Báo cáo Project 2

1. THÔNG TIN SINH VIÊN

Quách Trung Tín - 21130061

2. THỐNG KÊ MỨC ĐỘ HOÀN THÀNH

STT	Các chức năng	Mức độ hoàn thành	Sinh viên thực hiện
1	Combining Spatial Enhancement	100%	Quách Trung Tín
2	Find Contours Function	100%	Quách Trung Tín
3	Dice Recognition	100%	Quách Trung Tín

3. PHÂN TÍCH VÀ MÔ TẢ THUẬT TOÁN

a. Combining Spatial Enhancement

- Mô tả thuật toán:

```
image_path = 'C:\\Users\\Trung Tin\\Tài liệu\\IUH\\Xử lý ảnh\\Labs\\Pictures\\bone.jpg'
bone_a = plt.imread(image_path)
if bone_a is None:
    print("Hình ảnh không đọc được. Vui lòng kiểm tra lại đường dẫn.")
else:
    bone_a = bone_a.astype(np.float32)
    plt.imshow(bone_a, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')
```

bone_a = bone_a.astype(np.float32):
Dòng này chuyển đổi dữ liệu của ảnh (dạng mảng) sang kiểu float32. Việc chuyển đổi này thường được thực hiện để chuẩn bị cho các phép tính trên ảnh như xử lý ảnh, vì các phép tính số thực thường yêu cầu kiểu dữ liệu này.

```
laplacian = cv2.Laplacian(bone_a, cv2.CV_32F, ksize=3)
bone_b = cv2.normalize(laplacian, None, 0, 1, cv2.NORM_MINMAX)

plt.imshow(bone_b, cmap='gray')
plt.title('Laplacian Filter')
plt.axis('off')
```

laplacian = cv2.Laplacian(bone_a, cv2.CV_32F, ksize=3)

- **cv2.Laplacian()**: Đây là hàm trong OpenCV dùng để áp dụng toán tử Laplacian lên hình ảnh. Toán tử Laplacian là một phép toán đạo hàm bậc hai, giúp phát hiện các cạnh trong ảnh (edge detection) bằng cách tính toán sự thay đổi cường độ sáng của ảnh.
- **bone_a**: Đây là ảnh nguồn (đã được chuyển đổi thành float32 trong đoạn code trước đó). Hình ảnh sẽ được sử dụng để tính Laplacian.
- **cv2.CV_32F**: Tham số này chỉ định kiểu dữ liệu của ảnh đầu ra, trong trường hợp này là kiểu số thực 32-bit (float32), tương tự như kiểu đã chuyển đổi cho ảnh gốc bone_a.
- **ksize=3**: Đây là kích thước của kernel được sử dụng trong phép tính Laplacian. Kernel là một ma trận nhỏ mà ta "quét" qua từng điểm ảnh để tính toán. Kích thước kernel là 3x3, thường được sử dụng trong các bộ lọc phát hiện cạnh để làm mịn và giảm nhiễu.

Hàm này sẽ tính toán ảnh đạo hàm bậc hai của hình ảnh bone_a, lưu trữ kết quả vào biến laplacian, thể hiện các cạnh của ảnh.

bone_b = cv2.normalize(laplacian, None, 0, 1, cv2.NORM_MINMAX)

- **cv2.normalize()**: Hàm này dùng để chuẩn hóa giá trị của một hình ảnh về một phạm vi cụ thể.
- **laplacian**: Đây là hình ảnh đã được tính Laplacian từ bước trước.
- **None**: Đối số này được để trống vì kết quả sẽ được trả về trực tiếp trong biến bone_b, không cần truyền thêm mảng đầu ra.
- **0, 1**: Đây là khoảng giá trị mà ảnh sẽ được chuẩn hóa. Ở đây, giá trị của các điểm ảnh sẽ được chuẩn hóa về khoảng từ 0 đến 1.
- **cv2.NORM_MINMAX**: Đây là phương pháp chuẩn hóa NORM_MINMAX, nghĩa là các giá trị trong mảng sẽ được điều chỉnh sao cho giá trị nhỏ nhất sẽ là 0 và lớn nhất sẽ là 1, giúp cải thiện việc hiển thị và xử lý ảnh.

Ảnh Laplacian sẽ được chuẩn hóa để đảm bảo các giá trị nằm trong khoảng từ 0 đến 1, lưu vào biến bone_b.

```
bone_c = cv2.subtract(bone_a, bone_b)

plt.imshow(bone_c, cmap='gray')
plt.title('Sharpened a b')
plt.axis('off')
```

Dòng code này thực hiện phép trừ giữa hai hình ảnh bone_a và bone_b bằng hàm cv2.subtract(). Đây là phép toán pixel-by-pixel, trong đó giá trị của mỗi điểm ảnh trong ảnh thứ hai sẽ bị trừ ra khỏi giá trị của điểm ảnh tương ứng trong ảnh thứ nhất.

```
gx = cv2.Sobel(bone_c, cv2.CV_64F, 1, 0, ksize=3)
gy = cv2.Sobel(bone_c, cv2.CV_64F, 0, 1, ksize=3)
bone_d = cv2.magnitude(gx, gy)

plt.imshow(bone_d, cmap='gray')
plt.title('Sobel')
plt.axis('off')
```

gx = cv2.Sobel(bone_c, cv2.CV_64F, 1, 0, ksize=3)

- **cv2.Sobel()**: Hàm Sobel được sử dụng để tính toán đạo hàm bậc nhất của ảnh theo một hướng cụ thể (hướng x hoặc y). Sobel là một bộ lọc giúp phát hiện cạnh bằng cách tính đạo hàm của ảnh.
- **bone_c**: Đây là ảnh đã qua xử lý sau khi trừ ảnh gốc với ảnh Laplacian (bone_c).
- **cv2.CV_64F**: Tham số này chỉ định kiểu dữ liệu đầu ra của hình ảnh là số thực 64-bit (float64). Điều này là cần thiết để đảm bảo độ chính xác trong các phép tính toán học, vì gradient có thể có giá trị âm hoặc lớn hơn.
- **1, 0**: Hai tham số này chỉ định rằng chúng ta muốn tính đạo hàm bậc nhất theo trục x. Cụ thể:
    - **1**: Tính đạo hàm theo hướng x.
    - **0**: Không tính đạo hàm theo hướng y.
- **ksize=3**: Kích thước kernel là 3x3, thường được dùng để làm mịn và giảm nhiễu trong quá trình tính toán.

Biến **gx** sẽ lưu trữ gradient của ảnh theo hướng x, tức là sự thay đổi cường độ ánh sáng dọc theo trục x.

gy = cv2.Sobel(bone_c, cv2.CV_64F, 0, 1, ksize=3)

- Tương tự như dòng trước, nhưng ở đây, các tham số **0, 1** chỉ định rằng chúng ta muốn tính đạo hàm theo trục y thay vì trục x:
    - **0**: Không tính đạo hàm theo hướng x.
    - **1**: Tính đạo hàm theo hướng y.

