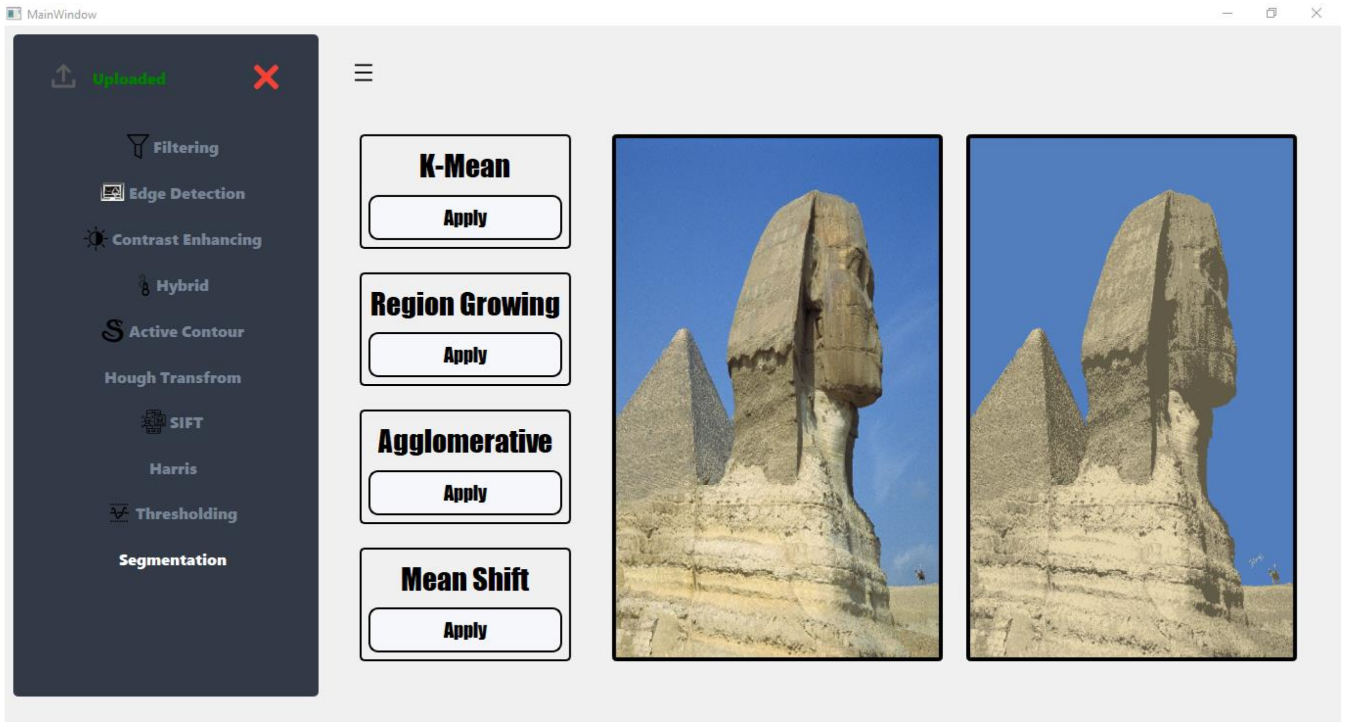
8. Finally, the function converts the output image from the Lab color space back to the BGR color space and returns the resulting segmented image.

Application Screenshots:

1) Thresholding Tab:



2) Segmentation Tab:


MainWindow
Uploaded
Filtering
Edge Detection
Contrast Enhancing
Hybrid
Active Contour
Hough Transfrom
SIFT
Harris
Thresholding
Segmentation
K-Mean
Apply
Region Growing
Apply
Agglomerative
Apply
Mean Shift
Apply