Exp.No:10 **Date :20-03-2025**

IMAGE SEGMENTATION

Aim:

To implement and compare different image segmentation techniques, including Mean-Shift, K-Means Clustering, Thresholding, Graph Cut, and Region Growing.

Algorithm:

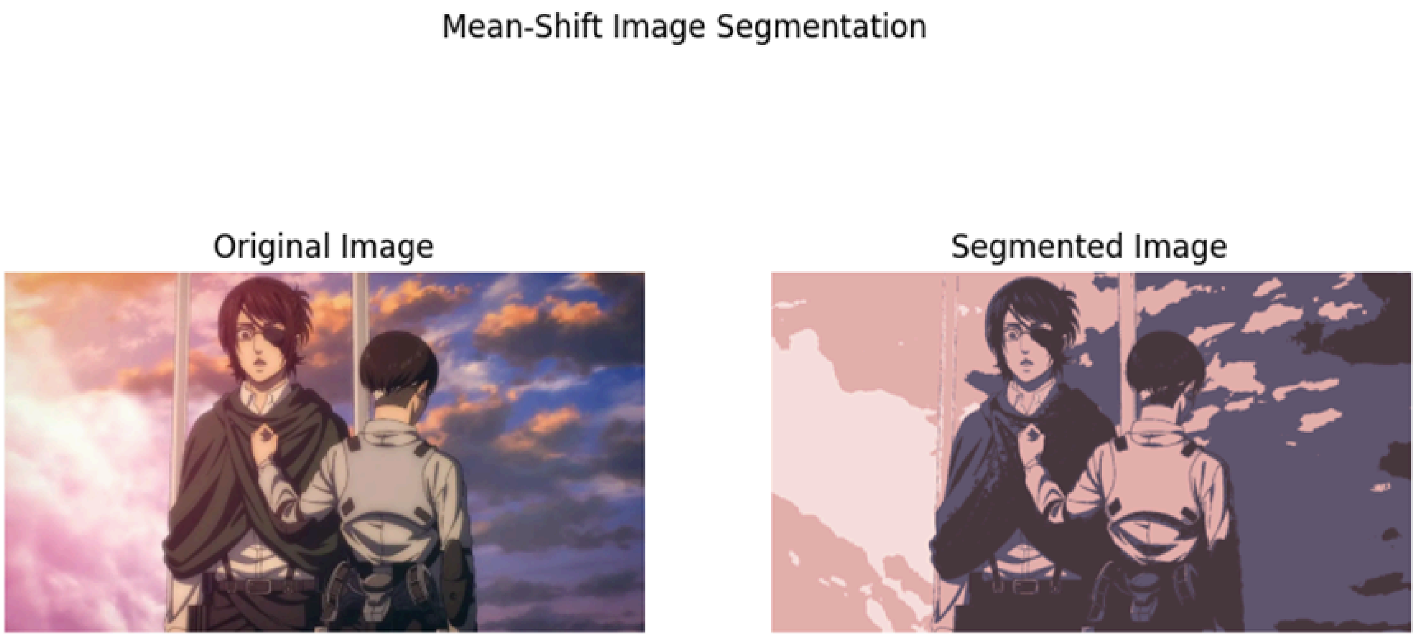
1. Mean Shift Algorithm

1. Load and preprocess the input image.
2. Convert the image to a 2D array of pixel values.
3. Estimate the bandwidth for clustering.
4. Apply the Mean-Shift clustering algorithm.
5. Assign each pixel to the nearest cluster center.
6. Reshape the clustered pixels to reconstruct the segmented image.
7. Display the original and segmented images.

Code Implementation:

```
from sklearn.cluster import MeanShift, estimate_bandwidth
import numpy as np
import cv2
import matplotlib.pyplot as plt

image_path = "aot.jpg"
image = cv2.imread(image_path)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

flat_image = image.reshape((-1, 3))

bandwidth = estimate_bandwidth(flat_image, quantile=0.1, n_samples=500)

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(flat_image)

labels = ms.labels_
cluster_centers = ms.cluster_centers_

segmented_image = cluster_centers[labels].reshape(image.shape).astype(np.uint8)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image)
```

```
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(segmented_image)
plt.title("Segmented Image")
plt.axis("off")
plt.suptitle("Mean-Shift Image Segmentation")
plt.show()
```