Biến **gy** sẽ lưu trữ gradient của ảnh theo hướng y, tức là sự thay đổi cường độ ánh sáng dọc theo trục y.

bone_d = cv2.magnitude(gx, gy)

cv2.magnitude(): Hàm này tính độ lớn của gradient (magnitude) tại mỗi điểm ảnh từ hai thành phần gradient theo trục x (gx) và trục y (gy). Công thức được sử dụng là: magnitude=gx2+gy2\text{magnitude} = \sqrt{gx^2 + gy^2}magnitude=gx2+gy2 Độ lớn của gradient giúp xác định cường độ thay đổi tại mỗi điểm ảnh, cho ta biết mức độ sắc nét hoặc biên độ của cạnh.

Biến **bone_d** sẽ lưu trữ cường độ gradient tại mỗi pixel, tức là một ảnh mới thể hiện mức độ thay đổi về cường độ ánh sáng của từng điểm ảnh. Hình ảnh này sẽ làm nổi bật các cạnh và đường viền trong ảnh bone_c.

```
bone_e = cv2.blur(bone_d, (5, 5)).astype(np.float32)

plt.imshow(bone_e, cmap='gray')
plt.title('Smoothed 5 5')
plt.axis('off')
```

bone_e = cv2.blur(bone_d, (5, 5))

- **cv2.blur()**: Đây là hàm trong OpenCV dùng để làm mờ hình ảnh bằng cách áp dụng bộ lọc trung bình (average filter). Bộ lọc này hoạt động bằng cách tính giá trị trung bình của các điểm ảnh trong một vùng nhất định (kernel) và thay thế giá trị điểm ảnh trung tâm bằng giá trị trung bình đó.
- **bone_d**: Đây là ảnh nguồn, kết quả của bước tính cường độ gradient trước đó (ảnh chứa các cạnh và đường viền được phát hiện).
- **(5, 5)**: Đây là kích thước của kernel (5x5). Kernel này xác định kích thước của vùng mà hàm blur sẽ tính trung bình. Kích thước lớn hơn sẽ làm cho ảnh mờ hơn.

Hình ảnh đầu ra sau phép làm mờ sẽ là **bone_e**, trong đó các chi tiết sắc nét hoặc nhiễu nhỏ sẽ bị làm mờ để tạo ra hiệu ứng mềm mại hơn.

.astype(np.float32)
Sau khi làm mờ, kết quả của hàm cv2.blur() có thể là kiểu dữ liệu mặc định của ảnh (có thể là float64 hoặc uint8). Dòng này chuyển kết quả về kiểu dữ liệu float32 để đảm bảo tính tương thích với các bước xử lý ảnh trước đó và tránh lỗi trong các bước xử lý ảnh sau này.

Ảnh **bone_e** sẽ là phiên bản làm mờ của **bone_d**, với các giá trị pixel được lưu dưới dạng số thực 32-bit (float32).

```
bone_f = cv2.multiply(bone_c, bone_e)

plt.imshow(bone_f, cmap='gray')
plt.title('Product c e')
plt.axis('off')
```

Dòng code này thực hiện phép nhân từng pixel giữa hai hình ảnh bone_c và bone_e bằng hàm cv2.multiply(). Đây là phép toán pixel-by-pixel, trong đó giá trị của mỗi điểm ảnh trong ảnh bone_c sẽ được nhân với giá trị tương ứng trong ảnh bone_e.

```
bone_g = cv2.addWeighted(bone_a, 1, bone_f, 0, 0)

plt.imshow(bone_g, cmap='gray')
plt.title('Sharpened a f')
plt.axis('off')
```

Dòng code này thực hiện phép cộng tuyến tính có trọng số giữa hai ảnh bone_a và bone_f bằng hàm cv2.addWeighted(). Phép cộng tuyến tính này giúp kết hợp thông tin từ hai hình ảnh theo các hệ số trọng số (weights) được chỉ định.

Cụ thể:

bone_g = cv2.addWeighted(bone_a, 1, bone_f, 0, 0)

- **cv2.addWeighted()**: Hàm này thực hiện phép cộng tuyến tính giữa hai ảnh với các trọng số cụ thể. Công thức tổng quát của hàm này là: output=α×src1+β×src2+γ\text{output} = \alpha \times \text{src1} + \beta \times \text{src2} + \gammaoutput=α×src1+β×src2+γ Trong đó:
  - src1: Ảnh thứ nhất.
  - src2: Ảnh thứ hai.
  - \alpha: Trọng số của ảnh thứ nhất.
  - \beta: Trọng số của ảnh thứ hai.
  - \gamma: Hằng số được cộng vào toàn bộ ảnh (thường là 0).

bone_a, 1:

- bone_a: Ảnh gốc ban đầu (sau khi được chuẩn hóa thành kiểu float32 ở bước trước).
- 1: Đây là trọng số của ảnh bone_a, nghĩa là ảnh này sẽ được nhân với 1, giữ nguyên giá trị của các pixel trong bone_a.

bone_f, 0:

- bone_f: Ảnh đã qua nhiều bước xử lý, bao gồm cả Laplacian, Sobel, và phép nhân với ảnh làm mờ.
- 0: Trọng số của ảnh bone_f là 0, nghĩa là ảnh bone_f sẽ không đóng góp bất kỳ thông tin nào vào kết quả của phép cộng.

0 (gamma):

Đây là một hằng số được thêm vào toàn bộ kết quả sau khi cộng hai ảnh. Trong trường hợp này, giá trị là 0 nên sẽ không có gì thay đổi.

bone_g: Kết quả của phép cộng tuyến tính. Do ảnh bone_f được nhân với trọng số 0, kết quả **bone_g** sẽ chỉ là bản sao của ảnh bone_a (ảnh gốc), vì chỉ có bone_a được nhân với trọng số 1.

```
bone_h = (np.power(bone_g.astype(np.float64) / 255.0, 0.6) * 255).astype(np.uint64)

plt.imshow(bone_h, cmap='gray')
plt.title('Power-law trans')
plt.axis('off')
```

bone_g.astype(np.float64)

- **astype(np.float64)**: Chuyển đổi dữ liệu của ảnh bone_g từ kiểu dữ liệu hiện tại (có thể là float32 hoặc uint8) sang kiểu số thực 64-bit (float64). Điều này đảm bảo độ chính xác cao hơn khi thực hiện các phép toán tiếp theo.

/ 255.0

- Chia tất cả các giá trị pixel của ảnh bone_g cho 255.0 để chuẩn hóa chúng về khoảng giá trị [0, 1]. Điều này là cần thiết khi thực hiện các phép toán gamma correction, vì giá trị cường độ pixel của ảnh thường nằm trong khoảng 0 đến 255 (với kiểu dữ liệu uint8).

np.power(..., 0.6)

- **np.power()**: Hàm tính lũy thừa của các phần tử trong mảng. Trong trường hợp này, mỗi giá trị pixel (sau khi đã được chuẩn hóa về [0, 1]) sẽ được nâng lên lũy thừa **0.6**.
- **Gamma Correction**: Đây là một dạng của phép biến đổi gamma. Gamma correction thường được sử dụng để điều chỉnh độ sáng hoặc độ tương phản của hình ảnh.
    - Nếu **gamma < 1** (như 0.6 trong trường hợp này), ảnh sẽ trở nên sáng hơn vì các giá trị nhỏ sẽ được tăng cường.
    - Nếu **gamma > 1**, ảnh sẽ tối hơn vì các giá trị lớn sẽ bị nén xuống.
- Trong trường hợp này, lũy thừa **0.6** sẽ làm cho ảnh sáng hơn bằng cách tăng cường các giá trị pixel nhỏ.

