CP1

Anastasia Sharangia

Code:

```
import cv2
import numpy as np

# Function for DBSCAN
def manual_dbscan(points, eps, min_samples):
    def distance(p1, p2):
        return np.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

    def region_query(point, points, eps):
        neighbors = []
        for idx, p in enumerate(points):
            if distance(point, p) < eps:
                neighbors.append(idx)
        return neighbors

    def expand_cluster(points, point, neighbors, cluster, eps, min_samples,
visited, clusters):
        cluster.append(point)
        visited[point] = True
        for neighbor in neighbors:
            if not visited[neighbor]:
                visited[neighbor] = True
                neighbor_neighbors = region_query(points[neighbor], points, eps)
                if len(neighbor_neighbors) >= min_samples:
                    neighbors.extend(neighbor_neighbors)
            if neighbor not in cluster:
                cluster.append(neighbor)
        clusters.append(cluster)

    visited = [False] * len(points)
    clusters = []
    noise = []

    for i, point in enumerate(points):
        if not visited[i]:
            visited[i] = True
            neighbors = region_query(point, points, eps)
```

```
            if len(neighbors) < min_samples:
                noise.append(i)
            else:
                cluster = []
                expand_cluster(points, i, neighbors, cluster, eps, min_samples,
visited, clusters)

    return clusters, noise



# Function to calculate Euclidean distance
def euclidean_distance(p1, p2):
    return np.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# Function to apply manual edge detection using Sobel operator
def sobel_edge_detection(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=np.float32)
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=np.float32)

    grad_x = cv2.filter2D(gray, -1, sobel_x)
    grad_y = cv2.filter2D(gray, -1, sobel_y)

    magnitude = np.sqrt(grad_x**2 + grad_y**2)
    magnitude = np.uint8(np.clip(magnitude, 0, 255))

    return magnitude

# Load the video
video_path = './content/cp1.mp4'
cap = cv2.VideoCapture(video_path)

# Background Subtractor to detect moving objects (cars)
fgbg = cv2.createBackgroundSubtractorMOG2(history=600, varThreshold=20,
detectShadows=True)

# Parameters for tracking
min_contour_area = 1100
frame_time = 1 / cap.get(cv2.CAP_PROP_FPS)
car_tracks = {}
car_id_counter = 1
frame_count = 0
```

```
if not cap.isOpened():
    print("Error: Could not open video.")
else:
    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Resize the frame for faster processing
        frame = cv2.resize(frame, (640, 360))

        # Apply background subtraction and pre-process mask
        fgmask = fgbg.apply(frame)
        blurred_mask = cv2.GaussianBlur(fgmask, (5, 5), 0)
        _, thresh_mask = cv2.threshold(blurred_mask, 127, 255, cv2.THRESH_BINARY)

        # Morphological operations to clean the mask
        kernel = np.ones((5, 5), np.uint8)
        clean_mask = cv2.morphologyEx(thresh_mask, cv2.MORPH_CLOSE, kernel)
        clean_mask = cv2.dilate(clean_mask, kernel, iterations=2)

        # Apply manual Sobel edge detection
        edges = sobel_edge_detection(frame)

        # Find contours and filter out small ones
        contours, _ = cv2.findContours(clean_mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        filtered_contours = [cnt for cnt in contours if cv2.contourArea(cnt) >
min_contour_area]

        # Calculate centroids of contours in current frame
        current_centroids = []
        for contour in filtered_contours:
            M = cv2.moments(contour)
            if M["m00"] != 0:
                cx = int(M["m10"] / M["m00"])
                cy = int(M["m01"] / M["m00"])
                current_centroids.append((cx, cy))

        # Track cars using Euclidean distance threshold
        unmatched_centroids = set(current_centroids)
        for car_id, track in car_tracks.items():
```

```
            if track:
                last_position = track[-1]
                closest_centroid = None
                closest_distance = float('inf')

                # Find closest centroid within threshold
                for centroid in current_centroids:
                    distance = euclidean_distance(last_position, centroid)
                    if distance < closest_distance and distance < 200:
                        closest_centroid = centroid
                        closest_distance = distance

                # Update track if match found
                if closest_centroid:
                    track.append(closest_centroid)
                    unmatched_centroids.discard(closest_centroid)

        # Assign unmatched centroids to new tracks (up to 4 cars)
        if True:
            for centroid in unmatched_centroids:
                car_tracks[car_id_counter] = [centroid]
                car_id_counter += 1

        # Draw contours for visual feedback
        for contour in filtered_contours:
            hull = cv2.convexHull(contour)
            cv2.drawContours(frame, [hull], -1, (0, 255, 0), 2)

        # Draw centroids on the frame
        for centroid in current_centroids:
            cv2.circle(frame, centroid, 5, (0, 0, 255), -1)

        # Combine the edge detection with the original frame for one output
        edges_colored = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
        frame_with_edges = cv2.addWeighted(frame, 0.7, edges_colored, 0.3, 0)

        cv2.imshow('Moving Cars with Edges and Centroids', frame_with_edges)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
```

```python
cv2.destroyAllWindows()

# Calculate average speed for each car
car_speeds = {}
for car_id, track in car_tracks.items():
    if len(track) < 120:
        continue

    total_distance = 0.0
    for i in range(1, len(track)):
        total_distance += euclidean_distance(track[i - 1], track[i])

    # Convert distance per frame to pixels per second
    total_time = len(track) * frame_time
    if total_time > 0:
        average_speed = total_distance / total_time
        car_speeds[car_id] = average_speed

# Output the average speed of each car
for car_id, speed in car_speeds.items():
    print(f"Car {car_id}: Average Speed = {speed:.2f} pixels/second")
```

Input:

