Task : Detect water drop-shaped as defects in image using Python and the OpenCV library.

Overview :

This script detects water drop-shaped defects in an image using edge detection and contour filtering.

Step-by-Step Working of the Script

The script processes an image to detect **water drop-shaped** defects based on edge detection and contour filtering. It then highlights the detected defects and saves the output.

Main Steps:

1. **Load the Image** – Read the image from the given path.
2. **Preprocessing** – Convert the image to grayscale and apply Gaussian blur.
3. **Edge Detection** – Use the Canny edge detector to find potential defect boundaries.
4. **Find Contours** – Extract contour information from the detected edges.
5. **Filter Contours** – Identify contours that resemble water drop shapes based on:
    - Contour closure
    - Size and perimeter
    - Circularity and convexity
    - Overlap with existing detected contours
6. **Draw Defects on a Black Image** – Highlight the detected defects.
7. **Save and Display the Output Image** – Save the final result as `"defect_detection_contour.jpg"`.

Function Breakdown

1. `is_contour_nearly_closed(contour, tolerance=30)`

This function checks whether a given contour is **nearly closed** based on the difference between the first and last points.

Working:

- Retrieves the **first** and **last** points of the contour.

- If the difference in x and y coordinates between these points is within the **tolerance (default = 30 pixels)**, the contour is considered nearly closed.

Why?

- Water drops tend to have a closed or nearly closed boundary, so this filter helps in detecting them.

2. `calculate_iou_rectangles(contour1, contour2)`

Calculates the **Intersection over Union (IoU)** between the bounding boxes of two contours.

Working:

1. **Extracts** the bounding rectangle `(x, y, width, height)` for each contour.

2. **Finds the intersection** region between the two rectangles.

3. **Computes IoU**:

$$\textbf{IoU} = \text{Area of Intersection/Area of Union}$$

4. If no intersection, returns **0**.

Why?

- Ensures that detected defects **do not overlap excessively**, reducing false positives.

3. `detect_water_drops_edge_mask(image_path)`

This is the **main function** that performs the entire defect detection pipeline.

Working:

1. **Read Input Image**
   - Uses `cv2.imread(image_path)` to load the image.
   - If the image is **not found**, prints an error message and exits.
2. **Convert to Grayscale**
   - `cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)` converts the image into a grayscale version for better edge detection.
3. **Create a Black Background Image**
   - `black_image = np.zeros_like(image)` creates an empty black image of the same size to draw detected defects.
4. **Apply Gaussian Blur**
   - `cv2.GaussianBlur(gray, (1,1), 0)` reduces noise while preserving edge information.
5. **Edge Detection**
   - `cv2.Canny(blurred, 170, 250)` detects edges using Canny's algorithm.
6. **Find Contours**
   - `cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)` extracts contours from the edges.
7. **Process Each Contour**
   - **Check if it's nearly closed** using `is_contour_nearly_closed(contour)`.

- **Filter by area & length**:
  - Ignore too small (<100 pixels) or too large (>10,000 pixels) defects.
- **Calculate Shape Metrics**:
  - **Circularity**: Determines how round a contour is.
  - **Convexity**: Measures how well a contour fits inside its convex hull.
- **Check Overlap with Existing Detections**:
  - Uses `calculate_iou_rectangles()` to avoid duplicate detections.

8. **Draw Valid Defects**
   - `cv2.drawContours(black_image, [contour], -1, (0, 255, 0), 2)` highlights detected defects in **green**.
9. **Save Output Image**
   - Saves `"defect_detection_contour.jpg"` using `cv2.imwrite(output_image, black_image)`.
10. **Print Results**

- Displays the number of defects detected and confirms that the output file has been saved.

Results :