*** 255**

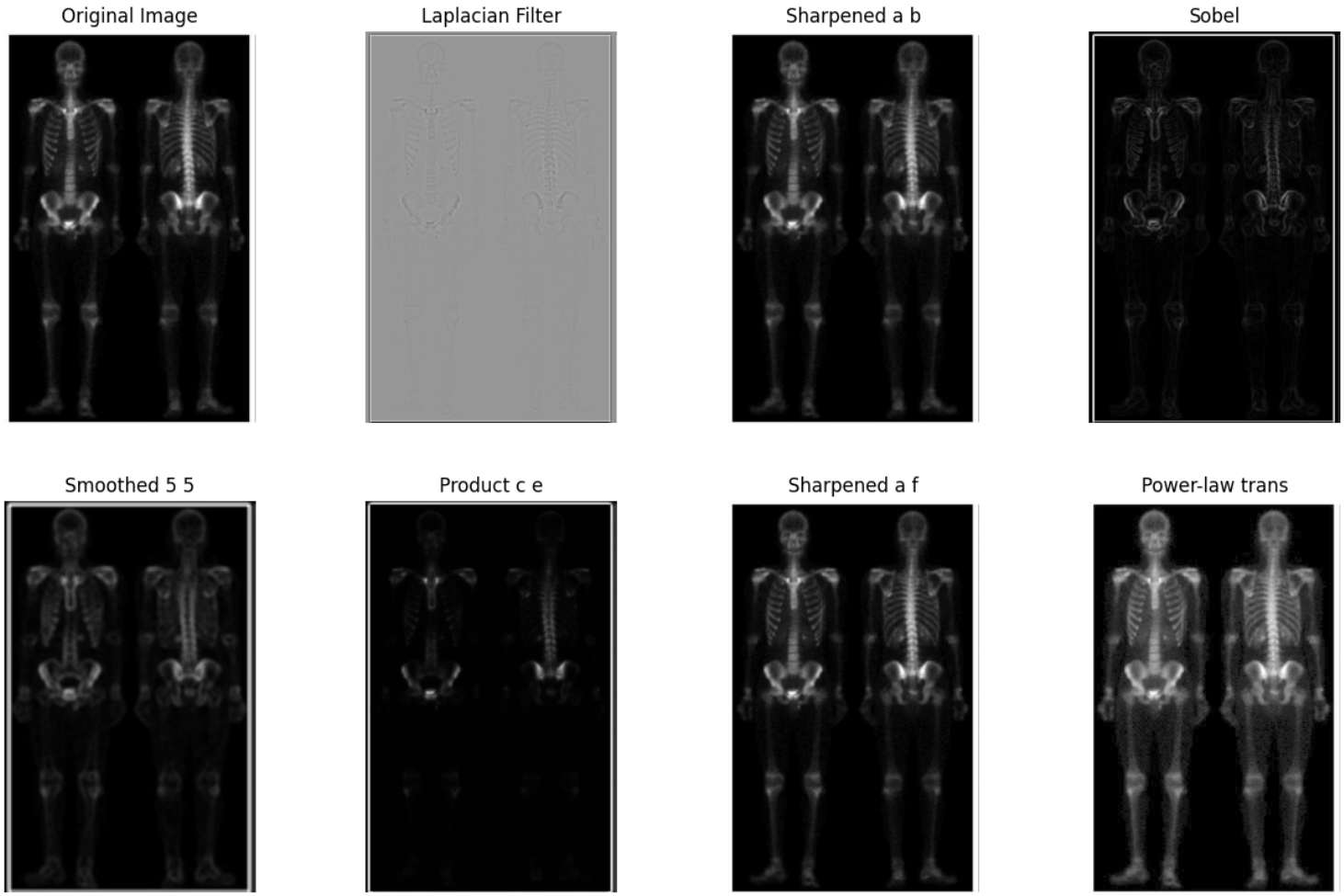
Sau khi thực hiện phép biến đổi gamma, giá trị pixel của ảnh vẫn nằm trong khoảng [0, 1]. Để đưa ảnh trở lại khoảng giá trị chuẩn của một ảnh 8-bit, tất cả các giá trị pixel sẽ được nhân với 255.

astype(np.uint64)

Cuối cùng, ảnh kết quả được chuyển về kiểu dữ liệu số nguyên 64-bit không dấu (uint64). Điều này có thể được thực hiện để tránh mất mát dữ liệu khi thực hiện các phép toán tiếp theo. Tuy nhiên, đối với ảnh thông thường, kiểu dữ liệu uint8 hoặc uint16 thường đủ. Việc sử dụng uint64 có thể là để đảm bảo rằng các phép tính không gây ra lỗi tràn số (overflow).

bone_h: Ảnh kết quả sau khi áp dụng gamma correction (với gamma = 0.6). Hình ảnh sẽ sáng hơn so với ảnh gốc bone_g, đặc biệt là ở các vùng tối của ảnh.

- Kết quả:



b.Find Contours Function

- Mô tả thuật toán:

```
image_path = 'C:\\Users\\Trung Tin\\Tài liệu\\IUH\\Xử lý ảnh\\Labs\\Pictures\\Dino.jpg'
img = plt.imread(image_path)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (5, 5), 0)

#Thresholding
_, thresholded = cv2.threshold(blur, 100, 255, cv2.THRESH_BINARY)

#RETR_EXTERNAL
contours_ext, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
external_contours = cv2.drawContours(np.zeros_like(img), contours_ext, -1, (0, 255, 0), 2)

#RETR_LIST
contours_list, _ = cv2.findContours(thresholded, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
list_contours = cv2.drawContours(np.zeros_like(img), contours_list, -1, (0, 255, 0), 2)
```

blur = cv2.GaussianBlur(gray, (5, 5), 0)

- **cv2.GaussianBlur()**: Áp dụng bộ lọc Gaussian để làm mờ ảnh. Bộ lọc này giúp giảm nhiễu và làm mịn ảnh trước khi thực hiện ngưỡng (thresholding).
- **(5, 5)**: Kích thước kernel (5x5), nghĩa là vùng xung quanh mỗi pixel mà bộ lọc Gaussian sẽ xem xét.
- **0:** Độ lệch chuẩn của bộ lọc Gaussian. Nếu là 0, OpenCV sẽ tự động tính toán dựa trên kích thước kernel.

Kết quả: Ảnh blur đã được làm mờ, giúp giảm nhiễu và các chi tiết không cần thiết.

_, thresholded = cv2.threshold(blur, 100, 255, cv2.THRESH_BINARY)

- **cv2.threshold()**: Áp dụng phép biến đổi ngưỡng nhị phân lên ảnh.
- **blur**: Ảnh đầu vào đã làm mờ.
- **100**: Giá trị ngưỡng (threshold). Các pixel có giá trị xám nhỏ hơn 100 sẽ được gán giá trị 0 (màu đen), và các pixel có giá trị lớn hơn hoặc bằng 100 sẽ được gán giá trị 255 (màu trắng).
- **255**: Giá trị đầu ra tối đa, tức là giá trị cho các pixel vượt qua ngưỡng (trắng).
- **cv2.THRESH_BINARY**: Loại ngưỡng, biến đổi ảnh thành nhị phân (đen hoặc trắng).