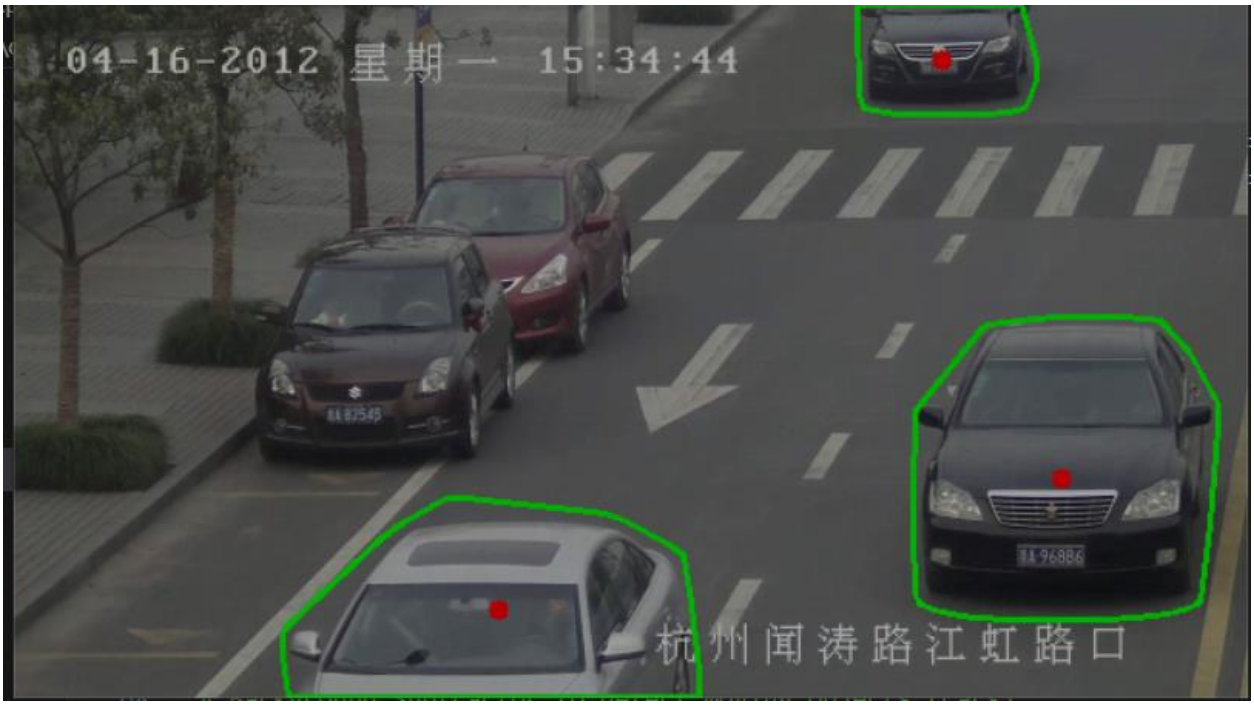
Video of moving cars with traffic camera.

Output:

Speed output

```
Car 8: Average Speed = 134.36 pixels/second
Car 15: Average Speed = 134.21 pixels/second
Car 16: Average Speed = 94.85 pixels/second
```

Video output kinda looks like this



Explanation:

I started by writing an edge detection function that applies the Sobel operator to detect edges in an image. First, it converts the image to grayscale, then defines Sobel filters (`sobel_x` and `sobel_y`) for detecting horizontal and vertical gradients. These filters are applied to the grayscale image using `cv2.filter2D`. The function calculates the gradient magnitude (edge intensity) and returns the result as an 8-bit image.

Next, I wrote a function to calculate the Euclidean distance between two points. This is essential for tracking moving objects (in this case, cars) based on their positions and is also required for clustering.

For clustering, I implemented the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm manually to group points based on their proximity. The implementation includes several helper functions:

- distance(p1, p2): Calculates the Euclidean distance between two points.
- region_query(point, points, eps): Finds all points within a specified distance (`eps`) of the given point.
- expand_cluster(...): Expands a cluster by recursively adding neighbors that meet the density condition (based on `min_samples`).

This implementation returns a list of clusters and a list of noise points, enabling me to identify meaningful groups in the data.

After writing these functions, I moved on to video processing, applying the written functions to video frames using OpenCV. I began by reading the video file and using a background subtraction algorithm (`cv2.createBackgroundSubtractorMOG2`) to isolate moving objects, such as cars. To refine the mask, I applied Gaussian blur, thresholding, and morphological operations (closing and dilation). These steps cleaned the mask and highlighted the moving objects.

I then applied the Sobel edge detection to each frame, which helped emphasize edges for better visualization. After this, I used contour detection to identify objects in the mask. I filtered out small contours (below a certain area threshold) to focus on larger objects like cars. The centroids of these contours were calculated and stored for use in clustering and tracking.

For car tracking, the program matched each car's current position with its previous position using Euclidean distance. If a match was found, the new position was added to the car's track. If no match was found, a new track was created for the car. To manage this efficiently, unmatched centroids were tracked and assigned to new cars when necessary.

Finally, I calculated the average speed for each tracked car. After processing all the frames, the program summed the Euclidean distances between consecutive centroids in each car's track to compute the total distance traveled. The total time was calculated based on the number of frames in the track and the frame time (`frame_time`), derived from the video frame rate. The average speed was then computed as the total distance divided by the total time.

To avoid printing out random or nonsensical clusters caused by issues like camera shake, I added a constraint: tracks that lasted for fewer than 120 frames were ignored.

This process ensures a NOT reliable system for detecting, tracking, clustering, and calculating the speed of cars in a video.

Constraints:

Does not work on moving camera

Does not work on shaky camera

Does not work on a video that has too much objects

Does not work with sudden movements

And has a lot of other problematic constraints, but well i tried :') .