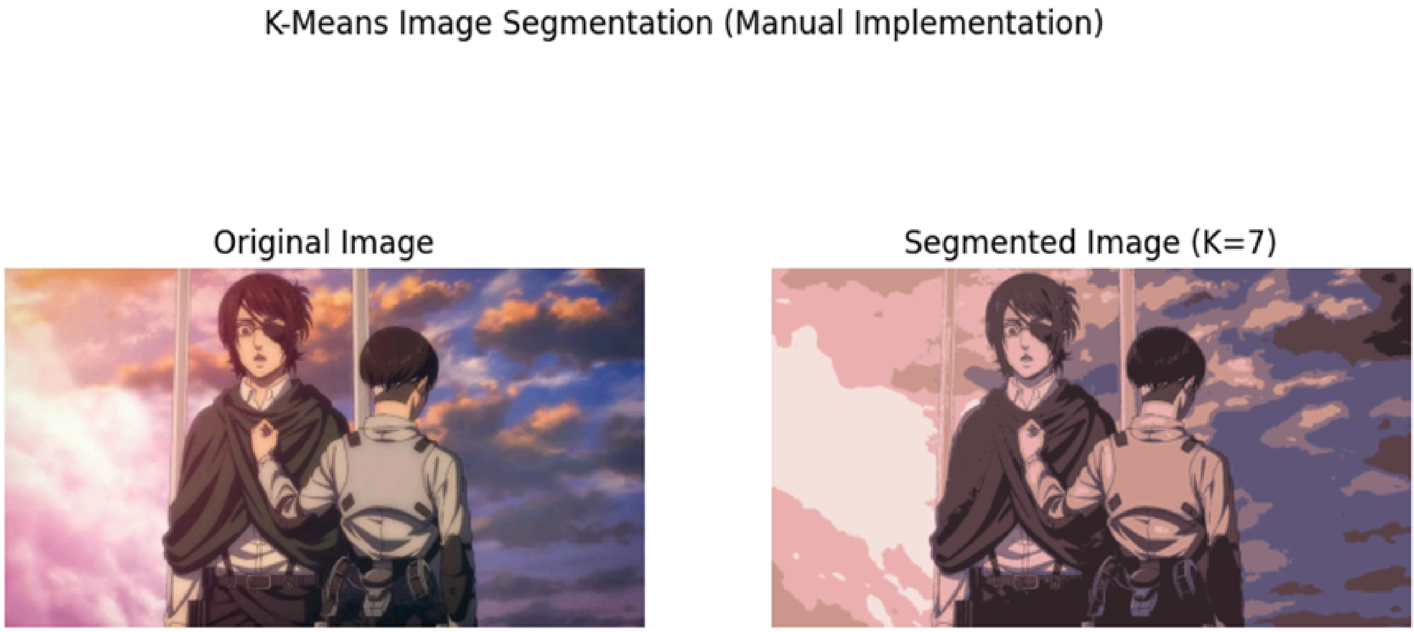
2. K-Means Clustering

1. Convert the image into a 2D pixel array.
2. Initialize cluster centers randomly.
3. Assign each pixel to the nearest cluster.
4. Update cluster centers as the mean of assigned pixels.
5. Repeat the process until convergence.
6. Reshape pixels to reconstruct the segmented image.
7. Display the original and segmented images.

Code Implementation:

```
pixels = image.reshape((-1, 3))
pixels = np.float32(pixels)
K = 7
max_iterations = 100
tolerance = 1e-4
centers = pixels[np.random.choice(pixels.shape[0], K, replace=False)]

def compute_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2, axis=1))
```

```
for iteration in range(max_iterations):
    distances = np.array([compute_distance(pixels, center) for center in centers])
    labels = np.argmin(distances, axis=0)

    new_centers = np.array([pixels[labels == k].mean(axis=0) if np.any(labels == k) else centers[k]
for k in range(K)])

    if np.all(np.linalg.norm(new_centers - centers, axis=1) < tolerance):
        break
    centers = new_centers

segmented_pixels = centers[labels].astype(np.uint8)
segmented_image = segmented_pixels.reshape(image.shape)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title("Original Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(segmented_image)
plt.title(f"Segmented Image (K={K})")
plt.axis("off")

plt.suptitle("K-Means Image Segmentation (Manual Implementation)")
plt.show()
```