Kết quả: Ảnh thresholded là ảnh nhị phân với các đối tượng được làm nổi bật.

RETR_EXTERNAL

contours_ext, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
external_contours = cv2.drawContours(np.zeros_like(img), contours_ext, -1, (0, 255, 0), 2)

- **cv2.findContours()**: Tìm các đường viền trong ảnh nhị phân.
  - **thresholded**: Ảnh nhị phân đầu vào.
  - **cv2.RETR_EXTERNAL**: Chế độ tìm kiếm chỉ lấy các đường viền ngoài cùng (external contours), bỏ qua các đường viền bên trong các đối tượng.
  - **cv2.CHAIN_APPROX_SIMPLE**: Phương pháp nén các điểm trên đường viền, giúp giảm số lượng điểm không cần thiết (chỉ lưu các điểm quan trọng).
- **contours_ext**: Danh sách các đường viền tìm thấy (chỉ các đường viền ngoài).
- **cv2.drawContours()**: Vẽ các đường viền tìm thấy lên một ảnh mới.
  - **np.zeros_like(img)**: Tạo một ảnh đen có cùng kích thước với ảnh gốc.
  - **contours_ext**: Các đường viền cần vẽ.
  - **-1**: Vẽ tất cả các đường viền.
  - **(0, 255, 0)**: Màu xanh lá cây cho đường viền.
  - **2**: Độ dày của đường viền.

Kết quả: Ảnh external_contours chứa các đường viền ngoài cùng của các đối tượng trong ảnh.

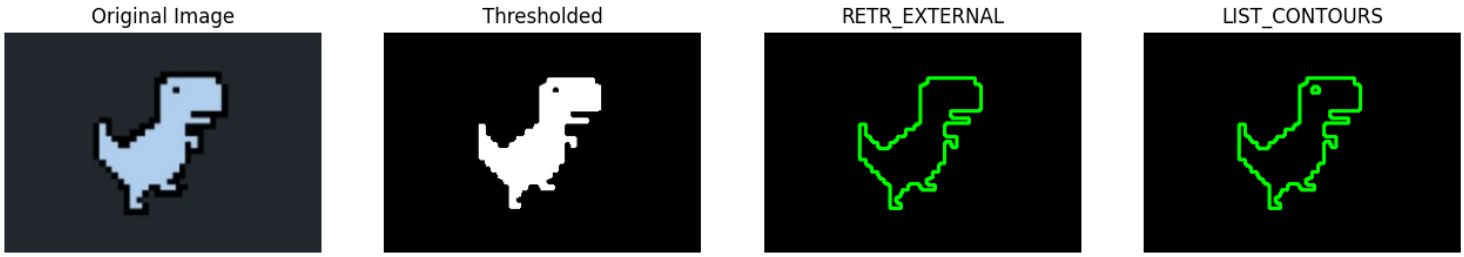
RETR_LIST
contours_list, _ = cv2.findContours(thresholded, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
list_contours = cv2.drawContours(np.zeros_like(img), contours_list, -1, (0, 255, 0), 2)

- **cv2.RETR_LIST**: Chế độ tìm kiếm tất cả các đường viền, bao gồm cả đường viền bên trong và bên ngoài, nhưng không duy trì thứ bậc phân cấp giữa các đường viền (không phân biệt đường viền ngoài và đường viền trong).
- **contours_list**: Danh sách tất cả các đường viền tìm thấy (bao gồm cả bên trong và bên ngoài).
- **list_contours**: Ảnh chứa tất cả các đường viền (cả ngoài và trong) được vẽ lên.

Tổng kết:

- **external_contours**: Ảnh chứa chỉ các đường viền ngoài của các đối tượng.
- **list_contours**: Ảnh chứa tất cả các đường viền (cả ngoài và trong).

- Kết quả:



c.Dice Recognition

- Mô tả thuật toán:

```
image_path = 'C:\\Users\\Trung Tin\\Tài liệu\\IUH\\Xử lý ảnh\\Labs\\Pictures\\1.jpg'

img = plt.imread(image_path)

rgb_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(rgb_img)
plt.axis('off')
```

Dòng code này đọc một hình ảnh từ đường dẫn cụ thể và sau đó chuyển đổi nó từ không gian màu BGR (Blue-Green-Red) sang RGB (Red-Green-Blue) để hiển thị bằng thư viện matplotlib.

```
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
plt.imshow(gray_img, cmap='gray')
plt.axis('off')
```

Dòng code này thực hiện việc chuyển đổi ảnh màu sang ảnh xám và hiển thị nó với matplotlib.

```
detected_edges = cv2.Canny(gray_img,9, 150, 3)

kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (9,9))

close = cv2.morphologyEx(detected_edges, cv2.MORPH_CLOSE, kernel, iterations=2)

circles = cv2.HoughCircles(close,cv2.HOUGH_GRADIENT,1.1,20,param1=50,param2=30,minRadius=5,maxRadius=55)
print(circles)

plt.imshow(close, cmap='gray')
plt.axis('off')
```

1. **Phát hiện các cạnh với Canny**

cv2.Canny(): Hàm này dùng để phát hiện các cạnh trong ảnh bằng thuật toán Canny.

- **gray_img**: Ảnh xám đầu vào (đã được chuyển đổi từ ảnh màu).
- **9**: Giá trị ngưỡng dưới (lower threshold).
- **150**: Giá trị ngưỡng trên (upper threshold).
- **3**: Kích thước của bộ lọc Sobel (dùng để tính gradient trong ảnh).

Kết quả: **detected_edges** là ảnh chứa các cạnh được phát hiện trong ảnh xám. Canny Edge Detection giúp phát hiện các cạnh mạnh trong ảnh.

2. **Áp dụng phép toán hình học (Morphological Operation) - Close**

cv2.getStructuringElement(): Tạo một kernel (hình dạng cấu trúc) để thực hiện phép toán hình học.

- **cv2.MORPH_RECT**: Tạo kernel hình chữ nhật.
- **(9, 9)**: Kích thước của kernel (9x9).

cv2.morphologyEx(): Thực hiện phép toán hình học với ảnh, cụ thể là phép toán đóng (MORPH_CLOSE).

- **detected_edges**: Ảnh chứa các cạnh phát hiện được.
- **cv2.MORPH_CLOSE**: Phép toán "đóng", giúp loại bỏ các điểm đen nhỏ và nối các đoạn rời rạc của cạnh lại với nhau.
- **iterations=2**: Lặp lại phép toán này 2 lần để làm mịn các đường viền.

