**Theory:**

**1. Preprocessing the Image:**

* In this step, we preprocess the input image to enhance its quality and reduce noise.
* We convert the image to grayscale and apply Gaussian blur to smoothen it.

**2. Detecting Elements:**

* After preprocessing, we perform edge detection using the Canny edge detection algorithm.
* Then, we find contours in the edge-detected image to identify shapes and objects.
* Based on the contours' properties, we classify them into different categories such as rectangles, circles, logos, and text boxes.

**3. Printing Detected Elements:**

* Finally, we print out the detected elements along with their positions.

**Code Explanation:**

import cv2 import numpy as np from google.colab.patches import cv2\_imshow

* We import the necessary libraries: **cv2** for computer vision operations, **numpy** for numerical operations, and **cv2\_imshow** from Google Colab patches for displaying images.

pythonCopy code

def preprocess\_image(image): """ Preprocesses the input image by converting it to grayscale and applying Gaussian blur. Args: - image: Input image (BGR color space) Returns: - preprocessed\_image: Preprocessed image (grayscale) """ # Convert the image to grayscale gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) # Apply Gaussian blur to reduce noise blurred = cv2.GaussianBlur(gray, (5, 5), 0) return blurred

* The **preprocess\_image** function takes an input image and preprocesses it by converting it to grayscale and applying Gaussian blur.
* Grayscale conversion simplifies the image by removing color information, making it easier to detect shapes and objects.
* Gaussian blur reduces noise in the image, which helps in improving the accuracy of subsequent processing steps.

**def preprocess\_image(image):**

**"""**

**Preprocesses the input image by converting it to grayscale and applying Gaussian blur.**

**Args:**

**- image: Input image (BGR color space)**

**Returns:**

**- preprocessed\_image: Preprocessed image (grayscale)**

**"""**

**# Convert the image to grayscale**

**gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)**

**# Apply Gaussian blur to reduce noise**

**blurred = cv2.GaussianBlur(gray, (5, 5), 0)**

**return blurred**

* The **detect\_elements** function detects different types of elements (rectangles, circles, logos, text boxes) in the input image using contour detection.
* We first preprocess the image by converting it to grayscale and applying Gaussian blur.
* Then, we perform edge detection using the Canny edge detection algorithm to detect edges in the image.
* Next, we find contours in the edge-detected image and classify them based on their properties (number of vertices).
* Detected elements are stored in lists and organized into a dictionary for easy access.

def detect\_elements(image\_path):

"""

Detects elements in the input image using contour detection.

Args:

- image\_path: Path to the input image

Returns:

- detected\_elements: Dictionary containing detected elements and their positions

"""

# Read the image

image = cv2.imread(image\_path)

# Preprocess the image

preprocessed\_image = preprocess\_image(image)

# Perform edge detection

edges = cv2.Canny(preprocessed\_image, 50, 150)

# Find contours in the edge-detected image

contours, \_ = cv2.findContours(edges.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

# Initialize lists to store detected elements

detected\_rectangles = [] # Rectangular elements

detected\_circles = [] # Circular elements

detected\_logos = [] # Logos (rectangles with specific aspect ratio)

detected\_text\_boxes = [] # Text boxes (irregular shapes)

# Iterate through detected contours

for contour in contours:

# Calculate contour area

area = cv2.contourArea(contour)

# Ignore small contours (noise)

if area > 100:

# Approximate the contour to a polygon

epsilon = 0.02 \* cv2.arcLength(contour, True)

approx = cv2.approxPolyDP(contour, epsilon, True)

# Get the number of vertices

vertices = len(approx)

# Classify contours based on the number of vertices

if vertices == 4:

# Rectangle

detected\_rectangles.append(cv2.boundingRect(approx))

# Check if the rectangle resembles a logo based on aspect ratio

x, y, w, h = cv2.boundingRect(approx)

aspect\_ratio = w / h

if 0.5 < aspect\_ratio < 2:

detected\_logos.append((x, y, w, h))

elif vertices >= 8:

# Circle

(x, y), radius = cv2.minEnclosingCircle(contour)

detected\_circles.append((int(x), int(y), int(radius)))

elif vertices > 4:

# Text box

detected\_text\_boxes.append(cv2.boundingRect(approx))

# Prepare dictionary to store detected elements

detected\_elements = {'Rectangles': detected\_rectangles,

'Circles': detected\_circles,

'Logos': detected\_logos,

'Text Boxes': detected\_text\_boxes}

return detected\_elements

The detect\_elements function detects different types of elements (rectangles, circles, logos, text boxes) in the input image using contour detection.

We first preprocess the image by converting it to grayscale and applying Gaussian blur.

Then, we perform edge detection using the Canny edge detection algorithm to detect edges in the image.

Next, we find contours in the edge-detected image and classify them based on their properties (number of vertices).

Detected elements are stored in lists and organized into a dictionary for easy access.

def print\_detected\_elements(detected\_elements):

"""

Prints the detected elements along with their positions

Args:

- detected\_elements: Dictionary containing detected elements and their position

Returns:

- None

"""

for element\_type, elements in detected\_elements.items():

print(f'Detected {element\_type}:')

for index, element in enumerate(elements, start=1):

print(f'{element\_type} {index}: {element}')

print('\n')

* The **print\_detected\_elements** function prints the detected elements along with their positions.
* It iterates through the dictionary of detected elements and prints each type of element along with its index and position.

**# Example usage**

**image\_path = '/content/Screenshot 2024-03-23 233800.png' # Replace with the path to your image**

**detected\_elements = detect\_elements(image\_path)**

**print\_detected\_elements(detected\_elements)**

* Finally, we provide an example usage of the code by specifying the path to the input image.
* We call the **detect\_elements** function to detect elements in the image and store the results in **detected\_elements**.
* Then, we print the detected elements using the **print\_detected\_elements** function.

Top of Form