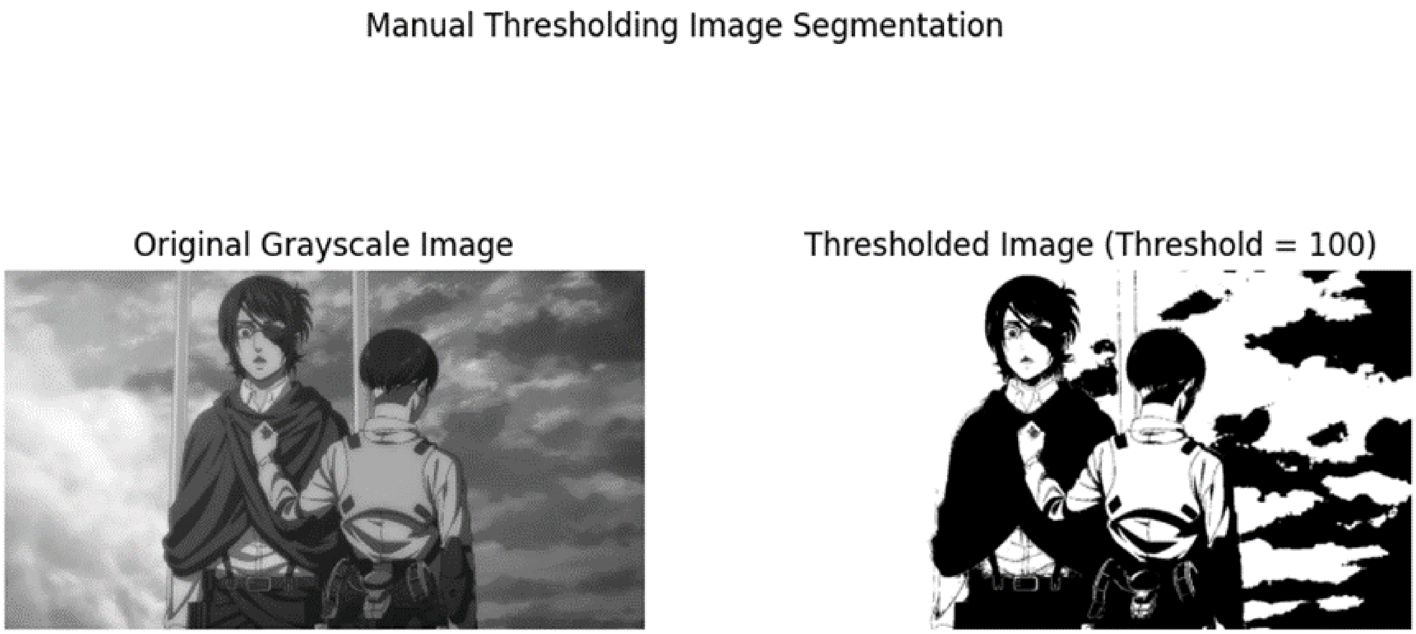


3. Thresholding

1. Convert the image to grayscale.
2. Set a threshold value.
3. Assign pixels to foreground or background based on threshold.
4. Display the original and thresholded images.

Code Implementation:

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

threshold_value = 100

binary_image = np.where(gray > threshold_value, 255, 0).astype(np.uint8)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(gray, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(binary_image, cmap='gray')
plt.title(f"Thresholded Image (Threshold = {threshold_value})")
plt.axis("off")