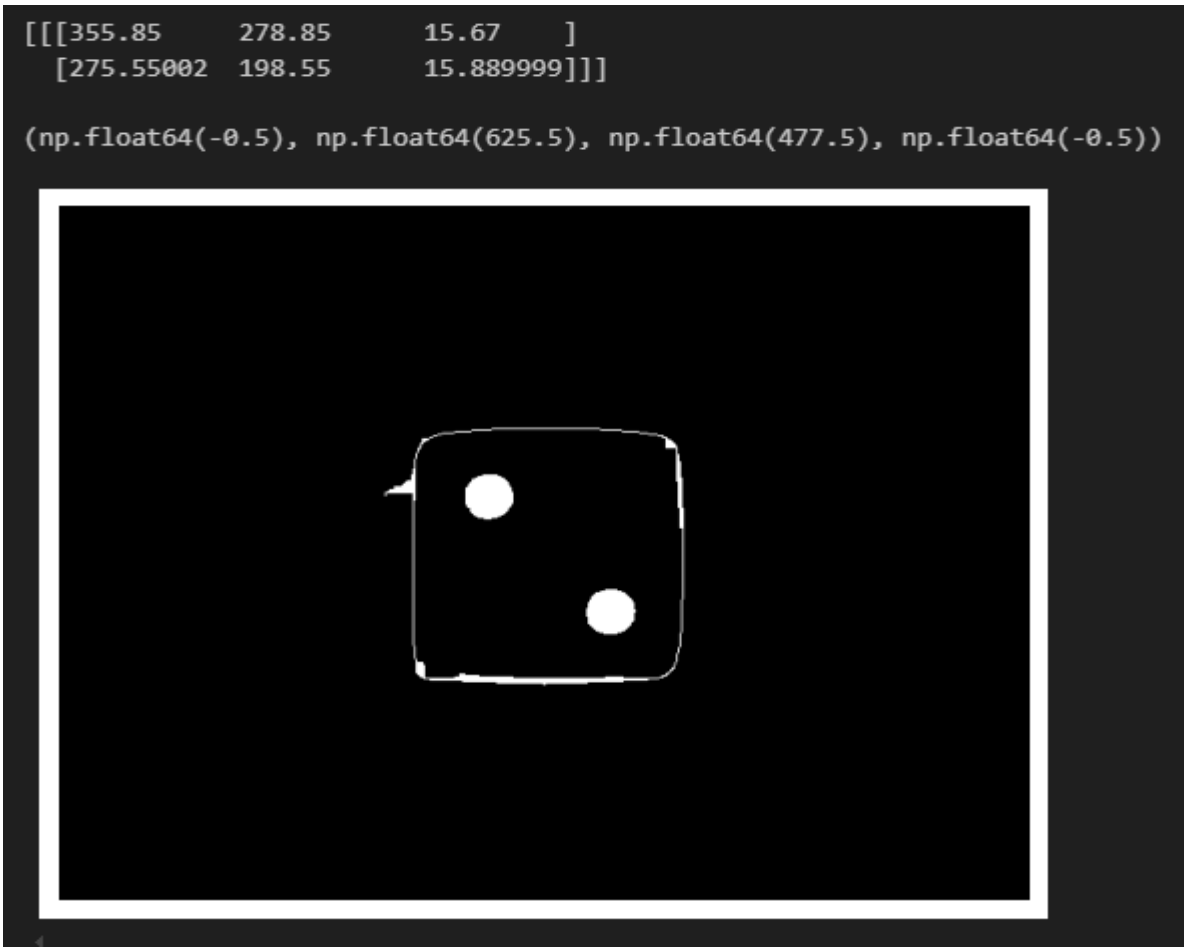
Kết quả: **close** là ảnh đã được xử lý với phép toán đóng, giúp làm mượt các cạnh và nối liền các đoạn rời rạc.

3. Phát hiện hình tròn với Hough Transform

cv2.HoughCircles(): Hàm này sử dụng phương pháp Hough Transform để phát hiện hình tròn trong ảnh.

- **close:** Ảnh đầu vào đã qua phép toán hình học.
- **cv2.HOUGH_GRADIENT:** Phương pháp Hough Transform sử dụng gradient (độ dốc).
- **1.1:** Tỷ lệ ảnh để giảm độ phân giải (càng nhỏ càng chi tiết, nhưng tốn nhiều tài nguyên).
- **20:** Khoảng cách tối thiểu giữa các tâm tròn được phát hiện.
- **param1=50:** Ngưỡng cho bộ lọc Canny được sử dụng trong quá trình phát hiện tròn (ngưỡng cao).
- **param2=30:** Ngưỡng cho quá trình phát hiện các vòng tròn (ngưỡng thấp).
- **minRadius=5:** Bán kính tối thiểu của các hình tròn cần phát hiện.
- **maxRadius=55:** Bán kính tối đa của các hình tròn cần phát hiện.

Kết quả: **circles** chứa thông tin về các vòng tròn đã phát hiện (nếu có). Dữ liệu này sẽ bao gồm tọa độ tâm và bán kính của từng hình tròn.



```
circles=circles[0,:]
print(circles)
```

Dòng mã **circles = circles[0, :]** và **print(circles)** được sử dụng để xử lý kết quả từ hàm **cv2.HoughCircles()** và in ra các vòng tròn đã phát hiện.

Giải thích về **circles** sau khi sử dụng **cv2.HoughCircles()**

Kết quả của **cv2.HoughCircles()** là một mảng 3D có dạng **(1, N, 3)**:

- 1: Chỉ có một lớp (layer), vì hàm Hough Transform trả về một mảng với 1 lớp chứa các vòng tròn được phát hiện.
- N: Số lượng vòng tròn được phát hiện.
- 3: Mỗi vòng tròn được mô tả bởi ba giá trị:
    - Tọa độ x của tâm vòng tròn (float)
    - Tọa độ y của tâm vòng tròn (float)
    - Bán kính của vòng tròn (float)

Lý do dùng circles[0, :]

- **circles[0, :]**: Chỉ lấy lớp đầu tiên (vì có một lớp duy nhất) và tất cả các vòng tròn (tức là **:** lấy toàn bộ các phần tử theo chiều thứ hai). Điều này chuyển mảng từ dạng **(1, N, 3)** thành dạng **(N, 3)**, nơi mỗi hàng trong mảng là thông tin của một vòng tròn.

Sau khi thực hiện bước này, mảng **circles** sẽ có dạng **(N, 3)**, mỗi vòng tròn được mô tả bởi ba giá trị: tọa độ x, tọa độ y và bán kính.

```
[[355.85     278.85      15.67    ]
 [275.55002  198.55      15.889999]]
```

```
for i in circles:
    # draw the outer circle
    cv2.circle(rgb_img,(int(i[0]),int(i[1])),int(i[2]),(0,255,0),2)
    # draw the center of the circle
    cv2.circle(rgb_img,(int(i[0]),int(i[1])),2,(0,0,255),3)

print(len(circles))
plt.imshow(rgb_img)
plt.axis('off')
```

1. Vẽ vòng tròn ngoại vi

cv2.circle(rgb_img, (int(i[0]), int(i[1])), int(i[2]), (0, 255, 0), 2)

- **rgb_img:** Ảnh mà bạn muốn vẽ lên.
- **(int(i[0]), int(i[1])):** Tọa độ tâm của vòng tròn được lấy từ mảng **circles**. **i[0]** là tọa độ **x** và **i[1]** là tọa độ **y** của tâm vòng tròn. Hàm **int()** dùng để chuyển đổi chúng thành số nguyên, vì tọa độ phải là số nguyên.
- **int(i[2]):** Bán kính của vòng tròn, cũng được lấy từ mảng **circles**. **i[2]** là bán kính của vòng tròn, và chúng ta chuyển đổi nó thành số nguyên.

- **(0, 255, 0)**: Màu của vòng tròn (xanh lá cây).
- **2**: Độ dày của đường viền vòng tròn.

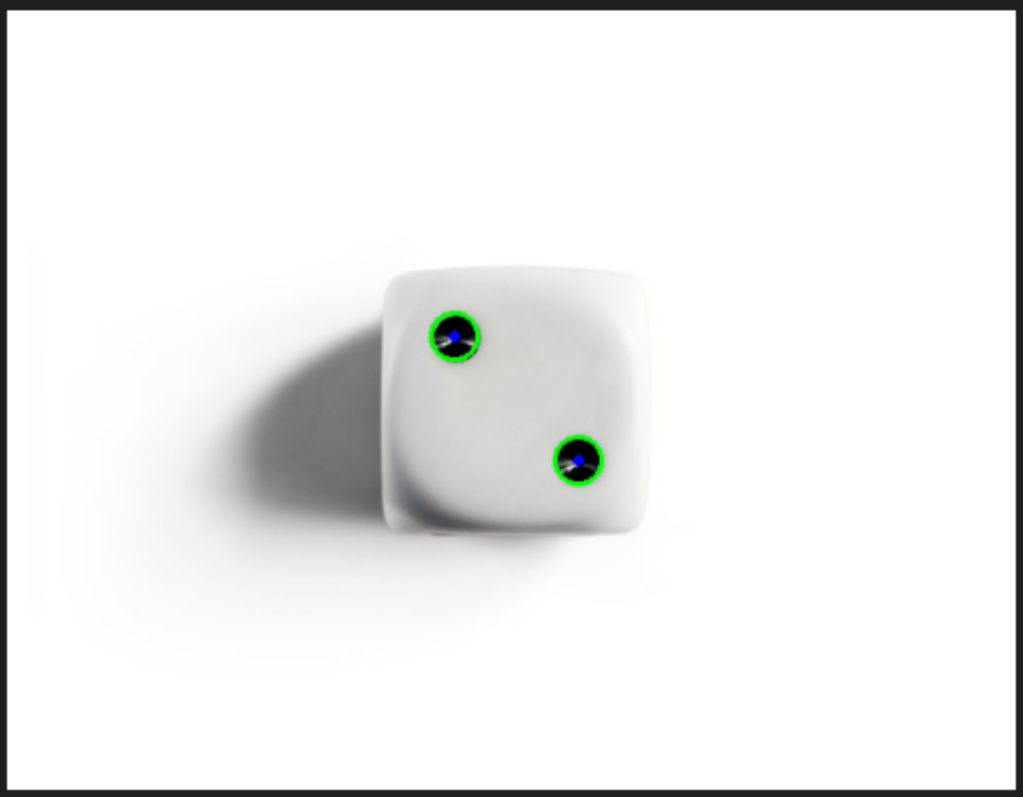
2. Vẽ tâm của vòng tròn

cv2.circle(rgb_img, (int(i[0]), int(i[1])), 2, (0, 0, 255), 3)

- **(int(i[0]), int(i[1])):** Tọa độ tâm của vòng tròn (giống như trên).
- **2**: Bán kính của vòng tròn nhỏ để vẽ tâm (rất nhỏ, chỉ để đánh dấu tâm).
- **(0, 0, 255):** Màu của điểm tâm vòng tròn (đỏ).
- **3:** Độ dày của đường viền điểm tâm vòng tròn.

```
2

(np.float64(-0.5), np.float64(625.5), np.float64(477.5), np.float64(-0.5))
```



```
contours, hierarchy = cv2.findContours(close, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
print(len(contours))
print((hierarchy[0]))
```

1. Tìm các đường viền
contours, hierarchy = cv2.findContours(close, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

- **close:** Ảnh đầu vào đã được xử lý trước đó (thường là ảnh nhị phân sau khi thực hiện các phép biến hình hoặc ngưỡng).
- **cv2.RETR_EXTERNAL:** Chỉ lấy các đường viền bên ngoài (không lấy các đường viền bên trong). Điều này có nghĩa là chỉ những đường viền lớn nhất sẽ được tìm thấy, phù hợp với việc phát hiện các đối tượng riêng biệt trong ảnh.
- **cv2.CHAIN_APPROX_SIMPLE:** Sử dụng phương pháp nén các điểm đường viền để giảm số lượng điểm lưu trữ. Chỉ lưu lại các điểm cần thiết để tạo thành đường viền, điều này giúp giảm kích thước dữ liệu và cải thiện hiệu suất.

2. Số lượng đường viền

print(len(contours))

- **len(contours):** In ra số lượng đường viền đã được tìm thấy trong ảnh. Mỗi đường viền được lưu trong một phần tử của mảng contours.

3. Phân cấp đường viền

print((hierarchy[0]))

- **hierarchy:** Là một mảng cung cấp thông tin về mối quan hệ giữa các đường viền. Cấu trúc của nó thường là:
  - Mỗi đường viền được mô tả bằng bốn giá trị:
    - [Next, Previous, First Child, Parent]
    - Next: Chỉ số của đường viền tiếp theo trong cùng một cấp.
    - Previous: Chỉ số của đường viền trước đó trong cùng một cấp.
    - First Child: Chỉ số của đường viền con đầu tiên (nếu có).
    - Parent: Chỉ số của đường viền cha (nếu có).
- **hierarchy[0]:** In ra cấu trúc phân cấp cho tất cả các đường viền tìm thấy. Nếu không có đường viền con, giá trị sẽ là -1.

Tổng kết:

- **cv2.findContours():** Tìm kiếm các đường viền trong ảnh nhị phân đã cho, chỉ lấy các đường viền bên ngoài và nén dữ liệu đường viền.
- In số lượng đường viền: Số lượng đường viền tìm thấy sẽ được in ra, giúp bạn đánh giá mức độ phức tạp của hình ảnh.
- In cấu trúc phân cấp: Cung cấp thông tin về mối quan hệ giữa các đường viền, có thể hữu ích trong việc phân tích cấu trúc hình ảnh phức tạp hơn.

```
1
[[-1 -1 -1 -1]]
```

```
x0, y0, w0, h0= cv2.boundingRect(contours[0])
cv2.rectangle(rgb_img, (x0,y0),(x0+w0,y0+h0), (0,255,0),5)

plt.imshow(rgb_img)
plt.axis('off')
```

1. Tính toán hình chữ nhật bao quanh

x0, y0, w0, h0 = cv2.boundingRect(contours[0])

- **contours[0]**: Đây là đường viền đầu tiên trong danh sách các đường viền đã tìm thấy. Nếu không có đường viền nào, bạn nên kiểm tra trước khi thực hiện dòng này.
- **cv2.boundingRect()**: Hàm này tính toán hình chữ nhật nhỏ nhất bao quanh đường viền được cung cấp.
- **Trả về**: Bốn giá trị:
  - **x0**: Tọa độ x của góc trên bên trái của hình chữ nhật.
  - **y0**: Tọa độ y của góc trên bên trái của hình chữ nhật.
  - **w0**: Chiều rộng của hình chữ nhật.
  - **h0**: Chiều cao của hình chữ nhật.

2. Vẽ hình chữ nhật

cv2.rectangle(rgb_img, (x0, y0), (x0 + w0, y0 + h0), (0, 255, 0), 5)

- **rgb_img**: Ảnh mà bạn muốn vẽ lên.
- **(x0, y0)**: Tọa độ của góc trên bên trái của hình chữ nhật.
- **(x0 + w0, y0 + h0)**: Tọa độ của góc dưới bên phải của hình chữ nhật, được tính bằng cách cộng chiều rộng và chiều cao với tọa độ góc trên bên trái.
- **(0, 255, 0)**: Màu sắc của hình chữ nhật (xanh lá cây).
- **5**: Độ dày của đường viền hình chữ nhật.