plt.suptitle("Manual Thresholding for Image Segmentation")
plt.show()
```

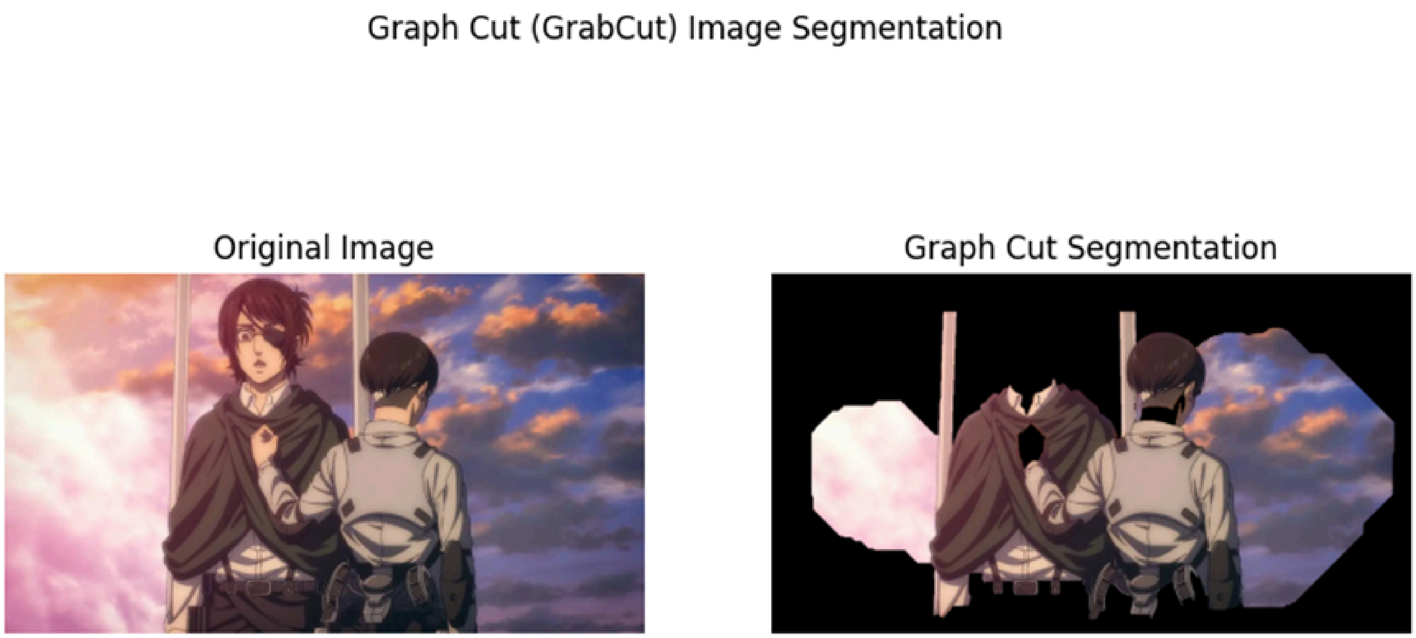


4. Graph Cut Segmentation

1. Define a rectangular region for segmentation.
2. Create background and foreground models.
3. Apply the GrabCut algorithm to classify pixels.
4. Modify and refine the segmentation mask.
5. Display the segmented image.

Code Implementation:

```
mask = np.zeros(image.shape[:2], np.uint8)
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)

rect = (75, 75, image.shape[1] - 100, image.shape[0] - 100)

cv2.grabCut(image, mask, rect, bgd_model, fgd_model, 5, cv2.GC_INIT_WITH_RECT)

mask_2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

segmented_image = image * mask_2[:, :, np.newaxis]

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(segmented_image)
plt.title("Graph Cut Segmentation")
plt.axis("off")
plt.suptitle("Graph Cut (GrabCut) Image Segmentation")
plt.show()
```



5. Region Growing

1. Convert the image to grayscale.
2. Select a seed point for segmentation.
3. Compare neighboring pixels based on intensity difference.
4. Add similar pixels to the segmented region.
5. Display the segmented image.

Code Implementation:

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
seed_point = (100, 100)
threshold = 10

segmented = np.zeros_like(gray, dtype=np.uint8)
to_process = [seed_point]

while to_process:
x, y = to_process.pop(0)

if segmented[y, x] == 0:
  segmented[y, x] = 255

  for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (1, 1), (-1, 1), (1, -1)]:
    nx, ny = x + dx, y + dy

    if 0 <= nx < gray.shape[1] and 0 <= ny < gray.shape[0]:
      if segmented[ny, nx] == 0 and abs(int(gray[ny, nx]) - int(gray[y, x])) < threshold:
        to_process.append((nx, ny))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(gray, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(segmented, cmap='gray')
plt.title("Region Growing Segmentation")
plt.axis("off")

plt.suptitle("Manual Region Growing Image Segmentation")
plt.show()
```

Manual Region Growing Image Segmentation

Inference

1. **Mean-Shift Clustering** performs well on images with distinct color clusters.
2. **K-Means Clustering** is efficient but requires manual selection of cluster numbers.
3. **Thresholding** is simple but sensitive to illumination changes.
4. **Graph Cut Segmentation** effectively separates foreground from the background.
5. **Region Growing** works well for object-based segmentation but depends on the seed point.

Result

Each method has its strengths and is suitable for different segmentation tasks.