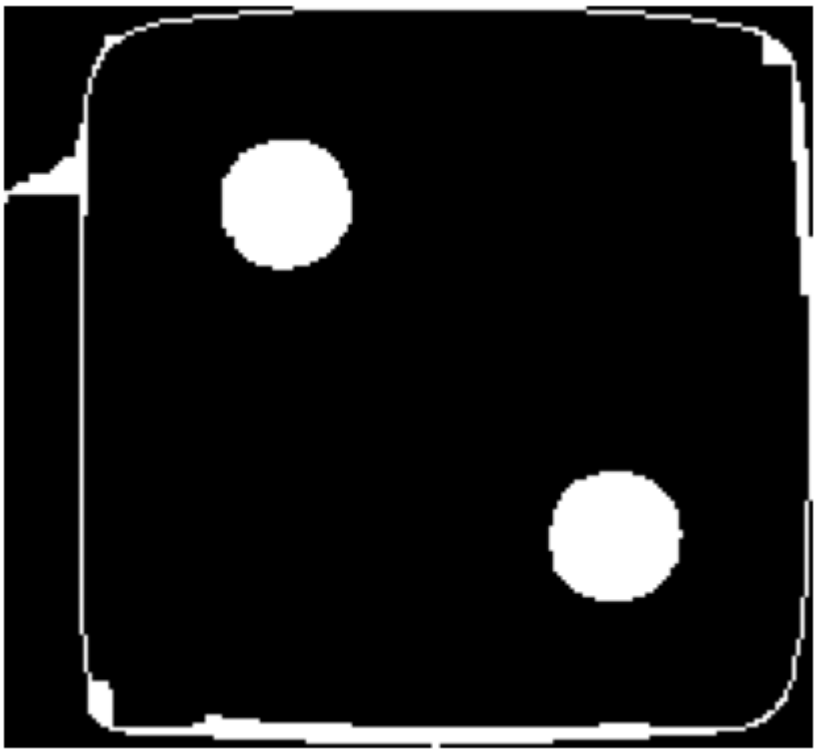
```
dice0 = close[y0:y0+h0, x0:x0+w0]

plt.imshow(dice0, cmap='gray')
plt.axis('off')
```

Trích xuất vùng ảnh: Đoạn mã này cắt một vùng từ ảnh nhị phân close dựa trên tọa độ và kích thước của hình chữ nhật bao quanh đối tượng đầu tiên đã được phát hiện.

dice0 = close[y0:y0+h0, x0:x0+w0]

- **close:** Ảnh nhị phân mà bạn đã xử lý, thường là kết quả của các bước trước đó như phát hiện cạnh hoặc biến hình.
- **y0:y0+h0:** Chỉ định các hàng (rows) mà bạn muốn trích xuất, từ y0 (tọa độ y của góc trên bên trái của hình chữ nhật) đến y0 + h0 (tọa độ y của góc dưới bên phải của hình chữ nhật).
- **x0:x0+w0:** Chỉ định các cột (columns) mà bạn muốn trích xuất, từ x0 (tọa độ x của góc trên bên trái của hình chữ nhật) đến x0 + w0 (tọa độ x của góc dưới bên phải của hình chữ nhật).



```
circles0 = cv2.HoughCircles(dice0,cv2.HOUGH_GRADIENT,1.3,20,param1=50,param2=30,minRadius=5,maxRadius=55)
print(len(circles0[0]))
```

Dòng mã này sử dụng hàm **cv2.HoughCircles()** để phát hiện các hình tròn trong vùng ảnh đã trích xuất **dice0.**

1. Phát hiện hình tròn

circles0 = cv2.HoughCircles(dice0, cv2.HOUGH_GRADIENT, 1.3, 20, param1=50, param2=30, minRadius=5, maxRadius=55)

- **dice0**: Vùng ảnh mà bạn đã trích xuất trước đó, trong đó bạn muốn phát hiện các hình tròn.
- **cv2.HOUGH_GRADIENT**: Phương pháp phát hiện hình tròn sử dụng kỹ thuật gradient. Đây là một trong những phương pháp phổ biến và hiệu quả trong OpenCV để phát hiện hình tròn.
- **1.3:** Tỉ lệ ảnh. Tham số này cho biết tỉ lệ của ảnh đã vào (1.3 nghĩa là kích thước của ảnh vào sẽ được thay đổi thành 1/1.3).

- **20:** Khoảng cách tối thiểu giữa các tâm hình tròn được phát hiện. Điều này có nghĩa là nếu hai hình tròn gần nhau hơn 20 pixels, chỉ một trong số chúng sẽ được giữ lại.
- **param1=50:** Tham số đầu tiên cho thuật toán Canny Edge Detector. Giá trị này xác định ngưỡng cao cho việc phát hiện cạnh.
- **param2=30:** Tham số thứ hai cho thuật toán Hough Circle Transform. Giá trị này xác định ngưỡng cho việc nhận diện các hình tròn; giá trị cao hơn sẽ dẫn đến việc phát hiện ít hình tròn hơn.
- **minRadius=5:** Bán kính tối thiểu của hình tròn cần phát hiện.
- **maxRadius=55:** Bán kính tối đa của hình tròn cần phát hiện.

2. In số lượng hình tròn phát hiện

print(len(circles0[0]))

- **len(circles0[0]):** Số lượng hình tròn được phát hiện trong vùng ảnh **dice0.** Nếu hàm phát hiện thành công, **circles0** sẽ chứa thông tin về các hình tròn đã phát hiện dưới dạng một mảng. Mỗi hình tròn sẽ được mô tả bởi ba giá trị: tọa độ x, tọa độ y và bán kính.

Tổng kết:

- **Phát hiện hình tròn:** Đoạn mã này sử dụng thuật toán Hough Circle Transform để phát hiện các hình tròn trong vùng ảnh đã trích xuất.
- **In số lượng hình tròn:** Số lượng hình tròn đã phát hiện sẽ được in ra, giúp bạn đánh giá mức độ thành công của quá trình phát hiện.

```
cv2.putText(rgb_img, f'score: {len(circles0[0])}', (x0, y0), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,255), 2)
plt.imshow(rgb_img)
plt.axis('off')
```

Dòng mã này sử dụng hàm **cv2.putText()** để thêm văn bản vào ảnh, cụ thể là hiển thị số lượng hình tròn đã phát hiện trong vùng ảnh **dice0.** Dưới đây là giải thích chi tiết cho từng phần trong mã:

1. Thêm văn bản vào ảnh

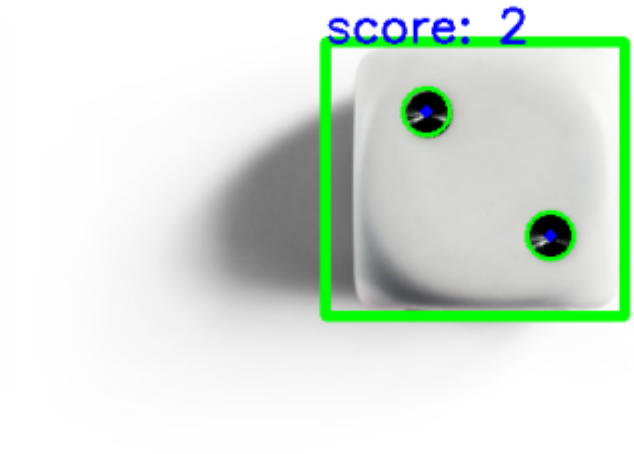
cv2.putText(rgb_img, f'score: {len(circles0[0])}', (x0, y0), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,255), 2)

- **rgb_img:** Ảnh mà bạn muốn thêm văn bản vào. Đây là ảnh đã được chuyển đổi sang dạng RGB.
- **f'score: {len(circles0[0])}'**: Văn bản bạn muốn thêm vào ảnh. Đây là một chuỗi định dạng (f-string) trong Python, trong đó **len(circles0[0])** là số lượng hình tròn được phát hiện trong vùng **dice0.** Kết quả sẽ là một chuỗi như **score: 3** nếu có 3 hình tròn được phát hiện.
- **(x0, y0)**: Tọa độ (x, y) nơi văn bản sẽ được vẽ trên ảnh. Trong trường hợp này, tọa độ này là góc trên bên trái của hình chữ nhật bao quanh (bounding box) của đối tượng mà bạn đã tính toán trước đó.
- **cv2.FONT_HERSHEY_SIMPLEX:** Kiểu phông chữ sử dụng để vẽ văn bản. Đây là một phông chữ đơn giản và dễ đọc.
- **1**: Kích thước phông chữ, giá trị này có thể thay đổi để làm cho văn bản lớn hơn hoặc nhỏ hơn.
- **(0, 0, 255)**: Màu sắc của văn bản dưới dạng BGR (Blue, Green, Red). **(0, 0, 255)** là màu đỏ.
- **2**: Độ dày của đường viền văn bản. Giá trị này càng lớn, đường viền văn bản càng dày.

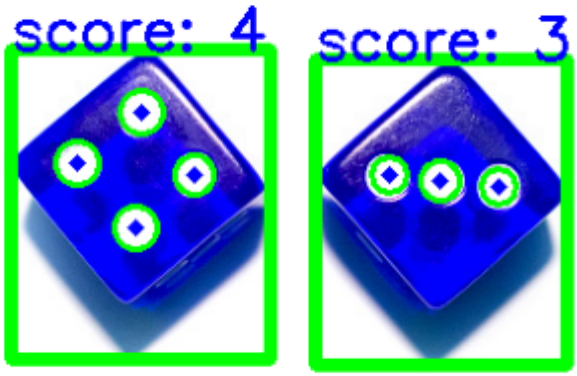
Tổng kết:

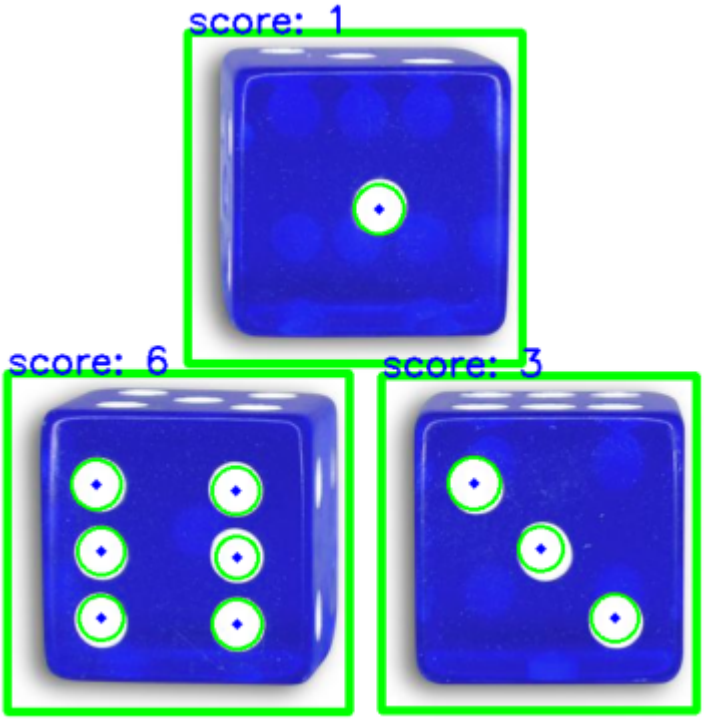
- **Thêm văn bản vào ảnh:** Đoạn mã này thêm văn bản vào ảnh để hiển thị số lượng hình tròn đã phát hiện, với văn bản được vẽ bằng màu đỏ và hiển thị tại tọa độ **(x0, y0).**

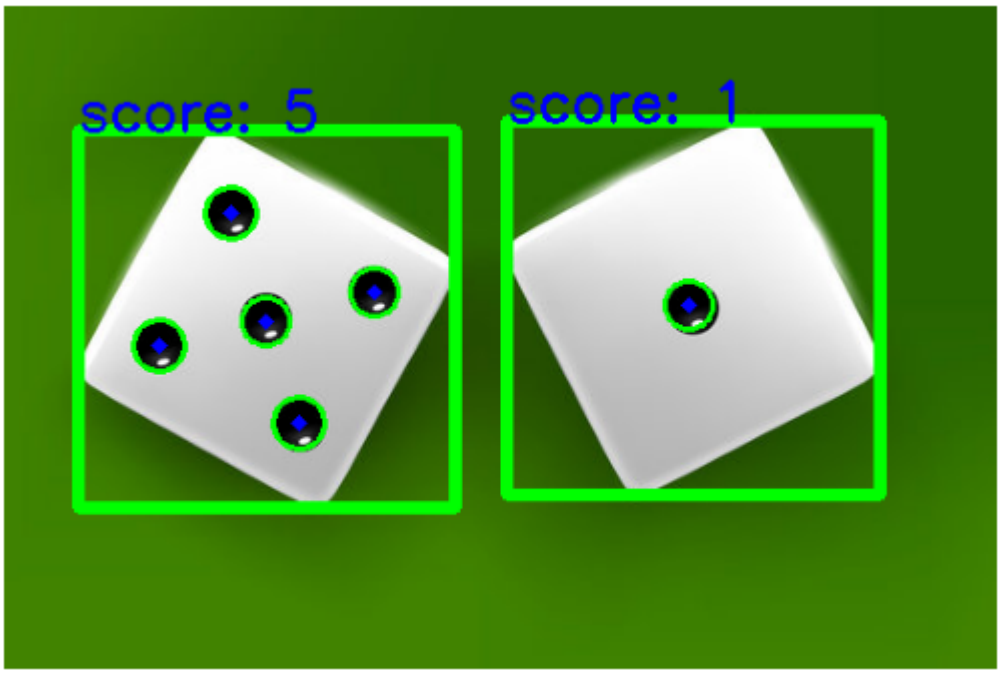
- **Hiển thị kết quả:** Sau khi văn bản được vẽ, ảnh được hiển thị với văn bản này để bạn có thể dễ dàng thấy số lượng hình tròn đã phát hiện.



Làm tương tự cho các hình còn lại




score: 1
score: 6
score: 3


score: 5
score: 1